Kieran Hogan

C12561353

Distributed Systems

Assignment 1

19th of November 2015

DECLARATION

I declare that this work, which is submitted as part of my coursework, is entirely my own, except where clearly and explicitly stated.

Signed: [illegible]

Date: 19/11/15

How to start the server and clients.

server.bat file:

```
REM Usage: java AuctionServer port

java -classpath . AuctionServer 1234
```

client.bat file:

```
REM this bat file runs the server. Usage: java AuctionClient host port name

java -classpath . AuctionClient localhost 1234 kieran
```

Design Document

This application is a Client-Server tier application. It can support multiple clients. There are 5 java source files required for the Auction system to work. The AuctionServer which invokes AuctionServerThread, manages all of the auction logic. This includes analysing bids, controlling the auction timer and feeds the auction information to the client side. The AuctionClient which invokes AuctionClientThread, manages the client end. This simply involves processing bids, and sending them to the server side. The item file is used to create the auction objects. *I have used the chat server application from week 4 given in the labs as the base of my application*. This is because it already implements the required base functionality and elements. I then built the rest of the application on top of it.

Client Side

The client side is made up of the AuctionClient and AuctionClientThread classes. The Auction Client is started with command line arguments of the host, the port and the name. This is handled in the main() method, and when run correctly creates the AuctionClient object. When it runs the, AuctionClient begins by creating the output and input streams in start(), as well as creating a new thread for the client. It does this by using AuctionClientThread. AuctionClientThread manages the server's communication messages to the client side. Every time a client is created, a thread is created for that client.

The AuctionClient also runs a check on the user input. Since it is an auction, the input needs to be an integer value. Other inputs won't be accepted, and will be caught by the numberFormatException. If the number is accepted, it is then sent to the server as a valid bid value, where further processing is done. It is important to note that the AuctionServer needs to be running before the AuctionClient can run and connect.

Server Side

The server side is made up of the AuctionServer and AuctionServerThread classes. The Auction Server is started with one command line argument, the port. This is handled in the main() method, and when run correctly creates the AuctionServer object, similarly to how the AuctionClient works. When it runs the, AuctionServer binds to a port, adds the objects to a list, and creates the server thread. The server also calls the Timer() method. The timer method manages all of the timer based logic in the application and starts the auction. The timer uses the ScheduledExecutorService to call the method repeatedly, and is set to do so every second. If the timer is greater or equal to 0, it prints the time and decrements by 1. For the ease of testing and rapid demoing, my time is set to 9, allowing for auctions to be done quicker. In reality, this can be set to whatever by changing the timeDefault value.

When the timer hits 0, the application checks the current items highest bid. If the bid is less than the reserve price, the current item is changed to the next item in the list. If the number of items is equal to 2, it ensures that it alternates to the item not on auction. Otherwise, the current item is reset to the first item in the list. The client is then sent the updated information regarding if the item was sold or not. This is also printed on the server. If the bid is greater than or equal to the reserve price, similar logic is used. The only difference is that once the item is sold, it's removed from the list. The index of the items is also updated here, because since the list has had an item removed, the other items have moved up. I have also tried to implement that when all of the auction items are sold, it closes gracefully. However this function does not work, and the timer remains at 0.

The run() method here waits for clients to connect, and when they do the addThread() method is called. This method creates a thread for each client that connects. On connection, they are updated with the current item being auctioned, and the time left for the item auction. This method also catches exceeding the max clients. The remove method is invoked whenever a client disconnects from the server. This method uses notifyAll() which allows it to always be listening and aware of disconnects. When a client is disconnected, the current bid gets reset. I found this was the fairest way to handle clients leaving the auction.

The broadcast method is used for responding to messages received from the client. If a message has been received it means the bid has been received (has passed client side int verification), so it must be a valid integer. If the new bid is bigger than the current items current highest bid, it is changed to the new bid. This is then broadcast to the users. The user who bid is notified that they are the new highest bidder, and the other users are also notified of the new highest bidder. If the new bid was found not to be larger than the current highest bid, then the user who put the bid in is messaged to say the bid wasn't accepted due to being too low. The Item objects are also created in the AuctionServer, and then placed into an array list. Each Item has a name, a current bid, a reserve bid and an index of where it is in an array list, which gets updated when items are sold and removed.