This project runs an auction system that can support multiple clients and multiple backend servers.

**Auction logic**

The system can run 3 different types of auctions – forward, reverse, and double auction. The forward auction is an auction as most people would know it – a seller puts an item up for sale, names a starting price, and buyers bid on it until the seller decides to close the auction. The starting price is a suggestion – bidders are free to start at any price, but all bids must be higher than the bid before it. The reserve price is a seller’s internal limit – if, upon closing, the highest bid is lower than the reserve price, the sale does not happen. Sellers can sell multiple copies of the same item at once – If a seller has 2 plates, they can create 2 different auctions for “Plate”. These can have different descriptions, but they do not have to – as the items may be completely identical.

The double and reverse auctions make use of a grouping system I call “Listings”. A listing has a name and a description for it. All individual items belonging to a specific listing are considered to be homogeneous and interchangeable. An individual item shares the name and description of all the items in the listing but has their own price and seller. When these auctions are closed, the individual items get removed from the listing, but the listing remains, possibly containing other items from other sellers.

Listings are created when a buyer or seller attempts to place an order for an item that does not have a listing with the same name. They are offered to create a listing, and their new order becomes the first entry for the listing. Other users can then add their own offers – a listing does not belong to any specific user. The creator of a listing has no special privileges or control over that listing.

For the reverse auction, sellers add their items to a listing at a fixed price. A buyer can come in at any time, view a listing, and buy the item with the lowest price in that listing. The other sellers’ postings remain in the listing, and a future buyer can come in and buy them. For the double auction, both buyers and sellers can place orders in a listing at their desired price. Whenever a new order is added, it checks for matches. If there is a sell order with a selling price lower than any buy order, a match is made between the lowest-price sell order and the highest-price buy order, and the transaction is complete. The other orders – both buy and sell – remain in the listing for future matches. Users can also cancel their orders before they are fulfilled.

Besides the 3 auctions, there is also a text message system – when a noteworthy event happens, such as an auction closing with a user as a winner, the user gets sent a message informing them what happened. Users cannot send messages to each other directly – only the system sends messages to users at relevant times.

**Architecture**

The system follows a client-server architecture, with the ability to support multiple clients, and the server being a distributed system with 1 frontend server and many backend replicas. The client deals with showing the user relevant information on a command-line, and with sending the frontend server requests via RMI, and displaying the responses. It does not do any business logic or store any data. The only exception is creating an *Account* object from a username, email and password. This involves generating a salt for the password and hashing them. This used to be done server-side in previous iterations of the project, however, random generation of salts broke active replication of backend servers – the servers would each generate different salts, which would make the password hashes different and break passing accounts by reference, because the accounts would have different values in their *password* field in each backend replica. Moving this functionality to the client side does not create any extra security risk, because there is still no way to access another user’s account without knowing their username and password.

The frontend server binds to Java’s *rmiregistry* and is connected to the *JGroups* channel with all the backend servers. It takes the requests received from the client, invokes the equivalent methods in the backend servers, checks if all the responses are the same and, if they are, returns the response back to the client. The frontend server has no data stored in it and performs no business logic besides forwarding requests to backend servers and checking for consensus.

The backend server is the where both the data is stored (via a separate *ServerState* class) and the business logic is performed. It has methods for every action a user might take, and those methods get invoked by the frontend server when a client makes a request. Upon start-up, the backend server gets the state of the *JGroups* channel via an *RpcDispatcher* call to the channel coordinator. If the coordinator is the frontend server, the frontend server routes the request to the next oldest member of the channel. This state transfer, together with the fact that all requests are sent to all backend servers, ensures that the state of all backend servers will remain consistent.

**Class relationships**

A diagram of a company

Description automatically generated

Figure 1. High-level class diagram of the project

Figure 1 shows the relationships between the classes in the project. They have been logically grouped in a way that also reflects the directory structure of the project. The classes in the SharedObjects group are all used by almost all of the other classes, with few exceptions – thus the package-to-package arrows, instead of cluttering the diagram with drawing a line between every combination. *Client* is the main class for the client – it deals with logging in/creating an account, and when a user selects a type of auction, it delegates the functionality to the respective class. All of the *Client* classes connect to the server using *rmiregistry*, and have access to an instance of the *IRemoteAuction* stub. *FrontendServer* implements *IRemoteAuction*, processes the RMI requests, and calls the appropriate methods from the *BackendServer* (there can be multiple backend servers running) using a *JChannel* with an *RpcDispatcher* on top. Finally, *BackendServer*’s methods compute and store the data in *ServerState*, a wrapper object for multiple fields. *AuctionItem* and its subclasses represent a single listing, and *Bid* is a simple class with different uses based on what type of auction is running.

Part 3:

Idea is 1 frontend server, presumed to be unable to crash, and many backend replicas. Frontend uses rpc to make backends execute same methods in same order, maintaining same state. Problem – account creation uses SecureRandom, every backend will generate different salts/password hashes. Solution – create accounts client-side, and make “createAccount” take an account as a parameter. Should be safe.

State transfer doesn’t work on a channel that has an rpcdispatcher on it. No idea why. Found stackoverflow question mentioning that, tested, that’s the case