General information

This project aims to create an auction system using a client-server architecture and a distributed system with active replication for servers. It can support any number of concurrently running clients and backend server replicas. To start a new server replica, create a new process of *BackendServer*. To stop/crash a replica, kill its process. More information on how to run the system can be found in *readme.txt.*

Auction logic

The system can run 3 different types of auctions – forward, reverse, and double auction. The forward auction is the standard, well-known type of auction – 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 physical 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 any 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

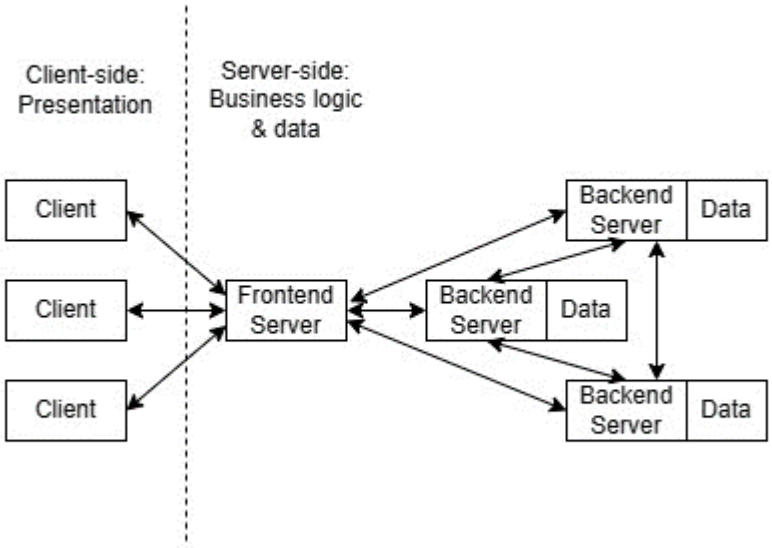

Figure 1. Architecture diagram

Figure 1 shows the architecture of the system. The system follows a client-server architecture, with the ability to support multiple clients, and the server being a distributed system with one 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 Java 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 and the requests are deterministic, ensures that the state of all backend servers will remain consistent. The backend server is, therefore, both the business logic and the data tier of the architecture.

Class relationships

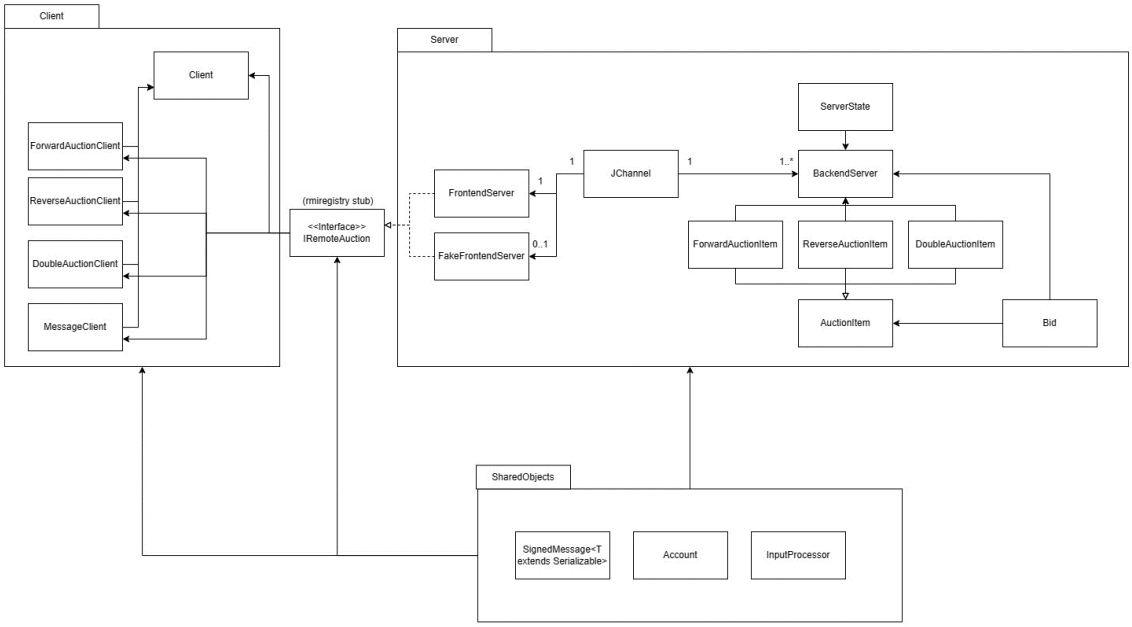

Figure 2. High-level class diagram of the project

Figure 2 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 (or the message system), it delegates the functionality to the respective class. All of the *Client* classes connect to the server using *rmiregistry* and have access to the same 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. Each *BackendServer* maintains its own state. *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.

**Note – the* IRemoteAuction *Java file is in the* SharedObjects *folder in the directory structure*.

Design rationale

Access control

I am using accounts, defined in the *Account* class, to represent individual users. An account can be both a buyer and a seller in any or multiple auction types, however, it cannot be both for the same exact listing at the same time. Users create accounts via the client and get a reference to their *Account* object from the server by logging in with a username and password. The passwords are salted and hashed with SHA-256. If an issue arises when hashing a password, a custom *InvalidPasswordException* is thrown. After getting a reference to their *Account*, the client stores it and passes it back to the server as an argument whenever a method requires it. Storing the account locally after logging in simulates a session from the user's point of view. All methods that require authentication take an argument of type *Account* for who is making the request. Clients never have access to any *Account* object that isn't theirs, due to only being able to get an *Account* object by 1) creating one locally from a username and password or 2) via the login() function of the server. Either option requires knowing the username and password of the other user.

Digital signatures

Digital signature functionality is implemented in the *SignedMessage<T extends Serializable>* class. Every method in the RMI stub returns a *SignedMessage*, and the client verifies the message using the server's public key. All backend servers share the same private key, and the frontend server does not have a key – it only relays the responses from the backend. The keys are stored in *.jks* files – private key is stored in the server folder, public key is stored in the client folder. I used generics to make this a general class for any return type – the generic class extends *Serializable* because a) the message needs to be hashed and b) all possible messages already had to be serializable because of the requirements of Java RMI.

SignedMessage contains the message itself, a hash (SHA-256) of the message, and the signature of the sender. The signature is the hash of the message encrypted with the sender's private key (RSA). It also contains a method for validation and some helper methods, including a method for validating consensus of multiple server replicas.

The server, after computing the message it wants to return to the client, encapsulates it in *SignedMessage* and returns it. The client unwraps the message and checks if the signature is valid. To test this, I added a *FakeFrontendServer* class, which, if used, puts itself between the client and the real frontend server. It alters the return value of one of the methods (placing a forward auction bid) without changing the signature, and it can optionally re-sign all messages sent by the real frontend with its own (wrong) private key. The client correctly detects when the fake server tampers with the message and when it re-signs messages.

Active replication

I use one of *JGroups*' building blocks, *RpcDispatcher*, to maintain backend replicas. There is no replica manager – replicas are created by starting a new *BackendServer* process and automatically join the channel. Replicas are removed by killing their process. The backend servers communicate with the clients via the frontend server. The frontend server's main functionality is to take the request from the client and convert it into an *RpcDispatcher.callRemoteMethods()* call. The call is sent to every member of the cluster (the

frontend server is set to discard own calls). I made every method of the backend server deterministic – with the same initial state and the same request with the same arguments, it will result in the same end state. This is the reason for moving creation of *Account* objects client-side, as the random generation of salts for an account was non-deterministic. Together with getting the initial state upon start-up (see next section), this makes the backend replicas maintain consistent state. However, this implementation does not account for partial/byzantine backend server errors, so there is still a possibility that states with become inconsistent. If they do, and they return different values to a method call, the frontend server will catch it by comparing the responses and will throw a custom *NoConsensusException* to the client.

Fault tolerance

Fault tolerance is achieved by transferring the network's state to any newly joined backend server, making so that, no matter when a backend server joined, it will have the same state upon startup. State transfer was surprisingly difficult to implement, due to a quirk in *JGroups* – if an *RpcDispatcher* is setup on a channel, the standard state transfer functionality of the channel (getState() and setState() methods of *MessageListener*) does not work. A possible solution is to use another channel only for state transfer, however, I decided to use *RpcDispatcher* again, as it seemed like the easiest/fastest to implement solution. Upon startup, a backend server calls getState() from the channel coordinator using *RpcDispatcher.callRemoteMethod()*. If the coordinator is the frontend server (which has no state), the frontend server forwards the request to the next oldest member of the channel. This has achieved the desired result – no matter how many backend servers crash/leave the channel, if a single functioning backend server remains, the state is maintained, and new backend replicas can get the state from the surviving server. This also allows for complete backend replica turnover.