Distributed Systems Coursework Report

by Nellie Mugattarova

December 12, 2023

1 Architecture

(see Fig. 1 for a diagram representing the Architecture)

- **Presentation Layer**

  The application has one client for both sellers and buyers, with appropriate permissions hadled by the business logic, for example, one user cannot close another user's auction. The helper class, InputHandler, helps sanitize input and ensure a more user-friendly interaction.

- **Processing Layer**

  The BackendServer classes implement all of the functions of the RMI registry, so the user side does very little work (of input sanitation).

- **Data Layer**

  There are 4 hashtables with all of the information: *auctionItems*, which stores items that can be a part of an auction, by their ID, which is assigned iteratively through a static field. This table was mostly created for the ease of use (so the testing is much quicker). Any auctions created with a new item add the item into the list and there's no way to add an item without creating an auction, since this list is intended to be a lookup table or as a view of past items you can auction.

  Next hashtable, *openAuctions*, stores currently open for bidding auctions, by auctionID (also assigned at the class's static id counter). It stores all types of auctions together, as opposed to different tables for each type, since it makes the system more extendable: if there's a new type of auction, there's no need to create a new database structure, just the getter of that type will be enough to obtain it from the database.

  The two last hashtables go together: *passwordSalt* stores password salt for a user's email, and *accounts* stores UserAccount objects with the key being the hashed salted password. This way there's no password stored anywhere

in the system (UserAccount also has no such field), so, arguably, it's a little more secure. I'll talk about user authentication more in the business logic section below.

In addition, for the class relationships see Fig. 2

2 Business Logic (in detail)

2.1 Access Control

As with the client application, there is only one user identity. The same user can act as a buyer and a client in an auction, even their own.

The users have correlating user accounts. They are free to log in and out on different connections, retaining their data. It is possible to log in on multiple connections with the same account. Once users log in to the system, they get a UserAccount object back, which will act as a verification for actions.

The users own the auctions they created, and only passing their UserAccount object will verify their identity in order to close any auctions they own, or place bids under their name. The account system is simple (a log in with email and password), but employs additional security measures, as described in the data layer section.

Salt for passwords is usually randomly generated, but for the active replication to work, there needs to be a way to get the same salt for the same user every time. This job is delegated to the FrontendServer. It generates the salt and passes it onto all of the backends running. Once it is generated, it's stored in the database, so future replicas will get the same information (through the passwordSalt hashtable).

2.2 Digital Signature

Communication is done via SignedMessage objects, that act as a wrapper for the server response. It consists of the response itself and the server signature. Since it was valid to assume the key exchange has already happened, I generated a key pair in a .der format, so it's readable by Java, distributed it in correct folders and treated them like keys in memory.

Now, there's a lot of java libraries that handle key generation/key reading etc. Before arriving at my current solution, I have written a key generation helper class, but decided against it, since it was causing a lot of issues compared to not using it, like distributing keys to the user and the server, since a singleton class would have different instances at the two replicas. There were a lot of workarounds to be done, so I decided against this method.

2.3 Active Replication

A single frontend server binds to the RMI registry, and cannot fail, create another instance of it, etc. Through JGroups, at least one backend server must be created for the service to actually work, since the frontend does very little for the actual application.

Frontend performs no job in terms of business logic(except for the part mentioned in access control), but coordinates the replicas with RpcDispatcher, which allows to invoke the same method in all of the replicas. It does, however, ensure that all of the responses match, and only after ensuring, does it direct the message to the user.

2.4 Fault Tolerance

Due to how JGroups, rmiregistry and RpcDispatcher interact, the built-in and recommended getState() and setState() functions do not work. One way to overcome this is to set up another JChannel for transferring state, but then the issues of port coordination arise, since they can connect to the same port as for the main communication. I have decided to take a different route: a wrapper object BackendState contains the data, and this can be shared between the replicas. So whenever a new replica joins the cluster, it asks the coordinator of the cluster (chosen by JGroups) to transfer state (BackendServer implements getState), then sets own state to the received one.

This worked well, until I decided to leave a backend server that joined after the initial start running alone. JGroups' cordinator choosing system typically puts the oldest replica in the cluster to be the coordinator. It also has no way of differentiating between the frontend and the backend servers, so this results in the frontend server becoming the coordinator. Since it has no data, the coordinator either needs to be reassigned, or redirect the inquiry to a backend server, and I do the latter, which is simpler and less time-consuming (if the former is possible at all).

The frontend gets the view of the cluster, and since the view is updated as a list and by age, it can be deduced that since frontend is being asked this, it must be the oldest in the cluster (at index 0), and if there is a backend active, it must be the next in the list (at index 1). So I callRemoteMethod on that specific replica (since there's guaranteed to be at least one, otherwise there's no way to save a state at all) and get state from it.

2.5 Attachments

Figure 1: Overview of the architecture

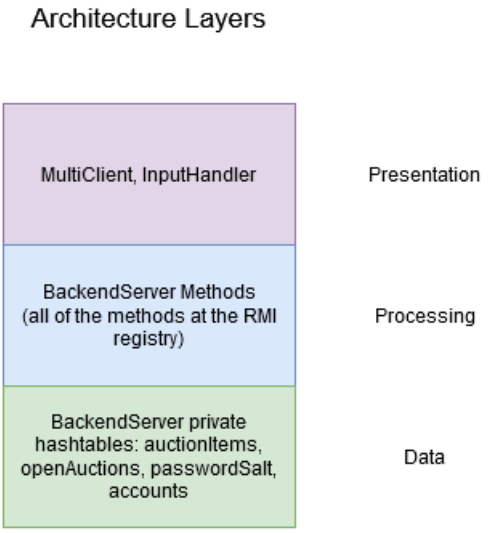

Figure 2: General class diagram of the project