Project Report: 1

We've decided that we'll split major work for the server and the client. However, this doesn't exclude us from working together on the server or the client application. With that being said, our application is currently under a placeholder name titled `Project-Voxx`. It's a global chat application where a random user can drop in and chat with people that are connected and registered to the server.

Project Voxx is managed using `Gradle` and is subdivided into three modules: `voxx-commons`, `voxx-client`, and `voxx-server`. The module names are pretty self-explanatory, `voxx-commons` contains code that both the client and the server would use. The module `voxx-client` will contain the code for the client. And lastly, the `voxx-server` module contains the code for the server.

Since the project proposal, we've dedicated most of our times brainstorming and designing the server and the communication protocol. The implementation of the server is pretty much almost done as well. All that's really needed is some polishing and testing.

Project-Voxx Server Protocol

This documentation specifies what the server for Project-Voxx is going to accept with its corresponding response and messages that a server could send. Therefore, a client needs to anticipate non-requested messages from the server.

Server-Client Communication Overview

At its core, Project-Voxx protocol is going to be transported through `websocket`. The communication is text based, therefore, all the message received and sent by the message is going to be a `String`. To have a clear and easy to parse message, we've opted into using the `json` syntax for our messages. However, the `json` syntax needs to be flattened and `must not have any break line`. Use the following regex replacement on your `json` string before sending requests to the server:

Java

```
"<your jason string here>".replaceAll("\\s{2,}|\\n","");
```

Python

```
import re

re.sub(r'\s{2,}|\n', '', "<your jason string here>")
```

Server Connection

When a client connects to the server, the connection is not affiliated with any user until the client socket sends a request `ru` or `register user` to the server. If the user is with the same username is not registered, the socket connection will be bound to that user and every request from or to the client is in the context of that user.

Request format

Before we can list down all the valid request that a client can make to the server, let's talk about the format of a request. The format of a request is pretty straight forward and looks like the following:

```
{
  "request-id": "request key",
  "params": {
    "param-1": "Some parameter 1",
    "param-2": "Some parameter 2"
  }
}
```

As we can see, the format is in the `json` syntax. This makes it so that we can serialize and deserialize an object through a `websocket` easily. The attribute named `request-id` is the name of the request that we are making to the server. Depending on the parameters that the request accepts, we need to provide the right amount of parameter that goes with that request, and we must put them in the attribute `params` and attribute params must match the attribute name defined in the request documentation.

For undefined requests the server is going to respond with the following message:

```
{
  "response-id": -1,
  "body": {
    "message": "{request-id} is not a valid request"
  }
}
```

We can see that the `response-id` is `-1` meaning that the request does not exist. And a body attribute with
a `message`.

For valid requests, each request type will have their own unique response, and they will be specified below.

Requests

The following headers will show specifications of each request.

Register User

Here is the request body that you must send to the server to have a valid register user request

```
{
  "request-id": "ru",
  "params": {
    "uname": "{username}"
  }
}
```

Since Project-Voxx is a non-persistent drop in public chat channel, user do not need a password. The only requirement for user registration is a username and that it's not taken by any other users in the live server.

If the client socket sends a request with a username that's already taken the server will respond with the following:

```
{
  "response-id": 0,
  "body": {
    "message": "{request username} is already taken"
  }
}
```

A response ID `0` means that the request is invalid due to improper parameters or unsatisfied requirements.

If the user registration is successful, the server will respond with the `UID` for the user. A UID is a unique identifier that could be associated with a user or a message. Since the UID contains the timestamp, it allows us to sort users based on when they registered to the server. The following is an example of a successful user registration:

```
{
  "response-id": 1,
  "body": {
    "user": {
      "uid": 6884583347257344,
      "uname": "{username}"
    }
  }
}
```

Sending a Chat Message

When sending a message to the server, the client socket must have a bound `user` first. Therefore, the client socket must request an `ru` (register user) first. With that being said, here's an example request for sending a message:

```
{
  "request-id": "sm",
  "params": {
    "message": "{message}"
  }
}
```

However, nothing can stop a client form sending a send message request even without sending a user registration request first. If this is the case, the server will respond with the following invalid request response:

```
{
  "response-id": 0,
  "body": {
    "message": "User registration is required before sending a message!"
  }
}
```

When a chat message request was handled properly the server should respond with the following message:

```
{
  "response-id": 1,
  "body": {
    "message": {
      "message-uid": 6884583351369728,
      "message-content": "{message sent}"
    }
  }
}
```

Getting User List

A request that the client can make to get all the registered user in the server.

The request body for getting the user list is super simple and does not require any parameter:

```
{
  "request-id": "ul"
}
```

Since this request does not require any parameter/argument, the response will always be a `1`. And here's an example of a possible response from the server:

```
{
  "response-id": 1,
  "body": {
    "users": [
      {
        "uid": 6884583351369729,
        "uname": "{username1}"
      },
      {
        "uid": 6884583351373824,
        "uname": "{username2}"
      },
      {
        "uid": 6884583355506688,
        "uname": "{username3}"
      },
      {
        "uid": 6884583355506689,
        "uname": "{username4}"
      },
      {
        "uid": 6884583355510784,
        "uname": "{username5}"
      },
      {
        "uid": 6884583359643648,
        "uname": "{username6}"
      }
    ]
  }
}
```

As you can see, the response body has one attribute named `users` that contains a `json` array of users. If there is no user in the server, the array will be an empty array.

Update Messages

Update messages are messages that are sent by the server to the clients to update clients about changes that happens in the server. This is for when a user sends a chat message to the server, a new user registers, or when a user disconnects. The client can do whatever they want to do with the update messages, but they're there so that the clients can display up-to-date information in the server.

New User Update Message

This update message is sent to each of the client when a new user is registered so that clients can update their user list (if being tracked). Here's what the update message looks:

```
{
  "update-message": "nu",
  "body": {
    "user": {
      "uid": 6884583359643649,
      "uname": "{username}"
    }
  }
}
```

New Chat Update Message

This is sent by the server to all the clients when a user sends a new chat message. This excludes the sender of the message.

```
{
  "update-message": "ns",
  "body": {
    "sender": {
      "uid": 6884583359643650,
      "uname": "{sender username}"
    },
    "message": {
      "uid": 6884583363784704,
      "content": "{some message}"
    }
  }
}
```

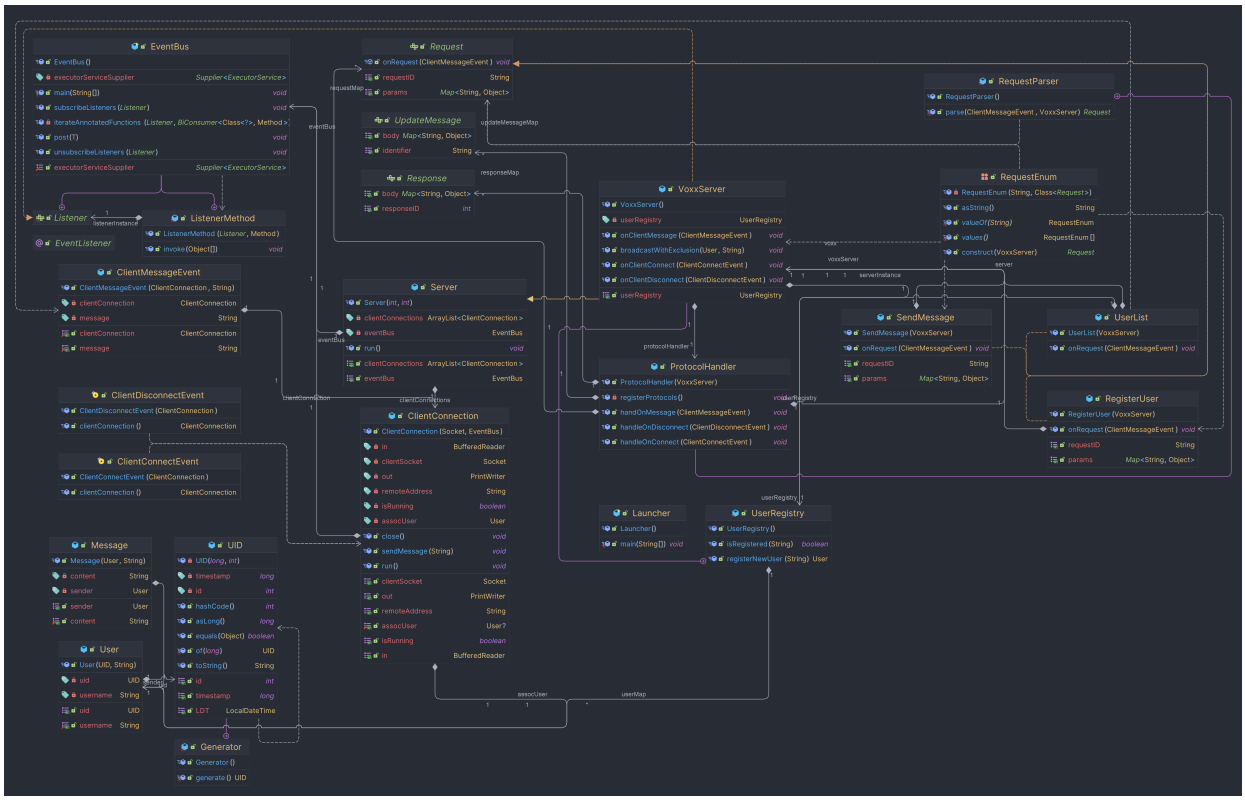
User Disconnect Update Message

This update message is sent by the server when a user (client) disconnects from the server.

```
{
  "update-message": "ud",
  "body": {
    "user": {
      "uid": 6884583363784705,
      "uname": "{username}"
    }
  }
}
```

UML Diagram

The following image contains the UML diagram of all the implemented code that we have so far:



For an interactive UML diagram, you can go to draw.io and download the UML file for the picture above here. To open the UML file, click `File > Open from > Device..` and browse your explorer/finder to find the downloaded UML file.

Modification from Proposal

There are two big changes from our original proposal. Initially we were thinking of having data persistence. However, we've moved away from that and now our chat application will only have in-memory persistence and will lose all data when the server shuts down. The server will also does not store any messages since messages are immediately broadcast to all the clients and the client is required to keep track of messages. With that being said, new users that joins will not see chat history

Another big change that we made is that we got rid of private and public chat channels. Users can no longer directly message another user, making Project-Voxx a purely drop-in single channel public chat service.

We've also got criticized that the UID documentation that we had on our proposal was out of placed. But we'd love to take this opportunity to defend the existence of this data now since it's now a fundamental part of Project-Voxx because it allows to make sure that users are unique but more importantly, it allows us to order message and user based on creation because the timestamp is embedded into the UID.