Voxx is a desktop/command-line, non-persistent, drop-in, public chat channel, where users can chat with other connected users with a degree of anonymity. Users could also host their own Voxx server to set up a communication medium for their own use cases.

Goals

Since Voxx is a simple chat application, we only need a few simple goals as well.

- ✅ **Multi-Client Server**: It is critical that the server is able to serve and handle multiple clients that connect to the server.
- ✅ **Reliable Connection**: The connection between the client and the server needs to be stable, and all the features like sending messages and getting updates must still work until the client disconnects from the sever.
- ✅ **Robust Communication Protocol**: The communication between the server and the client must be straight forward and easy to understand.
- ✅ **Deployable Clients**: Voxx clients that's easy to install for the end user/clients.
- ✅ **Deployable Server:** Make it easy for users to be able to start their own voxx server for their own use case.

Use cases

Since Voxx is a simple, non-persistent, chat application, we don't really have an extensive list of use cases. However, for the use cases that we do have, it changes a bit when using the cli and the desktop client.

- **Changing server**
    - Allows the user to change the Voxx server that they want to connect to.
    - For the desktop client: this can be done by right-clicking the connection status circle and typing the new address.
    - For the cli client: this is changed by passing in a parameter `-a <address>`
- **Registering a User**
    - Before a client can send chat messages to the server, the client connection needs to register a user for that connection first.
    - For desktop client: this is done by inputting a username in the desktop client and by pressing "Start Chatting"
    - For cli client: this is changed by passing in a parameter `-u <username>`

- **Sending a Chat Message**
    - This use case is when a user wants to send a message to the server, allowing other connected users to see the message as well.
- **List Connected User**
    - For the CLI client, this allows the user to list all the connected users in the server.
    - This use case is not available for the desktop client because it has a sidebar that contains a list of connected users.
- **Disconnecting from the Server**
    - This is essentially just closing or quitting from the application. However, it's important that we properly handle disconnects so that we could broadcast an update message correctly.

These are just simple user use cases when using Voxx. However, for a more general use case/usage for our application, here are just a few things Voxx could be used for.

- **Anonymous support groups**: Voxx can be used as a platform for anonymous support groups where people can connect with others who are going through similar struggles. The fact that messages are not logged or saved can create a sense of
privacy and safety for users.
- **Event-based chat**: Voxx can be used as a platform for event-based chat channels where people can connect and chat with others who are attending the same event. For example, people attending a conference, music festival, or sporting event can use Voxx to chat with each other.
- **Study groups:** Voxx can be used as a platform for study groups where students can connect and chat with each other about their coursework. The fact that messages are not logged or saved can create a sense of privacy and security for students who may be concerned about their academic performance.
- **Gaming communities**: Voxx can be used as a platform for gaming communities where players can connect and chat with each other about their favorite games. The fact that messages are not logged or saved can create a sense of privacy and security for players who may be concerned about their online reputation.
- **Language exchange**: Voxx can be used as a platform for language exchange where people can connect with others who are looking to practice speaking a different language. The fact that messages are not logged or saved can create a sense of privacy and safety for users who may be hesitant to speak in a new language with strangers.

These are just a few potential use cases for Voxx. With some creative thinking, you can likely come up with many more!

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 with the same username is not registered in the server, the socket connection will be bound to that user and every request from or to the client is in the context of that user.

Once a socket client registers a user, we now have an established `Response-Request` connection to the server. However, there's another optional connection that you can establish which is called the `Update Message` connection.

Response-Request Connection

This connection from the client handles all the requests that are coming in from it to the server. This connection is essentially a blocking connection where a client sends a request to the server and will wait for a response. *If you're implementing you're own client, you may want to set a socket timeout.*

Update Message Connection

The update message connection is a supplemental connection that a client sets up to receive update messages from the server. This connection must be setup correctly and a `Response-Request` connection must be established before this connection is set up. Specification for setting up this connection could be found under the heading Update Message

Request format

Before we can list down all the valid requests 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:

```json
{
  "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:

```json
{
  "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 does not need a password. The only requirement for user registration is a username, and that it is 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 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": {
      "uid": 6884583351369728,
      "content": "{message sent}"
    }
  }
}
```

Getting User List

A request that the client can make to get all the registered users 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 Message

Update messages are messages that are sent by the server to the clients to update clients about changes that happen 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 are there so that the clients can display up-to-date information from the server.

Before the client can establish a proper `Update-Message` connection. The `Response-Request` **must be established first and have a registered user**. Once established, we need to make another socket connection to connect to the server to serve as an `Update-Message` connection. It is also important to note that this new socket connection **needs to send keep alive messages**. The keep-alive configuration is flexible as long as it is sending it. To set up this new keep-alive connection as an `Update-Message` connection, we must send the following request using this new socket connection **not the response-request** connection:

```
{
    "request-id": "su",
    "params": {
        "main-user": "<main-username>"
    }
}
```

As you can see, this request needs a `main-username` as a parameter. This is why the `Response-Request` needs to be established first and a user needs to be registered (which returns the generated user).

When this request is sent and the main user exists, this connection will be set as a `supplemental` connection of the `Response-Request` connection and the server will send update messages to it.

Messages

The following are the messages that a client must anticipate from the server.

New User Update Message

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

```
{
  "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": "nm",
  "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}"
    }
  }
}
```

System/App Class Design

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. However, on top of these three modules, there's another "module" (it's really a python package) that is a git submodule where the python client is hosted, and it's called `voxx-client-cli` since this is a command line interface client for Voxx.

Voxx Commons

As briefly mentioned before. This module contains all the code that is going to be used throughout the whole project. However, the socket abstraction layer is mainly used in the `voxx-server` module.

This module contains three main modules, `esal`, `model`, and `protocol`

Abstraction Layer (esal)

ESAL or **E**than's web**s**ocket **a**bstraction **l**ayer is a package that makes implementing the Voxx server easier. This was achieved by implementing an event-based socket server. But before we can have an event-based system, an event bus is needed.

Event Bus

The event bus for the abstraction layer is annotation based, and the annotation's retention policy is set for runtime, therefore, the event bus relies on the Reflection API. Moreover, the event bus is also multithreaded to make listener invocations non-blocking (which we need to pay attention to for when we make our listeners).

At its core, the design of the event bus is pretty straight forward. When we construct an event bus, we can immediately subscribe listeners as demonstrated here:

```java
// An event could be any class.
class SomeEvent { ... }

class SomeEvent2 { ... }

class SomeListeners implements EventBus.Listener {

    @EventListener
    private void onSomeEvent(SomeEvent event) {
        System.out.println("SomeEvent happened!")
    }

    @EventListener
    private void onSomeEvent2(SomeEvent2 event) {
        System.out.println("SomeEvent2 happened!")
    }

    public static void main(String[] args) {
        EventBus bus = new EventBus();
        bus.subscribeListeners(new SomeListeners()) // <--- Listener subscription here
    }
}
```

The `EventBus#subscribeListener(EventBus.Listener listener)` method will reflectively iterate through all the declared methods inside that `Listener` class and will extract Methods that are annotated with `@EventListener`. It is important to note that we actually construct the listener class because we need to store it in the `ListenerMethod` class so that we can successfully pass it to the method invocation later via reflection api.

We can now look at how to `post` an event so that we could invoke listener methods. Here's a demo on how to post the events defined above and a corresponding output:

```
public static void main(String[] args) {
        EventBus bus = new EventBus();
    bus.subscribeListeners(new SomeListeners()) // <--- Listener subscription here

    bus.post(new SomeEvent()); // <--- Post with no runnable that runs after.
    System.out.println();
    bust.post(new SomeEvent2(), () -> System.out.println("Done posting SomeEvent2"));
// <--- Post with runnable
}
```

Output:

```
SomeEvent happened!

SomeEvent2 happened!
Done posting SomeEvent2
```

Now that we have a fully working event bus, we implemented the following events, `ClientConnectEvent`, `ClientDisconnectEvent`, and `ClientMessageEvent` that we need to post when implement the abstraction layer for the server.

Server Abstraction Layer

This implementation is very similar on how we implemented the server socket in class where we have a main thread that waits for connections and construct client workers (called `ClientConnection`) and run on a different thread. The only difference is that we take advantage of the `EventBus` so that whenever a new connection is accepted and a `ClientConnection` is constructed, we post a `ClientConnectEvent` with the matching `ClientConnection`.

```
var clientConnection = new ClientConnection(clientSocket, this); // "this" is the
Server instance
LOGGER.info(String.format("New client (%s)", clientConnection.getRemoteAddress()));
eventBus.post(new ClientConnectEvent(clientConnection), () ->
clientConnections.add(clientConnection));
```

Once posted, we then execute this `ClientConnection` on a different thread.

ClientConnection

This is also implemented very similarly to how we implemented a client worker in class. But as we mentioned before, instead of the abstraction layer handling an incoming message. We will pass down that responsibility to the listener of the `ClientMessageEvent` by posting this event and passing the message.

```
try {
    String inLine;
    while ((inLine = in.readLine()) != null && isConnected())
        eventBus.post(new ClientMessageEvent(this, inLine)); // <--- instance of
ClientMessageEvent and the line
    close();
} catch (IOException e) {
    Server.LOGGER.error(e.getMessage());
    close();
}
```

On top of posting the `ClientMessageEvent` we can also see that we are calling the `close` function when we exit the while loop or caught an exception. The `close()` function essentially closes the client socket and the in/out stream. Once closed, we post a `ClientDisconnect` Event.

```
public void close() {
    try {
        if (!clientSocket.isClosed()) {
            in.close();
            out.close();
            clientSocket.close();
            serverInstance.getClientConnections().remove(this);
            eventBus.post(new ClientDisconnectEvent(this));
            isRunning = false;
        }
    } catch (IOException e) {
        Server.LOGGER.error("Could not properly close connection! " + e.getMessage());
    }
}
```

The information above is everything in the `esal` package, and we will see how this is used when we get to the Voxx Server header.

Model (model)

This package contains “plain old java objects” `Message`, `User`, and `UID`. The first two objects are trivial objects, however, I want to focus on the UID because it is not that plain, and it contains very useful properties.

UID

UID or uniquely identifiable descriptors are essentially just a unique id that a `Message` and a `User` is assigned with to make sure that they are unique. This is done by implementing a similar system to Snowflake ID. But the main difference is that instead of using 64 bits, we are only using 54, and it is broken down like the following:

```
111111111111111111111111111111111111111111  111111111111
54                                       12            0
```

Field	Bits	Description	Retrieval
Timestamp	12-53	Millisecond since Epoch (Could be any arbitrary epoch that we chose)	(uid >> 12) + epoch
Incremental ID	0-11	If multiple creation request happens in the same timeline we increment this	uid & 0xFFF

The chosen epoch for the UID implementation is: `TIME_EPOCH = 0x64b62a60`

On top of this UID class, we also have an inner `Generator` class (factory class). It is a thread safe UID generator and will always produce a unique UID.

UID Property

- Since the UID contains a timestamp, that means we can take advantage of that information to show timestamps on our messages. Therefore, the UID class also comes with other utility functions that automatically convert that timestamp to local date time and format them.
- A UID is also easily transportable because it’s essentially just a binary data that you can convert to a number `long` for this instance.
- There is no data persistence in Voxx, but if there is, we can easily store objects in a database using UID. Which is the main motivator on developing the `Snowflake ID`

Protocol (protocol)

This package does not contain significant code since the protocol is outside the scope of the commons/abstraction layer. Therefore, the source code for the protocol defined above is implemented in the module `voxx-server`. However, this package contains two things, the interface for `Request` and `ProtocolUtil`

- ProtocolUtil
    - This class essentially just contains a static helper function that takes in JSON objects and returns it as a flattened string, ready to be sent to the server/client.
- Request
    - This is just an interface for a request that contains a function that implementors must implement to be considered as a Request.

Voxx Servers

This module is where the server for Voxx is actually implemented. Since we've made is so that the Server is actually event based, at its core, the server implementation is pretty simple, and it can be broken down like the following.

```java
public class VoxxServer extends Server implements EventBus.Listener {

    public VoxxServer() {
        // .... constructor code here.
        getEventBus().subscribeListeners(this);
    }

    @EventListener
    public void onClientConnect(ClientConnectEvent event) {
        // ... Code when a client connects.
    }

    @EventListener
    public void onClientMessage(ClientMessageEvent event) {
        // ... Code when a client sends a message
    }

    @EventListener
    public void onClientDisconnect(ClientDisconnectEvent event) {
        // ... Code when a client disconnects
    }
}
```

However, before we dive on how Voxx-server is using these events, we first need a few objects that would help us. Let's start with the `UserRegistry`.

User Registry

The user registry contains a concurrent hashmap (ConcurrentHashMap<String, User>) that stores the user object using the username as a key. Since we don't want to immediately register clients as a user, this class is not used until the socket client successfully sends a Register User Request. The main purpose of this class is to contain registered users and will be used to later for when a client sends a Get Users. To handle these requests, we heed a handler and we'll call this `ProtocolHandler`

ProtocolHandler

The protocol handler contains an inner class called `RequestParser` that parses incoming message. This parser will try to parse the message as a Json object and see if it would throw a JSONException, indicating that the message does not have a Json syntax. If it is, this parser will then look at the json attribute called "request-id" and find a matching ID that exist in the `RequestEnum` and reflectively construct and return that `Request`.

The ProtocolHandler also consist a function called `handleOnMessage(ClientMessageEvent event)` that would be called inside the event listener that we have above. With the request parser, this is what that function looks like:

```
public void handOnMessage(ClientMessageEvent event) {
    var req = RequestParser.parse(event, serverInstance);
    if (Objects.nonNull(req))
        req.onRequest(event);
}
```

Now that we've defined objects that we need. Let's now talk about how Voxx-server handles each event above.

On Client Connect

Since the client needs to send a register request before they could be added in the `UserRegistry`, the ClientMessageEvent does not really do anything other than logging that a new client connected to the server.

It is also important to note that this type of ClientConnection is still not set at this point. So we must wait if this connection is going to be set as a Response-Request Connection or an Update Message Connection

On Message

On this event, we are going to use the `handleOnMessage` function that could be found in the `ProtocolHandler`. This should automatically parse the message and construct the matching request and call the `onRequest` function that a request needs to implement.

```
@EventListener
public void onClientMessage(ClientMessageEvent event) {
    // ... code

    var msg = event.getMessage();
    Server.LOGGER.info("[Vox] Client said: " + msg);
    protocolHandler.handOnMessage(event);
}
```

On Client Disconnect

At this point, on the abstraction layer, the client is already disconnected. Therefore, we really can't do anything socket wise. So when Voxx gets this event, it does the following:

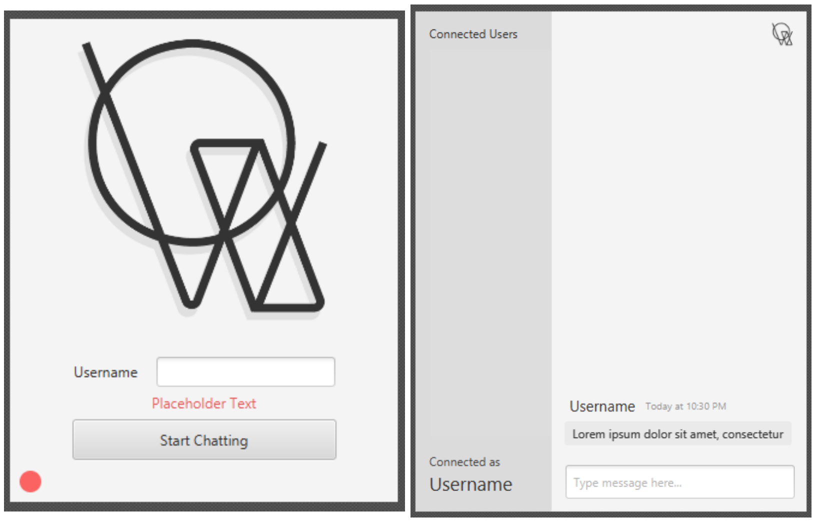
- Check if this client connection that disconnected was a supplemental connection or does not have an associated user.
  - If it doesn't, we don't need to do anything, so return and short the function.
- Get the associated user for that client connection and remove the user from the `UserRegistry`
- Broadcast that the client with the user info disconnected.

Voxx Client

This module contains the JavaFX client for Voxx.

User Interface

For Project Voxx Java client, we have two main scenes: `Login scene` and `Chatbox scene`



To allow us to switch scenes with ease, we also implemented a utility class called `PrimaryStageManager` that contains a function called `#setScene(String fxml, Consumer<T> controllerConsumer)`. This function allows you to pass in a consumer where the instance of the controller is passed just in case you have to call functions of the controller when the fxml is loaded.

Controllers

Currently, the Voxx java client connection is implemented using the classes `ChatController` and `LoginController`.

Login Controller

`LoginController` is responsible for controlling the login screen of the application. When the scene is loaded, and the controller is initialized, it will construct the `ConnectionTask` and run it on a different thread. By default, Voxx will try to connect to the server with the address `localhost` and port `8008` (*which is the voxx-server running on local machine*). If there is no voxx-server running on localhost, the connection indicator will turn red. To connect to a different server, you can click the red dot and provide the server address and port using the following format: `server_addres:port`. The controller will now then try to connect to that new server.

Once connected, we now need to register a new user. To do so, we need to provide a username and click the `Start Chatting` button. It will check if the username that is entered is valid. We have designated a valid username to be between 4-7 characters, also allowing for numbers and underscores, however, the username cannot start with numbers first. It then sends a request to the server to create a new user with the entered username. If the request is successful, the controller will now call `#setScene()` in the `PrimaryStageManager` to change the scene to the `Chatbox` scene.

Chat Controller

`ChatController` is responsible for controlling the user-interface experience of the application. It handles the display and sending of messages, updating user lists, and connecting to the server to receive updates.

Connection

Currently, the Voxx java client connection is implemented utilizing the class `ReqResClientConnection` and, optionally, `UpdateMessageConnection`.

Request-Response Connection

The implementation of the `ReqResClientConnection` follows the documentation above when it comes to the `Request-Response` connection. Therefore, any request sent from this connection type will be `blocking` and will always wait for a response from the server.

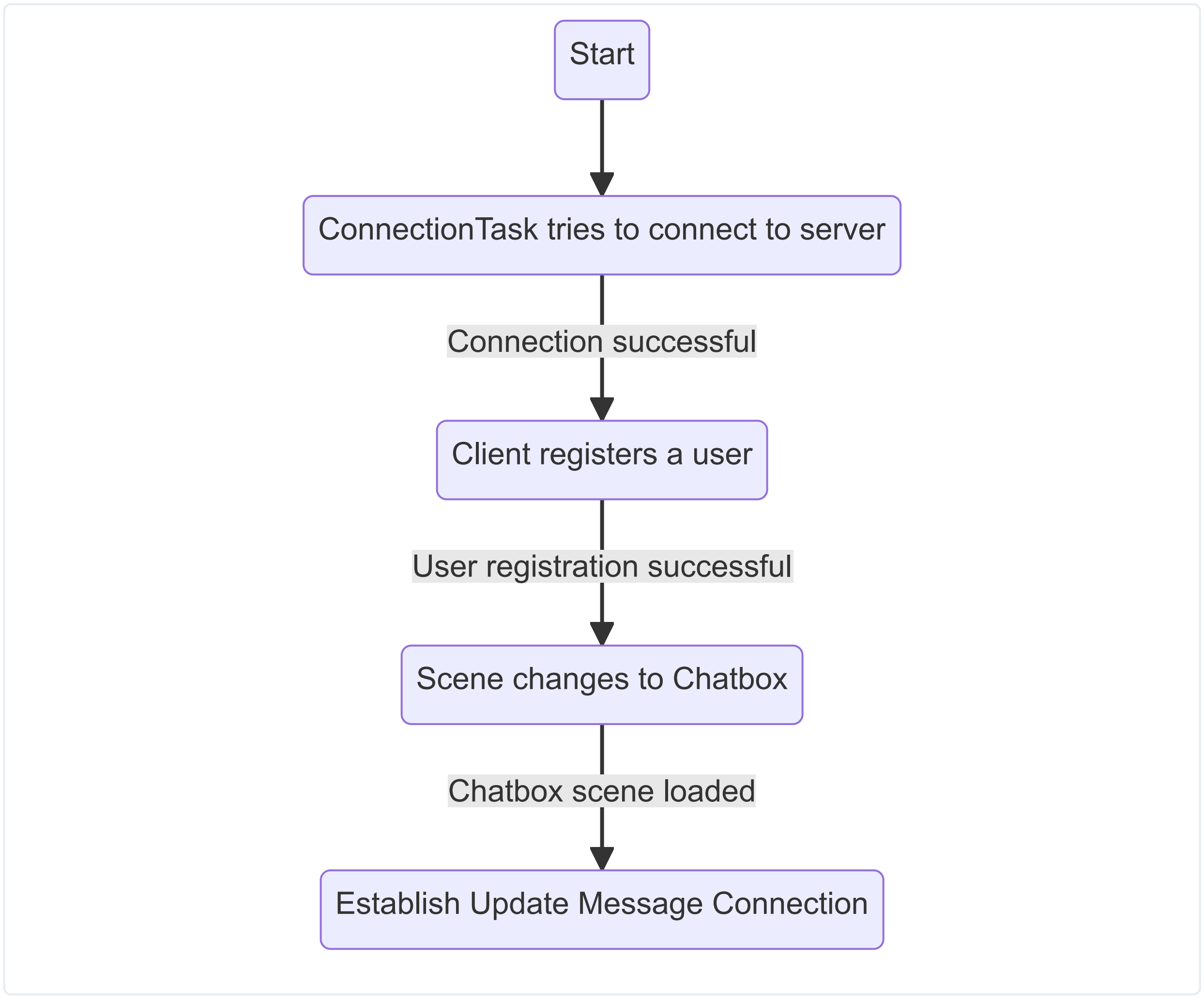
Update Message Connection

Like the `Response-Reques` connection, this connection type was also implemented by following the documentation about the Update message connection under the Protocol header.

ConnectionTask

On top of the two connection implementations above, we also wrote a JavaFX task that would try connection to the server with `n` number of times. By default, this will try connecting to the defined address 3 times.

Connection flow


Start
ConnectionTask tries to connect to server
Connection successful
Client registers a user
User registration successful
Scene changes to Chatbox
Chatbox scene loaded
Establish Update Message Connection

Voxx Client CLI

Voxx CLI is a command line interface client for Voxx, it's written in python and has its own repo and added as a git submodule for the main Voxx repository.

Voxx CLI initially starts as a pure CLI application when a connection has not been established yet. But as soon as the connection is established a valid user is registered, the CLI runs a text user interface (TUI) using the library `Textual`.

Modules

For Voxx cli, we have 4 modules. One of them is the main module which called `voxx` and inside of that module we have `connections`, `model` and `tui`.

voxx module

The voxx module is actually not really a python file, but it's just a directory with `__init__.py` and `__main__.py` in it. This package/module also contains the other modules. However, under this header, we'll just talk about the double underscore (dunder) files for now.

`__init.py__`

This file effectively makes it so that this package/directory is considered a module. This file also contains metadata like the app name, version, author, and description, that we're going to use for `setuptool` when we install or build distribution for the python application.

`__main__.py`

This is essentially considered to be the main file for the module. It essentially works as a main method for the module, but inside this file we also have a standard `if __name__ == __main__:` condition. This file is also the entry point of our cli application, and `setuptool` is directed to make a script for it.

Because this is our entry point, this is also where we initialized our CLI structure. Using Python's `argparse` module, we have the following commands available:

```
Usage: voxx-cli [options] <arg>

-h   --help                   show this help message and exit
-a   --address ADDRESS        voxx server address
-u   --user USERNAME          username to register as
-v   --version                show program's version number and exit
```

Unlike the JavaFX application voxx-cli does not need the registration scene, and all of that is by the command line. Therefore, when the text user interface or the TUI application is loaded, the connection to the server, both response-request and update message, has already been established. And this is done by passing in the command line arguments into the `establish_voxx_connection` function that could be found in the `connection` module.

connection module

This module is a pretty straight forward module. The paradigm used on this module is mixed, both procedural and object-oriented. The first part of the module essentially contains constants and the class definition of the connection type `ResReqClient` and the `UMClient` which inherits the request-response connection, but it also inherits a Thread. This is then followed by uninitialized global variables for those type classes that we are going to initialize later using the function `establish_voxx_connection(user: str, addr: tuple)`.

In this module, we also took advantage of Python decorators to register update message handlers that essentially just puts the function object into a dictionary. This dictionary is iterated over by the `UMClient` and see if there's any matching function. However, for a function to be considered as a handler for update messages, the function name must match the update message id. For example, we want to handle new message updates (the key is `nm`) then the function name needs to be `nm`.

```python
@um_handler
def nu(self, msg: SimpleNamespace) -> None:
    """Handles new user update message"""
```

Other than the `@um_handler` decorator, we also have a decorator named `@assert_rr` that essentially make sure that the request-response connection is established before any of the request functions can be used.

```python
def assert_rr(func):
    def wrapper(*args, **kwargs):
        try:
            if res_req_conn is None:
                console.print("RR connection not established!", style="bold red")
                return None
            return func(*args, **kwargs)
        except ConnectionResetError:
            return None
    return wrapper
```

This decorator does not really throw an exception since we don't want our program to crash when the `Request-Response Client` isn't available to use. Therefore, the decorator just returns a `None` and it essentially just act like the response from that request is `None`

This decorator is going to be used like this:

```python
@assert_rr
def register_user(username: str) -> SimpleNamespace:
    return res_req_conn.request({"request-id": "ru", "params": {"uname": username}})
```

Functions for the other request defined under the protocol heading are also defined in this module.

model module

This module is just a python version of the Java's `voxx.commons.module` package. The only difference is that the Python version of the UID does not have a generator since UID generation is going to be handled by the server anyway.

tui module

Like I've mentioned, once the connection is established from the connection module in the `__main__.py`, it will start the TUI application named Voxx. It's essentially a command line version of the `ChatBox` scene for the Java FX application. The only difference is that the TUI does not have the sidebar that shows all the connected users.

The Voxx TUI App

The class Voxx that could be found in the tui module is a subclass of a `Textual` `App`. Textual is a TUI library that allows us to render conventional UI models in a command line terminal. The TUI for Voxx is really simple, it just has one basic `screen` and inside that screen is a `VerticalScroll` container, an `Input` field, and a `Footer`.

The `VerticalScroll` will be used to insert two widgets, the `MessageBar` and the `NotificationBar`.

- Both MessageBar and NotificationBar are a subclass of the widget `Container` that has a border property and a `Static` text.

For the input field, it was just left default and no other property changes were done to it. The same case goes for the `Footer` but it will house the bindings we registered for the application.

Beyond the widgets, the update message handlers are also defined in this Voxx application class:

```
@um_handler
def nu(self, msg: SimpleNamespace) -> None:
    """Handles new user update message: called from thread-2"""
    self.call_from_thread(self._add_notif, f'{msg.body.user.uname} has connected',
'User Connect', None)

@um_handler
def nm(self, msg: SimpleNamespace) -> None:
    """Handles new message update message: called from thread-2"""
    sender = User(UID.of(int(msg.body.sender.uid)), msg.body.sender.uname)
    time = UID.of(int(msg.body.message.uid)).get_timestamp_string()
    self.call_from_thread(self._add_msg, msg.body.message.content, sender.username,
time)

@um_handler
def ud(self, msg: SimpleNamespace) -> None:
    """Handles user disconnect update message: called from thread-2"""
    self.call_from_thread(self._add_notif, f'{msg.body.user.uname} has disconnected',
'User Disconnect', None)
```

There is one important thing to note here; since the UpdateMessages are being listened to on a different thread, we need to make sure that we have our app call UI changes on the same thread of the TUI, this is why we use the function `App#call_from_thread` which is essentially the same as `Platform.runLater` for JavaFX.

Project State

All the modules described above are in its release state, and you can download your own copy of every component of this project. The server and the client are coded to be universal, so it works for both Windows, macOS, and should also work for Linux. However, because of the lack of a local Linux machine, both of the Voxx clients status on those machines is unknown (The server will run perfectly on a Linux machine).

Here is the instruction on how to get a ready-to-use artifact and executables for each component:

Voxx Server

Every commit we do on the main branch of the Voxx GitHub repository, a workflow will be triggered to build the artifact for `voxx-common` and `voxx-sever` and those artifacts are automatically uploaded to my maven repository repo.cyr1en.com. To get a server copy, you need to find the latest server artifact which can be found under `snapshots/com/cyr1en/voxx-server/1.0-SNAPSHOT`. Just scroll down and the naming convention of the artifact is `voxx-server-1.0-<build date>-<build number>.jar`. There are two ways to get the artifact, you can click the artifact and download it to your local machine. However, if you want to run the voxx server on a server, you can get the jarfile by using `wget`

```
wget https://repo.cyr1en.com/snapshots/com/cyr1en/voxx-server/1.0-SNAPSHOT/voxx-server-1.0-20230504.024024-40.jar
```

Once you have a copy of the voxx server, all you have to do now is run it. But before you could do that, we have to make sure that the machine you're running the Voxx server has the port `8008` open.

Once done, you can now run the server. You can use `tmux` or `screen` if you're accessing your server via `ssh`.

- For `tmux`

  ```
  tmux
  java -Xms256M -jar voxx-server-1.0-20230504.024024-40.jar
  ```

  - Once started you can press `ctrl + b` followed by `d` on your keyboard to detach from this window.
- For `screen`

  ```
  screen java -Xms256M -jar voxx-server-1.0-20230504.024024-40.jar
  ```

  - Once started you can press `ctrl + a` followed by `d` on your keyboard to detach from the session.
- Launching on local machine

```
java -Xms256M -jar voxx-server-1.0-20230504.024024-40.jar
```

- Just make sure you don't close the terminal window or else the server will shut down.

Voxx client

Voxx has two clients, a JavaFX client and a CLI client (written python). Here's an instruction on how to install each one of them:

- Voxx Desktop Client (JavaFX)
  - You can get the latest installer on the release page of Voxx here
    - An installer is available for windows and macOS. For now, there is no way to make an installer for Linux.
  - Note: The installer is not signed, therefore, your operating system will warn you when you run the installer
- Voxx CLI
  - The command line client for Voxx is currently distributed on `PyPi` and could be found here
  - As long as you have Python 3.7 or greater, you can easily install `voxx-cli` by doing the following!

    ```
    pip install voxx-cli
    ```

  - Once installed, you can just simply just run

    ```
    voxx-cli -h
    ```

    To understand how to use the cli application
  - A Voxx server is currently running on `voxx.cyr1en.com` on port `8008`
    - to connect to this server:

      ```
      voxx-cli -a voxx.cyr1en.com:8008 -u <username>
      ```