Elements of Software Construction Project Report

GTFS

A multi-user Android chat application with a Erlang backend.

By:

Glen Choo

Lau Siaw Young

Francisco Furtado

Ha Duc Tien

Table of Contents

Introduction

GTFS is a multi-user native Android chat application that is powered by a Erlang server backend. The main mode of communication between the frontend Android app and the server backend is achieved through JSON (JavaScript Object Notation) payloads over a TCP socket transport layer.

GTFS's core functionality mimics that of popular chat applications such as WhatsApp and WeChat, with a few key differences. Some of these key differences are in the form of extra features, such as the ability for users in chat rooms to create notes which can be seen and updated by all the users, as well as the ability to create events which users can vote on to reach a consensus.

System Requirements

To install the latest version of the GTFSapp for Android, you'll need:

- Mobile Device
- Android 4.4.2 (KitKat) or higher.

As for Aurora, its system requirements are detailed under the *Deploying Aurora* section.

System Design

Android frontend

About GTFS

As an IM service, GTFS was conceptualised to address many of the issues that we believe plague most IM services (WhatsApp, Facebook, Messenger etc). While these services pack varied services and effective UI, we found that these services did little to address the way real users use IM. Due to the ease of use and intuitiveness of most IM services, we had admittedly underestimated the scale of the task at hand and the difficulty of competing with existing applications without focusing on our distinguishing features. With the limited timeframe of the project in mind, we chose to focus on creating a minimalistic but

functional UI and solving these core problems rather than trying to replicate all the features of existing services e.g. profile pictures, statuses, media sharing.

GTFS is coded almost entirely in native Android and extensively utilises Android app components and helper classes on its frontend as well as the backend. The software development model we used attempts (as much as possible) to decouple the frontend and backend by using Activities to create a smooth front end and Services to do work in the background. As a result, issues (including those of concurrency) tend to reside in the interface between the two and will discussed in depth later on.

Key Features

Core IM features	**Distinguishing features**
-Sending/Receiving messages	-Message tagging and filtering
-Chatting with contacts	-Timed groups
-Notifications	-Chat notes
-Creating personal and group chats	-Chat events with notifications and invitations
-Searching for messages within chats	

We chose these features to target several deficiencies we identified with current services, particularly, in the field of group messaging.

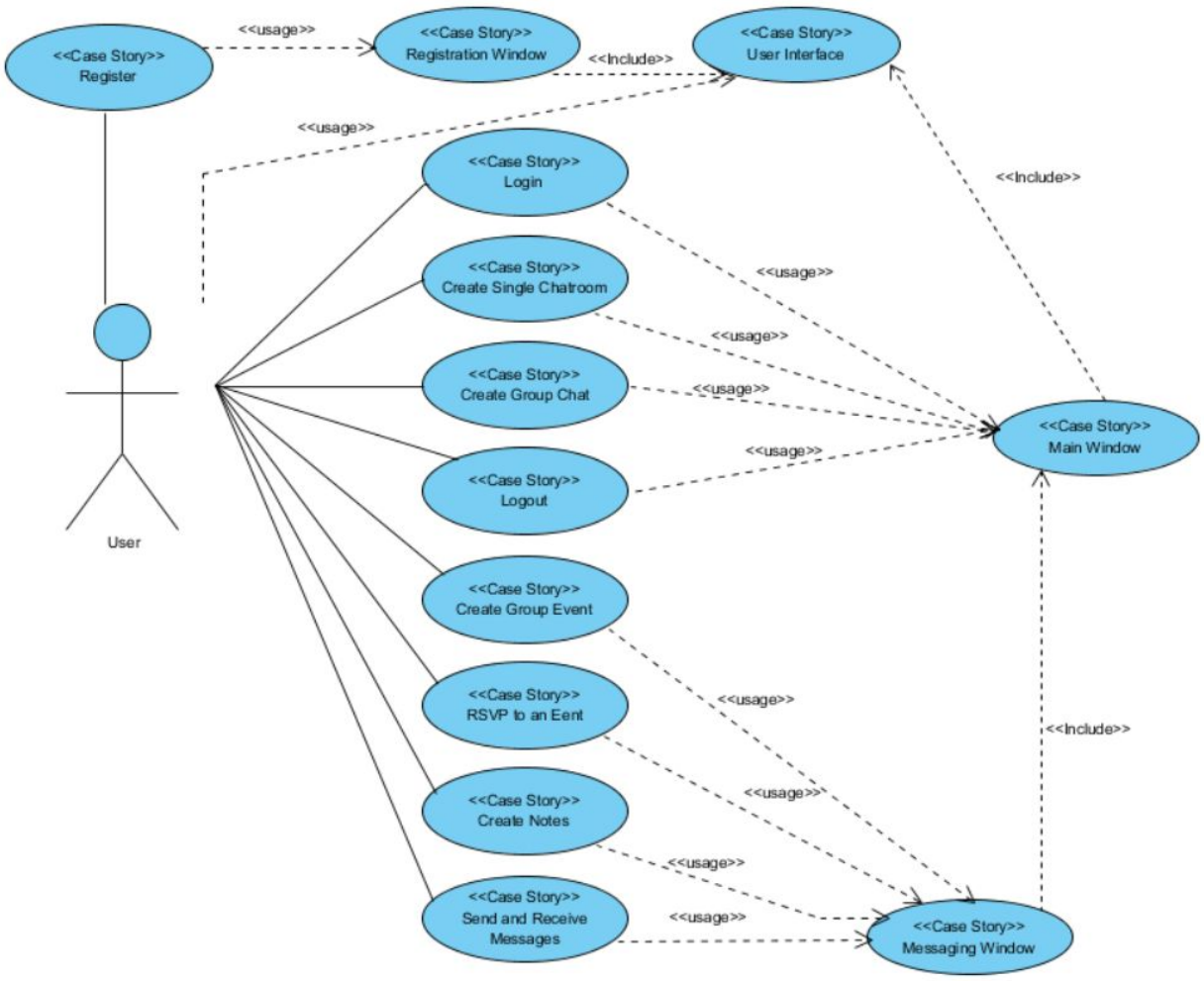

Figure 1: Use Case Diagram

Message tagging

The primary issue we identified in group messaging is message overload. As many of these groups consist of peers, a group chat becomes a natural place for social interaction. However, many such groups also serve a practical purpose in addition to this. A group chat among coworkers for instance may provide users with a platform for disseminating important information as well as sharing light-hearted banter. An especially difficult problem arises when important messages and unimportant messages interweave, resulting in a flood of messages that makes it difficult for users to track the flow of the conversation. Message tagging allows users to tag messages with a certain level of importance and by selecting the appropriate filters, users can selectively view messages with a certain tag. Tagged messages are denoted by a coloured circle next to

the messages for easy differentiation (currently red for Important and yellow for Nonsense). Initially, we had intended to allow users to specify tags and add multiple tags to each message. However we were unable to find a way to incorporate such a feature with a satisfactory UI. Regardless, a single tag with message importance will still allow for powerful filtering.

Timed groups

Group chats tend to be created for various reasons, many of which are transient. For such groups, having the group persist for extended periods of time results in unnecessary clutter on the app's home screen as well as possible feelings of social awkwardness by leaving the group and being afraid of 'making a statement'. While putting an artificial expiry on a social chat would be frowned upon, having a group expire upon the completion of a project would help keep the home screen streamlined and reduce the effort of users to leave and delete unwanted chats. Users can specify a certain duration (on the scale of minutes to years), after which the group will become inaccessible to the user.

Chat notes

As chats are often a place for collaboration, it can be helpful to have a repository of notes that users can modify at will. These can be used to store common pieces of information such as upcoming deadlines and details of events. These allow information to be offloaded from the main chat and allows users to find relevant information easily. Notes are synchronised with Aurora whenever the user attempts to view the notes. An existing issue is that notes can be edited by multiple users simultaneously and the note will only be updated with the last user's changes. Due to the expected rarity of such an event and the sophistication of the implementation involved, we have decided that this is a relatively minor issue.

Chat events

Even when chats are not constructed around a given event, chats tend to involve event planning such as going for a family holiday or spontaneous events such as meeting up

for drinks. Events provide a useful interface for users to create an event and view who is going for a given event. Users can create an event by specifying a name, date and time and the appropriate invitation is sent out to all members of the chat. More intricately planned events can be done using a combination of notes and events.

GTFS Frontend

The GTFS frontend is written entirely in native Android and consists of 14 activities and 6 custom adapters. However most of the activities are transient and are interacted with in very limited ways.

Messaging

Activities pertaining to messaging. The smallest package but contains the most central activities.

MainActivity contains what can be considered the home screen of the app. It provides a displays all the user's chats (with name, last message sent and number of unread messages) and allows users to access the relevant chat by tapping on it. It also provides navigation for users to create new chats, view contacts, edit their profile or change settings. When the activity is started, it fetches the names and IDs of the chats from the local database and populates the ListView using a ChatAdapter. The ChatAdaper retrieves the last message and number of unread messages based on the IDs given and populates accordingly.

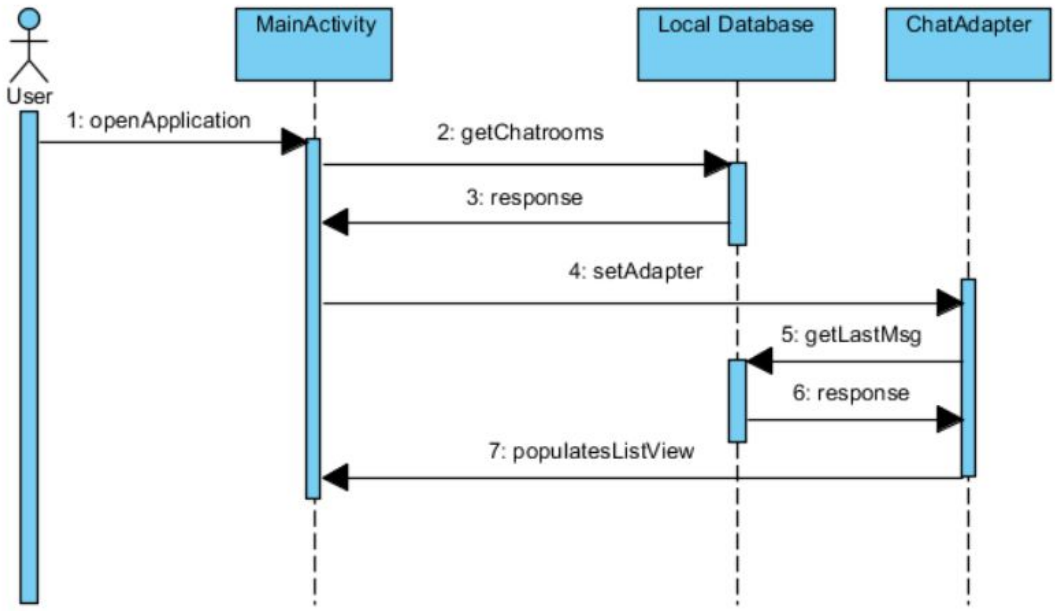

Figure 2: Sequence Diagram for MainActivity loading chats

MessagingActivity is easily the activity with the most complicated behaviour for three main reasons - populating, network communication and features. To populate a MessagingActivity with the relevant information, MessagingActivity first checks whether or not the chat is a group chat. If it is, the title is set to the group name, but if it is a personal chat, the title is set the the username of the recipient. Messages in a group chat contain the sender's name while personal chats do not. The action bar options also differ for groups and personal chats, for example, groups must be able to display group information while personal chats do not support events.

The true challenge is obtaining this information. If the chatroom ID is known, this information is a matter of querying the database appropriately. However, consider the case where a brand new chatroom is created. To avoid unnecessary lag, MessagingActivity can be started without the corresponding ID. Without the ID however, no messages can be sent out and the relevant information for the activity must be obtained by the Intent. In this scenario, sending is enabled by first sending a request to the server to create a new chatroom while the activity waits on a message containing the chatroom ID. When the appropriate message is identified, the ID is updated and sending can be done, all while providing the illusion of continuity to the user.

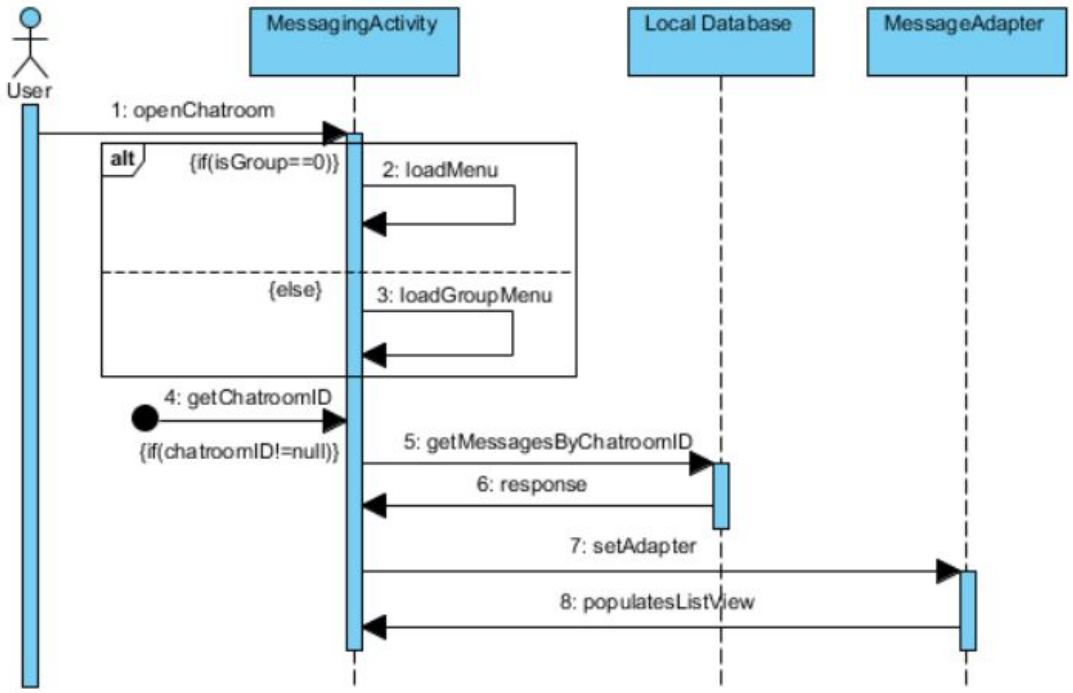

Figure 3: Sequence Diagram for MessagingActivity loading messages

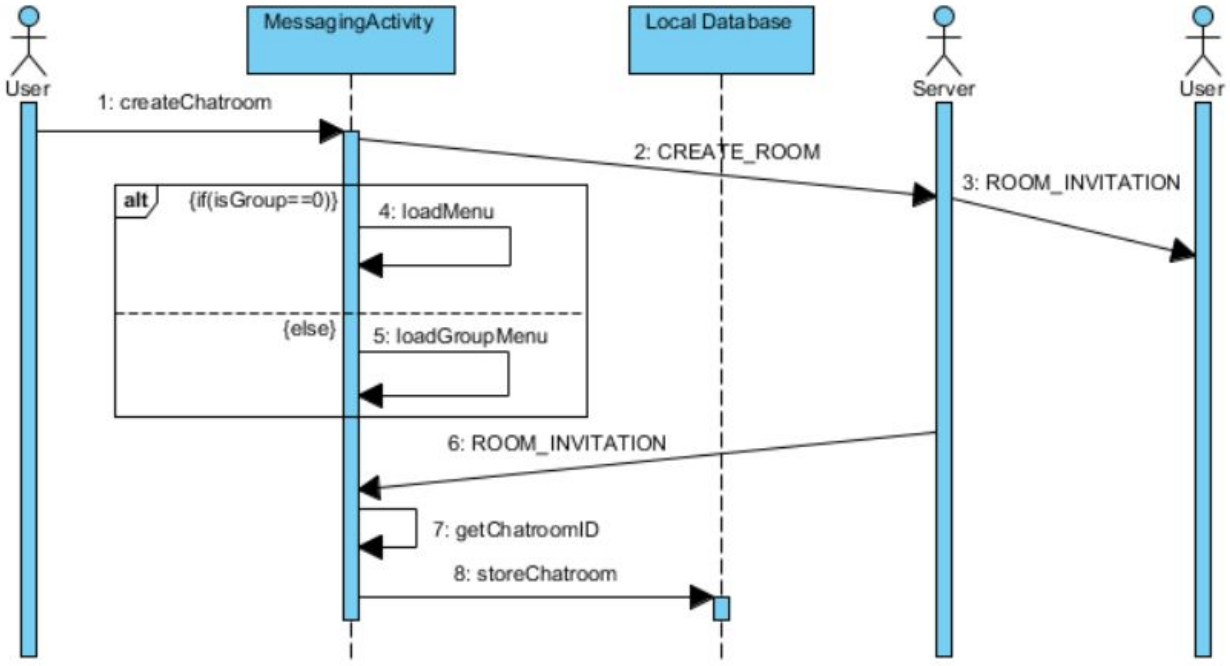

Figure 4: Sequence Diagram for creating new chatroom

A similar issue exists in sending out a new message. For the sake of simplicity and to ensure that messages are received by the server, messages are only stored and processed when the server sends the message back to the client. The server and client do not distinguish between the sender and receiver of the messages and thus any message sent is received by all users of the group. However, users expect messages to be sent the moment the send button is pressed. To again provide the illusion of continuity, the message sent is stored in a list (local to the activity) that backs the MessageAdapter. This means that the message appears on the screen but is not stored in the database. When a message is received, the UI is updated accordingly. If it is from the user himself, there will be no apparent change in the UI. The MessageAdapter is the part that differentiates between senders, putting the user's own messages on the right and other messages on the left. When the send button is pressed, a check is performed to see if the chatroom has expired during that time. If it has, sending will not be allowed and the UI will not be updated with the new message.

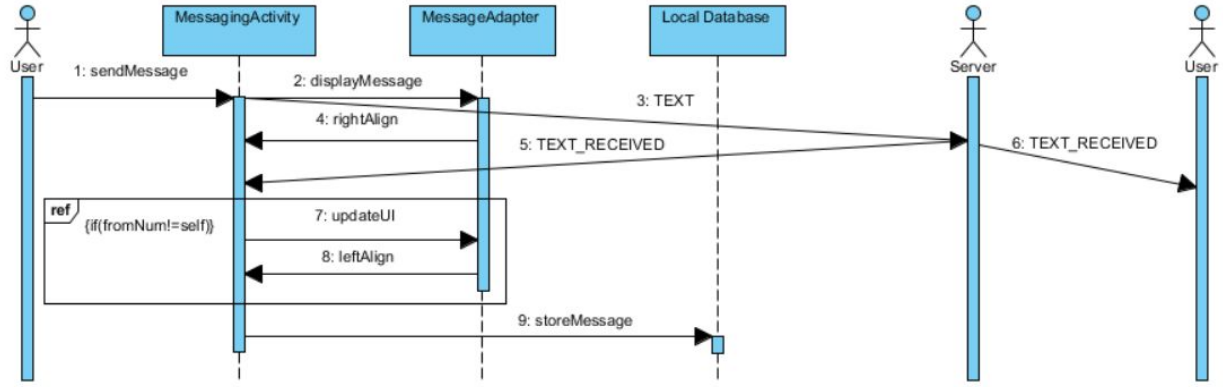

Figure 5: Sequence Diagram for sending messages in MessagingActivity

As specified earlier, most of GTFS's features lie in messaging, and thus MessagingActivity must support them. Searching for messages is supported on two levels: the MessageAdapter highlights the search term and stores a list of indices where the term occurs. Going forwards and backwards through search results is done by MessagingActivity selecting the appropriate index and focusing accordingly.

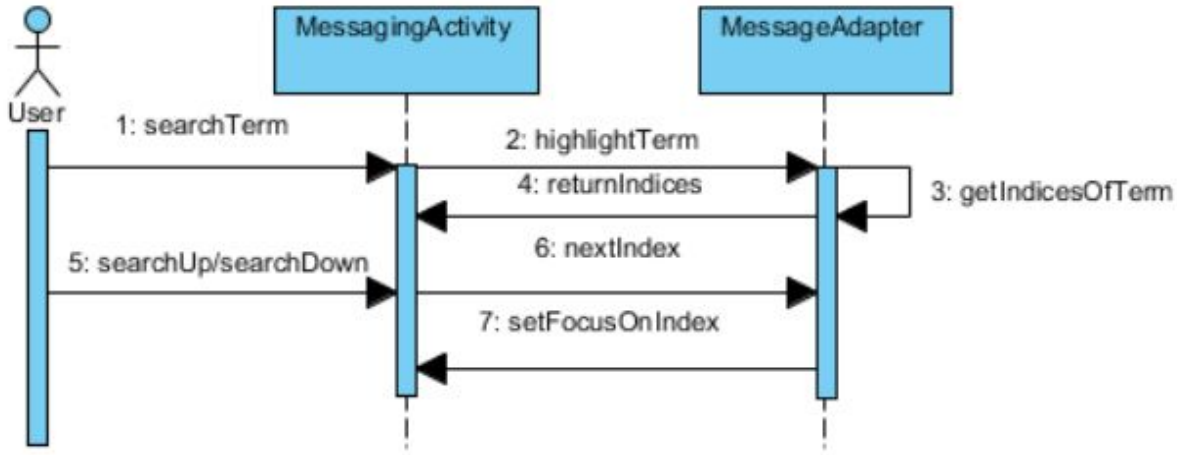

Figure 6: Sequence Diagram for searching in MessagingActivity

Similarly for message tagging, MessagingActivity sets the desired tags and the MessageAdapter filters the results based on the tags. MessagingActivity only suggests tags that are applicable in the current message and thus one cannot select a tag that results in no messages being displayed. The MessageAdapter also draws the appropriate coloured circles next the the messages. Event requests are handled by setting the visibility of a bar on the bottom of the screen and populating the bar appropriately.

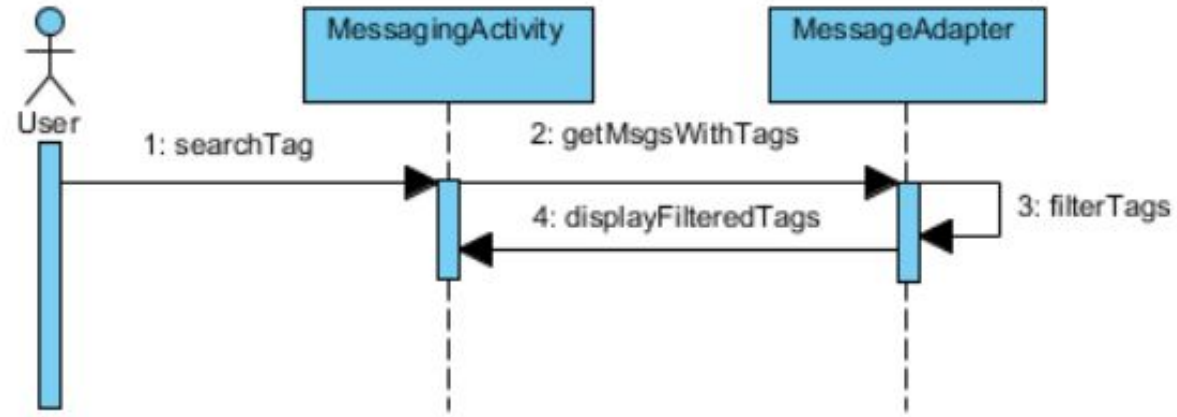

Figure 7: Sequence Diagram for filtering messages by tags in MessagingActivity

Group

Activities pertaining to group features and the creation of groups.

NewGroupActivity and AddContactToGroup together provide the ability to create a new group chat. NewGroupActivity allows users to decide on a name for the group and (optionally) an expiry. These are passed via intent to AddContactToGroup which provides the selection of users for the group and searching through users. These two activities are always called sequentially when a new group is created. The decision to split them into two activities is chiefly to provide a less cluttered UI for users.

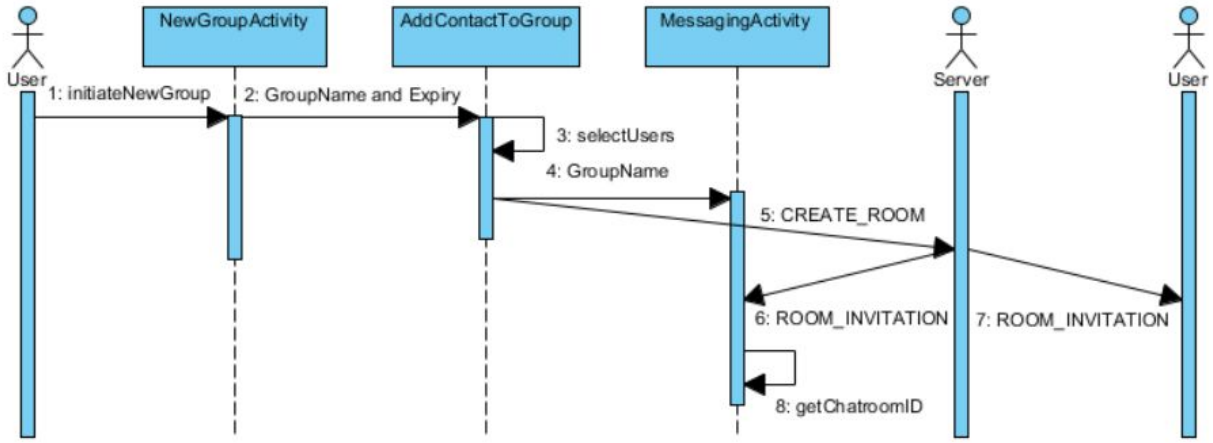

Figure 8: Sequence Diagram for creating a group chatroom

GroupInfoActivity provides relevant information of a group. This is done by querying the database for the relevant information using the group ID.

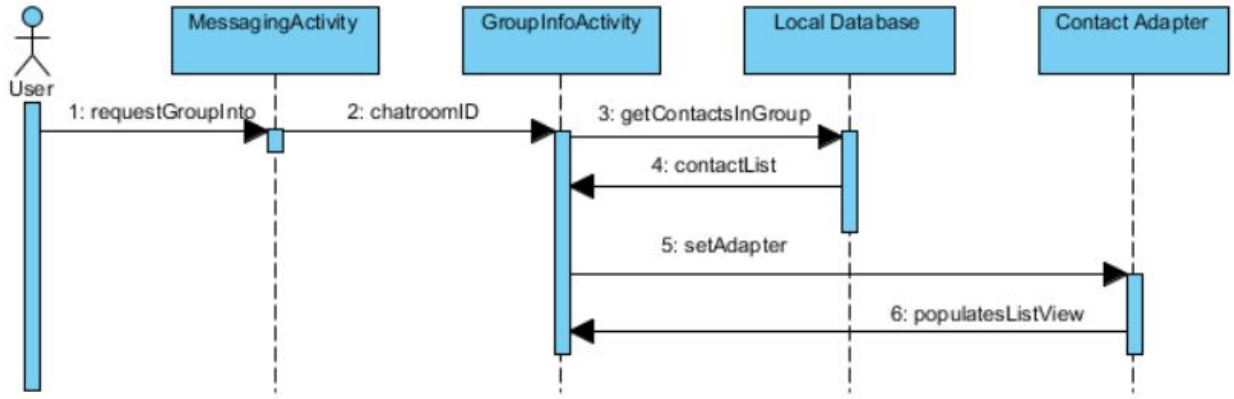

Figure 9: Sequence Diagram for viewing group users

EventInfoActivity provides information on existing events. This is done by querying the database for all events belonging to a group using the group ID. Clicking on an event gives a list of all users attending the event. The ListView is populated by an EventAdapter

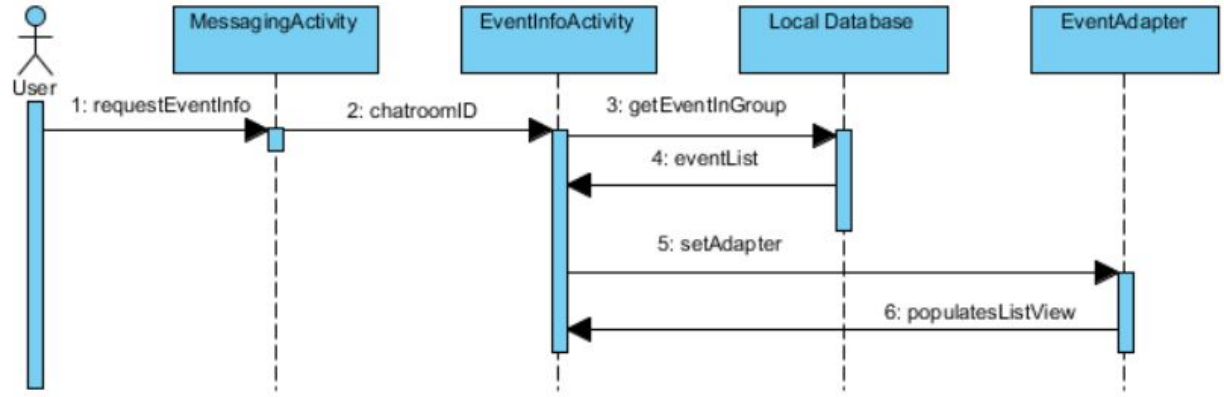

Figure 10: Sequence Diagram for viewing event info in group chatroom

EventsActivity allows users to create a new event. It provides date and time selection of the event along with the name. This is translated into epoch time in milliseconds (milliseconds since 1 Jan 1970) for unambiguous interpretation when received by the server and other users.

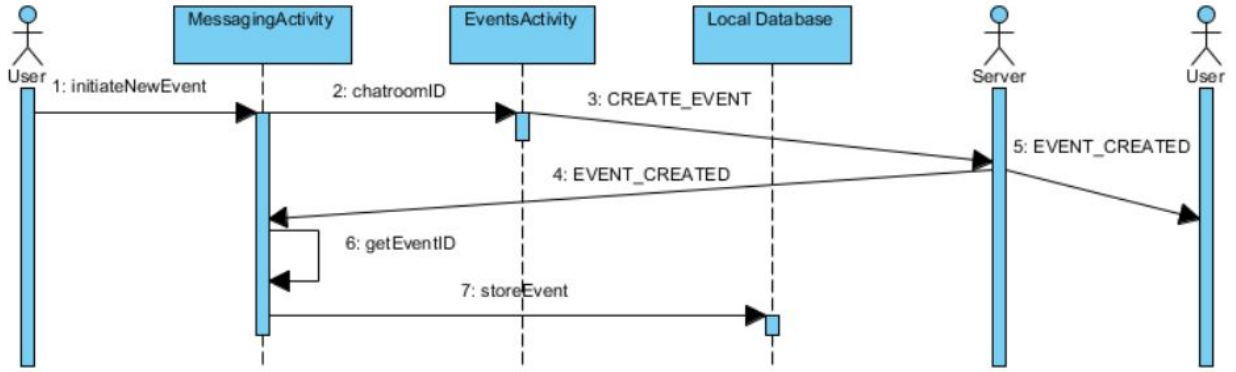

Figure 11: Sequence Diagram for creating new event in group chatroom

Notes

Activities pertaining to notes

NoteListActivity issues the request to sync notes with the server and populates the screen accordingly. The ListView is populated by a NoteAdapter. Clicking on a note allows the user to edit a note.

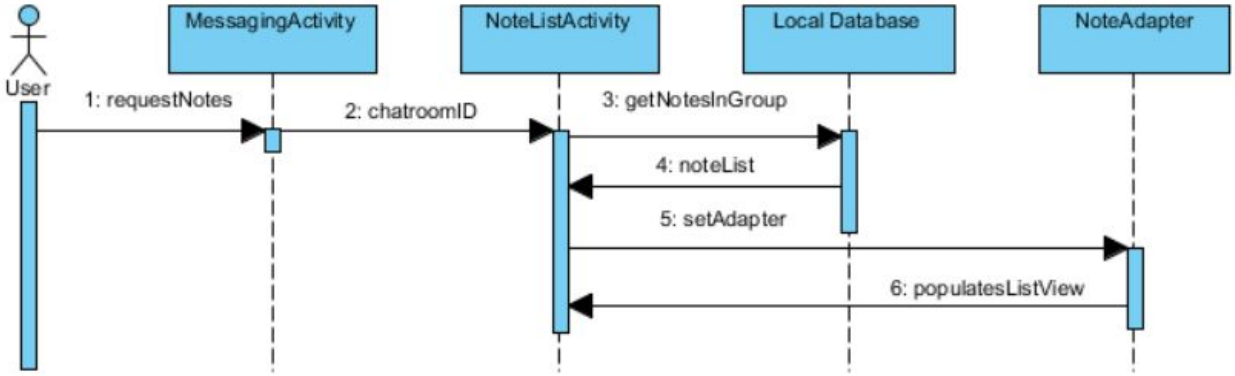

Figure 12: Sequence Diagram for viewing notes in chatroom

CreateNoteActivity allows users to create a new note for a group chat by specifying the title and body.

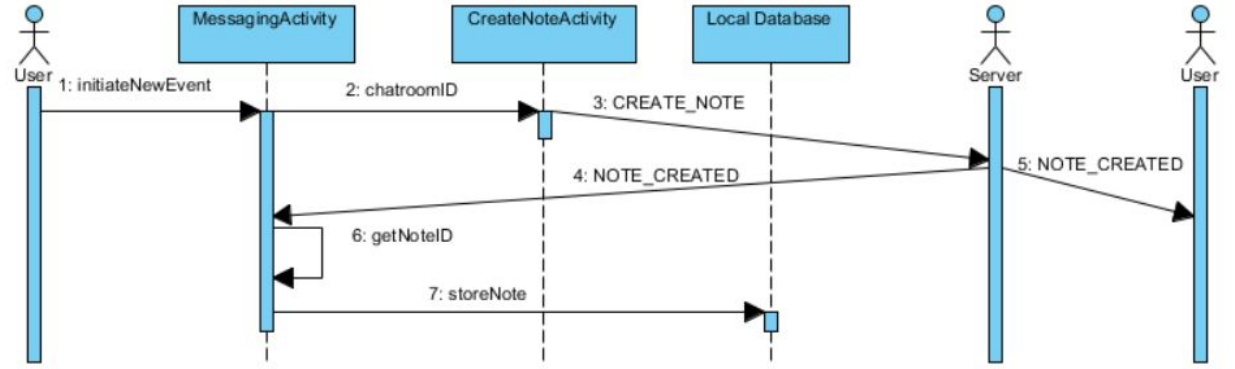

Figure 13: Sequence Diagram for creating a note in chatroom

EditNoteActivity allows users to view or make changes to an existing note. Because notes are always synchronized when NoteListActivity is started, the notes are always up to date the moment this activity is started. Once the activity is started however, the notes are not guaranteed to be up to date as another user may have edited the note in the meantime.

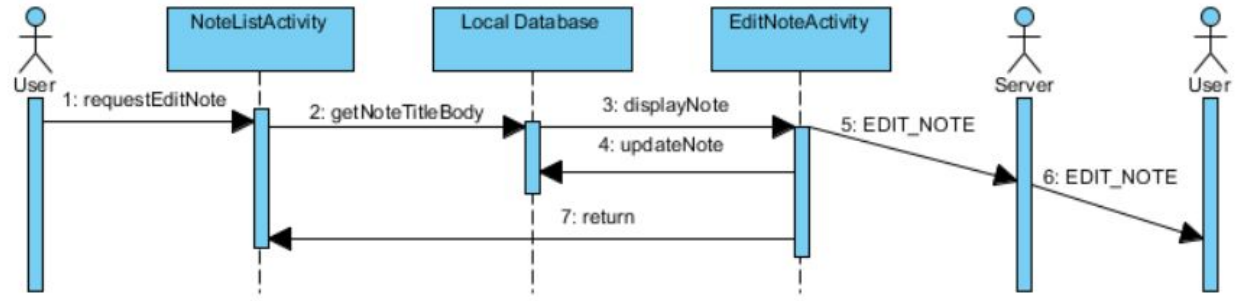

Figure 14: Sequence Diagram for editing a note in chatroom

Other Activities

LoginActivityCog is the main activity started when the user starts up GTFS. If shared preferences are found (if the user has logged in previously), LoginActivityCog retrieves them and immediately exits, presenting no UI. However if not, it brings up a screen for the user to enter his phone number, saves the phone number in the shared preferences and then brings the user to ProfileActivity. Initially we had intended for phone number verification through Twitter Fabric. However, Fabric has proven extremely problematic and due to the limited number of requests allows (only 10 a day), we have chosen to remove the functionality.

ProfileActivity allows users to create their profile. If successfully created, ProfileActivity stores the username in the shared preferences. When terminated, it opens up MainActivity.

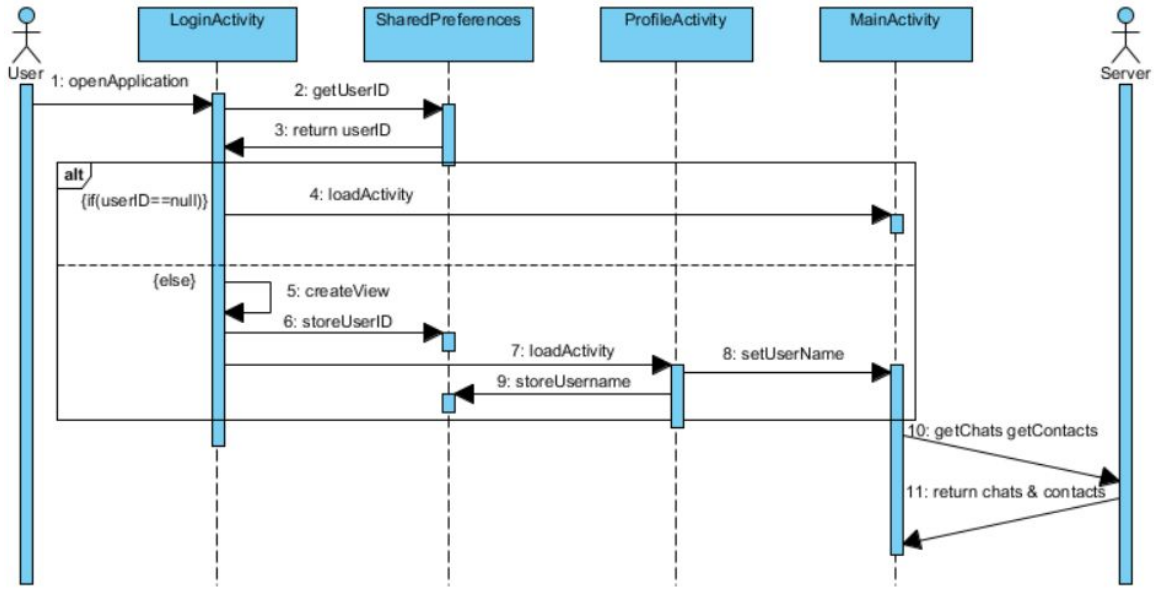

Figure 15: Sequence Diagram for logging in to application

ContactsActivity queries the user's phone for a list of contacts and performs a request to the Aurora for contacts who also use GTFS. ContactsActivity breaks the list of users into blocks of 15 and requests the server for matches. As one might expect, this is very costly in terms of network communication as every single contact must be checked with the server. Tapping on a contact on the screen creates a new personal chat with that user and brings the user to MessagingActivity. The ListView is populated by a ContactsAdapter.

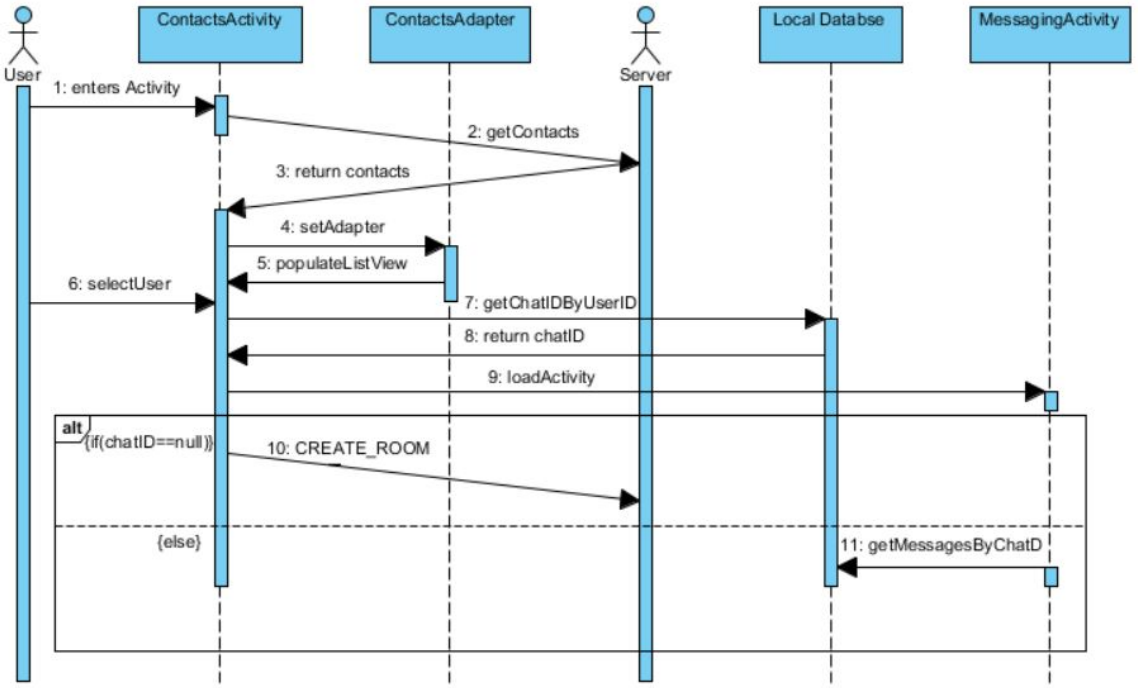

Figure 15: Sequence Diagram for viewing contacts available to message

GTFS Backend

The backend consists of three main components: the SQLite database (as interfaced by MessageDbAdapter), ManagerService and NetworkService. Broadly speaking, NetworkService handles all communication with the server and sends all outbound messages to ManagerService. ManagerService treats the messages accordingly, whether it is alerting the UI to updates or performing the appropriate database actions. GTFSClient is the custom application class and allows shared application-wide state. The only external library used in the backend is json-io[1], which allows the messages to be serialised to JSON for easy interpretation by Aurora.

MessageDbAdapter contains all methods used to access the SQLite database. These include storing new messages or notes, finding a username for a given ID or finding a list of all chats a user belongs to. MessageDbAdapter is designed not to modify the external state as much as possible. The exception is in the deleteExpiredChats() method which creates a notification to inform the user of deleted chats. We found this acceptable as we want every call that deletes expired chats to also create a notification.

[1] https://github.com/jdereg/json-io

ManagerService is the endpoint for most inbound messages. It extends Service, allowing it to run in the background even without an activity. It performs database operations whenever a new message is received for instance, or when a chatroom invitation is received. It also builds the notification for new messages received.

NetworkService is the interface between the client and the server. It extends IntentService, which allows new messages to be passed to it using the onHandleIntent() callback. This makes sending frugal as the service is only called upon when a relevant intent is sent. These messages are passed into a work queue to be processed sequentially. When created for the first time, NetworkService creates a thread that listens for incoming messages from the server every second. This is enforced by setting a flag in the Application class to check whether or not the listener thread has been created. Sending and receiving JSON objects is abstracted by the json-io package by creating new JsonReader and JsonWriter objects on an InputStream and OutputStream respectively. By using a Socket connected to the server, the relevant input and output streams are obtained, and reading and writing become a matter of calling the readObject() or writeObject() functions.

We chose to use such a model as we realised that most messages should not be handled by activities and we needed them to be handled by a service in the background. For smooth user experience, we had to offload as much work from the UI thread as much as possible. Using an IntentService to send messages seemed intuitive as it operates on a work queue system. However it was reasonable as well to have a different service to handle incoming messages as those are processor bound processes rather than the IO-bound processes of the NetworkService. Using the LocalBroadcastManager of Android also allows a straightforward implementation of message handling by defining a single callback in ManagerService to listen from a broadcast from NetworkService.

Supporting these main actors are the custom application class GTFSClient, stores state variables such as the username, ID and a data structure for storing message details to build notifications, as well as helper Java classes located in the Object package.

A special mention must be made to MessageBundle, which provides functions to create a valid message that can be accepted by Aurora. MessageBundle specifies a fixed number of types a message can have and requires mandatory fields to be defined in the constructor. A message is created by inserting the relevant values and the associated key values into a member Map, which is then serialised into JSON using the json-io's JsonWriter class. While the class provides methods to insert values without making the keys visible e.g. calling the putUsername() method to insert the username without having to define the key, creating a well-formed message is not the jurisdiction of the MessageBundle class, rather the creator of a new MessageBundle object must take responsibility to ensure that it fits the required fields as expected of Aurora.

Frontend - Backend Integration

As specified in the section GTFS Frontend, the frontend frequently interacts with the backend database to retrieve the relevant information to populate the views. Interactions become much more complex, however, when messages are involved.

Outbound Messages

All outbound messages resulting from user input originate in the frontend. The Activity obtains the relevant information from the user and creates a new MessageBundle object containing the relevant fields as required by Aurora. As mentioned in the MessageBundle section of GTFS Backend, creating a well-formed message is the responsibility of the Activity as neither MessageBundle nor NetworkService enforce any checks on the validity of the message. The message is sent to NetworkService by serialising the return value of MessageBundle.getMessage() (a Map containing the relevant key-value pairs) to JSON using json-io's JsonWriter class and adding it as an extra to an explicit Intent. NetworkService is then started using startService() and passing the explicity Intent. NetworkService deserialises the received message and writes it using the writeObject() method defined by json-io.

Inbound Messages

The UI is not affected directly by any outbound messages, only by inbound messages. As messages can be received asynchronously but requires communication between the backend and frontend, this is essentially a concurrency problem. When a message is received by NetworkService's listener thread, the message is sent to ManagerService by serialising the message to json, putting it as an extra and sending a local broadcast via LocalBroadcastManager. ManagerService deserialises the message and operates accordingly. To effect a change on the UI, ManagerService sends a local broadcast via LocalBroadcastManager. All activities that expect updates in their UI upon receiving some kind of message have an updateUI() method defined which allows them to respond to whatever relevant messages may be received. The updateUI() method allows the activities to read the received messages and perform the necessary changes. More often than not, this involves performing a database query to request for relevant information anew.

Aurora (Erlang backend)

Aurora is the working title for the Erlang backend that powers GTFS. Erlang as a language was chosen due to its natural ability as a language to power concurrent applications. The language itself was conceived with concurrency in mind, and all of its features are built to support this class of issues in computer programming. In the section below, we explain in greater detail what makes Erlang such a great language for building concurrent applications.

Aurora lives in the following public Github repository:
https://github.com/siawyoung/aurora-server

Why Erlang?

Erlang was conceived as a proprietary telecommunications language at Ericsson[2] in 1986 by Joe Armstrong (who continues to be an active core contributor), Robert Virding, and Mike Williams. Erlang rose to popularity as the programming language behind Ericsson's

[2] Erlang is a portmanteau of Ericsson and Language.

AXD301 switch, which was the first piece of software to achieve a system availability of nine '9's (99.999999999%)[3].

Erlang is a functional language, is dynamically and strongly typed, and uses eager evaluation (as opposed to Haskell, which is lazily evaluated).

Unlike most languages where concurrency and thread support is included as a library, Erlang provides built-in language support for creating and supervising processes. Erlang's explicit concurrency dictates that processes in Erlang use asynchronous message passing instead of shared variable state, thus sidestepping the need for explicit locks (although locks are still used elsewhere in Erlang programming, as you will read about later). Erlang processes are based on the Actor model - they are neither operating system processes nor virtual threads, but lightweight EVM (Erlang Virtual Machine) processes.

To put in perspective how lightweight Erlang processes are, Ulf Wiger, Senior Team Leader from Ericsson, performed a benchmark by spawning multiple processes and passing a message from one to another in a ring fashion. He reported[4] that on a machine running "64-bit erlang on a 1.5 GHz SPARC with 16 GB RAM", he was able "to create a ring of 20 million processes and pass a message around the ring. Average time to spawn a process: 6.3 us; average time to send & receive a message: 5.3 us". A newly created process takes about 300 words of memory, including 233 reserved words for its heap and stack area[5].

In other words, Erlang is a cool language built from the ground up for concurrency, unlike Java or other general purpose programming languages where concurrency support is clunky at best.

Aurora Concepts

Aurora is a JSON API server built in Erlang. The Android client will establish a TCP socket connection with Aurora, and communicate using JSON as the data format. JSON was

[3] http://ll2.ai.mit.edu/talks/armstrong.pdf
[4] https://groups.google.com/forum/#!original/comp.lang.functional/5kldn1QJ73c/T3py-yqmtzMJ
[5] http://www.erlang.org/doc/efficiency_guide/processes.html

chosen because of its wide adoption and availability of libraries, as well as ease of working with it as compared to XML or YAML.

Aurora itself recognizes only certain API endpoints in a specified format that the client must adhere to in order to get a response. Message validation is performed on every request and will be covered in more detail under the “Message Validation” section.
Request authentication and authorization is done with a session token that must be sent with every request. This session token is generated courtesy of Twitter’s Fabric system, which we use as our user verification layer, and provides our users with an onboarding flow similar to that of popular commercial chat apps.

Lastly, Aurora also maintains a private connection to a local Mnesia database system, which contains all of the data and state that passes through Aurora. Mnesia, like most things Erlang, lends itself well to distributed scaling, although we did not investigate that area in the course of this project.

Concurrency Issues

Concurrency issues in the context of Aurora can be broadly categorized into *application-level* and *database-level* concurrency issues.

Not unlike Java, Erlang’s TCP socket accepting function is a blocking call. Thus, we built Aurora to take advantage of Erlang’s ultra-lightweight processes and spawn a process to handle each incoming socket connection. Where it would be infeasible to do so in Java and JVM, it is very normal for Erlang and EVM to handle up to hundreds and thousands of concurrent processes without so much as a performance hiccup (of course, hardware considerations apply). This process spawning is discussed under “Lightweight Process Spawning”.

On the database level, Mnesia provides an easily-accessible API for developers to specify the kind of locks required for each database operation. Since all of the state is stored centrally in Mnesia, we need to pay particular attention to the application-database interface to make sure that we reduce any performance bottlenecks. Mnesia and

database-level concurrency issues are discussed towards the end of this report under "Mnesia Database".

Code Architecture

Aurora itself can be split into 5 monolithic files:

1. `chat_server.erl`
2. `tcp_server.erl`
3. `controller.erl`
4. `messaging.erl`
5. `validation.erl`

Aurora can be started in the Erlang console by entering the following commands:

`chat_server:install([node()]).`
This creates a Mnesia database and should only be run if it hasn't yet been created.

`mnesia:start().`
This boots the Mnesia database and its associated processes up.

`chat_server:start(8091).`
This runs Aurora's initialization function, which is provided below:
Line 13 to 16, `chat_server.erl`

```
start(Port) ->
    mnesia:wait_for_tables([aurora_users, aurora_chatrooms, aurora_message_backlog,
aurora_chat_messages, aurora_events, aurora_notes], 5000),
    controller:start(),
    tcp_server:start(?MODULE, Port, {?MODULE, pre_connected_loop}).
```

We first ensure that all of Mnesia's tables are online, then start a `controller` instance and a `tcp_server` instance with the specified port.

`chat_server.erl` is the file where Aurora's initialization function (`start()`) is located and where the main process loops are located. There are two loops - one for pre-authentication (`pre_connected_loop()`), another for post-authentication (`connected_loop()`).

`tcp_server.erl` is in charge of accepting incoming client connections and spawning a separate `chat_server` process to handle each connection. How exactly this process spawning is accomplished is discussed in "Lightweight Process Spawning".

`messaging.erl` and `validation.erl` are library modules. All server-to-client response methods are handled in `messaging.erl` and is covered under "Sending Client Responses and Handling Poor Network Connections". Validation involves checking for the presence and nullity of the various fields for each message type and is managed by the library module `validation.erl`.

And last but not least, `controller.erl` contains all of the business logic for Aurora and is the main interface between Aurora and the Mnesia database. It includes a private API for communicating with the database.

Lightweight Process Spawning

As mentioned earlier, Erlang processes are extremely lightweight. Therefore, we can exploit this to our advantage by spawning a process to handle each incoming socket connection. This process spawning happens in `tcp_server.erl`, which has been fully annotated and commented. This section will thus focus on a high-level overview.

`tcp_server` is initialized by `chat_server's start` function, and will spawn a single main acceptor socket that listens on the specified port (default 8091).

Line 49 to 51, `tcp_server.erl`

```
accept(State = #server_state{lsocket=LSocket, loop = Loop}) ->
    proc_lib:spawn(?MODULE, accept_loop, [{self(), LSocket, Loop}]),
    State.
```

When a new socket is connected, `tcp_server` calls itself asynchronously to spawn a new process that listens on the main acceptor socket, and then passes the connected socket object returned by the acceptor socket's accept function to `chat_server`'s `pre_connected_loop` function, where Aurora's main business logic is located.

Line 59 to 62, `tcp_server.erl`

```
accept_loop({Server, LSocket, {Module, LoopFunction}}) ->
    {ok, Socket} = gen_tcp:accept(LSocket),
    gen_server:cast(Server, {accepted, self()}),
    Module:LoopFunction(Socket).
```

External Libraries

The only external library used is `jsx`[6], an Erlang JSON library for producing and parsing JSON. With `jsx`, one can simply call `decode` and `encode` like such:

```
1> jsx:decode(<<"{\"library\": \"jsx\", \"awesome\": true}">>).
[{<<"library">>,<<"jsx">>},{<<"awesome">>,true}]
2> jsx:decode(<<"{\"library\": \"jsx\", \"awesome\": true}">>, [return_maps]).
#{<<"awesome">> => true,<<"library">> => <<"jsx">>}
3> jsx:decode(<<"[\"a\",\"list\",\"of\",\"words\"]">>).
[<<"a">>, <<"list">>, <<"of">>, <<"words">>]
```

to convert between UTF-8 encoded binaries containing JSON strings and native Erlang data types. jsx also takes the liberty of converting `true`, `false`, and `null` JavaScript values into their corresponding `true`, `false` and `null` atoms in Erlang. All input to be parsed should be UTF-8 encoded binaries, although `jsx` has some limited support for replacing invalid codepoints and poorly formed sequences.

In Aurora, `jsx` acts as a middleware layer. All incoming JSON requests from the Android client are parsed into native Erlang key-value maps using `jsx`, and all outgoing JSON responses from Aurora are converted into UTF-8 encoded binaries before being written into the TCP socket stream.

`jsx` also comes with several useful options. Firstly, we opted to convert JSON keys to atoms. Atoms in Erlang are literal constants, not unlike the concept of Enums in Java, or symbols in Ruby. Atoms in Erlang are identified by "variables" that start with a lower-case alphabetical letter (and yes, true Erlang variables must start with an upper-case alphabetical letter). The atom text is stored just once in the atom table in memory during

[6] https://github.com/talentdeficit/jsx

the running of the Erlang Virtual Machine (EVM). Atoms help to maintain consistency across the entire program when describing common types of data.

We also use the `return_maps` option in `jsx`, which coerces the `jsx` parsed output into Erlang maps. Erlang maps are a relatively new introduction to Erlang, having only been introduced from Erlang R17 in February 2014. As its name implies, Erlang maps are used to map unique keys to values.

JSON API over TCP

This section describes the API endpoint and their design and specifications in more detail. The table below is a short excerpt of the list of API endpoints.

Name	**Description**	**Client to Server JSON Payload**	**Server to Client JSON Payload**

AUTH	Authenticating	`{`
		`  “username”: String,`
		`  “session_token”: String,`
		`  “from_phone_number”: String,`
		`  “type”: “AUTH”`
			`{`
			`  “status”: Integer,`
			`  “type”: “AUTH”`
			`}`

TEXT	Client-to-client / client-to-group message	`{`
		`  “from_phone_number”: String,`
		`  “chatroom_id”: String,`
		`  “session_token”: String,`
		`  “message”: String,`
		`  “timestamp”: String,`
		`  “type”: “TEXT”`
		`}`
			`To: sending client`
			`{`
			`  “type”: “TEXT”,`
			`  “status”: Integer,`
			`  “message”: {`
			`  “message_id”: String,`
			`  “to_phone_number”: String`
			`  }`
			`}`
			
			`To: sending and receiving clients`
			`{`
			`  “message_id”: String,`
			`  “from_phone_number”: String,`
			`  “chatroom_id”: String,`
			`  “Message”: String,`
			`  “timestamp”: String,`
			`  “type”: “TEXT_RECEIVED”`
			`}`

Note in particular the third and fourth column, which describes the type of request payload that Aurora should expect from the client, as well as the type of response payload the client expects from Aurora.

For certain resources like notes and events, we implemented the related endpoints RESTfully[7]. For example, for notes, we have the following RESTful CRUD[8] actions:

GET_NOTES	Fetch all notes belonging to a room	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: String,
		“type”: “GET_NOTES”
		}
			{
			“chatroom_id”: String,
			“notes”: [
			{
			“note_id”: String,
			“note_title”: String,
			“note_text”: Text,
			“note_creator”: String,
			}
			],
			“type”: “GET_NOTES”
			
			}

CREATE_NOTE	Create a note	{
		“from_phone_number”: String,
		“session_token”: String,
		“note_title”: String,
		“note_text”: Text,
		“chatroom_id”: String,
		“type”: “CREATE_NOTE”
		}
			{
			“status”: Integer,
			“note_id”: Integer,
			“type”: “CREATE_NOTE”
			}

EDIT_NOTE	Edit a note	{
		“from_phone_number”: String,
		“session_token”: String,
		“note_id”: String,
		“note_text”: String,
		“chatroom_id”: String,
		“type”: “EDIT_NOTE”
		}
			{
			“status”: Integer
			}

DELETE_NOTE	Delete a note	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: String,
		“note_id”: String,
		“type”: “DELETE_NOTE”
		}
			{
			“status”: Integer
			}

Although CRUD maps very well to HTTP action verbs, in the absence of HTTP we can still borrow these concepts when writing our own custom API endpoints.

[7] REST stands for REpresentational State Transfer and is the major underlying architecture used to manage web resources.
[8] CRUD stands for Create, Read, Update, Delete. These are the 4 basic functions of persistent storage.

The exhaustive list of all API endpoints supported by Aurora can be found in the Appendix section - Table 1.

Message Validation

After the incoming data is determined to be valid JSON, message validation is performed on the JSON payload to ensure that the required fields are present for each message type and are not null. The functions used to accomplish this are as below:

Line 338 to 348, `validation.erl`

```
validate_fields(Fields, ParsedJson) ->

  F = fun(Field) ->
    maps:get(Field, ParsedJson, missing_field)
  end,

  Items = lists:map(F, Fields),
  check_missing_or_null(Items).

check_missing_or_null(Items) ->
  (lists:member(missing_field, Items)) or (lists:member(null, Items)).
```

In the `validate_fields` function we attempt to get each field from the `ParsedJson` variable, which at this point has been parsed by `jsx` into a Erlang map. If the field is missing, the `maps:get` function returns the atom `missing_field` instead. This local function is then mapped onto the list of fields, and the resulting list is checked for the presence of the `missing_field` or `null` atoms. Recall from the "External Libraries" section that the `null` atom is a result of jsx parsing the null data type from JavaScript.

The below is an example of the above validation functions being used:

Line 314-322, `validation.erl`

```
validate_get_events_request(ParsedJson) ->
  case validate_fields([from_phone_number, session_token, chatroom_id], ParsedJson) of
    true ->
        io:format("Message from validate_get_events_request: Invalid payload~n", []),
        invalid_request;
    false ->
        io:format("Message from validate_get_events_request: Valid payload~n", []),
        valid_request
  end.
```

When a `GET_EVENTS` message is sent from the client, Aurora expects to see a `from_phone_number`, a `session_token` and a `chatroom_id` field in the JSON payload. By feeding these fields as atoms into the `validate_fields` function, we can determine if any of these fields are missing or null by its return value.

Authentication and Authorization

User request authentication and authorization is achieved through the use of a session token that is generated by Twitter's Fabric platform, specifically with its Digits[9] module, which allows users to sign in with their mobile phone number. When users enter GTFS for the first time, they will be prompted to enter their phone number, and Digits will send a confirmation SMS containing a randomly-generated token to that mobile number. Upon entering the correct token, Digits will generate a session token. GTFS will then store it locally and send an `AUTH` message to Aurora with the user's phone number and session token. Aurora will check if the user has already registered before, and update his or her session token if so.

After this point, all user requests from GTFS to Aurora must include the session token. All incoming requests are checked for the session token and matched against the one stored in Mnesia. This authorization step is called synchronously for security reasons, and is a performance hit that we have decided to take on.

The particular line in the chat server that makes the synchronous call to the controller for authorization is as below:

Line 100 to 111, `chat_server.erl`

```
ParsedJson ->

                    AuthorizedStatus = call_authorize_request(ParsedJson),
                    MessageType = maps:get(type, ParsedJson),
                    PhoneNumber = maps:get(from_phone_number, ParsedJson),

                    if
                        AuthorizedStatus =/= authorized ->
```

[9] https://dev.twitter.com/twitter-kit/ios/digits

```
                            %% Session tokens don't match
                            messaging:send_status_queue(Socket, PhoneNumber, 4,
MessageType),
                            connected_loop(Socket);
```

Line 215 to 222, `chat_server.erl`

```
call_authorize_request(ParsedJson) ->
    Status = gen_server:call(controller, {authorize_request, ParsedJson}),
    io:format("Message from authorize_request method: ~p~n", [Status]),
    case Status of
        authorized   -> authorized;
        no_such_user -> no_such_user;
        _            -> error
    end.
```

Because the call is synchronous, we expect a return value from it, and that return value will used to determine if the user is allowed to proceed.

Line 56 to 79, `controller.erl`

```
handle_call({authorize_request, ParsedJson}, _From, State) ->

    TokenToCheck = maps:get(session_token, ParsedJson),
    PhoneNumber  = maps:get(from_phone_number, ParsedJson),

    case find_user(PhoneNumber) of
        #{username     := _UserName,
        session_token  := SessionToken,
        rooms          := _Rooms,
        current_ip     := _IPaddress,
        active_socket  := _Socket} ->

            case SessionToken == TokenToCheck of
                true ->
                    {reply, authorized, State};
                false ->
                    {reply, error, State}

            end;

        _ ->
            {reply, no_such_user, State}

    end.
```

Error Handling

Since Aurora is a custom-built JSON API served over TCP, we do not have the benefit of using HTTP status codes for client callbacks. Instead, we wrote our own status protocol

from scratch. The benefit to doing so is that we can have much more fine-grained callback behavior. The table of status code mappings are as below:

Status Code	Meaning
0	Invalid JSON
1	OK
2	Some required fields are missing in the payload.
3	Database transaction error.
4	Session token is invalid.
5	User or chat room ID is invalid.
6	Invalid message type.
7	Recipient socket is closed.
8	User does not have administrator rights.
9	User has already cast a vote/hasn't yet cast a vote (for events)

As you can see, writing our own protocol from scratch allows us to precisely define each possible error that can happen within the application flow, and to feed that back to the client for the appropriate callback functions to be called (e.g. Toast messages prompting the user to try again). Implementation wise, the status code is attached as a key-value pair in the JSON response returned by the server.

Controller Actions

`controller.erl` is by far the largest file in Aurora, and contains all of the business logic powering Aurora and GTFS. It communicates with `chat_server.erl` through the use of synchronous and asynchronous calls, termed in Erlang as `calls` and `casts`. Critical segments of the application flow - the user registration and authentication calls - are blocked synchronously for good reason. We have to ensure that these actions are completed before further controller actions can take place. It also contains a private CRUD API for interacting with the Mnesia database.

Since the source file is fully annotated, and for the sake of brevity, this report will present two representative examples that showcase how `controller.erl` handles asynchronous calls from chat_server.

Line 249 to 284, `controller.erl`

```
handle_cast({create_chatroom, ParsedJson, FromSocket}, State) ->

    FromPhoneNumber = maps:get(from_phone_number, ParsedJson),
    RawUsers = maps:get(users, ParsedJson),

    ValidateResult = validate_users(RawUsers),

    case ValidateResult of

        {ok, users_validated, UserInfo} ->

            DatabaseResult = create_chatroom(ParsedJson),
            case DatabaseResult of
                {ok, room_created, ChatRoomID, ChatRoomName, Users, Expiry} ->

                    messaging:send_message(FromSocket, FromPhoneNumber,
                    jsx:encode(#{
                        <<"status">>        => 1,
                        <<"chatroom_id">>   => ChatRoomID,
                        <<"type">>          => <<"CREATE_ROOM">>
                    })),

                    % send ROOM_INVITATION message to all invited users
                    F = fun(User) ->
                        send_chatroom_invitation(ChatRoomID, ChatRoomName, Users,
                                                 maps:get(active_socket, User),
                                                 maps:get(phone_number, User), Expiry, group)
                    end,
                    lists:foreach(F, UserInfo);

                error ->
                    messaging:send_status_queue(FromSocket, FromPhoneNumber, 3,
                    <<"CREATE_ROOM">>, <<"Database transaction error">>)
            end;

        error ->
            messaging:send_status_queue(FromSocket, FromPhoneNumber, 5, <<"CREATE_ROOM">>,
            <<"Some users are invalid">>)
    end,
    {noreply, State};
```

As indicated by the `create_chatroom` atom, this `handle_cast` function is in charge of the business logic behind creating a group chatroom.

From Appendix A, we can see that the fields required for creating a chatroom are as follows: `chatroom_name`, `session_token`, `from_phone_number`, `users`, and `expiry`.

The function starts off by first checking if all of the phone numbers in the users list are registered users by using the `validate_users` function. If at least one of the phone numbers are not valid, `validate_users` simply returns an `error` atom, which we handle by replying the sending client with a status 5 (user ID not valid). If all of the phone numbers are valid, then `validate_users` replies with the tuple `{ok, users_validated, UserInfo}`, where the `UserInfo` variable contains a list of maps mapping each user phone number to its active socket object.

Line 748 to 776, `controller.erl`

```
validate_users(Users) ->

    F = fun(UserPhoneNumber) ->

        DatabaseResult = find_user(UserPhoneNumber),
        io:format("~p~p~n", [UserPhoneNumber, DatabaseResult]),
        case DatabaseResult of
            #{username     := _UserName,
            session_token  := _SessionToken,
            rooms          := _Rooms,
            current_ip     := _IPaddress,
            active_socket  := Socket} ->

                #{phone_number  => UserPhoneNumber,
                  active_socket => Socket};

            no_such_user -> no_such_user
        end
    end,

    UserSockets = lists:map(F, Users),

    case lists:member(no_such_user, UserSockets) of
        true ->
            error;
        false ->
            {ok, users_validated, UserSockets}
    end.
```

We then proceed to create the chatroom with the `create_chatroom` function, which is one of the functions that interact directly with Mnesia.

Line 973 to 996, `controller.erl`

```
create_chatroom(ParsedJson) ->

    AdminUser    = maps:get(from_phone_number, ParsedJson),
    ChatRoomName = maps:get(chatroom_name, ParsedJson, null),
    Users        = maps:get(users, ParsedJson),
    Group        = maps:get(group, ParsedJson, true),
    Expiry       = maps:get(expiry, ParsedJson),

    ChatRoomID   = timestamp_now(),

    F = fun() ->
        Status = mnesia:write(#aurora_chatrooms{chatroom_id   = ChatRoomID,
                                                admin_user    = AdminUser,
                                                chatroom_name = ChatRoomName,
                                                group         = Group,
                                                expiry        = Expiry,
                                                room_users    = Users}),
        case Status of
            ok -> {ok, room_created, ChatRoomID, ChatRoomName, Users, Expiry};
            _  -> error
        end

    end,
    mnesia:activity(transaction, F).
```

If the record is unable to be written in Mnesia for some reason, then the transaction will fail and return an `error` atom, which is sent back as a status 3 (database transaction error). If the chatroom record was successfully created, then we send a status 1 back to the sending client, as well as a chatroom invitation message to all of the users in the chatroom (this is where `UserInfo` and the active socket objects come in).

Line 689 to 716, `controller.erl`

```
send_chatroom_invitation(ChatRoomID, ChatRoomName, Users, Socket, PhoneNumber, Expiry, Type)
->

    if
        Type == group ->

            messaging:send_message(Socket, PhoneNumber,
                jsx:encode(#{
                    <<"chatroom_id">>   => ChatRoomID,
                    <<"chatroom_name">> => ChatRoomName,
                    <<"users">>         => Users,
                    <<"type">>          => <<"ROOM_INVITATION">>,
                    <<"expiry">>        => Expiry,
                    <<"group">>         => true
                }));

        Type == single ->

            messaging:send_message(Socket, PhoneNumber,
                jsx:encode(#{
```

```
                    <<"chatroom_id">>   => ChatRoomID,
                    <<"users">>         => Users,
                    <<"type">>          => <<"SINGLE_ROOM_INVITATION">>,
                    <<"expiry">>        => Expiry,
                    <<"group">>         => false
                })
    end,

    add_room_to_user(ChatRoomID, PhoneNumber).
```

When the chatroom invitation is sent, `add_room_to_users` is also called to add the chatroom ID foreign key association to each user. This will be useful when we need to get all the rooms belonging to a user, as in the `GET_ROOMS` API endpoint.

The second example concerns the transfer of administrator rights of a chatroom from one user to another. Transferring administrator rights is necessary in some cases - for example, an administrator cannot leave the chatroom until administrator rights have been transferred away to another user.

Line 345 to 389, `controller.erl`

```
handle_cast({transfer_admin, ParsedJson, FromSocket}, State) ->

    FromPhoneNumber = maps:get(from_phone_number, ParsedJson),
    ToPhoneNumber   = maps:get(to_phone_number, ParsedJson),
    ChatRoomID      = maps:get(chatroom_id, ParsedJson),

    NewAdminUser = find_user(ToPhoneNumber),
    FoundRoom    = find_chatroom(ChatRoomID),

    if
        NewAdminUser == no_such_user ->
            messaging:send_status_queue(FromSocket, FromPhoneNumber, 5,
<<"TRANSFER_ADMIN">>, <<"No such user">>);

        FoundRoom == no_such_room ->
            messaging:send_status_queue(FromSocket, FromPhoneNumber, 5,
<<"TRANSFER_ADMIN">>, <<"No such room">>);

        % notice the "true" idiom used here - this is the "else" of Erlang
        % meaning that this branch will always be evaluated
        true ->
            case check_if_user_in_room(FoundRoom, FromPhoneNumber) and
check_if_user_in_room(FoundRoom, ToPhoneNumber) of
                false ->
                    messaging:send_status_queue(FromSocket, FromPhoneNumber, 5,
<<"TRANSFER_ADMIN">>, <<"User is not a member of the chatroom.">>);
                true ->
                    case check_if_user_admin(FromPhoneNumber, ChatRoomID) of

                    user_not_admin ->
```

```
                        messaging:send_status_queue(FromSocket, FromPhoneNumber, 8,
<<"TRANSFER_ADMIN">>, <<"User is not admin of the room">>);
                    user_is_admin ->
                        update_chatroom(change_admin, ChatRoomID, ToPhoneNumber),

                        messaging:send_status_queue(FromSocket, FromPhoneNumber, 1,
<<"TRANSFER_ADMIN">>, #{<<"chatroom_id">> => ChatRoomID}),

                        messaging:send_message(maps:get(active_socket, NewAdminUser),
ToPhoneNumber,
                            jsx:encode(#{
                            <<"chatroom_id">> => ChatRoomID,
                            <<"type">> => <<"NEW_ADMIN">>
                        }))

                    end

            end

    end,

    {noreply, State};
```

From Appendix A, we can see that the fields required for transferring administrator rights are as follows: `from_phone_number`, `to_phone_number`, `session_token`, and `chatroom_id`.

Assuming that `from_phone_number` refers to an actual user, we check the `to_phone_number` and the `chatroom_id` to obtain their records and make sure that they refer to existing records in the database (or else send a status 5).

Then, we check that both users are in the chatroom, and that the `from_phone_number` user is actually the existing administrator of the chatroom. If all these validations pass, we then call `update_chatroom` to update the chatroom record with the new administrator.

One especially innovative feature of Aurora, and one which may not be apparent on first glance, is that all identification numbers, including those uniquely identifying messages, chat rooms, notes and events, are generated using Erlang's `now()` BIF. According to Erlang's official documentation on the `now()` function, each call to `now()` is **guaranteed**[10] to return a monotonically increasing number representing the current epoch time.

[10] This guarantee is ensured through the use of a lock, although the performance hit is minimal (from anecdotal experience)

What this means, especially for the client side, is that all responses received by the client side can be uniquely identified *and* ordered in the correct order as it was received by the server. This has proven to be extremely helpful, especially when ordering is required between different entities, such as vote notifications and message notifications. This also allows us to use this generated IDs as unique primary keys for all records to be stored in Mnesia. This solution is both elegant and has been found well.

An example of the above in action, when text messages are written into Mnesia in the `create_chat_message` function:

Line 1128 to 1134, `controller.erl`

```
create_chat_message(ParsedJson) ->

    ChatRoomID      = maps:get(chatroom_id, ParsedJson),
    FromPhoneNumber = maps:get(from_phone_number, ParsedJson),
    Message         = maps:get(message, ParsedJson),
    TimeStamp       = maps:get(timestamp, ParsedJson),
    ChatMessageID   = timestamp_now(),

    … rest of function
```

Mnesia Database

Mnesia is a distributed, real-time, and relational database management system written in Erlang. Although it has cross-platform support, Mnesia's use case is mainly as a persistent storage for Erlang systems. Its query language is written in Erlang instead of SQL, which, similar to other ORMs[11] like Active Record in Ruby on Rails, allows developers to represent database transactions in the same language throughout the application. Furthermore, since Erlang is a functional language, transactions are represented very intuitively and naturally as so-called "functional blocks". These functional blocks are run with Mnesia's `transaction` function, like below:

[11] ORM stands for Object Relational Mapping.

Line 1258 to 1275, `controller.erl`

```
vote_event(ParsedJson) ->

    EventID         = maps:get(event_id, ParsedJson),
    FromPhoneNumber = maps:get(from_phone_number, ParsedJson),

    F = fun() ->

        [ExistingEvent] = mnesia:wread({aurora_events, EventID}),
        {aurora_events, _, _, _, _, Votes} = ExistingEvent,
        case lists:member(FromPhoneNumber, Votes) of
            true -> vote_already_cast;
            false ->
                UpdatedVotes = lists:append(Votes, [FromPhoneNumber]),
                UpdatedEvent = ExistingEvent#aurora_events{votes = UpdatedVotes},
                mnesia:write(UpdatedEvent)
        end
    end,
    mnesia:activity(transaction, F).
```

Mnesia achieves ACID[12] quite easily. Although Mnesia's API also supports "dirty" versions of all its operations, it compromises the *Atomicity* and *Isolation* aspects of ACID, so Aurora sticks to using safe operations throughout the application (although evidence seems to suggest that dirty operation performance can be up to 5-10x faster than their safe counterparts, which may warrant limited usage in certain endpoints, subject to real-life usage benchmarking).

In Aurora, Mnesia is set up with `disc_copies` options and all tables configured to the "`set`" type. The `disc_copies`[13] option sets up Mnesia to write to disk, as well as keep a copy of the database in memory. All operations will first be transacted in memory, before being written to disk. Setting the set type on all tables means that all keys must be unique. This is not a problem for Aurora as all entities identified by a uniquely generated timestamp ID (refer to "Controller Actions" section). By default, the primary key of a table is the first field as listed in the record. In Mnesia, tables of the `set` type are implemented with hash tables, and as such offer **constant time** lookups. This is a huge boon to database performance.

[12] ACID stands for Atomicity, Consistency, Isolation, Durability and is the litmus test for a well-implemented database system.

[13] The other options are `ram_copies`, which keeps the database only in memory, with an `dump_tables` function to write to disk, and `disc_only_copies`, which keeps the database only on disc to save memory at the cost of much slower performance.

Mnesia Locks

Mnesia uses standard two-phase locking (2PL) commit protocol with 5 types of locks:

1. Read locks
    a. A lock set on one replica of a record before it is read
2. Write locks
    a. A lock set on all replicas of a record before writing takes place
3. Read table locks
    a. Table wide lock, useful when a transaction traverses an entire table, instead of instantiating locks for each record in the table
4. Write table locks
    a. If a large number of transactions are to be written, a table wide write lock can be used
5. Sticky locks
    a. Only useful when used on replicated tables - a normal write lock on replicated tables across networks involves sending an RPC (remote procedure call) to set up and acquire networked locks across all replicas, whereas using a sticky lock will "persist" the lock on one of the nodes, so that further attempts to acquire the lock will simply use the sticky lock and not attempt to do a RPC (remote procedure call) to acquire networked locks. This is especially useful in a master-slave configuration where most of the operations are done on a single node, with the other nodes acting as backup

Create Operations

As stated earlier, all database operations to Mnesia in Aurora are based on CRUD (with function overloading). Within that paradigm, Aurora utilizes different types of locks for each type of operation.

When creating a new record, Aurora uses Mnesia's `write` function, which supports two types of locks, `write` and `sticky_write`. Since Aurora only uses Mnesia on 1 node, we use the default `LockKind`, which is `write`.

For example, when creating a new user:

Line 948 to 971, `controller.erl`

```
create_user(ParsedJson, Socket) ->
    PhoneNumber  = maps:get(from_phone_number, ParsedJson),
    UserName     = maps:get(username, ParsedJson),
    SessionToken = maps:get(session_token, ParsedJson),

    case inet:peername(Socket) of

        {ok, {IPaddress, _Port}} ->

            F = fun() ->
                mnesia:write(#aurora_users{phone_number = PhoneNumber,
                                           username = UserName,
                                           session_token = SessionToken,
                                           current_ip = IPaddress,
                                           active_socket = Socket}),
                mnesia:write(#aurora_message_backlog{phone_number = PhoneNumber})
            end,
            mnesia:activity(transaction, F);

        _ ->

            socket_error

    end.
```

Read Operations

When reading, we use Mnesia' `read` function, which supports three types of locks - `read`, `write`, and `sticky_write`. We use the `read` lock since the table is of the `set` type, meaning that all `read` operations will return either 1 or 0 records.

Line 1050 to 1071, `controller.erl`

```
find_chatroom(ChatRoomID) ->

    F = fun() ->
        case mnesia:read({aurora_chatrooms, ChatRoomID}) of
            [#aurora_chatrooms{chatroom_name = ChatRoomName,
                               room_users    = RoomUsers,
                               expiry        = Expiry,
                               group         = Group,
                               admin_user    = AdminUser}] ->

                #{chatroom_id   => ChatRoomID,
                  chatroom_name => ChatRoomName,
                  room_users    => RoomUsers,
                  expiry        => Expiry,
                  group         => Group,
                  admin_user    => AdminUser};
```

```
            _ ->
                no_such_room
        end
    end,
    mnesia:activity(transaction, F).
```

Another kind of read operation Aurora does involves traversing the whole table. This happens when we are searching for more than 1 record that fulfils a particular criteria. In such cases, Aurora will utilize a table wide read lock. For example, for the `FIND_NOTES` API endpoint, Aurora is supposed to return a list of all note records that belong to a certain chatroom. Since the primary key in the `aurora_notes` table is the note ID, we need to search instead for all notes that matches a certain chatroom ID foreign key.

Line 1149 to 1162, `controller.erl`

```
find_notes(ChatRoomID) ->
    F = fun() ->
        case mnesia:match_object({aurora_notes, '_', ChatRoomID, '_', '_', '_'}) of
            [] ->
                no_notes;
            Notes ->
                F = fun(Note) ->
                    {aurora_notes, NoteID, ChatRoomID, NoteTitle, NoteText, FromPhoneNumber}
= Note,
                    #{note_id => NoteID, chatroom_id => ChatRoomID, note_title => NoteTitle,
note_text => NoteText, from_phone_number => FromPhoneNumber}
                end,
                lists:map(F, Notes)
        end
    end,
    mnesia:activity(transaction, F).
```

We use Mnesia's `match_object` function, which takes a pattern as an argument with which to match against.

Update Operations

Many operations that Aurora does are update operations. In update operations, the transaction functional object is slightly more complicated. We want to read the record, create a new record based off the existing values (remember that Erlang, being a single assignment functional language, does not allow variables to change state), then write the new record back into the table.

In order to support such transactions, we need to use Mnesia's `read` function with a `write` lock. Naturally enough, Mnesia provides us with a helper function called `wread`, which is basically just a shortcut to writing `mnesia:read(Tab, Key, write)`, Mnesia's read function with a write lock explicitly specified.

Line 1246 to 1263, `controller.erl`

```
vote_event(ParsedJson) ->

    EventID         = maps:get(event_id, ParsedJson),
    FromPhoneNumber = maps:get(from_phone_number, ParsedJson),

    F = fun() ->

        [ExistingEvent] = mnesia:wread({aurora_events, EventID}),
        {aurora_events, _, _, _, _, Votes} = ExistingEvent,
        case lists:member(FromPhoneNumber, Votes) of
            true -> vote_already_cast;
            false ->
                UpdatedVotes = lists:append(Votes, [FromPhoneNumber]),
                UpdatedEvent = ExistingEvent#aurora_events{votes = UpdatedVotes},
                mnesia:write(UpdatedEvent)
        end
    end,
    mnesia:activity(transaction, F).
```

Delete Operations

And lastly, delete operations utilize a write lock by default.

Line 1202 to 1209, `controller.erl`

```
delete_note(ParsedJson) ->

    NoteID = maps:get(note_id, ParsedJson),

    F = fun() ->
        mnesia:delete({aurora_notes, NoteID})
    end,
    mnesia:activity(transaction, F).
```

Aurora utilizes 6 tables within Mnesia to store the various types of records. The table schemas are listed in full in Appendix A.

1. `aurora_users`

2. aurora_chatrooms
3. aurora_chat_messages
4. aurora_backlog_messages
5. aurora_events
6. aurora_notes

Sending Client Responses and Handling Poor Network Connections

`messaging.erl` is where all of the client response functions are located. Aside from the trivial functionality of sending response data back to the client by writing the data to the socket stream, messaging.erl also includes a nifty feature to handle cases where the recipient sockets may be closed. As a chat messaging app, this functionality is critical - we cannot expect users to always have a socket connection to the server at all times, can we? Unlike a game application, where activities happen in real time, we need to consider the case when users are simply offline from the application. In other words, we need to help users store their messages until they come back online.

Using the `gen_tcp` module's `send()` function and its return value, we can determine if the TCP packet was sent successfully. In the event that the active socket associated with a particular user has been closed for any reason at all (poor network connectivity, user has quit the app, etc.), we append the message to the user's message backlog, and store it in Mnesia (in the `aurora_message_backlog` table). The below code snippet shows the exact `send_message` function used to accomplish this:
Line 42 to 49, `messaging.erl`

```
send_message(Socket, PhoneNumber, Message) ->
    Status = gen_tcp:send(Socket, Message),
    case Status of
        ok -> ok;
        _  ->
            controller:append_backlog(PhoneNumber, Message),
            error
    end.
```

Then, when a client first establishes a socket connection, and upon any subsequent request, the client's active socket object is updated in Mnesia and the `send_backlog` function is called with the latest active socket object, flushing all backlog messages that

might have accumulated in the correct order to the client. Therefore, each client request serves double duty as a heartbeat message to the server, informing Aurora that "hey, I'm still alive and this is my socket object as of this moment. If I have any backlog messages, can you send them to me?"

Line 117, 212 to 213, `chat_server.erl`

```
.... after successful authorization in connected_loop

cast_update_socket(ParsedJson, Socket),

.... pattern match to correct API endpoint

cast_update_socket(ParsedJson, Socket) ->
    gen_server:cast(controller, {update_socket, ParsedJson, Socket}).
```

Line 139 to 143, `controller.erl`

```
handle_cast({update_socket, ParsedJson, SocketToUpdate}, State) ->

    update_user(socket, ParsedJson, SocketToUpdate),
    send_backlog(ParsedJson, SocketToUpdate),
    {noreply, State};
```

And the actual sending of the backlog (notice that if in the midst of sending the backlog messages the socket is disconnected, we add the rest of the unsent messages back into the backlog):

Line 777 to 816, `controller.erl`

```
send_backlog(ParsedJson, Socket) ->

    PhoneNumber = maps:get(from_phone_number, ParsedJson),
    io:format("Sending backlog messages..~n",[]),
    LeftoverMessages = [],

    case find_backlog(PhoneNumber) of

        undefined ->
            io:format("There are no backlog messages.~n",[]),
            no_backlog_messages;

        no_such_user ->
            error;

        [] ->
            io:format("There are no backlog messages.~n",[]),
            no_backlog_messages;
```

```
        Messages ->

            F = fun(Message) ->
                io:format("Backlog message:~n~p~n",[Message]),
                Status = gen_tcp:send(Socket, Message),

                case Status of
                    ok ->
                        chat_message_delivery_receipt(PhoneNumber, Message),
                        great;
                    _  ->
                        lists:append(LeftoverMessages, [Message])
                end

            end,

            lists:foreach(F, Messages),
            update_backlog(PhoneNumber, LeftoverMessages)


    end.
```

Additional Notes on Erlang

Due to the functional nature of Erlang, the process of writing programs lends itself very naturally to abstraction and refactoring. Because Erlang programs have very little state (variables in Erlang are single-assignment only and cannot be changed once they are assigned), functions must be written very atomically and composed layer-by-layer to form more complex functions. Erlang, like most other functional languages like Haskell, also do not have control structures in the traditional sense - that is, data in functions only have 1 direction of flow within a function.

These two features make debugging Erlang relatively straightforward as function output is very predictable - what you put in will *always* be what you get out of it. “If” statements and “case” statements in Erlang, although named as such, actually evaluate to expressions, and do not control flow. For example, in `validation.erl`, we have the `handle_list` function, which helps us detect if a list (of user phone numbers) has been converted to a binary string by `jsx`:

Line 353 to 360, `validation.erl`

```
handle_list(List) ->
    ParsedList = case is_binary(List) of
        true ->
            jsx:decode(List);
        false ->
            List
    end,
    convert_list_items_to_binary(ParsedList).
```

As you can see, the "case" statement will be evaluated to the value of `jsx:decode(List)` if the return value of `is_binary(List)` is `true`, and to the value of `List` if `false`, and then assigned to the variable `ParsedList`. This style of writing programs is starkly different from writing imperative programs. In Java, this would not be possible:

```
int getValue(String name) {
        returnValue = switch (name) { // OMG WHAT ARE YOU DOING
        case "Max":
             1;
             break;
        case "Elle":
             2;
             break;
       default:
             0;
             break;
        }
        return returnValue; // ILLEGAL
}
```

Functional programming also relies heavily on pattern matching, both within functions as well as from function to function. Pattern matching can match on number of variables, or on atoms.

Most, if not all, of the functions in `messaging.erl` are based on pattern matching on number of variables. For example:

Line 39 to 49, `messaging.erl`

```
send_message(Socket, Message) ->
    gen_tcp:send(Socket, Message).

send_message(Socket, PhoneNumber, Message) ->
    Status = gen_tcp:send(Socket, Message),
    case Status of
        ok -> ok;
        _  ->
            controller:append_backlog(PhoneNumber, Message),
            error
    end.
```

`send_message`, when called with only 2 variables, will invoke the version without support for backlogging, whereas when called with 3 variables, will include backlogging support.

An example of pattern matching against atoms can be found in the Mnesia API in `controller.erl`.

Line 1073 to 1087, `controller.erl`

```
update_chatroom(change_room_users, ChatRoomID, Users) ->
    F = fun() ->
        [ExistingRoom] = mnesia:wread({aurora_chatrooms, ChatRoomID}),
        UpdatedRoom = ExistingRoom#aurora_chatrooms{room_users = Users},
        mnesia:write(UpdatedRoom)
    end,
    mnesia:activity(transaction, F);

update_chatroom(change_admin, ChatRoomID, NewAdmin) ->
    F = fun() ->
        [ExistingRoom] = mnesia:wread({aurora_chatrooms, ChatRoomID}),
        UpdatedRoom = ExistingRoom#aurora_chatrooms{admin_user = NewAdmin},
        mnesia:write(UpdatedRoom)
    end,
    mnesia:activity(transaction, F).
```

We have two different `update_chatroom` functions. Thus, when invoking this function, we can specify which version to use by the use of atoms. If we include the `change_room_users` atom as the first parameter to the function, then the third parameter will be treated as the list of users to update the chatroom to, whereas if we include the `change_admin` atom, then the pattern will match against the version of `update_chatroom` that treats the third parameter as the user which is going to be the new administrator of the room.

This style of pattern matching is quite like the practice of function overloading in certain imperative languages like C++, but is much more intuitive to write, and extremely expressive, especially considering that Erlang is dynamically typed (and therefore has no concept of function headers based on parameter type or order).

Deploying Aurora

Aurora is managed and deployed with Git on a small Digital Ocean droplet with 1 CPU core and 512MB of RAM running Ubuntu 14.04 x64 (public IP address: `128.199.73.51`, Aurora listens on port 8091). The Erlang runtime is R17 (required for maps support). All commits are pushed to Github and pulled from the Aurora repository on the remote server. Compilation of the source files is automated with an Erlang Makefile (called an `Emakefile`) which dictates the location of source files and the output location of compiled `.beam` files. The Mnesia database lives locally in the same directory as Aurora.

System Testing

Activities

LoginActivityCog			
	Description	**Expected Result**	**Test Result**
1	To test if users are only allowed to log in with a Singapore mobile number. (8 digits starting with 8 or 9)	If correct, MainActivity is loaded. If wrong, Toast with error message is shown	Pass
MainActivity			
	Description	**Expected Result**	**Test Result**
1	To test if user logout properly.	After logout, user is expected to go through the login process	Pass
2	To test if corresponding messages are loaded when chatroom is selected.	MessagingActivity is loaded with the corresponding messages ordered latest from the bottom.	Pass

3	To test if chatroom expires on stipulated time.	If user is still in chatroom, user is unable to send messages. Else users are notified when chatroom expires.	Pass
ContactsActivity			
	Description	**Expected Result**	**Test Result**
1	To test if contacts displayed are registered users	Only registered users will be displayed in the activity	Pass
2	To test if chatroom loaded corresponds to contact selected	MessagingActivity is loaded. If chatroom exists, loads chatroom with messages else, opens request server for a chatroom ID	Pass
MessagingActivity			
	Description	**Expected Result**	**Test Result**
1	To test if the correct menu bar is loaded	If chatroom is a group, menu loaded has Events and New Events option.	Pass
2	To test if sending a message to other users is successful	Message for sender will be right aligned in the activity. Message for receiver will be left aligned in the activity.	Pass
3	To test if message tagging is correct	Messages with 'Important' tag will be displayed with a red circle. Messages with 'Nonsense' tag will be displayed with a yellow circle	Pass
4	To test if filter for messages works properly	Only messaged with filter tag are displayed. More than one filter is allowed	Pass
5	To test if the search function works properly	When character/s is keyed into the search bar, messages with the same sequence for the search term will be highlighted.	Pass
NewGroupActivity			
	Description	**Expected Result**	**Test Result**
1	To test if group name cannot exceed 25 characters	After 25 characters, textfield will stop add characters. Emoticons take up 2 characters	Pass

2	To test if expiry date set is the same as expiry date keyed	If timed group is not selected, expiry is 0. Else, expiry is based on user input	Pass
AddContactsToActivity			
	Description	**Expected Result**	**Test Result**
1	To test if contacts are added to the group	Only checked contacts will be added to the new group	Pass
EventsActivity			
	Description	**Expected Result**	**Test Result**
1	To test if user cannot select event date before current date	User is only able to select date from day onwards. Past days are forbidden	Pass
2	To test if event name and date time is stored and sent to others	User receives a notification of event in the MessagingActivity with correct title and date time	Pass
EventInfoActivity			
	Description	**Expected Result**	**Test Result**
1	To test if the information about the event is shown properly	Onclick, users who have voted for the event will be displayed in a pop up menu	Pass
CreateNoteActivity			
	Description	**Expected Result**	**Test Result**
1	To test if note is created with user specified title and body	Note is sent to users with specified title and body	Pass
EditNoteActivity			
	Description	**Expected Result**	**Test Result**
1	To test if body of note is displayed and editable upon save	Upon save, note body is replaced with new note budy. Other users will be updated.	Pass
NoteListActivity			
	Description	**Expected Result**	**Test Result**

1	To test if notes displayed belongs to chatroom	Only notes that belong to chatroom is displayed	Pass
ProfileActivity			
	Description	**Expected Result**	**Test Result**
1	To test if username is changed on users confirmation	On save, user is able to change username which is sent to server	Pass

Services

NetworkService			
	Description	**Expected Result**	**Test Result**
1	Able to send JSON object	Server replies with status=1	Pass
2	Able to receive JSON object	Received message logged	Pass
3	Able to authenticate with server	Server replies with type=AUTH, status=1	Pass
4	Able to send and receive in background	Same as cases 1 and 2 but with app in background	Pass
5	Able to receive JSON strings from activities and send to server	JSON message logged, Server replies with status=1	Pass
ManagerService			
	Description	**Expected Result**	**Test Result**
1	Able to receive JSON string	Message logged	Pass
2	Able to build notification	Notification matches format depending on unread messages from chats: 1 Chat: <Chat Name/Sender Name> <Unread message> <Unread message>	Pass

		N chats: Messages from <N> chats <Unread message> <Unread message>	
3	Text message handled appropriately	Database contains text message information, insertion is executed without throwing an exception, relevant notification is built	Pass
4	Group invitation handled appropriately	Database contains new group information, insertion is executed without throwing an exception	Pass
5	Single chat invitation handled appropriately	Database contains new chat information, contact is updated with corresponding chatID, insertion is executed without throwing an exception	Pass
6	Able to import chatrooms	Database contains chatroom information matching Aurora's, insertion is executed without throwing an exception	Pass
7	Able to import contacts	Database contains contact information matching Aurora's, insertion is executed without throwing an exception	Pass
8	Able to import notes	Database contains note information matching Aurora's, insertion is executed without throwing an exception	Pass
9	Able to import events	Database contains event information matching Aurora's, insertion is executed without throwing an exception	Pass
10	Sends UPDATE_UI broadcast after each message	Relevant activities receive UPDATE_UI broadcast	Pass
MessageDbAdapter			
	Description	**Expected Result**	**Test Result**
1	Text message stored correctly	storeMessage() executes without throwing an exception, database contains new message	Pass

2	New single chat stored correctly	createSingleChat() executes without throwing an exception, database contains new single chat, contact is updated with chatID	Pass
3	New group chat stored correctly	createGroupChat() executes without throwing an exception, database contains new group chat	Pass
4	Chatrooms imported correctly	importChatrooms() executes without throwing an exception, database chats table matches chatrooms specified in the message	Pass
5	Contacts imported correctly	importContacts() executes without throwing an exception, database contacts table matches contacts specified in the message, chatID column not null if there is an existing personal chatroom with the user	Pass
6	Notes imported correctly	importNotes() executes without throwing an exception, database notes table matches notes specified in the message	Pass
7	New note stored correctly	createNote() executes without throwing an exception, database contains new note	Pass
8	Able to retrieve all messages for a given chatID	getChatMessages() executes without throwing an exception, cursor contains details of all messages for given chatID	Pass
9	Able to retrieve the users for a given chatID	getUserForGroup() executes without throwing an exception, cursor contains userIDs for given chatID	Pass
10	Able to retrieve all chats	getChats() executes without throwing an exception, cursor contains details of all chats	Pass
11	Able to retrieve all undeleted chats	getUndeletedChats() executes without throwing an exception, cursor contains details of all chats flagged as not deleted	Pass
12	Able to retrieve all contacts	getContacts() executes without throwing an exception, cursor contains details of all contacts	Pass

13	Able to retrieve contact for a given phone number	getContact() executes without throwing an exception, cursor contains contact details corresponding to given phone number	Pass
14	Able to delete group chat	deleteGroupChat() executes without throwing an exception, database has chat flagged as deleted	Pass
15	Able to delete expired group chats	deleteExpiredChats() returns an array of deleted chatroom IDs without throwing an exception, database has all chats with nonzero expiry less than the current time flagged as deleted, if chats are deleted a notification containing all the deleted chats is built	Pass
16	Able to retrieve chatroom name for a given chatroom ID	getChatroomName() returns the corresponding chatroom name if the chat corresponds to a group chat, otherwise it returns the username corresponding to the other user of the personal chat without throwing an exception	Pass when contacts table is updated appropriately with chatID
17	Able to retrieve chatroom ID for a given chatroom name	getChatroomID() returns a corresponding chatroom ID without throwing an exception	Pass, but possible unintended behaviour if multiple chats share the same name
18	Able to retrieve whether a chatroom is a group for a given chatroom ID	isGroup() returns boolean value of whether the chat is a group chat without throwing an exception	Pass
19	Able to retrieve whether a chatroom is deleted for a given chatroom ID	isChatDeleted() returns boolean value of whether the chat is deleted without throwing an exception	Pass
20	Able to retrieve the personal chatroom ID corresponding to a given user ID	getChatIDForUser() returns the chatID of the corresponding chat without throwing an exception	Pass
21	Able to mark all messages in a given chat as read	clearRead() flags all messages for a given chatroom ID chat as read without throwing an exception	Pass

22	Able to retrieve the number of unread messages of a chatroom	getUnreadCount() returns the number of unread messages of the chat corresponding to the given chatroom ID without throwing an exception	Pass
23	Able to retrieve the username of a personal chatroom	getUsername() returns the username corresponding to the given chatroom ID without throwing an exception	Pass
24	Able to retrieve the username corresponding to a phone number	getUsernameFromNumber() returns the username corresponding to the given phone number without throwing an exception	Pass
25	Able to retrieve the latest message of a chatroom	getLatestMessage() returns the message ID corresponding to the last message of the chatroom corresponding to the given chatroom ID without throwing an exception	Pass
26	Able to retrieve the ID, title and body of notes corresponding to a chatroom	getNoteIDTitleBody() executes without throwing an exception, cursor contains details of all notes corresponding to the chatroom with the given chatroom ID	Pass
27	Able to retrieve the tags of the messages in a given chat	getTagsForChat() returns a list of used tags in a chat without throwing an exception	Pass
28	Events imported correctly	importEvents() executes without throwing an exception, database contacts table matches events specified in the message	Pass
29	New event stored correctly	insertEvent() executes without throwing an exception, database contains new event	Pass
30	Able to retrieve the event name for a given event ID	getEventNameByID() returns the event name corresponding to the given event ID without throwing an exception	Pass
31	Able to retrieve the event ID for a given event ID	getEventIDByName() returns an event ID corresponding to the given event name without throwing an exception	Pass, but possible unintended behaviour if multiple events share the same name

32	Able to retrieve the events corresponding to a given chat	getEventsForChat() executes without throwing an exception, cursor contains the events of the chat correponding to the given chatroom ID	Pass

Server

Aurora contains a self-running test suite written in Erlang that simulates all of the supported functionality, and more importantly, all of the expected errors to be raised. The test suite consists of 3 “clients” that connects using TCP to a running instance of Aurora, and communicates with it with JSON (using, of course, `jsx` to encode our messages). Aurora’s outputs are then captured and asserted against the expected output. Part of the test suite can be found in Appendix A.

Extensions

The communication protocol between the GTFS client and Aurora is not secure - meaning that we are currently still susceptible to eavesdropping. The most obvious solution would be to switch to using TLS sockets. As far as we can tell, changing the socket type will be fairly trivial for both client and server, and we will definitely be looking into implementing that in the near future for better security. Performance benchmarks will also be very interesting to implement to spot bottlenecks, especially in the context of concurrent applications.

Conclusion

For brevity’s sake, we have not discussed GTFS and Aurora in the fullest detail in this report. However, we hope that it will give you an insight into the complexities involved in building an API client-server system from scratch. As a service, we believe GTFS provides a solution for many problems plaguing most mainstream chat services and coupled with Aurora’s robust support, GTFS can be a serious consideration for those looking for a better chatting experience.

Appendix

Table 1: A list of all API endpoints supported by Aurora.

Type	Use	Expected request from client to server	Expected response from server to client

AUTH	Authenticating	{
		“username”: String,
		“session_token”: String,
		“from_phone_number”: String,
		“type”: “AUTH”
		}
			{
			“status”: Integer,
			“type”: “AUTH”
			}

TEXT	Client-to-client / client-to-group message	{
		“from_phone_number”: String,
		“chatroom_id”: String,
		“session_token”: String,
		“message”: String,
		“tags”: String,
		“timestamp”: String,
		“type”: “TEXT”
		}
			To: sending client
			{
			“type”: “TEXT”,
			“status”: Integer,
			“message”: {
			“message_id”: String,
			“to_phone_number”: String
			}
			}
			
			To: sending and receiving client
			{
			“message_id”: String,
			“from_phone_number”: String,
			“chatroom_id”: String,
			“Message”: String,
			“tags”: String,
			“timestamp”: String,
			“type”: “TEXT_RECEIVED”
			}

GET USERS	Get all users who are friends of the user	{
		“from_phone_number”: String,
		“session_token”: String,
		“users”: [Array Of Phone Numbers],
		“type”: “GET_USERS”
		}
			{
			“users”: [
			{
			“username”: String,
			“phone_number”: String
			}
			],
			“type”: “GET_USERS”
			}

CREATE SINGLE CHAT ROOM	Create a single chat room	{
		“from_phone_number”: String,
		“to_phone_number”: String,
		“session_token”: String,
		“expiry”: EpochTime,
		“type”: “CREATE_SINGLE_ROOM”
		}
			To: sending client
			{
			“status” : Integer,
			“chatroom_id”: String,
			“type”: “CREATE_SINGLE_ROOM”
			}

			To: receiving client
			{
			“chatroom_id”: String,
			“users”: [Array of Phone Numbers],
			“expiry”: EpochTime,
			“group”: false,
			“type”: “SINGLE_ROOM_INVITATION”
			}

CREATE CHAT ROOM	Create a chat room	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_name”: String,
		“users”: [Array of Phone Numbers (String)],
		“expiry”: EpochTime,
		“type”: “CREATE_ROOM”
		}
			To: sending client
			
			{
			“status”: Integer,
			“chatroom_id”: String,
			“type”: “CREATE_ROOM”
			}
			
			To: receiving client
			{
			“chatroom_id”: String,
			“chatroom_name”: String,
			“users”: [Array of Phone Numbers],
			“expiry”: EpochTime,
			“type”: “ROOM_INVITATION”
			}

INVITE USER TO CHAT ROOM	Inviting a user to a chat room (only for group chats)	{
		“from_phone_number”: String,
		“to_phone_number”: String,
		“chatroom_id”: String,
		“session_token”: String,
		“type”: “ROOM_INVITATION”
		}
			To: sending client
			
			{
			“status”: Integer,
			“type”: “ROOM_INVITATION”
			}
			
			To: receiving client
			{
			“chatroom_id”: String,
			“chatroom_name”: String,
			“users”: [Array of Phone Numbers],
			“expiry”: EpochTime,
			“type”: “ROOM_INVITATION”
			}

TRANSFER ADMIN RIGHTS	Transfer admin rights to another person in the chat room (only for group chats)	{
		“from_phone_number”: String,
		“to_phone_number”: String,
		“chatroom_id”: String,
		“session_token”: String,
		“type”: “TRANSFER_ADMIN”
		}
			To: sending client
			{
			“status”: Integer,
			“chatroom_id”: String,
			“type”: “TRANSFER_ADMIN”
			}
			
			To: receiving client
			{
			“chatroom_id”: String,
			“type:”: “NEW_ADMIN”
			}

LEAVE A CHAT ROOM	Leave a chat room (only for group chats? tentatively)	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: String,
		“type”: “LEAVE_ROOM”
		}
			To: sending client
			
			{
			“status”: Integer,
			“chatroom_id”: String,
			“type”: “LEAVE_ROOM”
			}

GET ALL CHAT ROOMS OF A USER	Fetch all chat rooms belonging to a user	{
		“from_phone_number”: String,
		“session_token”: String,
		“type”: “GET_ROOMS”
		}
			{
			“chatrooms”: [
			{
			“chatroom_id”: String,
			“chatroom_name”: String,
			“users”: [Array of Phone Numbers],
			“expiry”: EpochTime
			},
			{
			“chatroom_id”: String,
			“chatroom_name”: String,
			“users”: [Array of Phone Numbers],
			“expiry”: EpochTime
			}
			],
			“TYPE”: “GET_ROOMS”
			}

GET_NOTES	Fetch all notes belonging to a room	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: String,
		“type”: “GET_NOTES”
		}
			{
			“chatroom_id”: String,
			“notes”: [
			{
			“note_id”: String,
			“note_title”: String,
			“note_text”: Text,
			“note_creator”: String,
			}
			],
			“type”: “GET_NOTES”
			
			}

CREATE_NOTE	Create a note	{
		“from_phone_number”: String,
		“session_token”: String,
		“note_title”: String,
		“note_text”: Text,
		“chatroom_id”: String,
		“type”: “CREATE_NOTE”
		}
			{
			“status”: Integer,
			“note_id”: Integer,
			“type”: “CREATE_NOTE”
			}

EDIT_NOTE	Edit a note	{
		“from_phone_number”: String,
		“session_token”: String,
		“note_id”: String,
		“note_text”: String,
		“chatroom_id”: String,
		“type”: “EDIT_NOTE”
		}
			{
			“status”: Integer
			}

DELETE_NOTE	Delete a note	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: String,
		“note_id”: String,
		“type”: “DELETE_NOTE”
		}
			{
			“status”: Integer
			}

ADD_EVENT	Create an event attached to a room	{
		“from_phone_number”: String,
		“session_token”: String,
		“event_name”: String,
		“event_datetime”: String,
		“chatroom_id”: String,
		“type”: “CREATE_EVENT”
		}
			{
			“event_id”: Integer
			}

GET_EVENTS	Get all events in a room	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: Integer,
		“type”: “GET_EVENTS”
		}
			{
			“events”:[
			{
			“votes”: [Array of Phone Numbers],
			“event_id”: String,
			“event_name”: String,
			“event_datetime”: String,
			}],
			“chatroom_id”: String,
			“type”: “GET_EVENTS”
			}

EVENT_VOTE	Vote for an event	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: String,
		“event_id”: Integer,
		“type”: “EVENT_VOTE”
		}
			To sending client:
			{
			“status”: 1,
			“message”: {
			“event_id”: Integer
			},
			“type”: “EVENT_VOTE”
			}
			
			To: sending and receiving client
			{
			“from_phone_number”: String,
			“event_id”: String,
			“vote_timestamp”: String,
			“chatroom_id”: String,
			“type”: “EVENT_VOTE_RECEIVED”
			}

EVENT_UNVOTE	Unvote for an event	{
		“from_phone_number”: String,
		“session_token”: String,
		“chatroom_id”: String,
		“event_id”: Integer,
		“type”: “EVENT_UNVOTE”
		}
			To sending client:
			{
			“status”: 1,
			“message”: {
			“event_id”: Integer
			},
			“type”: “EVENT_UNVOTE”
			}
			
			To: sending and receiving client
			{
			“from_phone_number”: String,
			“event_id”: String,
			“vote_timestamp”: String,
			“chatroom_id”: String,
			“type”: “EVENT_UNVOTE_RECEIVED”
			}

Table 2: Aurora’s Mnesia database schema

```
-record(aurora_users, {phone_number, username, session_token, rooms, current_ip,
active_socket}).
-record(aurora_chatrooms, {chatroom_id, chatroom_name, room_users, admin_user, expiry,
group}).
-record(aurora_message_backlog, {phone_number, messages}).
-record(aurora_chat_messages, {chat_message_id, chatroom_id, from_phone_number, timestamp,
message, tags}).
-record(aurora_events, {event_id, chatroom_id, event_name, event_datetime, votes}).
-record(aurora_notes, {note_id, chatroom_id, note_title, note_text, from_phone_number}).
```

Table 3: An Excerpt of Aurora’s test suite

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% AURORA EUNIT TEST SUITE
%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%
% Test 0:
% First things first, let's establish 3 clients and authenticate them
%%%%%%%%

{ok, SocketA} = gen_tcp:connect({127,0,0,1}, 8091, []).
gen_tcp:send(SocketA, jsx:encode(#{<<"username">> => <<"clientA">>, <<"session_token">> =>
<<"1234">>, <<"from_phone_number">> => <<"12341234">>, <<"type">> => <<"AUTH">>,
<<"timestamp">> => <<"2:24:51">>})).

% assert("Status sent: 1 with type: <<"AUTH">>")
```

```
{ok, SocketB} = gen_tcp:connect({127,0,0,1}, 8091, []).
gen_tcp:send(SocketB, jsx:encode(#{<<"username">> => <<"clientB">>, <<"session_token">> =>
<<"2345">>, <<"from_phone_number">> => <<"23452345">>, <<"type">> => <<"AUTH">>,
<<"timestamp">> => <<"2:24:51">>})).

% assert("Status sent: 1 with type: <<"AUTH">>")

{ok, SocketC} = gen_tcp:connect({127,0,0,1}, 8091, []).
gen_tcp:send(SocketC, jsx:encode(#{<<"username">> => <<"clientC">>, <<"session_token">> =>
<<"3456">>, <<"from_phone_number">> => <<"34563456">>, <<"type">> => <<"AUTH">>,
<<"timestamp">> => <<"2:24:51">>})).

% assert("Status sent: 1 with type: <<"AUTH">>")

%%%%%%%%
% Test 1:
% After authenticating, users are unable to reauthenticate on the same socket connection
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"username">> => <<"clientA">>, <<"session_token">> =>
<<"1234">>, <<"from_phone_number">> => <<"12341234">>, <<"type">> => <<"AUTH">>,
<<"timestamp">> => <<"2:24:51">>})).

% assert("Status sent: 6")

%%%%%%%%
% Test 2:
% Test that a single chat room can be created
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"from_phone_number">> => <<"12341234">>,
<<"session_token">> => <<"1234">>, <<"to_phone_number">> => <<"23452345">>, <<"type">> =>
<<"CREATE_SINGLE_ROOM">>, <<"expiry">> => <<"123123123123">>})).

%
assert(""{\"chatroom_id\":\"1430139888260476\",\"expiry\":123123123123,\"group\":false,\"typ
e\":\"SINGLE_ROOM_INVITATION\",\"users\":[\"12341234\",\"23452345\"]}"}")

%%%%%%%%
% Test 3:
% Test that users cannot be invited into a single chat room
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"session_token">> => <<"1234">>,
<<"from_phone_number">> => <<"12341234">>,  <<"to_phone_number">> => <<"34563456">>,
<<"chatroom_id">> => <<"1430139888260476">>, <<"type">> => <<"ROOM_INVITATION">>,
<<"timestamp">> => <<"2:24:51">>})).

% assert("Status sent: 5 with type: <<"ROOM_INVITATION">> and message <<"Cannot invite
people into single chatroom">>")

%%%%%%%%
% Test 4:
% Test sending a text message in a single chat room
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"session_token">> => <<"1234">>,
<<"from_phone_number">> => <<"12341234">>,  <<"message">> => <<"Hi!">>, <<"chatroom_id">> =>
<<"1430139888260476">>, <<"type">> => <<"TEXT">>, <<"timestamp">> => <<"12:14">>, <<"tags">>
=> <<"Important">>})).
```

```
%
assert(""{\"chatroom_id\":\"1430139888260476\",\"from_phone_number\":\"12341234\",\"message\
":\"Hi!\",\"message_id\":\"1430141243505000\",\"tags\":\"Important\",\"timestamp\":\"12:14\"
,\"type\":\"TEXT_RECEIVED\"}"}")

%%%%%%%%
% Test 5:
% Test getting users
% This is actually two tests in one, because we include an invalid user as well
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"from_phone_number">> => <<"12341234">>,
<<"session_token">> => <<"1234">>, <<"type">> => <<"GET_USERS">>, <<"users">> =>
[<<"12341234">>, <<"23452345">>, <<"invalidnumber">>]})).

%
assert({\"type\":\"GET_USERS\",\"users\":[{\"phone_number\":\"12341234\",\"username\":\"clie
ntA\"},{\"phone_number\":\"23452345\",\"username\":\"clientB\"}]}"}")

%%%%%%%%
% Test 6:
% Test creating a group chat room with a backdated expiry time
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"from_phone_number">> => <<"12341234">>,
<<"session_token">> => <<"1234">>, <<"chatroom_name">> => <<"Good Chat">>, <<"users">> =>
[<<"12341234">>, <<"23452345">>], <<"type">> => <<"CREATE_ROOM">>, <<"expiry">> =>
<<"1000">>})).

% assert(""{\"chatroom_id\":\"1430141096770111\",\"chatroom_name\":\"Good
Chat\",\"expiry\":1000,\"group\":true,\"type\":\"ROOM_INVITATION\",\"users\":[\"12341234\",\
"23452345\",\"34563456\"]}"}")

%%%%%%%%
% Test 7:
% Test sending a message to an expired room
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"session_token">> => <<"1234">>,
<<"from_phone_number">> => <<"12341234">>,  <<"message">> => <<"Hi!">>, <<"chatroom_id">> =>
<<"1430141096770111">>, <<"type">> => <<"TEXT">>, <<"timestamp">> => <<"12:14">>, <<"tags">>
=> <<"Important">>})).

% assert("Status sent: 5 with type: <<"TEXT">> and message <<"The chatroom has
expired.">>"")

%%%%%%%%
% Test 8:
% Test creating a normal group chat room
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"from_phone_number">> => <<"12341234">>,
<<"session_token">> => <<"1234">>, <<"chatroom_name">> => <<"Good Chat">>, <<"users">> =>
[<<"12341234">>, <<"23452345">>], <<"type">> => <<"CREATE_ROOM">>, <<"expiry">> =>
<<"1440141096770111">>})).

% assert(""{\"chatroom_id\":\"1430141186944041\",\"chatroom_name\":\"Good
Chat\",\"expiry\":1440141096770111,\"group\":true,\"type\":\"ROOM_INVITATION\",\"users\":[\"
12341234\",\"23452345\",\"34563456\"]}"}")
```

```
%%%%%%%%
% Test 9:
% Test sending a text message to the group chat room
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"session_token">> => <<"1234">>,
<<"from_phone_number">> => <<"12341234">>,  <<"message">> => <<"Hi!">>, <<"chatroom_id">> =>
<<"1430141186944041">>, <<"type">> => <<"TEXT">>, <<"timestamp">> => <<"12:14">>, <<"tags">>
=> <<"Important">>})).

%
assert(""{\"chatroom_id\":\"1430141186944041\",\"from_phone_number\":\"12341234\",\"message\
":\"Hi!\",\"message_id\":\"1430141243507249\",\"tags\":\"Important\",\"timestamp\":\"12:14\"
,\"type\":\"TEXT_RECEIVED\"}"}")

%%%%%%%%
% Test 10:
% Test inviting a user into a group chat room when user is not admin
%%%%%%%%

gen_tcp:send(SocketB, jsx:encode(#{<<"session_token">> => <<"2345">>,
<<"from_phone_number">> => <<"23452345">>,  <<"to_phone_number">> => <<"34563456">>,
<<"chatroom_id">> => <<"1430141472309891">>, <<"type">> => <<"ROOM_INVITATION">>})).

% assert("Status sent: 8 with type: <<"ROOM_INVITATION">> and message <<"User is not admin
of the room">>")

%%%%%%%%
% Test 11:
% Test inviting a user into a group chat room when user is admin
%%%%%%%%

gen_tcp:send(SocketA, jsx:encode(#{<<"session_token">> => <<"1234">>,
<<"from_phone_number">> => <<"12341234">>,  <<"to_phone_number">> => <<"34563456">>,
<<"chatroom_id">> => <<"1430141472309891">>, <<"type">> => <<"ROOM_INVITATION">>})).

% assert(""{\"message\":\"User invited
successfully\",\"status\":1,\"type\":\"ROOM_INVITATION\"}"}")

--- REST OF TEST SUITE ---
```

Diagram 1: Activity Class Diagram

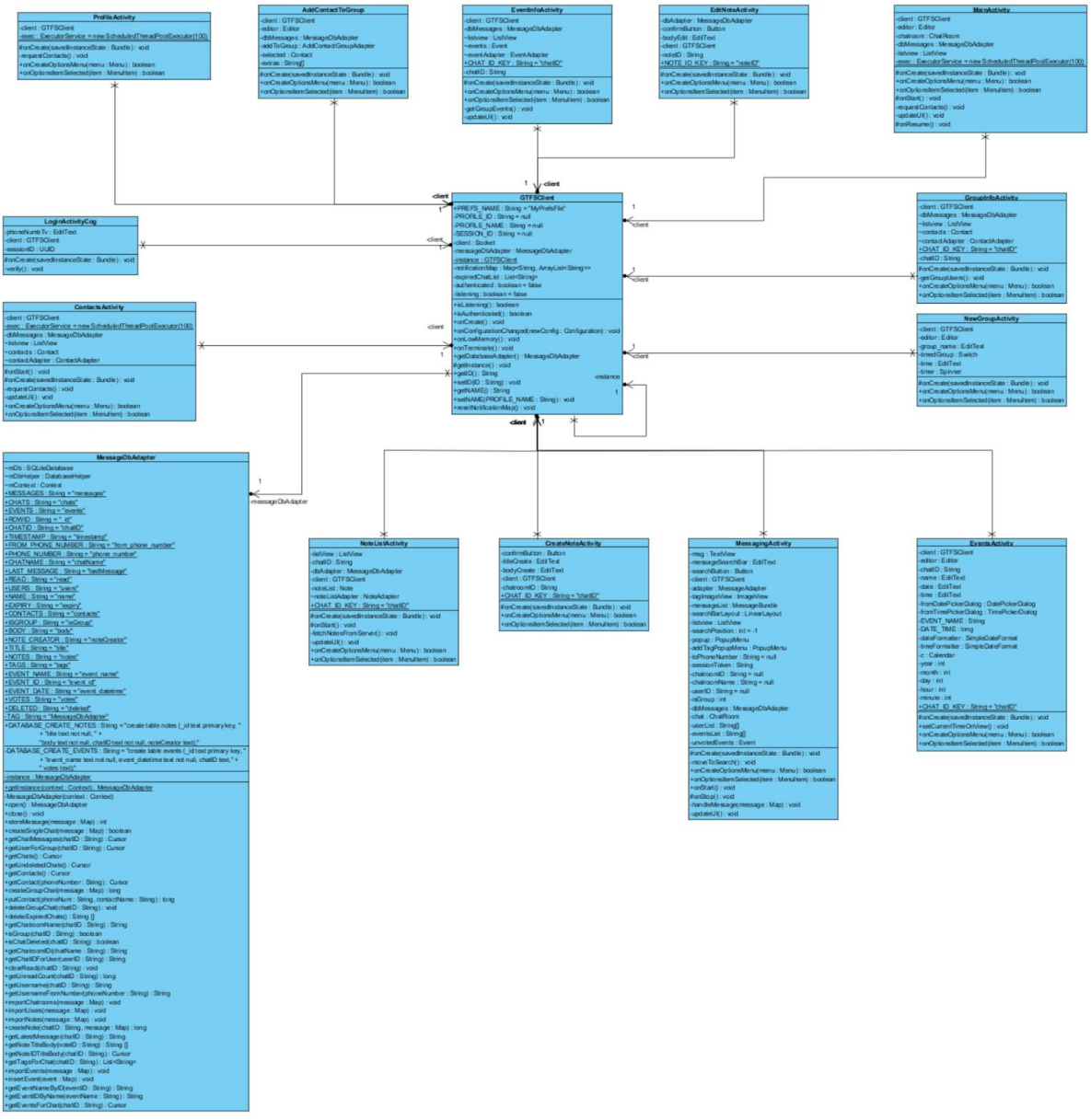

Diagram 2: Object Class Diagram

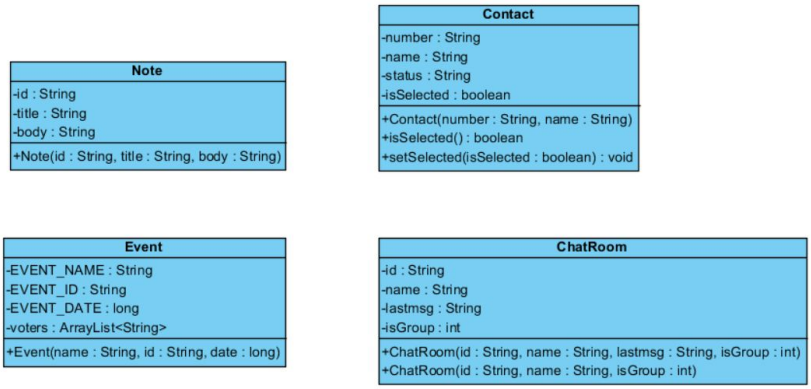

Diagram 3: Adapter Class Diagram

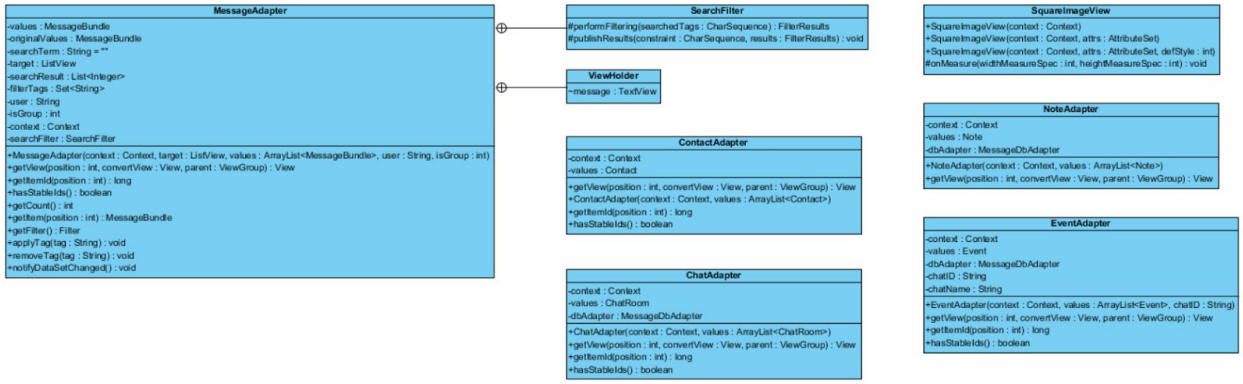

Diagram 4: Message Management Class Diagram

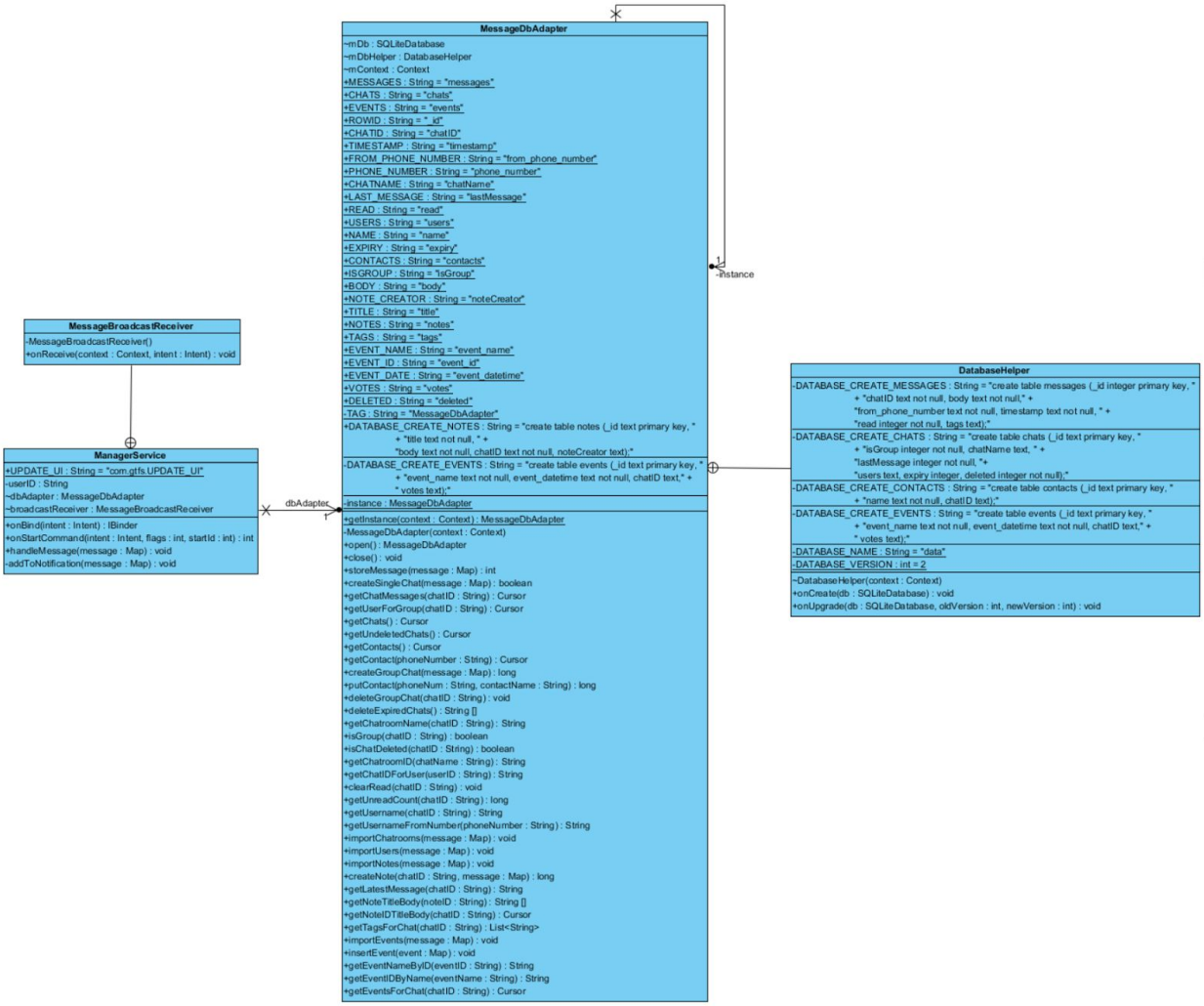

Diagram 5: Server Utilities Class Diagram

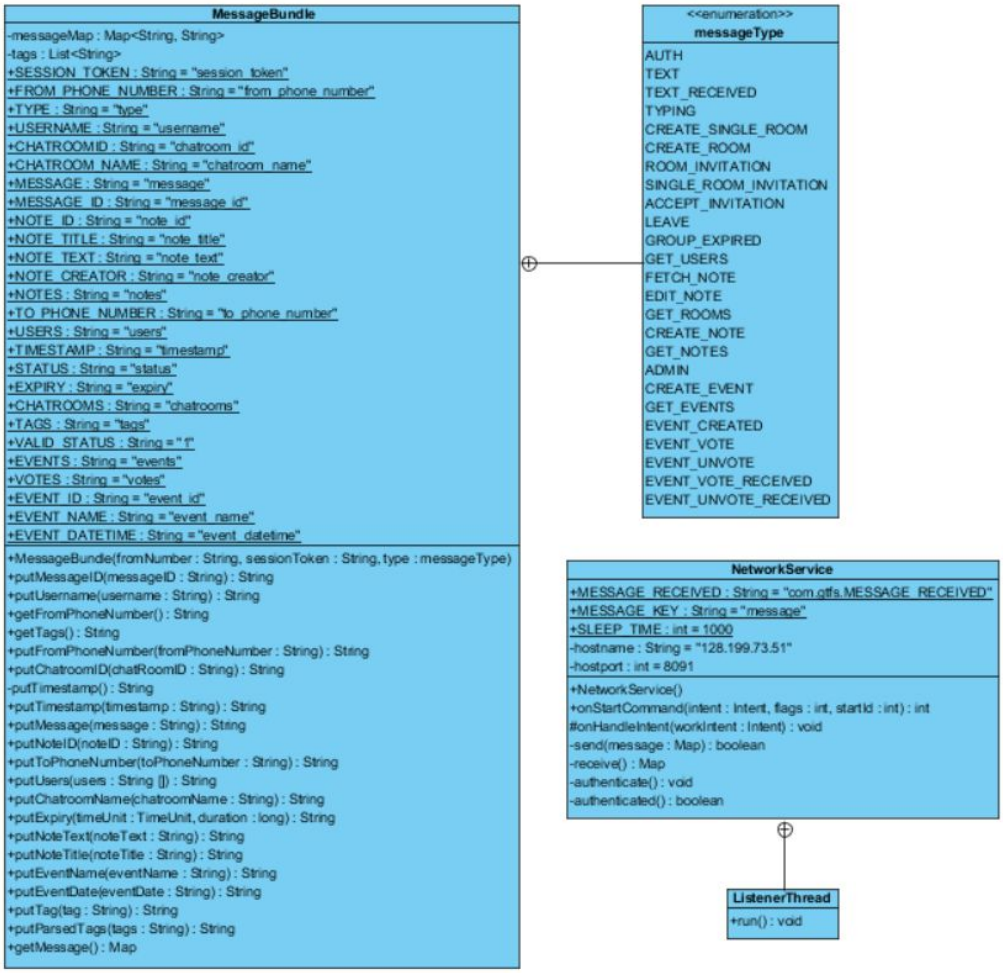

Diagram 6: Client Database Relationship Diagram