Team BusGen

BusTalk

Software Design Document

Michel Folkemark
Johan Iversen
Alexander Kloutschek
Kristoffer Knutsson
Daniel Wassbjer

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

This document serves to explain the architecture and system design of the BusTalk Android application and its server solution fully enough for software development to proceed with understanding on what to be built and how to build it.

1.2 Scope

This Software Design Document is for BusTalk, a chat application used for android to be used on the new electric busses in the city of Gothenburg, to chat with other people travelling on the same bus. The application server to work as an application on it's own, but could very well be integrated to another application. The server solution is open to for connections from any platform that can use JSON to work for cross platform chatting.

1.3 Definitions and Acronyms

- **Activity (Android) -** An activity is a GUI for a certain task. An Android application is usually comprised of a number of different activities.
- **Actionbar(Android) -** A bar that resides at the top of every activity which apart from displaying a title for the activity also can contain items that display information or provide a means of navigation.
- **Intent (Android) -** One of the parameters used to construct an Android activity.
- **JSON -** JavaScript Object Notation, is a lightweight data-interchange format that sends attribute-value pairs.
- **MVC -** Model-View-Controller, is a code pattern that splits the logic away from the presentation (view) to counter errors happening when changes are made in the presentation layer.
- **Service (Android)** - A process running in the background, unrelated to activities (see above) which run in the foreground and are easily destroyed. A service is not normally destroyed unless instructed to.
- **Session -** is a way to represent a conversation between two server endpoints
- **Singleton -** design pattern that ensures that only one instance of a class is created
- **Websocket -** protocol allowing simultaneous two-way conversation over a single TCP connection.

2. BUILD PROCESS

2.1 Android Application

The android application is built by first importing the project into android studio. There gradle takes care of building the project.

After the gradle building is finished, the application is installed by running the application on a connected phone or an emulator.

To run the application on a connected phone; debug mode has to be activated on it first. After activating debug mode you just need to connect the phone to the computer and it will show up as a runnable device after having pressed “run” in Android Studio.

2.2 Server Solution

The server uses Gradle as build tool. JavaEE is also required in order to build the server. For Eclipse there is a Java EE plugin (but also a Java EE version of Eclipse), and for IntelliJ, Java EE is only supported in the Ultimate edition.

Besides building the server, it may also be preferred to run the server through the IDE instead of building a .war file and re-deploying it manually after each build. The following text describes how to run the server within IntelliJ (Ultimate required), which starts the Tomcat server and deploys the .war file automatically when ran:

1. Open the server project (File > Open…) located at <project location>/BusGen/Server/.
2. Make sure Gradle is fully built.
3. Download Tomcat 8 and extract to wherever you see fit.
4. Edit your Run Configurations (Run > Edit Configurations…):
    a. Add a new configuration (Alt + Insert).
    b. Select Tomcat Server > Local.
    c. Configure the Application server:
        i. Add a new Application server (Alt + Insert).
        ii. At Tomcat Home, browse to the location of your Tomcat root folder and select it.
        iii. Click Okay until your back out at the Run/Debug Configurations window.

   d. Go over to the Deployment tab and add an artifact (Alt + Insert). Either of the standard .war and the exploded artifact should work, so select any of these.
   e. Press Okay once you have added the artifact.
5. You should now be able to run the server locally at port 8080. If you need to change ports it is easily done in the Run Configurations.

3. SYSTEM ARCHITECTURE

3.1 Android Application

3.1.1 Architectural Design

The client consists primarily of three different overarching sections:

- The activities: can be seen as the views that present the application to the user.
- The Client class: can be seen as the model of the application
- The ServerCommunicator class: takes care of communication to and from the server

These in turn communicate with each other via the EventBus class.

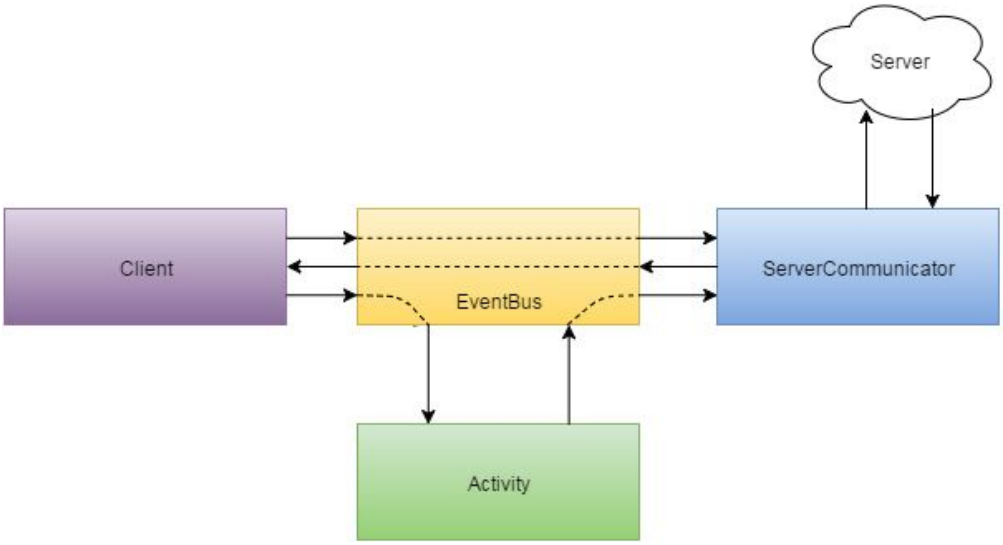


Figure 1: UML for android application

3.1.1.1 Activities

The application has three different activities which are responsible for presenting the application to the user. Each activity is relevant during different stages of the application's lifetime. To complement the activities there are the adapter classes, which are responsible for converting data into suitable view items in the activities.

BindingActivity
Extended by LoginActivity, MainChatActivity and UserActivity. Upon instantiation, binds to MainService and registers to the EventBus.

LoginActivity
This activity is responsible for presenting a login screen to the user, where the user gets to choose a nickname and an interest that is going to be visible to other users

later on in the application. The activity then relays the nickname typed in to the EventBus and depending on whether the nickname is available or not the activity notifies the user accordingly.

This activity is also responsible for handling the initialization of the central activity of the application; the MainChatActivity. After launching the MainChatActivity this activity gets terminated as it is no longer relevant.

MainChatActivity
This is the central activity of the application. It's in this activity the user can see the chat window and is able to communicate with other users connected to the chat. Apart from being responsible for sending/receiving the actual text messages to/from the EventBus it also reacts to messages sent to it via the EventBus containing information such as if a new user has joined/left the chat, if the connection to the bus router has been lost etc.

This activity also has an actionbar containing an item that shows the current number of users present in the chat as well as an item that launches the UserActivity.

MessageAdapter
This adapter converts MsgChatMessage objects into suitable message items that can be displayed in the MainChatActivity. These items consist of the actual text message, the name of the message sender, the date the message was sent as well as a background portraying a chat bubble.

UserActivity
This activity is responsible for presenting a list of the current users present in the chat along with their chosen interests.

UserAdapter
This adapter converts User objects into items that that can be listed in the UserActivity. The items contain the user's name and interest as well as an icon.

3.1.1.2 Client

Client
The Client class holds all the information comprising the user of the application and the chatrooms it is connected to.

Chatroom

A Chatroom has an ID number, a list of Users currently in that Chatroom and a list of messages that has been received since the user joined it.

3.1.1.3 ServerCommunicator

ServerCommunicator
The ServerCommunicator class acts as a mediator between the local client and a remote server. By using the JSONEncoder helper class it converts client side messages to JSON objects which are then sent to the server. Conversely, by using the JSONDecoder helper class, server side messages are parsed for interpretation by the client. Examples of messages from the server are connection status and incoming chat messages. Examples of messages by the client are room joining requests and outgoing chat messages.

3.1.1.4 EventBus

EventBus
The EventBus class is the mediator between all client side packages. An event is sent from either the Client class, any of the Activity classes or the ServerCommunicator class. Events are of the type ToServerEvent, ToClientEvent or ToActivityEvent depening on the chosen receiver. The classes registered to the event bus all implement onEvent(Event event) for handling events containing a message implementing the IServerMessage interface. The type of the message determines the response.

IServerMessage
Interface implemented by all internal messages on the client side. All events sent through EventBus contains a message implementing IServerMessage. Examples of these messages include MsgChatMessage which contains a chat message and MsgUsersInChatRequest which are sent to the server (After being encoded by JSONEncoder) to request a list of the users in a certain Chatroom.

3.1.1.5 Services

MainService
MainService is the service holding permanent instances of the classes Client, EventBus and ConnectionsHandler. After creating instances of these three classes, it registers client to the event bus. Every class extending BindingActivity binds to this on creation.

3.1.1.6 Utilities

JSONEncoder
Class responsible for converting IServerMessages from the client to JSON objects that are parsable by the server.

JSONDecoder
Class responsible for parsing JSON objects from the server to IServerMessages that are recognizable by the client.

ConnectionsHandler
Holder class for ServerCommunicator and PlatformCommunicator. Responsible for checking connection status and updating the information about next bus stop at intervals.

PlatformCommunicator
Mediator between the client and the Electricity platform. Responsible for fetching the data containing information about next bus stop.

ServerCommunicator
Responsible for establishing a connection to, and sending data to and from the server. Makes use of the JSONEncoder and JSONDecoder classes for handling messages.

WifiController
Have responsibilities with controlling if you're physically close enough to an accepted mac-adress, accepted for now being a bus line or a bus stop. If the user is close to one of these, it can retrieve the name of the bus or bus stop.

3.1.2 Technologies Used

JSON objects are used to send and receive information from the server. To gather information regarding busses the ElectriCity API is used.

3.1.3 Decomposition Description

***activity* package**
The *activity* package contains all the activities the application has.

***adapter* package**
The *adapter* package contains the two adapter classes responsible for converting data into the needed list items in the applications activities.

***events* package**
The *events* package contains all the events that are used to encapsulate messages so they can be posted on the EventBus

***model* package**
The logic representation of the application. Stores all vital information and has it ready for representation.

***service* package**
Helper classes for the application that handles various tasks not suited to keep in other parts of the project.

***utils* package**
Static classes keeping constants used for identifying busses and different message types.

3.2.3.2 Dependency Analysis

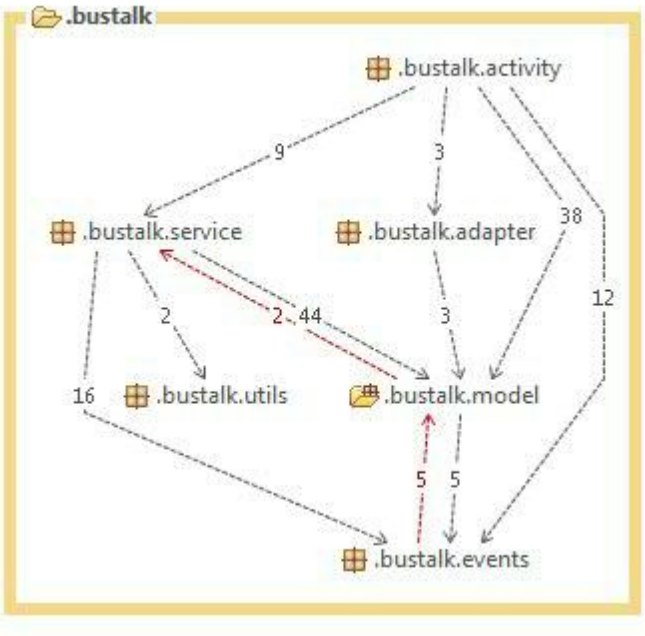


Figure 2: Dependency Analysis Client-Side. Due to the Client class being a singleton class, there are some circular dependencies.

3.1.4 System Operation

The first step in the client part of the application flow is that the user is greeted by the LoginActivity, where the user is prompted to insert his/her desired nickname and interest. The LoginActivity then relays the inserted information to the server via the EventBus as a message addressed to the ServerCommunicator. The ServerCommunicator translates the information into a JSON object which it then sends to the server.

If the nickname has already been chosen by another currently active user, the ServerCommunicator will receive a message from the server with that information. This message is then sent via the EventBus back to the LoginActivity, which in turn asks the user to choose another nickname.

In the more normal case where the nickname is available, the nickname of the Client's User will be set to the one requested. The LoginActivity will then proceed to request a list of all available Chatrooms from the server, and upon receiving that list asks the server to join the first one (the main chat). If receiving a confirmation of the joining (i.e. MsgNewUserInChat, and that user is me), the Client adds the Chatroom in question to its list of Chatrooms (List<Chatroom> chatrooms). Finally, an activity of type MainChatActivity is started, with the joined Chatroom as one of its intents. The no longer relevant LoginActivity is destroyed.

The MainChatActivity is where the different users get to communicate with one another. The sending of a message to other users in the chat is handled by creating a MsgChatMessage of the text the user has typed in. The MsgChatMessage then gets encapsulated by a ToServerEvent which in turn gets posted on the EventBus. After reaching the ServerCommunicator, the MsgChatMessage gets encoded into a JSON object and finally sent to the server.

The MainChatActivity's actionbar contains an item displaying the currenly active users in the chat as well as an item that leads to the UserActivity; where the currently active users nicknames are listed along with their selected interests.

3.1.5 Design Rationale

The "ViewHolder" design pattern was implemented in the adapter classes as there seems to be a consensus in the information gathered that it significantly improves the performance of a ListView by limiting the amount of calls to the method "findViewById".

The event bus design was chosen in favor of having classes of different packages subscribe to one another, or by having direct connections between classes. The reasons for this was to reduce dependencies and make it possible to send more focused requests and messages to all listeners affected.

Instead of having a single ‘message’ class with a type identifier, the choice was made to create separate classes for all IServerMessages. Since the relevant data to include is unique between most messages, using a common class would require the passing of variables not relevant, and maybe even unavailable to the sender, The validation of the message construction would have to be rigorous and the programmer would have to be very well versed in the different types of messages, since there would be no help given just from looking the constructor parameters themselves. So even though having unique classes for every type of message may seem a bit unintuitive at first, it was deemed the most usable solution.

3.2 Server Solution

3.2.1 Architectural Design

The server is in a way using a MVC pattern, even if it's not in the conventional way as the "View"-part haven't been implemented. The lack of need for this "View"-part together with the fact that it may have uses in the future (for example an admin tool) stood for the reason of choosing this pattern despite not implementing the View. This is not the typical case of MVC, but for a server solution that can be seen as a part of the model for the entire project (server and client), this is as close as we get. See figure 3.

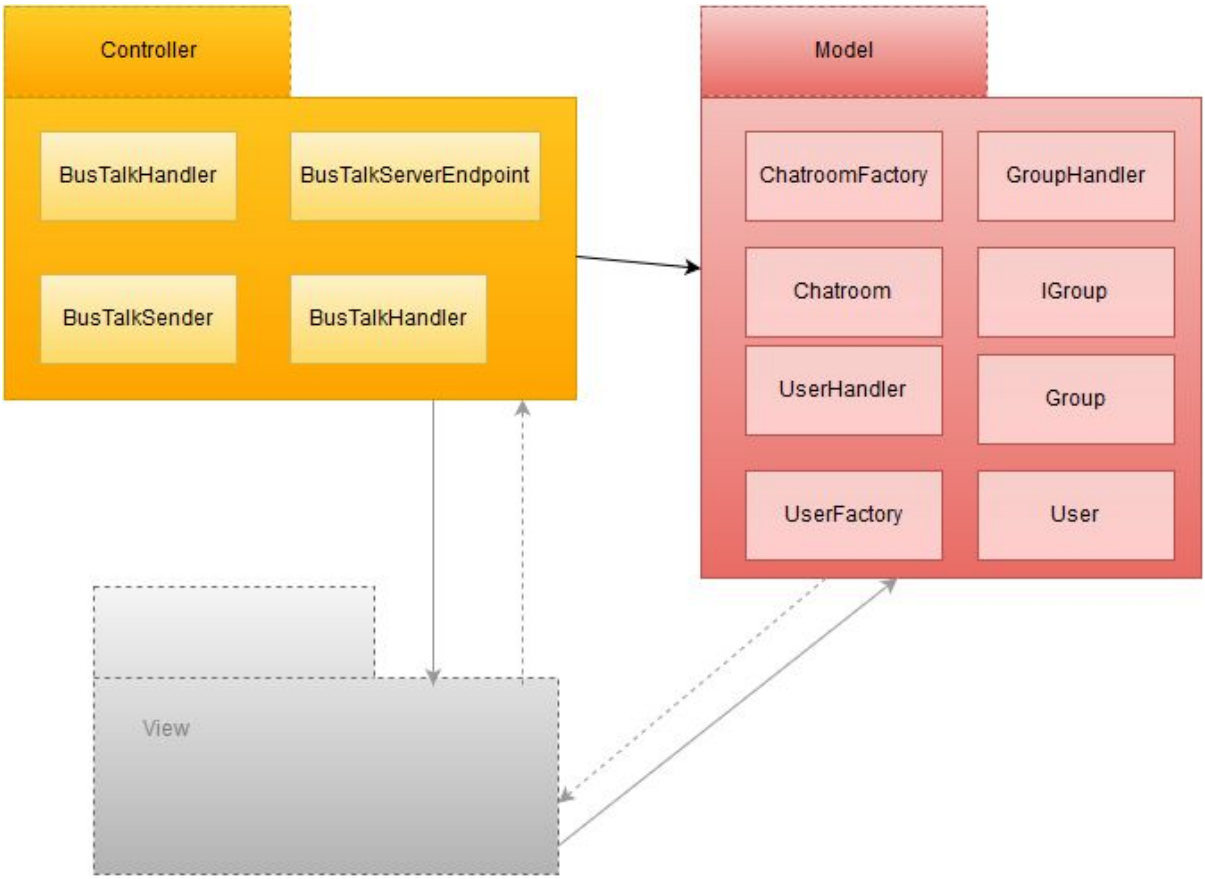


Figure 3: MVC with unimplemented "View"

BusTalkServerEndpoint

This class handles events from a client, such as when a connection is opened, closed, a message is received, etc. When a message is received, the message is forwarded to *BusTalkHandler*, which purpose is to handle all incoming messages. A new instance of this class is created for each client that connects to the server.

BusTalkHandler

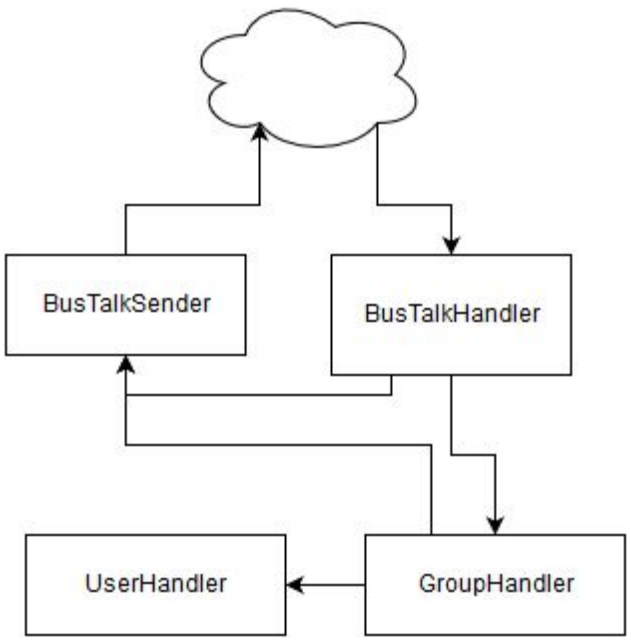
This class receives client messages from *BusTalkServerEndpoint*, checks what kind of message is received and decides what should be done, based on the message.

UserHandler
This class handles all user related things, such as disallowed nicknames, changing a user's information (nickname and interests), the connection between a *User* and a *Session*, etc.

ChatroomHandler
This class handles all that has to do with *Chatroom*s. If a user wants to join a chat room, create a chat room, etc., this is the class to call. It holds information about open chat rooms, linked to the group where the chat rooms are located.

BusTalkSender
This class acts as a post office, sending back messages to the users affected by an event. For example, when a user leaves a room, everybody in the room should get notified that a user has left, and that is what this class does. *BusTalkHandler* is the class that tells *BusTalkSender* what to send and when to send it.

JsonEncoder & JsonDecoder
These two classes make sure that data being sent to and from the server are interpreted correctly, and converted into the right objects. These works kind of as a layer between the incoming and outgoing messages, handling what is sent and received. These are used in *BusTalkServerEndpoint*.



As the server receives a message from a client (Figure 3), this message is sent to *BusTalkHandler* which interprets the message and tells *UserHandler* and/or *ChatroomHandler* what tasks to do. If required, *BusTalkHandler* also makes sure a message is sent back to the client by telling *BusTalkSender* what kind of message to be sent, while providing the required information for the message.

Figure 4: A simple overview of how

the server works.

3.2.2 Technologies Used

Websocket is being used to enable a full-duplex (simultaneous two-way) conversation between client and server.

The server uses JSON objects as input and output interpretation to be able to work as a server almost regardless of what platform the messages are being sent to and from.

3.2.3 Decomposition Description

3.2.3.1 Module Decomposition

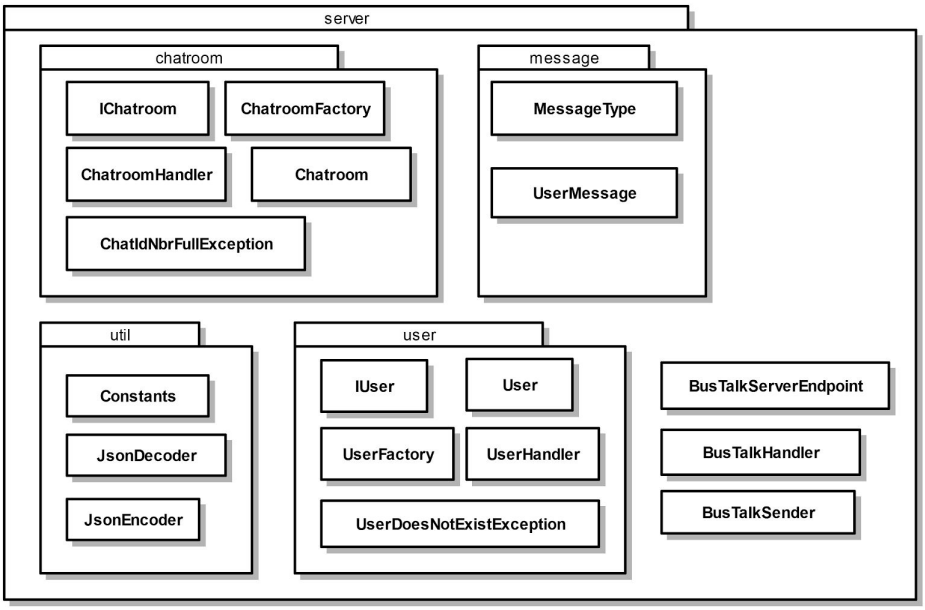


Figure 5: Package structure of the server.

***server* package**
The *server* package consists sub packages described below. Besides that it also contains three related to the connection and message handling of the server.

***chatroom* package**
The *chatroom* package handles everything that has to do with chat rooms. If a new chat room needs to be created, or a user wants to join a chat room, this is where that happens.

***message* package**
The *message* package consists of classes related to the incoming and outgoing messages. It holds the different types of messages that can be sent and received,

and the message class that is used on the server side (messages from clients are decoded into this object, and encoded when a message is sent to a client).

***user* package**
The *user* package handles all that has to do with users. Creation of users, user information and such is found here.

***util* package**
The *util* package consists of constants used within the server, but also the encoders and decoders for sent and received messages.

3.2.3.2 Dependency Analysis

Even if it looks somewhat cluttered, there are no circular dependencies and there are no immediate warnings with smelly code. See Figure 6.

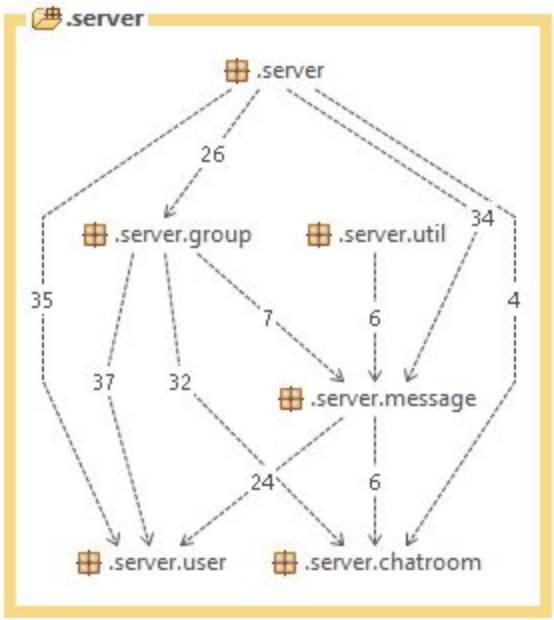


Figure 6: The overall view of the dependency between packages shown in STAN.

3.2.4 System Operation

When a client opens a connection to the server, a session is created. Each session have it's own instance of the BusTalkServerEndpoint class, which have a singleton instance of BusTalkHandler as their "entry point" to the server logic.

Communication between server and client is based on sending JSON-encoded object, which is decoded on the other side for interpretation.

When a message is received, BusTalkHandler will sort of what type of message that was received looking at the decoded JSON-object value associated with the string “type”. This will return an integer and what action to take is sorted out with a switch-case in the handleInput method.

When a received message has been handled and the appropriate action has been taken a message will be sent back to the client via the BusTalkSender if needed.

The communication between the client and the server is based on sending strings back and forth, and these strings are being converted to JSON object before being passed on to the server logic.

Each different type of message (request of chat room list, a chat message, join chat room...) has an identifier integer called “type”, and each of these types require some additional information.

3.2.5 Design Rationale

BusTalkHandler is a singleton due to the fact that every connected client gets their own instance of BusTalkServerEndpoint, and we want every instance to use the same server logic and this is exactly what using the singleton pattern offers.

4. HUMAN INTERFACE DESIGN

4.1 Overview of User Interface

User stories:
"As a user I want to be able to choose a nickname and optionally interests and then get logged in to the chat":
Upon launching the application the user is greeted by a login screen where he/she is prompted to type in his/her desired nickname and an optional interest. By clicking the login button the user then gets to arrive in the chat, provided the chosen nickname hasn't already been taken by another currently active user. See Figure 7.

"As a user i want to get a notification if my chosen nickname already is taken so that i can choose another one":
If the user by chance has chosen a nickname that has already been taken by another user a small notification window will appear for a short while on the screen informing the user of this.

"As a user I want to be able to see a list of all currently active members in the chat along with their chosen interests so that I can see who I am chatting with":
By clicking the actionbar item portraying three persons the user can reach a screen presenting a user list. See figure 9.

"As a user I want to be able to see information about the next bus stop while inside the chat so that I can continue chatting without lifting my head":
Between the actual chat window and the action bar, a small space is reserved for presenting the next bus stop. See figure 8.

4.2 Screen Images

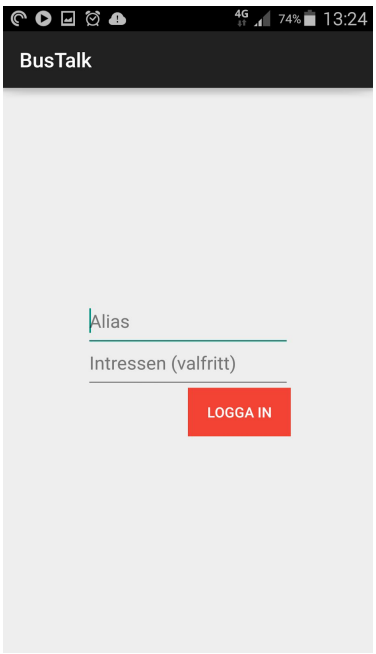


Figure 7: The login-screen

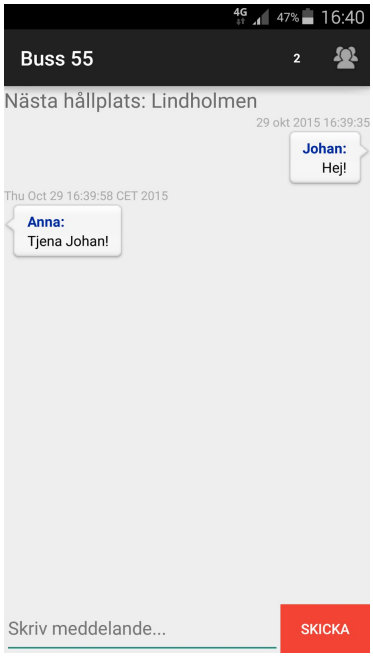


Figure 8: The chat-screen with next stop

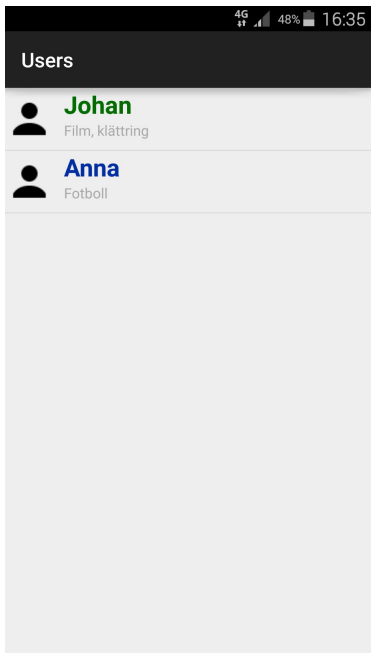


Figure 9: The screen listing the users

5. SOURCE REFERENCES

http://www.codeproject.com/Tips/897826/Designing-Android-Chat-Bubble-Chat-UI
The code for the adapter classes was initially derived from this website, and then tweaked to suit our application.

https://www.iconfinder.com/icons/309035/account_human_person_user_icon
The icon used in the UserAdapter class was taken from this website.

http://stackoverflow.com/questions/17541739/how-to-add-the-bubbles-to-textview-android
The background for a chatmessage was taken from this link.