Forside til eksamensopgave

MOBILAPPLIKATIONSUDVIKLING (MAU)

Underviser: Jacob Nordfalk

Applikationens navn og ikon
som det fremstår på en telefon:

Ripple

Studienr.:	s094357	Navn	Johannes Jørgensen

Nedenstående udfyldes kun ved gruppeprojekt

Studienr.:		Navn	

Studienr.:		Navn	

☐ Fortroligt

☑ Ej fortroligt
Må anvendes i undervisningen og til inspiration for andre studerende

☑ Teoriafsnit må viderebearbejdes
Andre må distribuere, remix, tweake, og bygge videre på teoriafsnittet, med kildeangivelse.
http://creativecommons.org/licenses/by/4.0/

Contents

Introduction

Project start

This report documents the creation of an Android app for personal location sharing. I have often found myself in situations where I needed the knowledge of real time updated locations of other individuals. The cases could be a festival where I would be building stages for live performance. You often need to know the location of Your supervisor so you can evaluate if you can reach him within minutes or tens of minutes. Normally this would be done by sending text messages but I wanted something that automatically would ping my location to my friends and vice versa.

Ripple is an app that seeks to fulfil exactly this through transmitting my location and showing the locations of my friends and myself on a map.

Problem formulation

I want to create an app, Ripple, that lets people share their locations in a fixed frequency. Ripple has to be structured as a container activity with different fragments occupying the container view. Ripple will use the XMPP protocol for transmission of locations and it will use the Quickblox as a BAAS and xmpp server. Ripple will focus on decoupled identities in such a way that your Ripple logins are disposable and decoupled from your established personal accounts like Facebook etc. This will reduce the chance of a being compromised as you can easily dump your account and create a new one. Ripple will seek to be secure by letting you decide which contacts you want to transmit to and a stretch goal is to implement Off-The-Record encryption on all XMPP traffic. Ripple shall plot all locations of enabled contacts on a map and all markers will be colored according to the time since their last location ping. If an enabled contact doesn't transmit pings for a certain period of time it will automatically be removed from the map for less cluttering.

Requirements

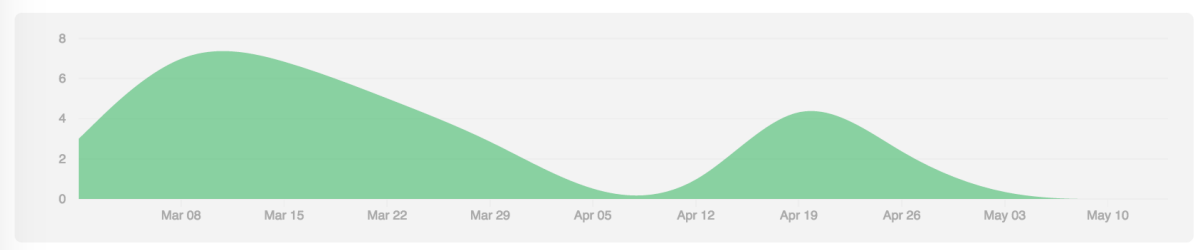
Name	Description
R1	Use Quickblox BaaS SDK for XMPP and user account managing
R2	Use a container activity and fragments
R3	Option to dispose account and create new account
R4	Add contacts to contact list and enable/disable transmission to them
R5	Automatically zoom in and show all enabled contacts on map
R6	Show all contacts on a map, if a contact does not ping its location within the allowed time frame it will be removed from the map. Within the time frame a color change will represent the freshness of its location ping.
R7	Enable end to end encryption using Off-The-Record encryption
R8	Delete contacts
R9	Brug lister og adaptere, netværkskommunikation, SharedPreferences

Process

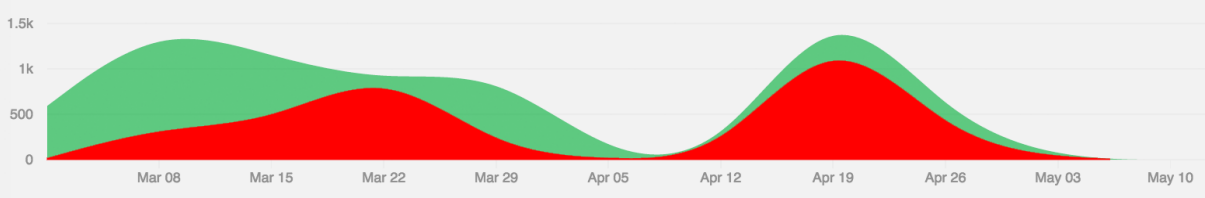
Overview

I decided early on that I would make an Android App for location sharing.

I had been working with the version control system GIT before and decided to use it for this project.



The figure above describes the amount of daily commits.



The figure above shows the amount of daily additions/deletions, (green/red), of lines.

Around start of April is a slow down of productivity, this is around the same period where I started researching how to change the whole app from a focus on activities and onto fragments. It is also the same time where I had to write a lot of deliverables to other classes I am attending.

I started by defining which subjects I would need to research in order to find the optimal solution before starting to implement them.

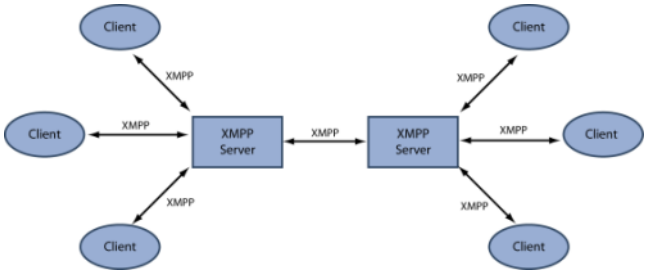
My main concerns from the start was:

- How to easily get XMPP up and running, best scenario would be without running the server myself
- Do I need a BaaS?
- How do I make the application secure?
- Should I put most weight on activities or fragments?

XMPP and BAAS

I started by researching which solutions existed for messaging between users. Originally I wanted my application to establish further security by having a decentralized protocol for communications.

However I soon figured out that decentralized communication is *bleeding edge* and in general not widely implemented. I dwelled at the extremely interesting *"Tele-Hash"* protocol by Jeremy Miller and wondered if I could make my own java implementation within the time frame but decided it was unrealistic. There is a short description of the Tele-Hash protocol in the appendices.



The xmpp protocol: client to server, server to server [1]

If it shouldn't be a decentralized protocol then the next best thing would be XMPP also originally invented by Jeremy Miller in the late 90's.

To create a XMPP system You need client implementations and a server. I tried to use the most recently updated XMPP libs for android I could find, called *"Smack"*. I setup an XMPP server on my home server and tried to connect to it. After 2 days of trial and error I still had not established a connection.

When searching for a solution to one of the endless undocumented XMPP exceptions from the *smack* driver I came across a reference to the Quickblox BaaS.

Quickblox used XMPP within but exposed a more simple XMPP API to the programmer.

I decided it was the way to go and a bonus was that Quickblox provided an XMPP service as well so I didn't have to run it myself.

Secutity and authentication

When I had decided on the Quickblox SDK I started looking for ways to implement security in the application. Security is critical since people are sharing their location and if a phone is compromised I wanted to reduce the likelihood of exposing all the locations of said phones contacts.

I had some ideas to implement security:

Make accounts disposable and decoupled The idea is to let accounts in my system be as naked as possible and decouple them from peoples private information, like facebook, phone, email etc. In this way you obfuscate the user so that the context is implicit, thereby overcoming a phone getting stolen and an unwanted entity gaining access to the map view. This entity will only be able to see where a number of usernames are located and nothing more. This way the thief would need knowledge of the context of a user for the information to have any value.

End to end encryption of all messages Early on I wanted the messages to furthermore be protected by encryption from end to end. Different options exist for this and they all have their specialities. I already knew about the PGP asymmetric encryption but it would break my idea of disposable accounts since pgp is public/private key encryption and therefore a key pair would have to be generated and stored on each account creation. This of course is no problem programmatically but if a solution existed providing end-to-end encryption without the keypairs I would prefer this.

The solution seemed to be Off-The-Record encryption which was invented specifically for instant messaging. The OTR protocol uses a combination of many *"crypto"* tricks to generate secure encryption per session and per message.

In this way I could make reasonably sure a message was not intercepted by a man-in-the-middle attack and the location of a user could be exposed.

Sadly I came to this conclusion too late into the process and didn't have time to implement the OTR protocol. However I managed to do the needed research and figure out which OTR library would have been suitable.

Activities and fragments

Through the process I had implemented my app with a focus on activities, those being the first knowledge acquired from the lessons. As I followed the Android lessons I was acquainted with the concept of fragments and how their life cycles are dramatically different from that of activities.

I rewrote the whole program to use fragments and decided to use a Singleton for storing information between fragments.

---

1 http://www.isode.com/whitepapers/xmpp.html

Theory

Quickblox is a powerful BaaS with SDK's for many different platforms focusing on providing easy communication such as push notifications, chat messaging, and easy user account management. Quickblox is a commercial BaaS with a free edition maxing out at 20 messages/second and 200000 monthly users. These features make it a good choice for a student project.

Quickblox creates a nice streamlined for android that hides away a lot of the complexity of the underlying libraries.

An example is that Quickblox uses XMPP for its chat protocol and uses the *smack* Android XMPP library but Quickblox adds a level of abstraction by hiding all *smack* functions and leaving a cleaner, more high level API for chat messaging.

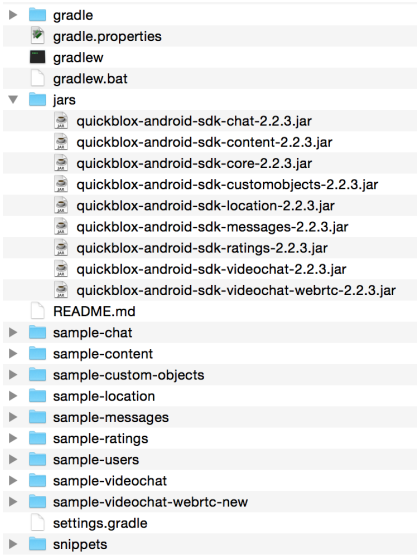
In this section i will try to explain the Quickblox SDK concerning the User and Chat parts. I will provide an implementation skeleton and a guide on how to install the library.

The Android SDK for Quickblox in general presents you with a callback-based asynchronous package of functions. An example for what this means is that if You want to login, You create a login *try* and a callback if login was successful. #TODO - explain why this is different - normal synchronous behaviour - etc This relieves your own code from having to deal with blocking API calls. See implementation/user for an example.

This section is inspired by the useful documentation provided by Quickblox on their website [2]

Installation

You download the Quickblox Android SDK from their website [2] as a zip with examples included. Below you can see how the downloaded sdk is organized:



The jars directory contains all the jars of the SDK; You choose which jars to embed in your project according to the parts of the Quickblox functionality you want.

You can always take a look in the different sample projects if you are uncertain which jars are needed for a certain part of the api.

Setup example

You need to tell Cradle to include the quickblox jars when building.

As an example i want to use the Users and Chat API's form the Quickblox SDK. The User API provides user authentication as well as means to letting users manage their user profiles. The Chat API provides user to user chat an the ability to check if another user is online.

For this example you would need the following jars: (i replaced the version with VERSION)

- quickblox-android-sdk-core_VERSION.jar
- quickblox-android-sdk-chat-VERSION.jar

The *core* jar contains all the core functionality of Quickblox, for example the User api. The *chat* jar contains all functions related to chat messaging.

You use the following syntax in your cradle settings

```
dependencies {
   compile files('libs/quickblox-android-sdk-core-2.1.jar')
   compile files('libs/quickblox-android-sdk-chat-2.1.jar')
}
```

And remember to define permissions for internet access in your manifest:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Implementation examples

User

To use the Users API you need to start by creating a session.

Session

Quickblox provides a nice *createSession* function that takes a callback as a parameter. Therefore you don't have to worry about the main thread being blocked while creating the session!

```
QBAuth.createSession(new QBEntityCallbackImpl<QBSession>() {

    @Override
    public void onSuccess(QBSession session, Bundle params) {
        /*
        YEAH you created your first Quickblox session!
        now go and have some Quickblox fun
        */
    }

    @Override
    public void onError(List<String> errors) {
        /*
        Too bad, there was an error establishing contact to the API server
        have a look in the error list for an explanation!
        */
    }
});
```

The Quickblox API expects You to implement some kind of state machine where each callback places you in a corresponding state. The createSession callback should lead either to a *session success* or *connection error* state.

Sign up

If you are in the *session success* state you are able to do API calls to Quickblox. Let's start by creating a user:

Let's create a user with the following information:

- username = karlmarx
- password = kapital
- phone number = 11223344

```
final QBUser user = new QBUser("karlmarx", "kapital");
user.setPhone("11223344")

QBUsers.signUp(user, new QBEntityCallbackImpl<QBUser>() {
    @Override
```

```
    public void onSuccess(QBUser user, Bundle args) {
        /*
        YEAH! you chose a unique, unused username and the API
        successfully created a new user
        */
    }

    @Override
    public void onError(List<String> errors) {
        /*
        Too bad, your new account was not accepted,
        there can be any number of reasons, have a look in the errors list ;-)
        */
    }
});
```

A Quickblox *user* can have many more fields set on it self both at creation and later on. These fields include:

- facebook id
- twitter id
- email
- tags (as a list of strings)
- website url

Sign in

When You have successfully signed up, You are allowed to sign in using the created user. You can sign in using any number of ways ranging from Twitter/Facebook tokens to using the native Quickblox users API.

Continuing on our example I will describe the process of logging in with a username and a password.

```
QBUser user = new QBUser("karlmarx", "kapital")

QBUsers.signIn(user, new QBEntityCallbackImpl<QBUser>() {
    @Override
    public void onSuccess(QBUser user, Bundle params) {
        /*
        Yeah you succesfully logged in!
        */
    }

    @Override
    public void onError(List<String> errors) {
        /*
        Too bad, Your credentials were rejected
        look in the errors list for forensics ;-)
        */
    }
});
```

This concludes the section on how to establish a Quickblox session, next up is sending a *hello world* chat message.

Chat

This section takes for granted that you have an authenticated session established. To begin chatting you need to establish some formalia beforehand. These formalia include the ones required by the xmpp protocol. More specifically you need to tell the xmpp protocol which frequency it will send an *"im online"* presence notification to keep you regarded as online. This notification is part of the xmpp protocol and is not a traditional *"push notification"*.

You do it like this:

```
if (!QBChatService.isInitialized()) {
    QBChatService.init(context);
}
QBChatService.getInstance().startAutoSendPresence(60);
```

Here we initialize the chat service if its not already initialized and then start transmitting presence notifications to Quickblox. If you want to handle changes in the connection you have to implement the *"ConnectionListener"* interface.

Chat "hello world"

Quickblox provides two ways to implement chat, 1-1 and group chat. I will describe 1-1 chat since it does not depend on a *group room* to exist beforehand.

To start a chat with another user you need to know the id of the user. If you don't know the id of the user, You can get it by using another known field of the user.

Here is an example of how to aqcuire the id of a user with username *"karlmarx"*:

```
QBUsers.getUserByLogin("karlmarx", new QBEntityCallbackImpl<QBUser>() {
    @Override
    public void onSuccess(QBUser user, Bundle args) {
        int user_id_of_karl_marx = user.getId()
    }

    @Override
    public void onError(List<String> errors) {
        /*
        Too bad you have not supplied right info, check errors list for explanations!
        */
    }
});
```

When you have the id of the user, You proceed to creating a chat with this user through the following steps:

Define a QBMessageListener of type QBPrivateChat

```
QBMessageListener<QBPrivateChat> privateChatMessageListener = new QBMessageListener<QBPrivat
    @Override
    public void processMessage(QBPrivateChat privateChat, final QBChatMessage chatMessage) {

    }

    @Override
    public void processError(QBPrivateChat privateChat, QBChatException error, QBChatMessage

    }

    @Override
    public void processMessageDelivered(QBPrivateChat privateChat, String messageID){

    }

    @Override
    public void processMessageRead(QBPrivateChat privateChat, String messageID){
    }
};
```

Define a QBPrivateChatManagerListener

```
QBPrivateChatManagerListener privateChatManagerListener = new QBPrivateChatManagerListener()
    @Override
    public void chatCreated(final QBPrivateChat privateChat, final boolean createdLocally) {
        if(!createdLocally){
```

```
            privateChat.addMessageListener(privateChatMessageListener);
        }
    }
};
```

Add the QBPrivateChatManagerListener to the QBChatService

```
QBChatService.getInstance().getPrivateChatManager().addPrivateChatManagerListener(privateCha
```

Create a QBChatMessage and send it

```
Integer opponentId = user_id_of_karl_marx;

try {
    QBChatMessage chatMessage = new QBChatMessage();
    chatMessage.setBody("Hello world");

    privateChat = privateChatManager.createChat(opponentId, privateChatMessageListener);
    privateChat.sendMessage(chatMessage);
} catch (XMPPException e) {

} catch (SmackException.NotConnectedException e) {

}
```

The exceptions can be quite non descriptive since they often refer to functions from inside the SDK jars.

This concludes the theory on the Quickblox android SDK

2(1, 2) http://quickblox.com/developers/Android#Download_Android_SDK.

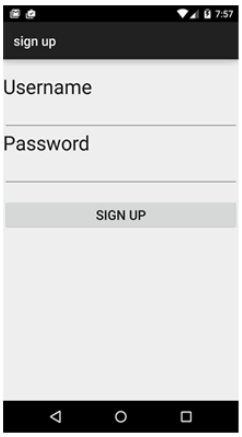
Design

Ripple consist of variations of four view:

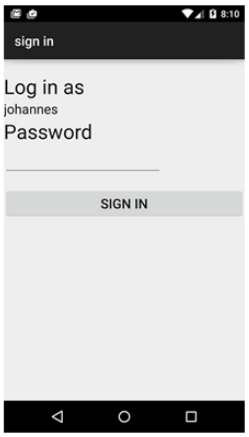
- Sign up
- Sign in
- Map
- Contacts list

Underneath You see the different views and a few variations on the *Map* view.

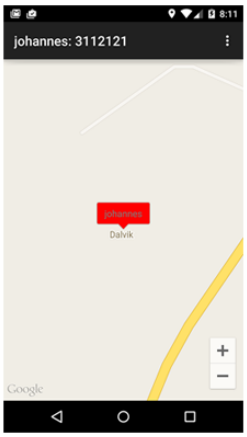
[A] - [SIGN UP]



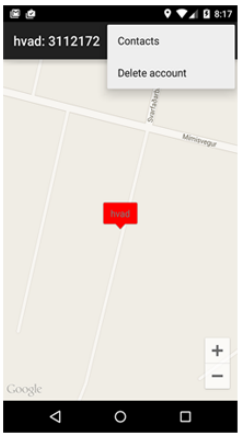
[B] - [SIGN IN]

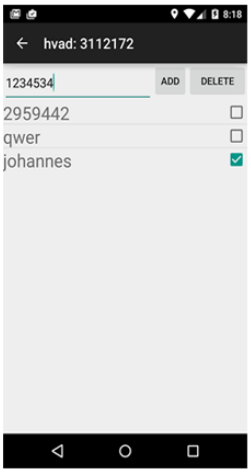


[C] - [MAP VIEW]



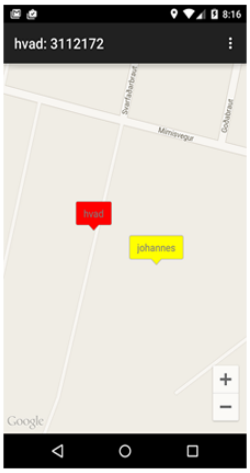
[D] - [MAP VIEW]





[D] - [CONTACTS]

[E] - [MAP VIEW]



[F] - [MAP VIEW]

At first I was experimenting with auto generated usernames to further obfuscate them as disposable and give people the impression that they were *"non-personal"* but it was hard to implement the functionality of figuring out which usernames were not taken at the BaaS so I went with people typing in usernames when signing up *(see Figure A)*

Markers

Letting people customize their usernames also gave me the option to use these to personalize the map markers. I decided to set the title of the container activity to a combination of username and user_id so it is easier when you are adding contacts (you just look at their screen and type in their user_id and press add)

The map markers change color as a function of the time-since-last-ping. In figure E and F you can see the functionality of the color change. We see in figure E that user *"hvad"* has added and enabled the user *"johannes"*. Johannes is green because *"hvad"* has just received a location update from *"johannes"*.

In figure F we see that *"johannes"* has stopped sending location updates to *"hvad"*, this has changed the color of the *"johannes"* marker into yellow, the step between *"healthy"* and *"removed from map*".

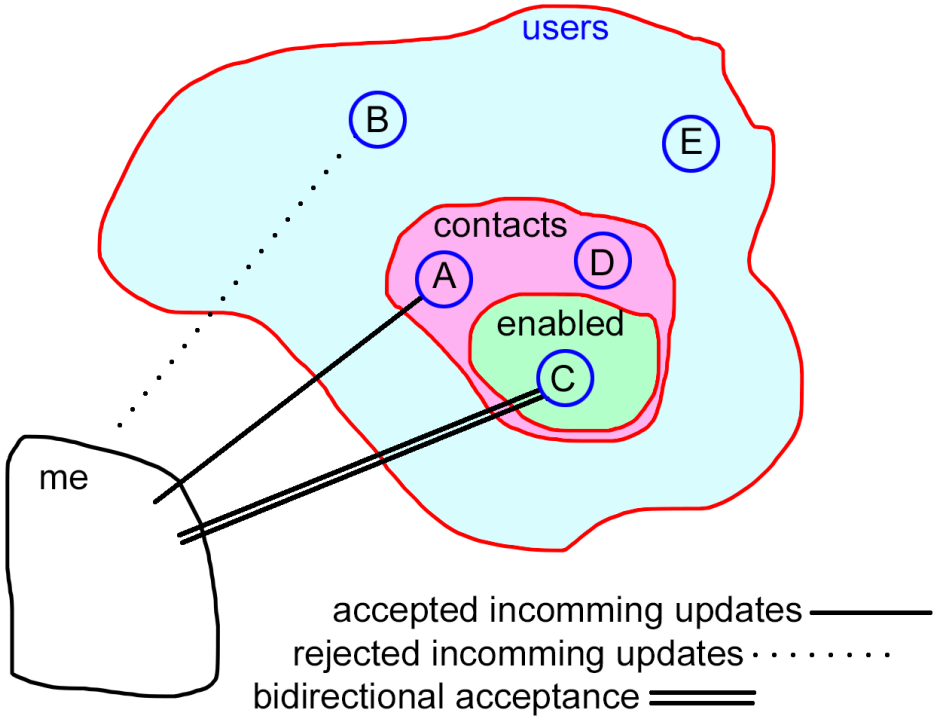
Contacts

Figure D shows the contacts view. This view is accessed from the action bar menu shown figure D. The contacts view lets you add a new contact, presently by typing in the user id of the contact. The view also lets you manage which contacts are enabled.

The contact list describes which users you accept location updates from, like a whitelist. It also features a toggle switch to enable the contact.

Enabling a contact means that you add the contact to the subset of contacts you transmit your location to.

The figure beneath describes the relationships:



1 : *accepted incoming updates:*

I accept, and draw markers for, incoming location updates from all contacts.

2 : *rejected incoming updates:*

I reject and ignore incoming location updates from users not in contact sub-set.

3: bidirectional acceptance:

The same as *1* with the added functionality that I actively transmit my location to users in this sub-sub-set at a fixed frequency.

Implementation

Persistence

I decided to use a singleton pattern for persisting the data.

ApplicationSingleton

Responsible for info regarding the current logged in user, the contact list and how to save/load this information from the preference manager.

It is implemented so that you ask them for an instance and if they internally have not populated their instance they will do so before returning it. #TODO - who are they

```
public static synchronized ApplicationSingleton getDataHolder() {
    if (dataHolder == null) {
        dataHolder = new ApplicationSingleton();
    }
    return dataHolder;
}
```

The ApplicationSingleton uses json to save load the settings as a single string from the preference manager. It packages all the contacts together with their *"UserDataStructure's"* and dumps them in a preference key.

Ripple will save current settings using json at a few occations:

1: Upon signing up

Ripple will clear the current list of contacts and save an empy one. Ripple will also set the *"current user"* on the ApplicationSingleton which will also be saved.

2: Upon signing in

Ripple will load the contacts list from json and set the *"current user"* on the ApplicationSingleton.

3: Upon manipulation contact list

The contact list will be saved every time a contact is added, enabled or disabled.

Currently the deletion of a contact is buggy and disabled. (see explanation in the *"Delete contact bug"* section below)

3: Upon account deletion

When you delete your account Ripple will clear all settings from the preference manager

Contacts

All users are Quickblox users. Quickblox users are designed to be personal and therefore have a lot of surplus meta data. In order not to bloat the persisted user data with this unused meta data I chose to implement my own user data structure.

UserDataStructure

The UserDataStructure is a property holding class with a few functions mostly in the getter/setter category: It has two fields:

```
private LatLng position;
private IconGenerator iconGenerator = null;
```

The position is geolocation of the contact and the iconGegenerator is an instance of a Google Maps Utilities IconGenerator for customizing the look of the contact marker. I used this specific library in order to easily change the color of each marker on the fly as well as generating the marker icon from a text string.

Contact list

The contacts are stored in the ApplicationSingleton in a hash map and an extra array list serves the purpose of assigning an index to each contact for use by the *"ContactListAdaptor"*.

```
  private Map<Integer, UserDataStructure> userContacts =
      new HashMap<Integer, UserDataStructure>();

  private List<Integer> indexToUserId =
      new ArrayList<Integer>();

The ContactListAdaptor is written so that instead of a manipulating a fixed set of entries i
```

Receive locations

The PrivateChatManager class is responsible for receiving all chat messages. It will discriminate between incoming messages based on their appearance in the contact list. If the sender of an incoming message is a contact then the PrivateChatManager will update that specific contact with its new location info. It will also update the contact with the username of the contact since this is not known when you add a contact using only the contacts user-id.

Update map markers and transmit location

The MapFragment executes a runnable when it receives focus. This runnable will reschedule itself on a fixed interval. The runnable is called *"locationsUpdatedRunnable"* and it is responsible for the following actions:

Update the map view

It will start by clearing all markers from the map. Then it will go through all contacts and add a marker if they fulfil a list of criteria: #TODO - there is only one criteria * The contact should have sent a location update within a defined period of time, else regarded as offline)

Three colors for the markers are used:

Red

The color of your own marker

Green

The color of a contact marker with fresh location update.

Yellow

The color of a contact marker with old location update

The runnable will focus the map view so it is centered and zoomed in such a way that the markers are all viewable and that they are not overlapping with the edge of the map, (padding)

Send position

The runnable is also responsible for transmitting the current location of the logged in user. The runnable will go through all contacts and send a *ChatMessage* to them if they are enabled.

The ChatMessage is a string formatted LatLon position.

Delete contact bug

When I had rewritten the application from a focus on activities to the use of a container activity I hadn't tested for regression bugs on the *"delete contact"* functionality. The result is that I have created a scenario where the implementation of such a feature will require a rewrite of major parts of the application.

As You can see in the preceding sections I'm referring to the Contacts hash map and the indexToUserId from a lot of different threads. This is no problem if I'm just appending to these data structures but if I'm removing from them then problems arise. In general what I experienced could be described as a deadlock problem. At the same time these three actions could happen:

- The ContactListAdaptor tries to remove a contact from the userContacts hash map and reorganize the indexToUserId array list.
- The PrivateChatManager tries to update the same contact with a new received update

- The MapFragment goes through the whole userContacts hash map to update the map and transmit current location to contacts.

These three actions cannot happen together and the result is long stack traces and asynchronous debugging for nights without end.

The solution

I propose two solutions to solve the deadlock issue at hand:

1. Implement the monitor design pattern so the userContacts and indexToUserId become governed by a monitor and only one entity can manipulate it at any given time.
2. Change the application so all resources and threads used during map view are freed when entering the contact list view. This would involve stopping the "PrivateChatManager" from listening and the "locationsUpdatedRunnable" from running. This would probably be the easiest solution.

Testing

I have only used informal testing in the develoment of Ripple. What I mean by informal testing is that I defined a list of actions I did every time I flashed the app to the emulator and based on the outcome being either: regression, accepted, failed, I debugged until the bugs were fixed.

I used these actions during development to test the condition of the app:

- Open and close the app repetitively
- Sign up and delete account
- Sign in and close app using task switch from android, open app and sign in again
- Sign up, close app, open app and sign in
- **Sign in and**
    - Open contact list and close contact list
    - Sign in, open contact list and add contact
    - Open contacts list, enable, disable contact
    - View MapFragment while a contact logs into another instance of the app. (check for communication)

Conclusion

I am quite happy with my app even if it didn't fulfill my ambitions. I started out with the project with experience from developing in java and xcode for iphone apps and thought this class would be easy but I was wrong.

The Android SDK is very fragmented because of their, admittedly nice, insisting on providing support for older API levels. This made it really hard to follow tutorials and code examples found on the internet because small changes to import statements could ruin the compiling of such, like examples depending on support libraries.

However I came up to speed and implemented the core of my ambitions. All Requirements except R7 and R8 is implemented and very functional. You can create accounts, login, add, enable, disable contacts, view them on the auto zoomed map etc.

I did not meet the requirement of supporting api level 15 since my emulator did not support this api level. I used genymotion and as seen in *appendice/genymotion* you can not create VM's with ice cream sandwich. I could have chosen the builtin emulator but the genymotion emulator gave me fast GPS mocking.

Process

I think I made a good decision to start out early on since I had problems getting to know the Android SDK. I used github for version control even though the use was sparse and mostly for marking some progress as well as a history of "undo's". I ended of with a good weight between research, programming and documentation. I wrote all my documentation using RestructuredText mark-up so I could generate the PDF from structured text instead of doing layout. This gave me a fast documentation workflow where I was not harassed by weird behaviour of an Office Suite. The source of both my documentation and the app are at the github repository. [3]

Met requirements

R1

I am using Quickblox sdk

R2

I am using a container activity with a switching fragments for ui.

R3

You are able to create and delete accounts, when deleted they are also deleted form the Quickblox server.

R4

You can add contacts and disable/enable them in a fragment.

R5

The map automatically zooms in to show all enabled contacts.

R6

The app automatically sorts contacts for viewing and discrimates on the freshness of the location updates.

R9

I have used adapter for the client list. Application singleton for persistence in runtime and state sharing. SharedPreferences for persistence between application executions. Network communication for Quickblox api.

Unmet requirements

R7

I used too much time organizing my project into a more maintainable fragment solution and in the end there was not time left to implement encryption. Encryption is not an easy task and the java libraries for doing OTR [4] would take much time to incorporate into using the Quickbox BaaS.

R8

As told about in the implementation section I had a deadlock bug in my application and the solution was outside the time frame of the project.

3 http://github.com/sloev/ripple
4 https://code.google.com/p/otr4j/wiki/QuickStart

Bibliography

When I developed the MapFragment, the following sites were a big help:

http://stackoverflow.com/questions/15098878/using-locationlistener-within-a-fragment
http://michalu.eu/wordpress/android-mapfragment-nested-in-parent-fragment/
http://stackoverflow.com/questions/13728041/move-markers-in-google-map-v2-android
https://blog.codecentric.de/en/2014/05/android-gps-positioning-location-strategies/

The DialogUtils class is taken from the Quickblox SDK: com.quickblox.sample.location.activities.dialogutils

In general the example samples from the Quickblox SDK were required in order to fulfil this project: http://quickblox.com/developers/Android

Appendices

Genymotion