## Popya

### Project description

During the project period we will make an android application. This will be a chat application, using mobile internet and global positioning to decide and make it possible who to chat with. This depends on the users maximum communication distance (see Illustration 1: Location based communication). Luuk and Michael will be able to communicate with each other but neither Luuk nor Michael will be able to talk to the other guy, because he is not in the maximum communication distance. The maximum communication range can be specified by the user.



Illustration 1: Location based communication

The application will connect you to all people within the specified range based on your global position. The position will be captured either by GPS or the approximated position coming from the cell id or a WiFi connection.

### Requirements

The following chapters will give an overview over the core requirements, which have to be fulfilled and all the nice to have features.

#### Core requirements

* Chat with people which are nearby.
* HTTPS for secure communication
* The maximum people to chat with is unlimited (is based on position radius, minimum radius is dependent on technology used for localization).
* People with different versions of the software should at least be able to talk to each other.
* Visualize the position of the chat partners on Google Maps
* Chat depending on movement (speed), when someone is for example in the train his messages shouldn't be sent to people who are standing still nearby the train track.

#### Nice to have requirements

* Send messages with pictures (from the camera or gallery).
* If pictures are supported use proxies for faster access.
* Send messages with files.
* Send messages with videos.
* Send messages with audios.
* Filtering of messages (no images, no files, only messages from a specific person).
* Allow private communications
* Allow private group communications

### Used technologies

* Android 2.1 for the client side implementation
  + GSON for JSON conversion
* Tomcat 6.0.24 for the server side implementation
  + HTTP POST based web service
  + JSON for transmitting data
  + Google Guava for the cache
  + Jersey for the web service implementation

### Design decisions

The following chapter describes all the design decisions and gives an overview of the architectural design of the application.

#### General design decisions

**Architectural overview**

The following diagram shows the layers of our architecture (see Illustration 2: Architectural overview). The whole project is split up into three sub projects. The first one is the client projects, which runs on the Android phone. It contains the user interface, the business logic and a background service, which is responsible for non-blocking sending and receiving messages from the server.

The second project is the API project. It contains the web service definition interface and all transfer objects.

The last project is the web server project which contains the implementation of the web service. The web server is responsible for receiving messages and distributing them correctly by using the current location of the user which has sent the message.



Illustration 2: Architectural overview

**Transfer objects**

Transfer objects are used for the communication with the server. This is necessary because the web server does not know the Android specific classes. It is also used to hide some information from the server.

**JSON**

JSON is used for interchanging data with the webserver. It is easy to use and there is a good JSON API available on Android (GSON).

**HTTP POST**

HTTP POST is used to communicate with the server. We have chosen HTTP POST because it is easy to use and does not have that much overhead compared to SOAP. Another reason is that SOAP is not well supported on Android. HTTP POST is also easier to test, because there are lots of browser plugins which can be used to test POST web services.

#### Client side design decisions

For sending and receiving messages a background service is implemented. All messages which should be sent will be stored in a queue inside the background service. After a defined period of time the background service will send all messages in the queue to the server.

This background service is also used to load new messages from the server. This is also executed in a defined time interval. The interval can be specified by the user. To inform the user interface that new messages are available, the observer pattern is used. The message background service informs all listeners about the new messages after loading them from the server (see Illustration 3: Background service communication).



Illustration 3: Background service communication

#### Server side design decisions

The server side of the Popya project is used to distribute the messages to the chat partners. This distribution is based on the current position of the users. To calculate the distance between two chat partners the haversine formula is used. The haversine formula is an equation important in navigation, giving distances between two points on a sphere.

The server uses a cache which contains all connected users with their unread messages. To ensure, that idle users will be disconnected automatically from the server, a cache is implemented, which removes all users after 10 minutes of idle time. Idle time means, that there was no call for sendMessage or getMessage on the server. For the cache, the guava library is used which has an implementation for a cache which removes keys from the map after a specified period of time.

### Implementation issues and solutions

1. Using SOAP for communication

We tried to use SOAP for the communication between the server and the client. But the API we used had some issues so we were unable to establish a connection.

To solve this issue we changed to a restful communication which was easier to use and easier to debug.

1. Update the message list view properly

At first the problem was that we had to exchange the whole list of messages if a new message arrived. We solved this by creating a method which fires a DataChangeEvent, provided by the MessageAdapter, which updates the model and adds just the new message item.

### Class diagram



Illustration 4: Client class diagram