SGN-35016 INTERNET OF THINGS AND MEDIA SERVICES PROJECT WORK - SNEAKSPEAK

Iiro Nykänen, Ville Vaarala

< student numbers >
< email addresses >
Tampere University of Technology

ABSTRACT

SneakSpeak is a self hosted group chat application. It conquers the market leaders Slack and Flowdock by giving organizations and teams ability to not to give 3rd parties access to the communication. The proof-of-concept consists of a Node.js GCM HTTP Connection Server and an Android client. The server can be hosted in a local network e.g. on a Raspberry Pi computer.

Methods and technologies used in the project work are explained in Methods chapter. Results and Conclusions will be written once the prototype is up and running.

1. INTRODUCTION

This document describes a simple messaging application proof-of-concept created as a project work for the course *Internet of things and media services*. The application consists of a Node.js server, an Android client and Thing™-integration.

The project originated from the realisation that nowadays different mobile instant messaging applications are very popular. Most of the people that own a modern smart phone have WhatsApp installed, the more critical users may prefer Telegram, and software developers are starting to migrate towards Slack. The thing these services have in common is that they provide the software as a service. All the messaging and information goes through their own servers, which may or may not be secure enough for your use. This project provides an alternative to these premade services. Since third party servers are deemed untrustworthy, the solution is for users to host their own private servers.

From this basic concept the project started to take shape. The first iterations of the core idea revolved around the messaging protocol; should we go with a ”normal“ http solution or if it were possible to use XMPP and websockets or if a server that only handled handshakes between clients would be a good idea. Eventually we moved on to a push notification system using Google Cloud Messaging API:s, but in the end steered towards a half and half solution by implementing upstream messaging via http.

The project was a long term implementation via exprerimentation, during which we ended up changing lots of things in both backend implementation and in the Android client application. The end result is a rough working prototype of what the real application could be, although for a real implementation the project would require better messaging protocols, security implementations, and other code base architecture revisions for the sake maintainability. Future additions could include a web, iOS and Windows Phone applications, but the project development will probably halt after the course.

2. METHODS

This section gives a quick glance at the technologies used in the application.

2.1. Android client

The Android application was written as a quick prototype to demonstrate the proof-of-concept. Code was written using the native SDK and a JVM-language called Kotlin, which compiles to Java bytecode. The main goal of the prototype was to allow messaging between devices via the provided self-hosted server. It is noteworthy that the code was written fast with functionality in mind, with little regards to maintainability or good practices. So if you take a look at it, don't judge us.

2.1.1. Structure

The overall application structure is partially messy but most of the classes have a clear area of responsibility. The code base can be roughly partitioned into activities and fragments, different recyclerview (list-component) implementations, a couple of manager singletons, gcm-communication classes and resultreceivers.

The application has three activities: MainActivity, which handles switching between lists of servers, users and channels; RegisterActivity, which surprisingly handles registration stuff; and ChatActivity, which is used display the chat window for private and channel messaging. Beside MainActivity most of the user interaction logic was implemented using Fragments. Due to the nature of the application there are lots of different lists. These were implemented using recyclerviews, each having their own adapter.

The application has four "manager" singletons: DatabaseManager, which handles data persistence (basically just servers); HttpManager, which is a bloated badly designed class that handles calls to server API; JsonManager, which is used to predefine possible serialization and deserialization routines; and SettingsManager, which could be used to save user settings (currently just saves registration data for application to remember).

Gcm-packages were required in the application in two places. During registration the client generates an unique token using RegistrationIntentService and sends it to the server, completing the registration. SneakGcmListenerService handles capturing downstream messages from the cloud, forwarding them to the fragment containing the chat logic. Registration and messaging data was forwarded internally using resultreceivers.

2.1.2. Development

The application was written with a sort of explorative mind set. A lot of the components used, like receivers (which basically were used to deliver messages internally from one class to another), were not too familiar and it took time to understand how they'd work. The application also served as a playground of sorts, and parts of it were initially implemented very differently. For example, at some point the UI-layouts were partially implemented with Anko DSL instead of the traditional XML-approach. After trying it out it was decided that XML would be the way to go and parts that used DSL were refactored.

During the development process we faced many problems with the platform. Some were easier to debug, others took lots of time. For example Proguard, which is a program used to obfuscate and minify Java bytecode, didn't cooperate at times and at the end of the project our `proguard-rules.pro`-file was roughly 150 lines including comments and whitespace. Due to obfuscation we also had some problems with debugging stacktraces, which were usually something like "`exception at a.b.c.d`".

2.2. Node.js server

The server is a Node.js application, more specifically Sails.js application. Users and channels are managed and stored on the server. Messages instead are just transmitted to the receiver or receivers without storing them to the database.

2.2.1. *Technology Choices*

Node.js was intuitive choice for backend application because both team members were already familiar with JavaScript and Node.js. Node.js is a JavaScript runtime which is often used as a web server application[1]. To make developing faster we chose to use Sails.js. Sails.js is a MVC (=Model View Control) framework built on top of Node.js[2]. There is plenty of documentation and support available online for both Node.js and Sails.js.

The use of Sails.js helped us not start from a scratch and accelerated the development process with integrated app generator and Blueprints API generator. There is also powerful Waterline ORM integrated in Sails.js. We chose to use the Waterline ORM with PostgreSQL adapter. First we planned to use SQLite database but the adapter for SQLite couldn't automate many-to-many associations as we wanted. Also using PostgreSQL enables us to use free version of Heroku for demo application.

Sails.js framework provides HTTP and WebSocket connections for API. For Google Cloud Messaging, GCM, connections we needed something else. First we implemented a GCM HTTP application server with sails-hook-push package[1]. Later we decided to use GCM also for upstream messages from the client to the application server. For that we had to fork and edit an existing node-gcm-ccs[2] library to implement the newest specification of GCM XMPP connection[3]. To add GCM XMPP listeners we created a Sails.js Project Hook.

Unfortunately we couldn't solve the problems with upstream messages. Some of the messages vanished or got delayed by hours. Because of this we added GCM listener equivalents to the HTTP API.

3. RESULTS

During the course we built a prototype for self hosted Android chat application with a possibility to add Internet of Things devices to send messages to chat channels. The prototype is split to two applications: Android client app and Node.js server application. We also hacked a Node.js package which can be used to send simple messags to a specific channel.

All the code is hosted and shared on GitHub with MIT open source license: `https://github.com/SneakSpeak/`

Some stuff about the Android application... Some stuff about the Android application... Some stuff about the Android application... Some stuff about the Android application...

The Node.js server can be hosted on almost any system which has Node.js and npm installed. Our implementation uses PostgreSQL database but one can choose the database they want to use and just change the Waterline adapter. Instructions on how one can host their own SneakSpeak server can be found on GitHub: `https://github.com/SneakSpeak/sp-server`.

The Node.js package for Internet of Things devices is extremely simple. It just sends messages to the specified channel on behalf of the user. This causes the user to not to receive messages sent by the device. The functionality is incomplete but demonstrates how real life measurements can be a part of project management and team collaboration.

4. CONCLUSIONS

As a course assignment the project was successful and met the goals that were set in the beginning. The prototype application works, but a real implementation would require some serious planning and architecture revisioning. The project was a learning experience for both of us and it was nice to be able to dedicate time and effort into areas that were genuinely interesting.

5/5 would recommend.

REFERENCES

[1] Google, *Implementing an HTTP Connection Server*, 2016, accessed 1.3.2016. [Online]. Available at: https://developers.google.com/cloud-messaging/http.