jamk.fi

CallTracker

Juha Sydänmaa

Bachelor's thesis
May 2018
Technology, communication and transport
Degree Programme in Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Sydänmaa, Juha	Bachelor's thesis	May 2018
		Language of publication: English
	Number of pages 29	Permission for web publication: x

Title of publication

CallTracker

Degree programme

Degree Programme in Software Engineering

Supervisor(s)

Rantala Ari

Assigned by

TeraKuu Oy

Abstract

The objective of this thesis was to create an Android application for detecting telephony events, and send information about the events to the client's server. The application should also be able to enable and disable call forwarding.

Later, the application was needed to be able to send SOAP requests to the client's web service, and receive data messages from the server, so that the application could be controlled remotely. The application was later named CallTracker.

CallTracker is written in Java. The communication from server to the application was achieved by implementing Google Cloud Messaging service for both application side and the server side.

As the result of this thesis, the client received an application, capable of all the requirements requested.

Keywords/tags (subjects)

Android, SOAP, Google Cloud Messaging

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Sydänmaa, Juha	Opinnäytetyö, AMK	Toukokuu 2018
		Julkaisun kieli: Englanti
	Sivumäärä 29	Verkkojulkaisulupa myönnetty: x

Työn nimi

CallTracker

Tutkinto- ohjelma

Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)

Rantala Ari

Toimeksiantaja(t)

TeraKuu Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli luoda Android-sovellus seuraamaan asiakasyrityksen puhelutapahtumia ja lähettämään niistä tietoa yrityksen palvelimelle sekä. Sovelluksella piti olla mahdollista aktivoida ja pysäyttää soitonsiirtopalvelu.

Myöhemmin, sovelluksen piti pystyä lähettämään SOAP-pyyntöjä asiakkaan verkkopalveluun sekä vastaanotta asiakkaan palvelimelta lähetettyjä viestejä. Tämä mahdollistaisi sovelluksen hallinoinnin etänä. Sovellukselle annettiin nimi CallTracker

CallTracker on kirjoitettu Java-ohjelmointikielellä. Palvelimelta sovellukseen tapahtuva kommunikointi toteutettiin Google Cloud Messaging–palvelun avulla.

Lopputuloksena oli mobiilisovellus, joka toteutti kaikki asiakkaan sille asettamat vaatimukset.

Avainsanat (asiasanat)

Android, SOAP, Google Cloud Messaging

Muut tiedot

Contents

1 Background.....3

2 Tools and technologies.....4

2.1 Android.....4

2.1.1 Java.....4

2.1.2 Android Studio.....5

2.1.3 ksoap2-android.....5

2.1.4 Google Cloud Messaging.....5

2.2 SOAP.....6

2.2.1 XML.....7

2.2.2 SoapUI.....7

3 The Application.....7

3.1 Detecting telephony state's changes.....8

3.2 Call forwarding.....10

3.3 Ending incoming call.....10

3.4 Communicating with the server.....11

3.4.1 Sending telephony event data.....12

3.4.2 Receiving data from server.....12

3.4.3 Sending a SOAP message to server.....13

3.5 Implementing Google Cloud Messaging.....17

3.5.1 GCM Message's structure.....17

3.5.2 Setup GCM for the application side.....19

3.5.3 GCM Server side.....20

3.6 Problems.....22

3.6.1 Sending data to server failed.....22

3.6.2 Operational redirecting not working properly when rejecting a call.....23

4 Results.....23

4.1 Futher development.....24

References.....25

Appendices.....26

Figures

Figure 1: Reflecting ITelephony and ending call.....11

Figure 2: Creating SoapObject.....14

Figure 3: Adding sub elements to the SoapObject.....14

Figure 4: Creating headers for the SOAP message.....15

Figure 5: Creating SoapEnvelope.....15

Figure 6: Performing a SOAP request.....16

Figure 7: Generating registration token.....20

Tables

Table 1: TelephonyManager's call states.....8

1 Background

During author's practical training in TeraKuu Oy (later client), an suggestion emerged of developing an application for sending information about company's call traffic to it's server. This would replace the client's Tasker implementation. The application was later named CallTracker.

TeraKuu Oy is a small software company located in Keuruu. They are specialized in developing FileMaker based information management systems. The company also provides technical support for their customers, which was the main reason for the need for stored information of the company's call traffic.

Tasker is a mobile application, which allows the user to automate tasks. These tasks are triggered when certain condition or event happens. In this case triggers were telephony states, and when the state would change, Tasker would perform HTTP request, containing data about the event, to client's server.

However, Tasker did not have support for detecting outgoing calls. This was the main reason why the client wanted to replace Tasker with a native Android application.

The client's initial requirements for the application with similar functionality what the Tasker task was capable of :

- Gathering information whenever a phone call is received, is answered, is ended or is started
- Sending acquired data to their web service
- Activating and deactivating call forwarding to a user's defined phone number.

Non- functional requirements were :

- The application should works on devices with Android version 4.1 or higher

Later, during the development phase, following requirement was also needed:

- Application should be able to be commanded by server side, by data messages received from server.

2 Tools and technologies

2.1 Android

Android is a popular, Linux based mobile operating system. It was originally developed by Android Inc. , but the company was bought by Google in 2005 (Wikipedia.com). It is built on Linux kernel and written largely in C. The Android SDK however, uses the Java language as the basis for the application. Android has been critized for fragmentation, for it's many different version. This makes it somewhat dificult for the developers, as the application needs to be developed so that it works with the oldest version that needs to be supported.

For this project, the fragmentation was not an issue, since the oldest Android version that needed to be supported was 4.1, and at the time of the development, the newest Android version was 6.0.

2.1.1 Java

Java programming language is a general-purpose, concurrent, classbased, object-oriented language. It is designed to be simple enough that many programmers can

achieve fluency in the language (J. Gosling, 2017). It was developed by James Gosling at Sun Microsystems in the 1990s.

2.1.2 Android Studio

Android Studio is an officially supported IDE, developed by Google. The first stable build, version 1.0, was released in December 2014. It was a replacement for the Eclipse Android Development Tool, which was the primary Android application development IDE before Android Studio's release.

2.1.3 ksoap2-android

The ksoap2-android project provides a lightweight and efficient SOAP client library for the Android platform (ksoap2-android Project). By default, Android does not provide any sort of SOAP library, so performing a SOAP requests requires using a third party library for handling SOAP requests.

CallTracker uses ksoap2-android for performing SOAP requests, because it is easy use, after figuring out what to do, and the data needed to be send was simple in structure. Library's main problem is, that the documentation is pretty nonexistent, but luckily the developers had gathered some useful links on the project's web site.

2.1.4 Google Cloud Messaging

Google Cloud Messaging (GCM) is a free service that enables developers to send messages between servers and client apps. This includes downstream messages from server to client apps, and upstream messages from client apps to server (Google.-com).

First the application registers for GCM by generating a registration token. This token is then send to the server side, which then sends a data message, containing the re-

gistration token, to GCM Connection Server. Then the GCM ConnectivityServer sends the data message to the client, identified by the registration token.

Google Cloud Messaging was used in order to have an option to remotely manage CallTracker by sending messages from server. For example, setting the phone number to which the incoming calls should be forwarded to, stopping the application for listening call state changes etc.

2.2 SOAP

SOAP (abbreviation for Simple Object Access Protocol) is a XML-based messaging protocol specification for exchanging structured information in the implementation of web services. It provides a way to communicate between applications running on different operating systems, with different technologies and programming languages. SOAP was designed in 1998 by Dave Winer, Don Box, Bob Atkinson and Mohsen Al-Ghosein for Microsoft.

SOAP message consists of four elements. These elements are Envelope, Header, Body and Fault.

Envelope is a mandatory part of SOAP message. It identifies the XML document as a SOAP message and indicates the start and the end of the message. It shows namespaces for the envelope schema definition.

Header is an optional element, containing specified application-level requirements. For example, the element can contain data for using services requiring user authentication.

Body is a mandatory element containing data exchanged in the SOAP message. The element must be within the envelope and it must follow any headers define for the message.

Should an error occur during processing the SOAP message, the response contains Fault element inside the body element. It contains specific information about the error.

2.2.1 XML

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents. XML documents are made up of storage units called entities (W3.org), which contain either parsed or unparsed data. It was developed by XML Working Group in 1996.

2.2.2 SoapUI

SoapUI is an open-source web service testing application. It is mainly used for testing SOAP and REST web services. SoapUI was originally developed by Eviware Software, which was aqquired by SmartBear Software in 2011.

SoapUI was mainly used to manually test the ksoap-androi2's generated SOAP message was valid, and that the client's web service accepted it and responsed correctly.

3 The Application

CallTracker consists of one Activity, which is used to start the service responsible for listening phone state changes manually. As the application's purpose is to gather information about phone's phone call activity while running in background, it should require little to no interaction with the user and progress should be independent. For this reason user interface is very simple in structure.

Other parts of the app are services, for handling phone state listening,and services required in order to make the app receive Google Cloud Messages (GCM) from client's server. In addition, there are some classes for making HTTP and SOAP requests and for logging possible exceptions and debug information.

3.1 Detecting telephony state's changes

Detecting telephony state changes in Android is achieved by implementing a custom PhoneStateListener, and registering it to receive notification of changes in specified telephony states. PhoneStateListener is a listener class for monitoring changes in specific telephony states on the device, including service state, signal strength, message waiting indicator (voicemail), and others (Android.com).

In this cause, the application needed to listen for changes in device's call state, so the custom PhoneStateListener needed to have override for onCallStateChanges callback.

This callback has two parameters :

- state : integer; the new telephony state. Refers to TelephonyManager's call states, see Table 1.
- phoneNumber : String, phone number of the caller.

Constant Name	Constant Value	Explanation
CALL_STATE_IDLE	0	No activity.
CALL_STATE_RINGING	1	A new call arrived and is ringing or waiting.
CALL_STATE_OFFHOOK	2	Offhook. At least one call exists that is dialing, active, or on hold.

Table 1: TelephonyManager's call states

As stated previously, the PhoneStateListener is started by a service (later PhoneStateListenerService), and keeps listening for call state changes as long as the service remains alive.

The intent, used to start the PhoneStateListenerService, can contain following data:

- boolean value to indicate if the service should be restarted if the application crashes.
- boolean value, to determinate if call forwarding for unanswered calls should be enabled.
- phone number, to which the incoming calls are forwarded to.
- delay, seconds before calls are being redirected.
- Whenever onCallStateChanges callback gets invoked, the new state is compared to previous states in order to detect the telephony action.

For example, if the state changes from 1 to 0, the incoming call was not answered. If the previous state was 1 and current state is 2, the call was answered. The appropriate request string is created according to the determined action.

However, outgoing calls can not be detected with PhoneStateListener (as the state changes from IDLE to OFFHOOK). Luckily Android has an action for detecting outgoing calls. By registering a BroadcastReceiver for action ACTION_NEW_OUTGOING_CALL, the receiver's onReceive callback gets invoked when there is an outgoing call about to be placed. The intent received contains the called phone number with key EXTRA_PHONE_NUMBER.

3.2 Call forwarding

By default, forwarding calls when user was busy, was activated, if the service received a phone number in it's intent. In addition, call forwarding when user did not answer to it, was activated if the service's intent contained a specified boolean variable.

Call forwarding is activated by dialing to **service_code*phone_number# ,

where service_code is , usually, any of the following:

- 21: forward all incoming calls
- 67: forward if busy
- 61: forward if not answered
- 62: forward if out of reach

67 forwarding service also has a optional parameter; **[delay]# , where delay indicates the delay in seconds the phone rings before it forwards the call. It can only be 5, 10, 15, 20, 25, or 30 . By default, the delay is 20 seconds, if it's not set.

Call forwarding can be deactivated by dialing to ##service_code#.

Programatically, call forwarding can be activated by sending an intent with action ACTION_CALL, and including a determinated forwarding string (Appendice 1).

3.3 Ending incoming call

The client wanted the app to be able to end the incoming call instantly, if there was already a call in progress. This case is mostly valid when call waiting is enabled, which allows multiple incoming calls to ring and be answered to.

The reason for this was that in this way the call could be forwarded as soon as possible. And because if not handled, it would have fouled the PhoneStateListener's logic.

Android has an hidden internal interface called ITelephony, which is used to interact with the phone. Being private, the interface cannot be accessed normally. But, It can be accessed by invoking TelephonyManager's private getITelephony() method. Call can be ended by invoking the interface's endCall() method (Figure 1).

There is two ways of doing this:

- Invoking methods directly
- Creating a similarly named interface, which is used to store the reflected interface

In order the latter case works, the created interface needs to have same package and class name as the ITelephony does.

```
Class c = Class.forName(TelephonyManager.class.getName());
Method m = c.getDeclaredMethod("getITelephony");
m.setAccessible(true);
ITelephony mITelephony = (ITelephony) m.invoke(mTelephonyManager);
mITelephony.endCall();
```

Figure 1: Reflecting ITelephony and ending call

3.4 Communicating with the server

The application communicates with the server in couple of ways. It sends data to server, when telephony state changes, the GCM's registration token updates or refreshes, or the server requests the application to send it's current status (settings used by the PhoneStateListener. In addition, the application can receive data from server, which is then used by the PhoneStateListenerService.

3.4.1 Sending telephony event data

After the call state changes, the service builds a HTTP request string for the corresponding call state change. The request contains the following data:

- Current call state. In case the current call state is idle, the call was either answered, missed or forwarded. In those cases, the corresponding action taken is send instead.
- caller's phone number,
- caller's name (if number is found from contacts),
- the called number, if the call is outgoing
- call direction
- device's id
- Call duration, when the call ends.

After building the request string, the requestt is then is executed in a AsyncTask class. AsyncTask allows to perform background operations and publish results on the UI thread without having to manipulate threads and handlers (Android.com).

If the network is not available, the string is stored to device's memory instead, and send next time when new request is send and the network is available.

3.4.2 Receiving data from server

When application receives a data message from server, custom GcmListenerService's (later GcmMessageHandlerService) onMessageReceived callback gets invoked (Appendice 2).

The received data is queried for following keys :

- number : phone number, to which the incoming calls are forwarded to.
- delay : delay as seconds before call is redirected, or the service waits before it ends the ringing call.
- operatorForwarding: a Boolean value indicating if the service should use normal operational call forwarding or, if not found or false, incoming call should be ended programmatically. This is mostly to avoid calls not being forwarded always, if they were ended, as discussed later.
- status : The server requests the application to send a SOAP message, containing the PhoneStateListener's current status
- stop: stop the service
- message : GcmMessageHandlerService builds a notification, with this key's content.

After the received data is quarried, the PhoneStateListenerService is then restarted with new intent, containing found values.

3.4.3 Sending a SOAP message to server

When requested by the server, or the GCM's registration token changes, the application sends a SOAP message, with following data:

- unique device id
- registration token, used by GCM to identify the application
- Status, string of PhoneStateListenerService's current status.

In addition, the SOAP message required to have headers, containing authentication data used to authenticate with the web service, to which the SOAP message was sent.

There are two alternatives for sending a SOAP request using ksoap2-android. For passing or receiving complex data, for example, objects or arrays, one should build a custom class, implementing KvmSerializable interface. The interface provides class with get and set methods for properties, used for mapping the properties. The class is basically the body element of the SOAP message.

If the data that needs to be retrieved or passed, is simple in structure, the library has a class called SoapObject, which already implements the said interface.

Since the information that needed to be passed to the client's server , consisted of couple of string variables, the latter way was used.

As seen in Figure 2, SoapObject's constructor takes two parameters, which are :

- namespace of the message's body element
- web service's operation name

```
SoapObject requestObject = new SoapObject(NAMESPACE_PHONETRACKER, METHOD);
```

Figure 2: Creating SoapObject

Then, SoapObject is populated with data by calling it's addProperty() method (Figure3). This method essentially creates a sub element for the SoapObject.

```
requestObject.addProperty("DeviceID",
        ((TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE)).getDeviceId());
requestObject.addProperty("Status", statusString);
requestObject.addProperty("DeviceToken", devToken);
```

Figure 3: Adding sub elements to the SoapObject

As the client's web service requires user authentication, the SOAP message needs to have headers, they are added by creating Element objects, which basically ad object representing a xml node, for each node element required for the header. Child ele-

ments are added and their types defined with addChild() method, as shown in Figure 4.

```
//headers
Element header = new Element().createElement(NAMESPACE_HEADER, "AuthenticatedUser");
Element Username = new Element().createElement(NAMESPACE_HEADER, "username");
Element password = new Element().createElement(NAMESPACE_HEADER, "password");
Username.addChild(Node.TEXT, WS_USER);
password.addChild(Node.TEXT, WS_PASSWORD);
header.addChild(Node.ELEMENT, Username);
header.addChild(Node.ELEMENT, password);
```

Figure 4: Creating headers for the SOAP message

Finally the SoapObject and the headers are included to a SoapEnvelope, as seen in Figure 5. SoapEnvelope is an object representing the SOAP message's envelope element.

SoapEnvelope also has a subclass, called SoapSerializationEnvelope, which offers SOAP serialization functionality, and has a method for getting the response message after sending the SOAP message. This subclass was used in this application in order to receive a response from the client's web service, making testing more easier.

```
SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
envelope.dotNet = true;
envelope.headerOut = new Element[1];
envelope.headerOut[0] = header;
envelope.setOutputSoapObject(requestObject);
```

Figure 5: Creating SoapEnvelope

In order to actually perform a SOAP call , a HttpTransportSE object needs to be initialized. The class has several constructors, in this application the one that takes the following parameters :

- Server's url
- Port number (default 443)

- Path to the web service's operation
- timeout

The SOAP call is performed by using created HttpTransportSE object's call() method (Figure 6).

Methods parameters are following:

- soapAction : the SOAP action, to which the message should be passed.
- envelope : the envelope containing the information for the SOAP call.

After the call method is called, the SoapSerliazisationEnvelope's getResult() method should contain the response from the web service, if the service sends one.

```
httpsTransportSE = new HttpsTransportSE(HOST, 443, uri.getPath(), 10000);
httpsTransportSE.debug = false;
httpsTransportSE.call(SOAP_ACTION, envelope);
Object result = envelope.getResponse();
Log.d(TAG,"result: "+ result.toString());
```

Figure 6: Performing a SOAP request

3.5 Implementing Google Cloud Messaging

In order to make the application receive messages send from server, app must be registered with GCM. The client app must obtain a unique registration token and pass it to the server side, which stores the token. The token received in process is the same client app instance identifier that the app server uses to send messages to the specific device.

It should be noted that GCM was implemented for this app before GCM was replaced with Firebase Cloud Messaging (FCM). Because of that, some of these instructions might be deprecated.

The GCM message has three fundamental components: the target, the options and the payload.

3.5.1 GCM Message's structure

Target

When server sends a message, it must specify a target, which is the destination of the message. Target is specified in to field of the message. Target can be a single registration token, a topic or a notification key, which is used for sending messages to device groups, meaning certain devices receiving the same message).

Target can also be a list of registration tokens. In this case, the message is send to devices having registration token included in list. This is defined with registration_ids field.

Options

The server can set various options when sending a message to a client app. In this project, the major option used was the one determining was the message collapsible or non-collapsible.

A non-collapsible message implies that each message is send to the device. Messages are non-collapsing by default , except for notification messages. Non-collapsible messages are typically used in chat messages and other more critical messages.

A collapsible message is a message that can be replaced my a new message containing the same collapse key if the message has yet to be delivered to the device. Noti-

fication is one of the most common use of collapsible messages. Only the most recent message is relevant. GCM allows a maximum of 4 different collapse keys to be used by the app server. Messages can be set collapsible by setting collapse_key field in the message.

This application uses collapsible messages , as their only function is to deliver data to the device, so the priority of the messages is not as high as , for example chat messages.

Payload

GCM provides two types of payload for downstream messaging (messaging from cloud to the application) :notification and data. Notification is more lightweight and has predefined set of user-visible keys. Data payload can contain custom key/value pairs. In addition, notification messages can contain an optional data payload. In this case, the payload is delivered when the users click on the notification.

In this case, the data payload was used, because it allows to include custom key/value pairs.

3.5.2 Setup GCM for the application side

GCM requires devices running Android 2.2 or higher with Google Play Store application installed in order to implement GCM on Android app. Pre 3.0 devices require users to set up their Google accounts. This is not required on devices running Android 4.0.4 or higher.

A new API project must be created in Google Console. Enable Google Cloud Messaging for the project in Api Manager. After following setup instructions, google-servises.json file can be downloaded and should be added to the projects module folder.

The next step is following lines on app level and project level build.gradle :

app level build.gradle :

```
dependencies {
compile 'com.google.android.gms:play-services-gcm:8.4.0'
}
apply plugin: 'com.google.gms.google-services'
```

project level build.gradle :

```
dependencies {
  classpath 'com.google.gms:google-services:2.1.0'
}
```

Google Play services SDK is also required in order to use GoogleCloudMessaging API. Google Play services SDK can be added to project via Android Studios' SDK Manager.

Android Manifest must include the declarations of following :

GcmReceiver, A receiver that receives GCM messages and delivers them to an application-specific GcmListenerService subclass.

GcmListenerService, a base class for communicating with Google Cloud Messaging. Enables various aspects of handling messages.

A service extending InstaceIDListenerService, base class to handle creating,updating and refreshing registration tokens.

In addition, the application must have implementations for custom InstaceIDListenerService and GcmListenerService

To send or receive messages, the app must get a registration token from InstanceID's getToken() method. The method has two parameters:

- senderID, which is project number, and can be found in google-services.json.
- specified scope, which in this case is GoogleCloudMessaging.INSTANCE_ID_SCOPE.

After registration token is generated or refreshed, it is passed to client's server via a SOAP message (Figure 7).

```
String registrationToken = InstanceID
        .getInstance(this).getToken(senderID, GoogleCloudMessaging.INSTANCE_ID_SCOPE);
```

Figure 7: Generating registration token

3.5.3 GCM Server side

The server side of Google Cloud Messaging consist of two components :

GCM connection servers, which are provided by Google. These servers take messages from an app server and send them to a client app running on a device. Google provides connection servers for HTTP and XMPP (Google.com).

An application server (app server) that sends data to a client app via GCM connection server.

Application server handles the following:

Communicating with client application.

Sending properly formatted requests to the GCM connection server.

Storing securely the server key and client registration tokens .

GCM offers two connection server protocols: HTTP and XMPP.

HTTP and XMPP messaging differ in the following :

Upstream/Downstream messages

- HTTP: Downstream only, cloud-to-device up to 4KB of data.
- XMPP: Upstream and downstream (device-to-cloud, cloud-to-device), up to 4 KB of data.

Messaging (synchronous or asynchronous)

- HTTP: Synchronous. App servers send messages as HTTP POST requests and wait for a response. This mechanism is synchronous and blocks the sender from sending another message until the response is received.
- XMPP: Asynchronous. App servers send/receive messages to/from all their devices at full line speed over persistent XMPP connections. The XMPP connection server sends acknowledgment or failure notifications asynchronously.

JSON

- HTTP: JSON messages sent as HTTP POST.
- XMPP: JSON messages encapsulated in XMPP messages.
- Plain Text
- HTTP: Plain Text messages sent as HTTP POST.
- XMPP: Not supported.
- Multicast downstream (messages send to multiple registration tokens)
- HTTP: Supported in JSON message format.
- XMPP: Not supported.

Having no need for upstream messaging, as the client wished to use existing SOAP web service for communication from application to the server, and XMPP having no support for multicast downstream messaging, HTTP messaging was used.

3.6 Problems

Overall the application development went smoothly, and there were only a couple of problems encountered during development.

However solving these problems was crucial for client and solving them took a long time.

3.6.1 Sending data to server failed

This error occurred when trying perform a HTTP request to the client's server.

The issue was caused by a SSLHandshakeException, reason being "Trust anchor for certification path not found". According to the Android documentation, SSLHandshakeException can happen for several reasons:

- The server certificate is not trusted by the system.
- The server certificate was self signed.
- The server configuration is missing an intermediate certificate authority

The issue was solved by setting a custom TrustManager for the HttpsUrlConnection, to trust a specific set of Certificate Authorities (CAs). The CA is fetched from server's certificate file, and uset to create a KeyStore, to initialize a custom TrustManager. A TrustManager is what the system uses to validate certificates from the server, and by

creating one form KeyStore with one or more CAs (Google.com), those will be the Certificate Authorities trusted by that specific TrustManager (Appendice 3).

3.6.2 Operational redirecting not working properly when rejecting a call

During the test period, conducted by the client, some of (approximately 5% according) programmatically rejected calls were not forwarded. The reason was a weird interaction between operator's forwarding service and rejecting calls; sometimes they were forwarded and sometimes not. Client had had this same issue with their Tasker task.

The issue was bypassed by using forwarding after delay, rather than ending call after certain time. Determining if the application should use forwarding with delay, or ending call after delay, is handled by checking if the received GCM message contains a specified boolean variable.

4 Results

The application was named CallTracker by the client. Once it is started, it handles listening changes in call states. When a change occurs, information about the current event is sent to the client's web service , which stores stores the received data to database. The application has been in use since summer 2016.

The application is capable of performing same tasks as the Tasker task did and in addition to detecting data about the outgoing calls. In addition, the application can be remotely controller and configured by messages sent by the server, something the Tasker was unable to do.

It is also more easier for the client to maintain and modify, compared to the Tasker implementation.

4.1 Futher development

Since the Google Cloud Messaging was deprecated in April 2018, and support for it will be removed in April 2019 (Google.com), CallTracker should be migrated to Firebase Cloud Messaging (FCM) service at some point.

The first registration to the GCM is done manually, it should be handled somewhere else, so that the registration for GCM does not require user interaction at all.

There is also a lot to improve on the application's user interface. As it was mentioned, the user interface is very minimalistic, mostly because not a lot effort was put in designing it, as it was not the main focus of the project.

And of course the existing code should be refactored frequently, to make the application performance more optimized.

References

Android.com - AsyncTask. Accessed on 14 April 2018. Retrieved from https://developer.android.com/reference/android/os/AsyncTask

Android.com - PhoneStateListener. Accessed on 14 April 2018. Retrieved from https://developer.android.com/reference/android/telephony/PhoneStateListener

Google.com – Google Cloud Messaging. Accessed on 14 April 2018. Retrieved from https://developers.google.com/cloud-messaging/gcm

Google.com – About GCM Connection Server. Accessed on 14 April 2018. Retrieved from https://developers.google.com/cloud-messaging/server

Google.com – Security with HTTPS and SSL. Accessed on 14 April 2018. Retrieved from https://developer.android.com/training/articles/security-ssl

Gosling, James; Joy, Bill; Steele, Guy; Bracha,Gilad; Buckley, Alex; Smith,Daniel, 2017 The Java Language Specification. Accessed 14 April 2018. Retrieved from https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf

Wikipedia.com – Android (operating system). Accessed on 14 April 2018. Retrieved from https://en.wikipedia.org/wiki/Android_(operating_system)

Wikipedia.com - SOAP. Accessed on 14 April 2018. Retrieved from https://en.wikipedia.org/wiki/SOAP

W3.org – Extensible Markup Language (XML) 1.0 (Second Edition) Accessed 14 April 2018. Retrieved from https://www.w3.org/TR/2000/REC-xml-20001006.pdf

Appendices

Appendice 1: Activating call forwarding programmatically

```
public void startForwarding() {
    Intent forwardingIntent;
    char firstChar = forwardingNumber.charAt(0);
    String notAnswered = "**61*" + forwardingNumber;
    if (firstChar != '+')
        forwardingNumber.replaceFirst(String.valueOf(firstChar), "+358");
    if(operatorForwarding) {
        if(delay > 0) {
            notAnswered += "**" + delay + "#";
        }
        else {
            notAnswered += "#";
        }

        forwardingIntent = new Intent(Intent.ACTION_CALL);
        forwardingIntent.setData(Uri.fromParts("tel", notAnswered, "#"));
        forwardingIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        try {
            startActivity(forwardingIntent);
        } catch (Exception e) {
            Log.d(TAG, "Unable to start forwarding (61) ", e);
        }
    }
```

Appendice 2. Querring received data message

```
@Override
public void onMessageReceived(String from, Bundle data) {
    Log.d(TAG, "onMessageReceived");
    Intent phoneListenerServiceIntent = new Intent(this, PhoneListenerService.class);
    try {
        Log.d(TAG, "data =" + data.toString());
        if (data.get("stop") != null) {
            stopService(phoneListenerServiceIntent);
            return;
        }
        if (data.getString("status") != null) {
                sendSoap();
        }

        if (data.getString("number") != null) {
            String forwardingNumber = data.getString("number");
            phoneListenerServiceIntent.putExtra(Intent.EXTRA_PHONE_NUMBER, forwardingNumber);
            phoneListenerServiceIntent.putExtra("restart", true);

            if (data.getBoolean("operatorForwarding")) {
                phoneListenerServiceIntent.putExtra("operatorForwarding", true);
            }
            if (data.get("delay") != null) {
                mDelay =  Integer.valueOf(data.get("delay").toString());
                phoneListenerServiceIntent.putExtra("delay", mDelay);
            } else {
                phoneListenerServiceIntent.putExtra("delay", 20);
            }
            getSharedPreferences("PhoneListenerData", Context.MODE_PRIVATE)
                    .edit()
                    .putString("forwardNumber", forwardingNumber)
                    .apply();

            startService(phoneListenerServiceIntent);
        }

        String message = data.getString("message");
        if(message != null && message.length() > 0)
            createNotification(from, message);
    } catch(Exception e) {
        Log.e(TAG, "Error occurred ", e);
    }
}
```

Appendice 3. Generating a custom TrustManager.

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// Load Certificate Authorities from inputStream.
InputStream inps = mContext.getResources().openRawResource(R.raw.teracloud);
Certificate ca = cf.generateCertificate(inps);
Log.e(TAG, "certificate ="+ ca.toString());
inps.close();
//Generate KeyStore , containing trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);

// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);
Log.e(TAG, "TrustManager initialized");

// Create an SSLContext that uses our TrustManager
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, tmf.getTrustManagers(), null);
SSLConnection.allowAllSSL();
Log.e("SSLContext init", "");

//// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL(params[0]);
Log.e(TAG, "URL = " +url.toString());
HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
urlConnection.setSSLSocketFactory(sslContext.getSocketFactory());
```