About This Document

This document is a technical article containing information on how to use the Google Cloud Messaging service in Android apps. To understand the contents of this article, one should know the basics of Android app development and Java EE (mainly servlets). The article will guide you through the process of creating a server app which sends messages to the GCM service and a client app that makes use of the notifications provided by the GCM.

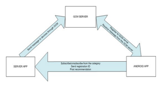
1. Introduction

1.1 What is the GCM and how it works?

The GCM abbreviation stands for Google Cloud Messaging for Android. It is a free service which is intended to help developers handle communication between the application deployed on the server and the apps installed on Android devices. The idea behind this is pretty simple. The Android app registers itself on the GCM server. The registration results in the generation of a unique identificator which is assigned to the device this app is installed on. This ID is then sent from the Android app to our server. From now on the server is able to use the GCM service for sending small messages (up to 4kb) to devices with the specified ID. It is possible to send a message to multiple receivers. These messages are usually called push notifications, since they are pushed to the device by the server. Such a solution takes the burden of constant polling for information updates from the Android app. There are two common applications of GCM messages. The first one is using them for sending very small notifications (with almost no data) telling our Android app that some data on the server changed and it should be downloaded. The second application is sending messages with a payload (e.g. text message) which is instantly consumed by the client app. Such an approach is presented in the sample app described in this article. The main responsibility of the GCM is queueing messages sent from the 3-rd party server and delivering them to registered devices. Altough the GCM does not guarantee the delivery of every message to every device it does its best and usually no notifications are lost. The nice thing is that messages can be stored in the GCM for up to 4 weeks. This proves to be useful in case when at the time of sending the message the device you are directing it is turned off. In such situations the GCM service will try to deliver the message later. To start using the service you have to enable it in the Google APIs Console (https://code.google.com/apis/console). Simply go to the Services tab, locate Google Cloud Messaging for Android and switch it on. If it is going to be your first project that uses any of the Google APIs, you have to create a Google APIs project beforehand. Please take notice of the Project number which will be used in the sample app as the Sender ID. Additionally, you have to generate a new server API key which will be used by the server app to authorize communication with the GCM. The GCM requires devices with Android 2.2 or higher that have the Google Play Store installed. You also need to add the gcm-server.jar library to your project.

1.2 Description of the sample app

The sample app developed for the purpose of this article is quite simple. It allows devices to register to the GCM server and unregister from it. After a proper registration, the user is able to subscribe to one of the four categories (IT, sport, music and other) and post recommendations of the content belonging to one of those categories. As mentioned in the previous section, the sample app uses GCM notifications to pass messages with some payload. Messages sent from our app contain the recommendation which is composed of a URL directing to some webpage and a short text message. This information is to be provided by the user. Additionally, the category the message concerns is attached. The recommendation is sent to our server that checks which devices are subscribed to the particular category and tells the GCM server to pass this message to those devices. It is also possible to unsubscribe from some category and therefore stop receiving notifications from it. Each message received by the client app is presented as an Android notification. The diagram presented below shows how the communication between the client app, server app and GCM service looks in general.


GCM SERVER
Send message to devices through GCM
Register to GCM server
Receive message from the GCM server
SERVER APP
Subscribe/unsubscribe from the category
Send registration ID
Post recommendation
ANDROID APP

[Diagram 1] Communication between the sample app components.

2. Development of the server

2.1 Structure of the server app

The server app developed for the purpose of this article is a basic Java EE web application. It contains the 7 servlets and one utility class whose main purpose is persisting the data about the registered devices and subscriptions. Apart from the mentioned components it also has the web.xml file which defines all the servlets and servlet mappings. The sample server is intended to be deployed on the Google App Engine application server, since it makes use of the App Engine’s Datastore i.e object, no-sql database. After a little bit of customization (changing the way data is stored) it is possible to run this app on any Java EE compliant application server. Here is the list of all servlets along with their responsibilities.

- MainServlet - presents information about the number of registered devices
- RegisterServlet - registers the device with the specified registration ID to the server
- UnregisterServlet –unregisters the device with the specified registration ID from the server
- SubscribeServlet - subscribes the device with the specified ID to the given category
- UnsubscribeServlet - unsubscribes the device with the specified ID from the given category
- PostRecommendationServlet - receives the recommendation, identifies devices subscribed to the given category and invokes SendMsgServlet
- SendMsgServlet - sends the message with the recommendatio to the GCM service

2.2 Handling requests from the client app

The main role of servlets used on the server is handling requests coming from the Android client app. There are four such servlets in this app: RegisterServlet, UnregisterServlet, SubscribeServlet, and UnsubscribeServlet. Their functionality is very simple. They parse the parameters passed to them and store these parameters in the Datastore for further reference. The code of one of them is below.

```
public class RegisterServlet extends HttpServlet {

   @Override
   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException {
      String id = request.getParameter("id");
      if (id == null || id.trim().length() == 0) {
         throw new ServletException("Parameter id not found");
      }
      Data.register(id);
      response.setStatus(HttpServletResponse.SC_OK);
   }
}
```

[Code 1] The registration servlet.

The RegistrationServlet is available on the server under the /register URL. It expects one parameter - id, which is the GCM generated registration id of the device. If the parameter is found in the servlet's request it is passed to the Data.register(String) method, which simply stores this id in the app's database. Additionally, the OK status code is returned with the response. When the id is missing a ServletException is thrown. The other three servlets work in exactly the same way. They differ only in the number of parameters.

There is one more servlet which is called directly by the client app. It is the PostRecommendationServlet. As mentioned before, it receives the recommendation data (category, content URL and the text message with the content's recommendation) and tells the SendMsgServlet to forward this recommendation to the specified client devices. It is more complicated than the previously described servlet, so it is worth to present its code.

```
public class PostRecommendationServlet extends HttpServlet {

   @Override
   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException {
      String registrationId = request.getParameter("id");
      if (registrationId == null || registrationId.trim().length() == 0) {
         throw new ServletException("Parameter id not found");
      }

      String category = request.getParameter("category");
      if (category == null || category.trim().length() == 0) {
         throw new ServletException("Parameter category not found");
      }

      String link = request.getParameter("link");
      if (link == null || link.trim().length() == 0) {
         throw new ServletException("Parameter link not found");
      }

      String recommendation = request.getParameter("recommendation");
      if (category == null || category.trim().length() == 0) {
         throw new ServletException("Parameter recommendation not found");
      }

      List<String> registeredDevices = Data.getDevices();
      if (!registeredDevices.isEmpty()) {
         Queue queue = QueueFactory.getQueue("cloudie");
         if (registeredDevices.size() == 1) {
            String regId = registeredDevices.get(0);
            List<String> categories = Data.getCategories(regId);
            if (categories != null && categories.contains(category)) {
               queue.add(TaskOptions.Builder.withUrl("/send").param(SendMsgServlet.ID,
regId).param(SendMsgServlet.CATEGORY,category).param(SendMsgServlet.LINK,
link).param(SendMsgServlet.RECOMMENDATION, recommendation));
               }
            } else {
            List<String> msgRecipients = new ArrayList<String>();
            for (String device : registeredDevices) {
               List<String> categories = Data.getCategories(device);
               if (categories != null && categories.contains(category)) {
                  msgRecipients.add(device);
               }
            }

            if (!msgRecipients.isEmpty()) {
               msgRecipients.remove(registrationId);
               String multicastKey = Data.storeList(msgRecipients);
               TaskOptions taskOptions =
TaskOptions.Builder.withUrl("/send").param(SendMsgServlet.MULTICAST_KEY,
multicastKey).param(SendMsgServlet.CATEGORY, category).param(SendMsgServlet.LINK,
link).param(SendMsgServlet.RECOMMENDATION, recommendation).method(Method.POST);
               queue.add(taskOptions);
            }
         }
      }
      response.setStatus(HttpServletResponse.SC_OK);
   }
}
```

[Code 2] The servlet for posting recommendation.

The first part of this servlet should already look familiar to you. It contains code which obtains four parameters (id, category, link and recommendation) from the request and stores them in the variables. In case any of the parameters is missing, a ServletException is thrown. The rest of the servlet identifies the devices that should be provided with the recommendation i.e. the devices that are subscribed to the particular category, and tells the SendMsgServlet to do the rest of the job. The sample app makes use of the mechanisms provided by the App Engine. It gets the task queue which is registered on the App Engine server and submits task requests to this queue. The queues provided by the App Engine are used to perform background tasks outside of the request. A task that calls SendMsgServlet with the specified parameters is created there. After creation it is added to the queue. It will be invoked by the App Engine. The similar functionality of making use of another servlet could be achieved without using the App Engine's goodies with the RequestDispatcher. The main difference is that the second servlet will be used in the foreground. The logic of the process of handling recommendations is very straightforward. At the beginning the list of IDs of all registered devices is obtained with the help of the Data.getDevices() method. Then the Data.getCategories(String registrationId) method is used for each ID on the list. It returns the categories the device with the given registraition ID subscribed to. Having the list, a check which tells if the currently handled recommendation should be sent to this device is performed. Depending on the number of registered devices, different types of tasks are used. If there is only one device that should be notified, task which contains the recommendation and the single ID is created. If the number of devices is greater than one, a second type of task is created. It contains the key of the list of devices to be notified. Such a list is stored in the Datastore with the Data.storeList(List<String>) method. Such an approach is used, since it is not possible to easily pass the list of values as the task's parameters. If you'd like to create a web app which does not use the App Engine, you could store this list in the database or in the application's session. The next section treats about the SendMsgServlet. It performs the final step of the recommendation submission handling i.e. it tells the GCM service to notify the specified devices.

2.3 Sending messages to the GCM server

The SendMsgServlet sends two types of messages to the GCM, depending on the number of devices to be notified: a single message (if there is only one recipient) or a multicast message (in case if there are multiple recipients). To be able to use the GCM service, it is needed to store an API Access Key somewhere in the application. In the case of this sample app, it is stored in the API_KEY attribute of the Data class. This key is the one generated in the Google APIs console. It should be generated as the key for the server apps. In case you would like to run this sample app, you have to set your own key as the API_KEY variable. If you are not sure if the key you have is proper, please go to the Google APIs Console and check that you have provided the correct one (it should be marked as the key for server apps). Moreover, you should check if the Google Cloud Messaging for Andoird service is enabled in your developer's console. Here is the code of the SendMsgServlet.

```
public class SendMsgServlet extends HttpServlet {
   // Sender used to send messages to GCM server.
   private final Sender sender = new Sender(Data.API_KEY);

   @Override
   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
IOException {
      String registrationId = request.getParameter(ID);
      if (registrationId != null) {
         sendSingle(registrationId, request, response);
         return;
      }
      String multicastKey = request.getParameter(MULTICAST_KEY);
      if (multicastKey != null) {
         sendMulticast(multicastKey, request, response);
         return;
      }
```

```
        response.setStatus(200);
        return;
    }

    private void sendSingle(String registrationId, HttpServletRequest request,
HttpServletResponse response) {
        Message msg = new Message.Builder().addData(CATEGORY,
request.getParameter(CATEGORY)).addData(LINK,
request.getParameter(LINK)).addData(RECOMMENDATION,request.getParameter(RECOMMENDATION)).bui
ld();
        Result result = null;
        try {
            result = sender.sendNoRetry(msg, registrationId);
        } catch (IOException e) {
            response.setStatus(200;
        }

        if (result != null) {
            if (result.getErrorCodeName().equals(Constants.ERROR_NOT_REGISTERED)) {
                Data.unregister(registrationId);
            }
        }
    }

    private void sendMulticast(String multicastKey, HttpServletRequest request,
HttpServletResponse response) {
        List<String> multicastIds = Data.getList(multicastKey);
        Message msg = new Message.Builder().addData(CATEGORY,
request.getParameter(CATEGORY)).addData(LINK,
request.getParameter(LINK)).addData(RECOMMENDATION,
request.getParameter(RECOMMENDATION)).build();
        try {
            sender.sendNoRetry(msg, multicastIds);
        } catch (IOException e) {
            Data.deleteList(multicastKey);
            response.setStatus(200);
        }
    }

}
```

[Code 3] The servlet for sending messages to GCM.

The servlet at the beginning checks parameters passed to it. If only a single ID was passed, it invokes the sendSingle(…) method. In case the multicast parameter is present, the sendMulticast (…) method is called. Before delving into details about these methods, there is one thing that should be mentioned. The SendMsgServlet has the sender attribute which is used to send messages to the GCM service. It is created by passing Data.API_KEY to the Sender's constructor. The sendSingle and sendMulticast methods are quite similar. At the beginning of each a message which will be sent to the GCM is constructed. Each message has the following parameters: category the message belongs to, link to the recommended content and the recommendation in the form of a short text. When a message is ready, it is sent to the GCM with the help of the sendNoRetry(…) method invoked on the Sender's instance. The first parameter of this method is the message that should be delivered. The second parameter is either a single registration ID or the list of IDs, depending on the type of message. It is possible to get the result of the communication with the GCM as presented in the sendSingle(…) method.

3. Development of the Android client

3.1 Initial project configuration

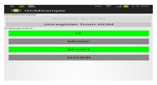
The Android client app is used for registering to the server, subscribing to the message categories, posting recommendations and receiving notifications. Before we delve into the source code of the sample app there are a few things that should be configured. First of all, you have to add the gcm.jar library to your Android project. The library can be found in the extras/google/gcm/gcm-client/lib subfolder contained in the Android SDK's folder. You have to install the Google Cloud Messaging for Android Library with the Android SDK Manager to have the gcm.jar library. There is also some configuration required in the app's manifest file. Its content is below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.sprc.gcmsample"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="16"
        android:targetSdkVersion="17" />

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

    <permission
        android:name="com.sprc.gcmsample.permission.C2D_MESSAGE"
        android:protectionLevel="signature" />

    <uses-permission
        android:name="com.sprc.gcmsample.permission.C2D_MESSAGE" />

   <uses-permission
```

```
        android:name="com.google.android.c2dm.permission.RECEIVE" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <activity
            android:name="com.sprc.gcmsample.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity
            android:name="com.sprc.gcmsample.CategoryActivity" />

         <receiver
            android:name="com.google.android.gcm.GCMBroadcastReceiver"
            android:permission="com.google.android.c2dm.permission.SEND" >
            <intent-filter>
                <!-- Receives the actual messages. -->
                <action android:name="com.google.android.c2dm.intent.RECEIVE" />
                <!-- Receives the registration id. -->
                <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
                <category android:name="com.sprc.gcmsample" />
            </intent-filter>
        </receiver>

        <service android:name="com.sprc.gcmsample.GCMIntentService" />
    </application>

</manifest>
```

[Code 4] The AndroidManifest.xml file.

At the beginning permissions used by the app are required. Apart from the typical ones like INTERNET, GET_ACCOUNTS and WAKE_LOCK, there are two permissions specific to the GCM service. The first one is the com.sprc.gcmsample.permission.C2D_MESSAGE that is the custom permission which makes sure that only our app can receive GCM messages. The second is com.google.android.c2dm.permission.RECEIVE which allows our app to receive GCM messages. Two activities are defined: MainActivity and CategoryActivity. MainActivity contains a few buttons which allow for the registration to and unregistration from the GCM server and for subscribing to the message categories.



[Image 1] The MainActivity of the client app.

The CategoryActivity allows to post the recommendation to the category and to stop the subscription of the notification form this category. Apart from the activities, a broadcast receiver which receives messages from the GCM service is defined. Also, GCMIntentService which is used for handling the GCM messages is defined.

3.2 Handling communication with the GCM service

Communication with the GCM service is performed in two places: in the MainActivity and in the GCMIntentService.To communicate with the GCM the Sender ID is required. This ID is the Project number from the Google APIs console. In case of this sample app, this ID is stored in the SENDER_ID variable of the Constants class. The MainActivity allows the app’s user to register to and unregister from the GCM service. An excerpt of this class with the most relevant parts is below.

```
public class MainActivity extends Activity {
   . . .
   private Button register;
   private Button unregister;
   private SharedPreferences preferences;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      register = (Button) findViewById(R.id.register_gcm);
      unregister = (Button) findViewById(R.id.unregister_gcm);

      . . .

      // Check if device is capable of using GCM
      GCMRegistrar.checkDevice(this);
      registerReceiver(messageReceiver, new
IntentFilter(Constants.DISPLAY_RECOMMENDATION));
   }

   public void onRegisterGCM(View v) {
      final String regId = GCMRegistrar.getRegistrationId(this);
      if (regId.isEmpty()) {
         GCMRegistrar.register(this, Constants.SENDER_ID);
```

```
        } else {
            if (!GCMRegistrar.isRegisteredOnServer(this)) {
                new Thread(new Runnable() {
                @Override
                public void run() {
                    boolean registered = NetworkOpsHelper.register(MainActivity.this, regId);
                    if (!registered) {
                        GCMRegistrar.unregister(MainActivity.this);
                            runOnUiThread(new Runnable() {
                                @Override
                                public void run() {
                                    toggleButtons("");
                                }
                            });
                        }
                    }
                }).start();
            }
        }
    }

    public void onUnregisterGCM(View v) {
        GCMRegistrar.unregister(this);
        . . .
    }



    @Override
    protected void onDestroy() {
        if (messageReceiver != null) {
            unregisterReceiver(messageReceiver);
        }
        GCMRegistrar.onDestroy(this);
        super.onDestroy();
    }

    private final BroadcastReceiver messageReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            MessageCategory category =
MessageCategory.getFromStringNumber(intent.getStringExtra(Constants.EXTRA_CATEGORY));
            String link = intent.getStringExtra(Constants.EXTRA_LINK);
            String recommendation = intent.getStringExtra(Constants.EXTRA_RECOMMENDATION);
```

```
         String output = "Latest recommendation:\n Category: " + category.name() + "\n
Content URL: " + link + "\n Opinion: " + recommendation;
         Log.i(TAG, output);
      }
   };
}
```

[Code 5] The excerpt of MainActivity.

MainActivity uses GCM through the GCMRegistrar class. In its onCreate(Bundle) method it calls GCMRegistrar.checkDevice(Context) to check if the device is capable of using GCM. If the device cannot use GCM, the UnsupportedOperationException will be thrown. The onRegisterGCM(View) method is invoked when the user clicks the 'Register to GCM' button. It performs the GCM registration process. At the beginning GCMRegistrat.getRegistrationId(Context) is called. This method returns the registration ID or an empty string if the device is not yet registerd. If it turns out that the latter is true, the GCMRegistrar.register(Context c, String senderId) method is called. The second parameter of this method has to be the SENDER_ID. When the GCM registration ID is not empty, we check if the device is registered to our server app. If it is not registered, the registration process request is sent to our server. It is done with the NetworkOpsHelper.register(Context c, String registration ID) helper method. This method has to be called in a new thread since Android prohibits performing network operations on the UI thread. The provided excerpt also shows that GCMRegistrar.unregister(Context c) has to be called to unregister the device from the GCM. Additionally, it presents the BroadcastReceiver named messageReceiver which is used to pass recommendations from the GCMIntentService to the MainActivity.

The rest of the communication with the GCM service is contained in the GCMIntentService which extends GCMBaseIntentService provided by the gcm.jar library. The mentioned service consists of several callback methods which are invoked when messages from GCM service are received. The service is not only responsible for handling GCM callbacks, but also sends requests to our server (e.g. to transfer the device's GCM registration ID) and passes recommendations to the MainActivity. Its most important part is presented below.

```
public class GCMIntentService extends GCMBaseIntentService {

   public GCMIntentService() {
      super(Constants.SENDER_ID);
   }

   @Override
   protected void onRegistered(Context context, String regId) {
      NetworkOpsHelper.register(context, regId);
   }

   @Override
   protected void onUnregistered(Context context, String regId) {
      if (GCMRegistrar.isRegisteredOnServer(context)) {
         NetworkOpsHelper.unregister(context, regId);
```

```
        }
    }

    @Override
    protected void onMessage(Context context, Intent intent) {
        transferMessage(context,
intent.getStringExtra(Constants.EXTRA_CATEGORY),intent.getStringExtra(Constants.EXTRA_LINK),
intent.getStringExtra(Constants.EXTRA_RECOMMENDATION));
    }

    private void transferMessage(Context context, String category, String link, String
recommendation) {
        Intent intent = new Intent(Constants.DISPLAY_RECOMMENDATION);
        intent.putExtra(Constants.EXTRA_CATEGORY, category);
        intent.putExtra(Constants.EXTRA_LINK, link);
        intent.putExtra(Constants.EXTRA_RECOMMENDATION, recommendation);
        context.sendBroadcast(intent);
    }
}
```

[Code 6] The excerpt of GCMIntentService.

The onRegistered(Context c, String regId) method is called when the registration to the GCM service succeeds. Its second parameter is the registration ID generated by the GCM. It uniquely identifies the device and is used by our server app to send messages. Therefore it is sent throught the NetworkOpsHelper.register(Context c, String regId) to our server app. The process of unregistering a device from the GCM and our server looks similarly. The most important method is the onMessage method. It is invoked each time a message from the GCM arrives. All data sent from our server is in the Intent passed to this method and can be now retrieved. At this point we get the category, content URL and the recommendation and transfers that information to the MainActivity through the broadcast receiver intent.

3.3 Sending requests to the server app

The last thing that needs description is handling the communication with our server. All code relevant to this is contained in the NetworkOpsHelper class. A part of it is presented below.

```
public class NetworkOpsHelper {
    public static boolean register(Context context, String registrationId) {
        String serverUrl = Constants.SERVER_URL + "/register";
        Map<String, String> parameters = new HashMap<String, String>();
        parameters.put("id", registrationId);
        long waitTime = Constants.WAIT_TIME_MS;
        // Using exponential back-off while trying to register on the server.
        for (int i = 0; i <= Constants.ATTEMPTS; i++) {
            try {
                postRequest(serverUrl, parameters);
                GCMRegistrar.setRegisteredOnServer(context, true);
                return true;
            } catch (IOException e) {
                if (i == Constants.ATTEMPTS) {
                    break;
                }
                try {
                    Thread.sleep(waitTime);
                } catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();
                    return false;
                }
            }
            waitTime *= 2;
        }
        return false;
    }

    public static void unregister(Context context, String registrationId) {
        String serverUrl = Constants.SERVER_URL + "/unregister";
        Map<String, String> parameters = new HashMap<String, String>();
        parameters.put("id", registrationId);
        try {
            postRequest(serverUrl, parameters);
            GCMRegistrar.setRegisteredOnServer(context, false);
        } catch (IOException e) {
            Log.i(TAG, "Unable to unregister from server");
        }
    }

    private static void postRequest(String serverUrl, Map<String, String> parameters) throws
IOException {
        URL url;
        try {
            url = new URL(serverUrl);
        } catch (MalformedURLException e) {
            throw new IllegalArgumentException("Wrong URL: " + serverUrl);
        }

        StringBuilder requestBody = new StringBuilder();
        Iterator<Entry<String, String>> it = parameters.entrySet().iterator();
        while (it.hasNext()) {
            Entry<String, String> entry = it.next();
            requestBody.append(entry.toString());
            if (it.hasNext()) {
                requestBody.append('&');
            }
        }

        byte[] data = requestBody.toString().getBytes();
        HttpURLConnection conn = null;
        try {
            conn = (HttpURLConnection) url.openConnection();
            conn.setDoOutput(true);
            conn.setUseCaches(false);
            conn.setFixedLengthStreamingMode(data.length);
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type",
"application/x-www-form-urlencoded;charset=UTF-8");
            OutputStream out = conn.getOutputStream();
            out.write(data);
            out.close();
            int status = conn.getResponseCode();
            if (status != 200) {
                throw new IOException("Request failed with status: " + status);
            }
        } finally {
            if (conn != null) {
                conn.disconnect();
            }
        }
    }
```

[Code 7] The excerpt of the NetworkOpsHelper class.

The helper class has the postRequest method which constructs and sends a POST request to the server. It needs to be supplied with the server's URL and the parameters to be passed with the request. This method is used by all the other methods from the presented class. The first method to be described is the register method. It sends to our server the device registration ID supplied by the GCM service. It uses the exponential back-off technique to increase the possibility that the message is successfully transferred to the server. The idea behind this approach is the following: the app tries to send the request, if it turns out that the server was unreachable it waits for a specified amount of time and retries the operation. Additionally, the waiting time is doubled. Three such attempts are performed, unless the operation succeeds the first time. The GCMRegistrar.setRegisteredOnServer(context, true) method is called to notify our client app that the registration was performed correctly. The rest of the methods from this class look alike. The main difference is the URL of the servlet that the requests are sent to.

We hope that you find this guide on using the Google Cloud Messaging service on the Android platform useful. If you have any issues, please let us know via the Samsung Developer forums.

Ref:http://developer.samsung.com/android/technical-docs/Using-Google-Cloud-Messaging#