How to connect to IoT Gateway and Server MQTT on Android devices

1. Elementary information

1.1. Send commands from the application to hardware

In case you would like to grant mobile application the hardware controlling privilege, you need to make the application publish 1 message to MQTT Broker. Even without being notified for the arrival of message, the application is still able to verify whether it is published to server or not.

If you desire to boost the likelihood the message turns up in hardware, feel free to modify QoS parameter of the message. Notwithstanding, with default QoS, the arrival probability is up to 90%.

1.2. Send data from IoT gateway to server

IoT gateway is an Android embedding device. As a consequence, students can get rid of actual hardware and utilize Android cell phones to implement a gateway. Steps to establish a connection from a phone to a group of sensors and wireless communication will be demonstrated thoroughly below.

2. Programming tutorial

2.1. Enlarge cellular phone's network

With a view to communicating with IoT devices, cell phones are required to broaden its network. In particular, the reason is that a system consisting of an operating system is incapable of handling real-time feature. Consequently, it's significant to append a microcontroller to our system. In fact, there is an abundance of prevalent microcontrollers at low cost, such as Arduino. Lying at the middle of an OS-system and the microcontroller is a connection, the simplest of which is USB UART.

A sample model comprising a sensor system, wireless communication and Gateway connection is illustrated below:

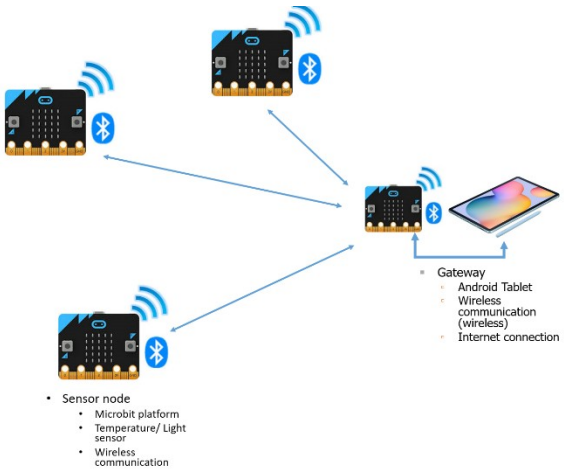

Figure 1: Sensor network connection

At this moment you might be wondering why there is a need to make an additional link between a cell phone or table and another microcontroller. The explanation is that mobile phone has no radiowave-based channels, dubbed as LoRa, which is specified for IoT network. Particularly, cell phone has to tie with external hardware component through USB OTG as such:

Figure 2: USB OTG extended for cellular phones

As a result, hardware connection plays a role as a middleware, transferring data from 1 node (sensor) to the phone. All of processing operations would be executed on the phone.

2.2. Advanced USB UART programming on Android Studio

In order to open a USB UART connection on Android mobile phone or tablet, you can refer to this github repository: https://github.com/mik3y/usb-serial-for-android

A shortened version could be found at this link:
https://www.dropbox.com/s/4045vcf1a90mg3t/TerminalUart.zip?dl=1

Unfortunately, might there appear several incompatibility bugs in Android Studio on your personal computer, or bugs caused by useless code segments, students are supposed to debug and eliminate on their own.

This is significant information:

Let’s supply libraries to build.graddle (app module)

```
android {

    allprojects {
        repositories {
            maven { url 'https://jitpack.io' }
        }
    }
}

dependencies {
        implementation 'com.github.mik3y:usb-serial-for-android:2.+'
}
```

Grant USB access rights in AndroidManifest, don’t forget to pay attention to device_filter file (located in res/xml/)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="uart.terminal.androidstudio.com.myapplication">
    <uses-feature android:name="android.hardware.usb.host" />
    <uses-permission android:name="android.permission.USB_PERMISSION" />

    <application

            <meta-data
                android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                android:resource="@xml/device_filter" />
        </activity>
    </application>

</manifest>
```

Beneath is the device_filter’s content, including general drivers of USB UART:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- 0x0403 / 0x6001: FTDI FT232R UART -->
    <usb-device vendor-id="1027" product-id="24577" />

    <!-- 0x0403 / 0x6015: FTDI FT231X -->
    <usb-device vendor-id="1027" product-id="24597" />

    <!-- 0x2341 / Arduino -->
    <usb-device vendor-id="9025" />
```

```
    <!-- 0x16C0 / 0x0483: Teensyduino  -->
    <usb-device vendor-id="5824" product-id="1155" />

    <!-- 0x10C4 / 0xEA60: CP210x UART Bridge -->
    <usb-device vendor-id="4292" product-id="60000" />

    <!-- 0x067B / 0x2303: Prolific PL2303 -->
    <usb-device vendor-id="1659" product-id="8963" />

    <!-- 0x1a86 / 0x7523: Qinheng CH340 -->
    <usb-device vendor-id="6790" product-id="29987" />

    <usb-device vendor-id="1155" product-id="22352" />

    <usb-device vendor-id="8208" product-id="30264" />

    <usb-device vendor-id="3368" product-id="516" />

</resources>
```

Grant privilege in MainActivity.java

```
PendingIntent usbPermissionIntent = PendingIntent.getBroadcast(this, 0, new
Intent(INTENT_ACTION_GRANT_USB), 0);
manager.requestPermission(driver.getDevice(), usbPermissionIntent);

manager.requestPermission(driver.getDevice(), PendingIntent.getBroadcast(this,
0, new Intent(ACTION_USB_PERMISSION), 0));
```

Attain information from hardware:

```
@Override
public void onNewData(final byte[] data) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            txtOut.append(Arrays.toString(data));
        }
    });
}
```

2.3. Subscribe and publish data to MQTT server

I would like to share with you some terms you might encounter while working with MQTT.

Publish and subscribe

In a server applying MQTT protocol, various sensor nodes, or MQTT client (or just client, for the sake of presentation), would make connection to an MQTT server (broker). Each client will subscribe to one or multiple topics, for instance “/client1/channel1”, “/client1/channel2”. Actually, this subscription process is analogous to Youtube subscription you have got familiar with. Every client receives data if any of the data points have data to send to the subscribed channel. Sending data is dubbed as “publish”.

QoS - Quality of Service

There are 3 QoS options during “publish” and “subscribe” progress:

- QoS0 Broker/client transfers data for only one time. Subsequently, sending process is confirmed by only TCP/IP protocol. It’s similar to the situation when you leave your girlfriend in the lurch.
- QoS1 Broker/client transfers data followed by at least 1 acknowledgement from the receiver, in other words there would be more than 1 message confirming that data is successfully obtained
- QoS2 Broker/client ensures that the destination only receives data once, requiring 4-step handshaking.

Clearly, the higher QoS, the longer the delay of package transferring. A package has the probability to be sent by any QoS, and clients have the capability of subscribing to any QoS requests. To state it differently, client would pick the best QoS from which it could seize information. For example, if a package is published with QoS2, and client subscribed to QoS0, broker will transmit this data with QoS0. Noticeably, another client subscribing to the same channel with QoS2 would receive information through QoS2.

Retain

If RETAIN is set to 1, broker is forced to save the package when it is published from Client. That package will be delivered to other client subscribing to the same channel in the future. When a client establishes a connection to Broker and subscribes, it would receive the ultimate package possessing RETAIN = 1 linked with all of the topics of duplicate subscriptions. Nonetheless, if a package with QoS = 1 and RETAIN = 1 is transmitted to Broker, it will eliminate all of the preceding packages storing RETAIN = 1. Furthermore, that package must be saved, although it could be deleted at some point in the future.

When publishing a package to Client, Broker has to set RETAIN = 1 if the package is the outcome of recent Client’s subscription (the same situation in which message notifies that subscription is successful). RETAIN has to be 0 if no consideration for subscription’s result is taken into.

Sample code exists at this url:

https://www.dropbox.com/s/r5n1p7u62he3s9u/CloudMQTT.zip?dl=1

Programming details are presented below.

2.3.1. Add WAKE_LOCK and MQTT service

An application making use of MQTT cloud needs WAKE_LOCK privilege and a service being executed hiddenly to listen to server. These operations are implemented in AndroidManifest.xml file. Please bear in mind 2 highlighted lines below, since they are 2 compulsory lines you need to supply to this file.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidthings.helloworld">
```

```
    <uses-permission
android:name="com.google.android.things.permission.USE_PERIPHERAL_IO"/>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application>
        <uses-library android:name="com.google.android.things" />

        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.IOT_LAUNCHER"
/>
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
        <service android:name="org.eclipse.paho.android.service.MqttService"/>
    </application>
</manifest>
```

Initially, because of dependency insufficiency, MqttService would alert a recognizable red error. Leave it as such, and continue progressing your study.

2.3.2. Add dependencies to MQTT

Library would be augmented into build.gradle (app Module), located inside Gradle Scripts. At first, you are demanded to add these 2 libraries to dependencies component:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-
core:3.0.2'
    compileOnly 'com.google.android.things:androidthings:+'

    implementation 'com.squareup.okhttp:okhttp:2.4.0'
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.0'
    implementation 'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
}
```

In succession, move up to android module, add repository to MQTT cloud service. Repositories would become obligatory when there pop up latest services from MQTT cloud, as such:

```
android {
    compileSdkVersion 26
    defaultConfig {...}
    buildTypes {...}

    repositories {
        maven {
            url "https://repo.eclipse.org/content/repositories/paho-snapshots/"
        }
    }
}
```

At last, click Sync Now with the aim of downloading fundamental dependencies. At present, MqttService in AndroidManifest would no longer raise any errors.

2.3.3. Add MQTTHelper

Returning to Java programming environment, we are in need of 1 more file, named as MQTTHelper, at the same level with MainActivity.java. So as to deal with this, right click on a higher level of MainActivity.java (application's package name), **choose New, and pick Java Class**, similar to below tutorial:

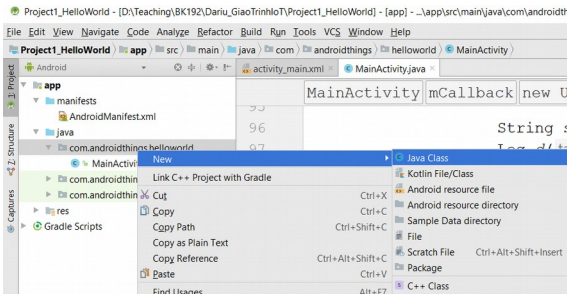

Figure 3: include a brand-new Java file

Another window would pop up, enter the file name and click OK.

A Java file would be generated, all of the code content is demonstrated below. If you wish to make a copy from this file, please start from **public class MQTTHelper.** In this file, there exists some final variables requesting for modification to make it compatible with our current server. Specifically,

- serverUri: path to server, for TCP protocol, with augmented Port information. These information is based on section 1, and server related information was displayed at the beginning of this report
- clientId: an identifier, you might set it for yourself. Nevertheless, please note that your identifier must be unique.
- subscriptionTopic: topic name from which subscriber desires to collect information. In this case, our device will subscribe to all of the channels pertaining to format "sensor/". Plus sign here means all of the characters are valid.

- username, password: authentication information into server, as shown in section 1.

While copying and pasting from this file, be cautious that it consists of several special characters (such as paging symbol). For simplicity, just add dependencies manually. A rule of thumb is to slowly paste code segments into your editor, instead of dealing with the whole file from the start.

```
import org.eclipse.paho.android.service.MqttAndroidClient;
import org.eclipse.paho.client.mqttv3.DisconnectedBufferOptions;
import org.eclipse.paho.client.mqttv3.IMqttActionListener;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.IMqttToken;
import org.eclipse.paho.client.mqttv3.MqttCallbackExtended;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class MQTTHelper {

    final String serverUri = "tcp://hairdresser.cloudmqtt.com:17681";

    final String clientId = "RP_001";
    final String subscriptionTopic = "sensor/+";

    final String username = "zwpkdaox";
    final String password = "0BxcZvUdbXkf";

    public MqttAndroidClient mqttAndroidClient;


    public MQTTHelper(Context context){
       mqttAndroidClient = new MqttAndroidClient(context, serverUri,
clientId);
        mqttAndroidClient.setCallback(new MqttCallbackExtended() {
            @Override
            public void connectComplete(boolean b, String s) {
                Log.w("mqtt", s);
            }

            @Override
            public void connectionLost(Throwable throwable) {

            }

            @Override
            public void messageArrived(String topic, MqttMessage mqttMessage)
throws Exception {
                Log.w("Mqtt", mqttMessage.toString());
            }

            @Override
            public void deliveryComplete(IMqttDeliveryToken
iMqttDeliveryToken) {

            }
        });
        connect();
```

```
    }

    public void setCallback(MqttCallbackExtended callback) {
        mqttAndroidClient.setCallback(callback);
    }

    private void connect(){
        MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
        mqttConnectOptions.setAutomaticReconnect(true);
        mqttConnectOptions.setCleanSession(false);
        mqttConnectOptions.setUserName(username);
        mqttConnectOptions.setPassword(password.toCharArray());

        try {

            mqttAndroidClient.connect(mqttConnectOptions, null, new
IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {

                    DisconnectedBufferOptions disconnectedBufferOptions = new
DisconnectedBufferOptions();
                    disconnectedBufferOptions.setBufferEnabled(true);
                    disconnectedBufferOptions.setBufferSize(100);
                    disconnectedBufferOptions.setPersistBuffer(false);
                    disconnectedBufferOptions.setDeleteOldestMessages(false);

mqttAndroidClient.setBufferOpts(disconnectedBufferOptions);
                    subscribeToTopic();
                }

                @Override
                public void onFailure(IMqttToken asyncActionToken, Throwable
exception) {
                    Log.w("Mqtt", "Failed to connect to: " + serverUri +
exception.toString());
                }
            });


        } catch (MqttException ex){
            ex.printStackTrace();
        }
    }

    private void subscribeToTopic() {
        try {
            mqttAndroidClient.subscribe(subscriptionTopic, 0, null, new
IMqttActionListener() {
                @Override
                public void onSuccess(IMqttToken asyncActionToken) {
                    Log.w("Mqtt","Subscribed!");
                }

                @Override
                public void onFailure(IMqttToken asyncActionToken, Throwable
exception) {
```

```
                    Log.w("Mqtt", "Subscribed fail!");
                }
            });

        } catch (MqttException ex) {
            System.err.println("Exceptionst subscribing");
            ex.printStackTrace();
        }
    }
}
```

2.4. Calling MQTT on MainActivity.java

Firstly, we call for object generation for MQTTHelper class, and a warmup method for MQTT service.

```
MQTTHelper mqttHelper;
private void startMQTT(){
    mqttHelper = new MQTTHelper(getApplicationContext());
    mqttHelper.setCallback(new MqttCallbackExtended() {
        @Override
        public void connectComplete(boolean b, String s) {

        }

        @Override
        public void connectionLost(Throwable throwable) {

        }

        @Override
        public void messageArrived(String topic, MqttMessage mqttMessage)
throws Exception {
                    }

        @Override
        public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {

        }
    });
}
```

Note that we're utilizing lambda function. Therefore, the system would generate respective override functions. Of all override functions are there ones which are ready to obtain data from subscribed topics. As a matter of fact, students need to process hardware controlling in **messageArrived** function. Lastly, let's call startMQTT in onCreate().

We might be able to construct a method transmitting data to MQTT server, including 2 parameters, which are ID and payload. For instance,

```
private void sendDataToMQTT(String ID, String value){

    MqttMessage msg = new MqttMessage();
    msg.setId(1234);
    msg.setQos(0);
```

```
    msg.setRetained(true);

    String data = ID + ":" + value;

    byte[] b = data.getBytes(Charset.forName("UTF-8"));
    msg.setPayload(b);

    try {
        mqttHelper.mqttAndroidClient.publish("sensor/RP3", msg);

    }catch (MqttException e){
    }
}
```

The most fundamental part in this method is our string concatenation from ID and value. It's not complicated to make this function complex. For instance, if ID=1, concatenate ID with Temperature, whereas if ID=2, let's merge ID with Light Level. Last but not least, data transmission would be wrapped in the try-catch segment, represented as a publish command. Inside this command is our naming for published channel as **sensor/RP3.** Moreover, students are compelled to modify information related to topics before publishing data to the server.