9
Monitoring Your Devices with MQTT

The goal of this chapter is to extend the Chronotherm application built in the previous chapters with a new feature that is capable of publishing the current collected temperature to a cloud web service. Moreover, we will create a new mobile monitoring application that shows published data to the users.

In this chapter, we will cover the following topics:

- Exploring the Internet of Things
- Working with the MQTT protocol
- Publishing Chronotherm data to a cloud web service
- Building a mobile app to read remote data

The Internet of Things

As an umbrella term, the **Internet of Things** (**IoT**) covers a broad number of applied domains, protocols, and applications that are related to the *interconnection of embedded and smart objects*. However, the IoT isn't just a set of technologies. We can see it as a vision in which the Internet extends itself into the real world, with the capability to control physical devices from the virtual world. Moreover, this grants users to query interconnected devices in a short time, obtaining useful information about their status and actions they're carrying out. This step is really important because it makes the *real world observable* through applications.

We may consider this the evolution of the Internet, which is grounded in the belief that in the foreseeable future, the number of connected devices will continue to grow thanks to the advances in engineering and microelectronics fields. Indeed, because these kinds of devices become smaller, cheaper, and with less energy consumption, they are already integrated in many everyday objects.

Imagine an alarm clock capable of sending a message to other home-connected devices to alert them when we wake up. Maybe the Web Radio application we built in *Chapter 5*, *Managing Interactions with Physical Components*, can switch on automatically after a few minutes and start playing our favorite songs. Perhaps the Chronotherm we finished building in *Chapter 8*, *Adding Network Capabilities*, can send the home temperature to our microwave oven, so it will change how hot to make our cup of tea. These improvements are just examples of what we will obtain if smart objects start to communicate with each other.

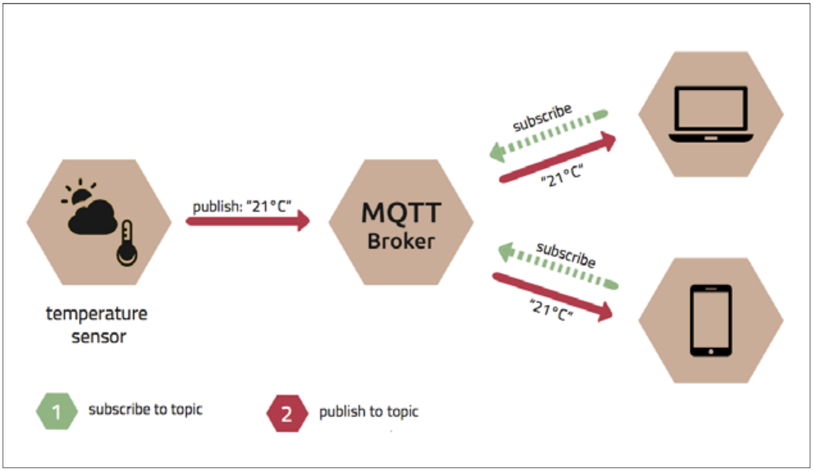
The MQTT protocol

We have many ways to interconnect our prototypes with other devices using private or public networks, such as the Internet. Whatever the communication channel, embedded and small devices need a well-defined lightweight protocol with a small code footprint, bandwidth-efficient, and with little battery consumption. There are many protocols capable of achieving the preceding requirements, but one of the most widespread at the moment is the **Message Queue Telemetry Transport** (**MQTT**) protocol.

This protocol uses the **publish/subscribe** paradigm in which the actors that send messages, called **publishers**, don't connect directly to specific receivers, called **subscribers**, but simply characterize their messages into classes without knowing who the recipients are. Indeed, an MQTT message includes a *topic* name that defines the message type, and this is used to distinguish this message among messages of different types. This implies that the publisher should assign a topic to every sent message, while the subscriber defines its interest to receive messages only for a particular topic.

Conversely to conventional point-to-point protocols, where a client interacts directly with a server (for example, a web server), the publish/subscribe messaging pattern makes it possible for two entities to exchange messages without being necessarily simultaneously online. This characteristic is realized through a third actor, called a **message broker**, with the responsibility to receive and dispatch messages between actors.

The broker could be a server, with or without authentication, that keeps track of the registered devices. Every time a new device registers itself with a broker, a TCP connection is kept alive until the subscriber or the publisher decides to disconnect. During message delivery, the broker plays an important role to *push* messages to all the subscribers that are interested in a particular message topic. The use of the MQTT protocol on our Chronotherm results in the following interaction schema that explains how messages are delivered between registered actors:



The Chronotherm, the laptop, and the mobile device register themselves with the MQTT broker, declaring their unique identifiers. The laptop and the mobile devices should provide their interest to listen to a particular topic that should be the same used by the Chronotherm application during the publishing phase. The message topic is a simple string that could have more hierarchy levels separated with a slash. This means if we want to publish the temperature of our living room, we can use a topic like this: `/home/living-room/temperature`. The hierarchy becomes really relevant when we subscribe our devices to more topics. If we suppose that we have another sensor in our kitchen, we will have a publisher that sends messages to the topic like this: `/home/kitchen/temperature`. In this case, if another device needs to subscribe to all temperature messages regardless of the location, we can subscribe it to multiple topics using a wildcard. MQTT provides two different wildcards:

- The + operator is a single-level wildcard that allows arbitrary values for one hierarchy. With the preceding example, the `/home/+/temperature` topic will subscribe to all temperature updates, both for the kitchen and for the living room.

- The `#` operator is a multilevel wildcard that allows subscribing to all the underlying levels. With the preceding example, the `/home/#` topic will subscribe to any topic that begins with the `/home` string.

In our case, the laptop and the mobile device must subscribe to the `/home/living-room/temperature` topic in order to receive temperature updates. Every time the Chronotherm detects a new temperature, it will publish the temperature to the broker with the living room topic so that it can push the published message to all the subscribed devices.

This architecture enables highly scalable and flexible solutions because the publishers and the subscribers are decoupled, and the ones that consume data are not directly related to the ones that produce data.

Preparing the cloud broker

The first step to let the Chronotherm prototype publish collected data through MQTT is to prepare and configure a message broker capable of managing our device's registrations and the publishing/subscribing phases. According to our needs, we can choose to prepare the message broker in a private network or in a public one such as the Internet. Whatever the performance and the network reliability that we need, we have at our disposal many open source projects that provides the MQTT protocol out of the box. If we want to manage a broker by ourselves, we can use **Mosquitto**, which is an open source project with full support for the MQTT protocol. You can find more information about the project at `http://mosquitto.org/`.

Anyhow, configuring a local server is not a simple task. Even if we are tempted to create our local broker in a self-hosted solution, we should take into account the drawbacks and implications of proceeding with this configuration. The installation of Mosquitto in a Windows or a Linux server operating system is greatly simplified nowadays, but keeping the system working requires other important tasks. If we want to access our broker when we aren't at home, our server should be reachable from the Internet using a proper configuration of our network together with a valid DNS entry. Furthermore, exposing a service to a public network could imply many security issues that require, for instance, a proper firewall configuration while mitigating any other threats caused by malicious users.

Many of the preceding issues can be resolved using a cloud **Virtual Private Server** (**VPS**) that simplifies the infrastructure layer of exposed services. Unfortunately, this approach will not mitigate all security concerns and still requires many system administration tasks to keep the software up to date while handling eventual incompatibilities caused by major releases of installed packages.

Whenever we're building our prototypes, to keep things simpler, it is always a good idea to delegate responsibilities and issues management to other services available on the Internet. For these reasons, and for the purpose of this chapter, we discard the idea of using a private broker in favor of a cloud-hosted solution; in our case, we will use the **CloudMQTT** web service.

The use of the CloudMQTT service is recommended for the purpose of this chapter because it's easy to use and it offers a free-tier that is good for the Chronotherm prototype. For your next project, you can freely choose any other cloud service or you can start using Mosquitto or any other open source projects to implement your own broker.

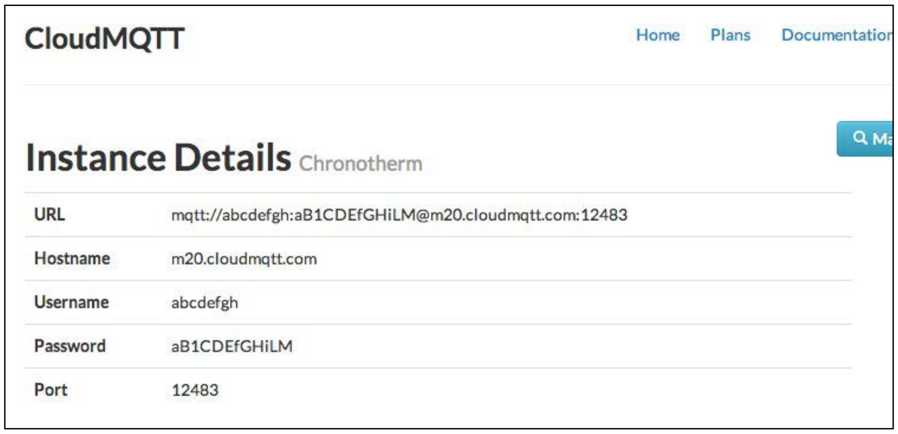
To begin the broker configuration, we have to register to the CloudMQTT service with the following steps:

1. Go to the service website `http://www.cloudmqtt.com`.
2. Select **Control Panel,** and in the **Sign up** form, write your e-mail before clicking on the **Sign up** button.
3. After a few minutes, you will receive the CloudMQTT service confirmation e-mail in your mailbox. Be sure to open the right e-mail and click on the *confirmation link* from your e-mail client.
4. You will be redirected to the **Create an account** form that you should compile with your own data.

Because we're going to use the available free-tier, the registration process will not require any sensitive information, such as your credit card details.

5. After the registration process is finished, we will be redirected to the **CloudMQTT Instances** screen, which lists all MQTT instances related to our projects. There, click on the **Create** button.
6. In the **Create new CloudMQTT Instance** form, write `Chronotherm` in the **Name** field and select the **Data center** nearest to your location. In the **Plan** field, be sure to select the **Cute Cat** plan, which is the free-tier version of the service. When you've finished, click on the **Create** button.
7. After the creation process, we will be redirected back to the instances list, but this time, we should find the **Chronotherm** instance. To get the access parameters, click on the **Details** button.

There, we will find all the information needed to let our publishers and subscribers connect to our broker. In particular, the values that we have to bear in mind are the **Username** and the **Password** because CloudMQTT uses basic authentication to provide access to their services. If you're going to use this service for your projects, remember to keep these values safe. The following screenshot is an example of a Chronotherm broker created with the CloudMQTT service:



Instance Details Chronotherm

URL	mqtt://abcdefgh:aB1CDEfGHiLM@m20.cloudmqtt.com:12483
Hostname	m20.cloudmqtt.com
Username	abcdefgh
Password	aB1CDEfGHiLM
Port	12483

Publishing Chronotherm data

With a working broker ready to accept connections, we can proceed with the Chronotherm physical application and start publishing the collected temperatures. This part consists of two main tasks:

- Implementing an abstraction to use the MQTT protocol
- Changing the `DataReader` class so that it will publish the temperature updates periodically

The first task could be easily achieved if we make use of a third-party library that implements the MQTT flow. For the scope of this chapter, we're going to use the **Paho library**, which is one of the most used open source client implementations for the MQTT protocol. In our case, we declare the Paho dependency in the Gradle build system, so the library classes will be available in the whole application.

[One of the most interesting characteristics of the Paho library is that it has many implementations for different programming languages. Indeed, it's available in C, C++, Java, JavaScript, Python, Go, and C#. An Android `Service` utility is also available, but we will use the plain Java implementation to better understand how the MQTT protocol works.]

To add Paho to our dependencies, perform the following steps:

1. In the `/build.gradle` file, add the Eclipse Maven repository inside the `repositories` parameter of the `allprojects` property:

   ```
   allprojects {
     repositories {
       jcenter()
       maven { url
       'https://repo.eclipse.org/content/repositories/paho-
       releases/' }
     }
   }
   ```

2. In the `/app/build.gradle` file, add the Paho dependency:

   ```
   dependencies {
     compile fileTree(dir: 'libs', include: ['*.jar'])
     compile 'com.android.support:appcompat-v7:21.0.3'
     compile
     'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.0.1'
     compile 'me.palazzetti:adktoolkit:0.3.0'
   }
   ```

3. Click on the **Sync now** button.

The next step is to create an abstraction for the Paho library, so we can easily reuse the initialization code later. To begin the implementation of this utility class, create a new package called `mqtt` in your namespace and add a new class named `MqttConnector`. This class should contain all connection settings, as you can see in the following code:

```
public class MqttConnector {
  private final static String BROKER_URL =
  "tcp://m20.cloudmqtt.com:12483";
  private static final String USERNAME = "abcdefgh";
  private static final String PASSWORD = "aB1CDEfGHiLM";
  private MqttConnectOptions mOptions;
  private MqttClient mClient;
  private String mTopic;
}
```

We set the `BROKER_URL` variable, according to the CloudMQTT parameters, using a TCP connection with the port at the end of the string. We also declare the instance credentials `USERNAME` and `PASSWORD` together with the `mOptions`, `mClient`, and `mTopic` parameters, which define the connection options, the `MqttClient` instance used to communicate with the broker, and the topic that will be used during the publishing and subscribing phases.

We proceed with the class constructor that should be defined as follows:

```
public MqttConnector(String clientId) throws MqttException {
  mOptions = new MqttConnectOptions();
  mOptions.setCleanSession(true);
  mOptions.setUserName(USERNAME);
  mOptions.setPassword(PASSWORD.toCharArray());
  MemoryPersistence persistence = new MemoryPersistence();
  mClient = new MqttClient(BROKER_URL, clientId, persistence);
}
```

We expect as an argument a unique identifier that will be used by the broker to recognize this device from the connection pool. We proceed by defining the connection options that are related to the basic authentication. The clean session flag is used during the connection establishment and defines how the broker treats the session with our client. It works as follows:

- If the flag is set to `False`, the system should persist the subscriptions between sessions and the device doesn't have any need to resubscribe to the chosen topics again during a reconnection
- If the flag is set to `True`, the client will have to resubscribe to the interested topics every time they reconnect

For the purpose of our application, we set the flag to `True`, avoiding the session persistence in the broker. In the last part, we declare the persistence mechanism used in the `MqttClient` instance to store the current session and MQTT messages that are sent or received to avoid duplicates. Using a file to achieve message persistency is a common approach followed by many applications that use the Paho library. However, to keep our application simple but less reliable, we will use a `MemoryPersistence` object.

Now, we should provide the following proxy methods to connect and disconnect from the broker. At the end of the class, add the following methods:

```
public void connect() throws MqttException {
  mClient.connect(mOptions);
}

public void disconnect() throws MqttException {
  mClient.disconnect();
}
```

In the `connect()` method, we have to be sure to use the `mOptions` object, otherwise the `CloudMQTT` instance will deny access for missing credentials. We also need two utilities that define the topic we're using during the publishing or the subscribing calls, and to define the `MqttCallback` interface called by the Paho library when our device receives a message from the broker. At the bottom of the class, add the following utility methods:

```
public void setTopic(String topic) {
  mTopic = topic;
}

public void setCallback(MqttCallback callback) {
  mClient.setCallback(callback);
}
```

The use of the `setTopic()` method is only to simplify the usage of our Paho wrapper. In real-world applications, you can publish and subscribe to different topics using the same connection. Indeed, your device could subscribe to five different topics, and in the same instance, it could publish some computed values to other topics.

Now we should put the two missing parts into the class. The first one is the `publish()` method that can be implemented with the following code:

```
public void publish(byte[] message) throws MqttException {
  MqttMessage mqttMessage = new MqttMessage();
  mqttMessage.setPayload(message);
  mqttMessage.setQos(0);
  mClient.publish(mTopic, mqttMessage);
}
```

Because the message payload is just a sequence of bytes, we expect a `message` instance represented by an array of bytes. We initialize a new `MqttMessage` object with the `message` payload and then publish the message to the broker using the chosen `mTopic` variable. One of the most important features related to MQTT is the capability to set the **Quality of Service** (**QoS**) on a per-message basis. Through this, we can choose between three different types of message deliveries:

- The value `0` indicates that the message must be delivered at most once. The message isn't persisted and will not be acknowledged across the network. This is the fastest but least reliable QoS because it's possible that the message isn't delivered due to a network error.
- The value `1` indicates that the message must be delivered at least once. The message is persisted in the device and it will be acknowledged across the network. This is the default QoS.

Under some circumstances related to an unreliable network, the publisher could send duplicates to the broker multiple times. This occurs when the connection drops after the broker sends back a `PUBACK` message to the publisher stating that the message is received but before the client receives it. In this case, the publisher will resend the message again after the reconnection, and the broker will treat it as a new flow sending out a duplicate. When we choose QoS 1 for our messages, we should always bear in mind that this case could happen.

- The value `2` indicates that the message must be delivered exactly once. The message is persisted in the device and it will be subject to a two-phase acknowledgement across the network. This is the most reliable QoS but waits for the two-phase acknowledgement with the broker.

Because we don't use any persistence mechanism and we don't need a high reliability, we use the fastest but less reliable QoS. The last step to conclude the class implementation is to add the `subscribe()` method for a chosen `topic` with the following code:

```
public void subscribe() throws MqttException {
  mClient.subscribe(mTopic);
}
```

This concludes the `MqttConnector` class implementation, and now we can proceed with adding the MQTT flow to our Chronotherm application.

We've implemented the `MqttConnector` class to provide an easy-to-read class that shows how the Paho library works. In your next prototypes, you could avoid creating this layer and use the library directly from your code.

Now we can proceed with publishing the collected temperatures while using the `MqttConnector` class in the `DataReader` class. Our approach is to open the MQTT broker connection just before the startup of our `SensorThread()` method so that it can start sending detected temperatures after the value has been retrieved through the ADK. We will close the broker connection after we shut down all schedulers.

To achieve this implementation, we should update our code through the following steps:

1. At the top of the `DataReader` class, add the highlighted declaration:

   ```
   private AdkManager mAdkManager;
   private MqttConnector mMqttClient;
   private Context mContext;
   ```

2. In the `start()` method of the class, before the scheduler's initialization, add the client connection with the broker that will identify this device as a Chronotherm, while setting up the publishing topic for our hypothetical living room:

   ```
   public void start() {
     try {
       mMqttClient = new MqttConnector("chronotherm");
       mMqttClient.setTopic("/home/living-room/temperature");
       mMqttClient.connect();
     }
     catch (MqttException e) {
       // manage connection errors
     }
     // Start thread that listens to ADK
     // ...
   }
   ```

3. Handle the disconnection in the `stop()` method, when the MQTT broker connection is no longer required:

   ```
   public void stop() {
     mSchedulerSensor.shutdown();
     mSchedulerWeather.shutdown();
     try {
       mMqttClient.disconnect();
     }
     catch (MqttException e) {
       // manage exceptions
     }
   }
   ```

4. Inside the `SensorThread` `run()` method, add the highlighted code to publish the detected temperature to our broker:

```
@Override
public void run() {
  Message message;
  // Reads from ADK and check boiler status
  AdkMessage response = mAdkManager.read();
  float temperature = response.getFloat();
  boolean status = isBelowSetpoint(temperature);
  try {
    mMqttClient.publish(response.getBytes());
  }
  catch (MqttException e) {
    // manage exceptions
  }
  // Updates temperature back to the main thread
  // ...
}
```

With this last step, we can update the physical application version to `0.4.0` in the `/app/build.gradle` file and upload the Chronotherm application to start publishing detected temperatures to the CloudMQTT broker.

Writing a mobile app to read published data

Now that we have a Chronotherm application that publishes data to our cloud broker, we can proceed with writing a mobile Android application that reads the detected temperature published by the Chronotherm. To begin the implementation of a simple monitoring system, we need to start a new Android project with the following steps:

1. From Android Studio, select **File** and then **New Project**.
2. In the **Application name** option, choose **ChronothermSub**.
3. When you have to choose the current SDK, select a valid SDK for your owned Android mobile device. If you don't own an Android device or if you want to use the built-in Android emulator, choose the **API level 19** option.

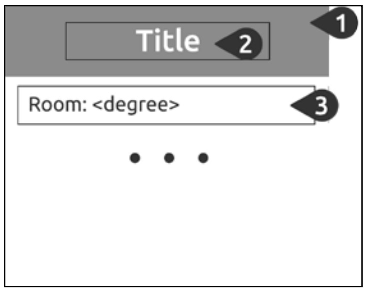
If you don't own an Android device or you don't want to use it, you can find more information about how to use the Android emulator at `http://developer.android.com/tools/devices/emulator.html`.

4. When you need to choose the activity type, select **Blank Activity**.
5. In the **Activity name** option, choose **Subscriber** and click on **Finish**.

Before we begin to work on the application layout, we should add to the `AndroidManifest.xml` file. Just before the `<application>` tag, add the following permission to use the Internet in our application:

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

We can now proceed with changing the default layout to realize an activity that should provide all details regarding the current detected temperature in our living room. All the required components can be summarized in the following mock-up that defines in which order the elements will be created:



According to the preceding layout, we should:

1. Create a background frame with a different color to provide a block in which we can put the title of our application.
2. Add the title of this application at the top of the layout.
3. Provide the list of all the connected devices indicating the name where they are located. In our case, we will only add one entry that indicates the detected temperature for our living room. The `degree` value will be updated using the MQTT broker connection through the same topic defined in the Chronotherm application.

Keeping in mind this layout design, we can proceed with replacing the standard theme with the following steps:

1. In the `res/values/dimens.xml` resource file, add the following definitions:

   ```
   <resources>
     <dimen name="activity_frame_height">80dp</dimen>
     <dimen name="list_margins">16dp</dimen>
     <dimen name="title_size">30sp</dimen>
     <dimen name="body_size">20sp</dimen>
   </resources>
   ```

2. In the `res/values/styles.xml` resource file, add the following colors:

   ```
   <resources>
     <color name="picton_blue">#33B5E5</color>
     <color name="white">#FFFFFF</color>
     <style name="AppTheme"
     parent="Theme.AppCompat.Light.DarkActionBar">
     </style>
   </resources>
   ```

3. In the `res/layout/activity_subscriber.xml` file, replace the default `LinearLayout` view with the following:

   ```
   <LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
     xmlns:tools="http://schemas.android.com/tools"
     android:orientation="vertical"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     tools:context=".Subscriber">
   </LinearLayout>
   ```

4. Create a `FrameLayout` view nested in the root `LinearLayout`:

   ```
   <FrameLayout
     android:layout_width="match_parent"
     android:layout_height="@dimen/activity_frame_height"
     android:background="@color/picton_blue">

     <TextView
       android:text="Home monitor"
       android:gravity="center"
       android:textColor="@color/white"
       android:textSize="@dimen/title_size"
       android:layout_width="match_parent"
       android:layout_height="match_parent" />
   </FrameLayout>
   ```

5. Create a new `LinearLayout` nested in the root `LinearLayout` that will contain all the connected physical devices:

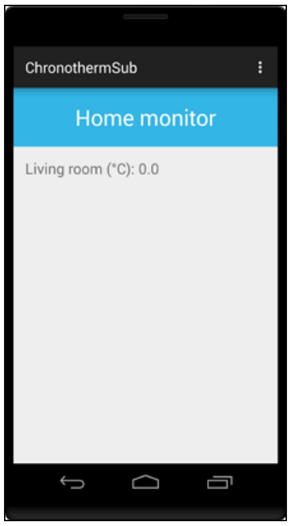
```
<LinearLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingLeft="@dimen/list_margins"
  android:paddingRight="@dimen/list_margins"
  android:paddingTop="@dimen/list_margins"
  android:paddingBottom="@dimen/list_margins">
</LinearLayout>
```

6. In the preceding `LinearLayout`, add the following views that show our living room temperature:

```
<TextView
  android:text="Living room (°C): "
  android:textSize="@dimen/body_size"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content" />

<TextView
  android:id="@+id/living_room"
  android:text="0.0"
  android:textSize="@dimen/body_size"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content" />
```

According to the previously defined styles, the following is the expected layout:



Living room (°C): 0.0

Subscribing to the Chronotherm topic

Now that we have a proper layout, we should start writing the code to register the mobile application with our broker subscribing it to the temperature topic. Like we did for the Chronotherm prototype in this chapter, we should add the Paho library dependency through the following reminders:

1. In the `/build.gradle` file, add the Eclipse Maven repository.
2. In the `/app/build.gradle` file, add the Paho dependency.
3. Click on the **Sync now** button.

Then, we should import the `MqttConnector` class we wrote for the Chronotherm physical application so that we can reuse its code in the subscriber. To achieve this step, create the `mqtt` package in your application namespace and copy and paste the `MqttConnector.java` file from the Chronotherm application to the mobile application.

Like we did in the Chronotherm application, we should provide a different thread that connects with the broker and listens to the incoming messages. We can achieve this by creating an `AsyncTask` thread that will use the `MqttConnector` class. However, when we subscribe the application to a particular topic, we should provide a proper class instance that implements the `MqttCallback` interface, used when a new message is received. For this purpose, we add this interface to our `Subscriber` class through the highlighted code:

```
public class Subscriber extends ActionBarActivity implements
MqttCallback {
  private TextView mLivingRoom;
  private Handler mHandler;
  // ...
}
```

In this part, we're defining an `mHandler` object that we will use to dispatch the message from the background thread to the main UI thread. During the activity creation, we can get the reference for the `mLivingRoom` `TextView`, providing the implementation for the `mHandler` object at the same time. In the `onCreate()` activity callback, add the highlighted code:

```
setContentView(R.layout.activity_subscriber);
mLivingRoom = (TextView) findViewById(R.id.living_room);
mHandler = new Handler(Looper.getMainLooper()) {
  @Override
  public void handleMessage(Message msg) {
    mLivingRoom.setText(msg.obj.toString());
  }
};
```

When the mHandler object receives a new message from the background thread, we will change the mLivingRoom text using the received Message object.

> The Message object should not be confused with the MqttMessage object that we will use next. The Handler and the Message classes are part of the Android system's framework for managing threads and are not related to the MQTT protocol. If you need more information about how to communicate with the UI thread, you can find the official Android documentation at https://developer.android.com/training/multiple-threads/communicate-ui.html.

To implement the MqttCallback interface, we should override the following listed methods:

```
@Override
public void messageArrived(String topic, MqttMessage mqttMessage)
{
  Message message = mHandler.obtainMessage(0, mqttMessage);
  message.sendToTarget();
}

@Override
public void connectionLost(Throwable throwable) {
  Toast.makeText(this, "Connection lost with the broker",
  Toast.LENGTH_SHORT);
}

@Override
public void deliveryComplete(IMqttDeliveryToken token) {
  // noop
}
```

The messageArrived() method is called when we receive a new message and it provides the received MQTT message with its topic. In our case, we should create a Message object to dispatch the mqttMessage instance to the main thread. The connectionLost() and the deliveryComplete() callbacks are called, respectively, to manage any disconnections with the broker, and to manage all acknowledgments received during network delivery, according to the chosen QoS.

The last step to complete our application is to create the `AsyncTask` class to receive messages from the cloud broker. To achieve this missing part, perform the following steps:

1. In the `mqtt` package, create the `MqttReceiver` class.
2. In the `MqttReceiver` class, add the highlighted code to extend the `AsyncTask` class:

   ```
   public class MqttReceiver extends AsyncTask<Void, Void, Void> {
     private Subscriber mCaller;
   }
   ```

3. Add the following class constructor that stores the `Subscriber` activity reference:

   ```
   public MqttReceiver(Subscriber caller) {
     this.mCaller = caller;
   }
   ```

4. Implement the following `doInBackground()` method:

   ```
   @Override
   protected Void doInBackground(Void... params) {
     try {
       MqttConnector client = new MqttConnector("receiver");
       client.setTopic("/home/living-room/temperature");
       client.connect();
       client.setCallback(mCaller);
       client.subscribe();
     }
     catch (MqttException e) {
       // manage connection errors
     }
     return null;
   }
   ```

   When the `AsyncTask` class is executed, we initialize the `MqttConnector` class using the `receiver` unique identifier. We set the same topic name used in the Chronotherm application, after which we connect to the broker. As last steps, we set the `MqttCallback` parameter using the caller activity and then we `subscribe` the device with the chosen topic.

Now that our `AsyncTask` class is ready to receive MQTT messages, we should start the background thread when the activity runs. In the `Subscriber` class, add the following activity callback:

```
@Override
protected void onResume() {
  super.onResume();
  new MqttReceiver(this).execute();
}
```

If two devices with the same identifier connect to the broker, the last connected device will grab the connection, disconnecting the others. Because we're reading MQTT messages through the `MqttReceiver` thread, if multiple threads run together, only the last one will receive the proper notifications while the others will terminate their execution.

After we've finished this missing part, we can upload this application to our Android smartphone or emulator, and if the Chronotherm and the mobile application are connected to the Internet, we will see temperature updates from our mobile device.

Summary

In this chapter, we explored the world of the *Internet of Things*, in which physical devices communicate with each other providing useful information to users and a new kind of interaction when they aren't at home. We took a look at the MQTT protocol, which is one of the most used protocols to exchange messages between devices. Before starting the implementation of the protocol, we registered with a cloud broker, compatible with MQTT, that provides basic authentication for our clients out of the box.

Then, we used the Paho library as our MQTT implementation, and after a little exploration of its powerful APIs, we added the MQTT capabilities to our Chronotherm, while publishing temperature updates to subscribed devices. To provide a simple monitoring system for our users, we built a new standalone mobile application that is subscribed to the Chronotherm updates.

Now that we have accurately designed our first IoT device, we may start thinking about our next prototype and how it can play a part in a larger ecosystem of devices. An important step that we should take in account when we want to expose a feature through a physical device and the MQTT protocol, is to provide a documentation for our published data and topics. This approach will grant other developers the capability to write their third-party applications or prototypes around your idea, and this could be the key factor for the success of your devices.