Lab 4: MQTT-based Twitter App Part II

Design and Development of Mobile Applications
E. De Coninck, S. Leroux, P. Simoens
2015-2016

In this lab session we will finish the twitter application designed in last session. We will be using the MQTT messaging protocol for mobile-cloud communication. In addition, we will learn to work with `Service` and `Notification` components.

1 MQTT protocol

MQTT is a lightweight message queuing protocol that was originally designed for machine-to-machine communication. The protocol was incepted to run on computationally constrained sensor devices with restricted power with extremely low and brittle wireless bandwidth availability. As of today, MQTT is one of the dominant protocols in the Internet of Things.

The small footprint on devices, the low power usage, the minimised data packets and the efficient distribution of information to one or many receivers makes MQTT also ideal for mobile applications. In addition, deploying such a small footprint in a data center results in a very dense, reliable and fast communication platform. These advantages for both mobile and cloud side of the communication were also acknowledged by Facebook, who decided to adopt MQTT in the Facebook Messenger chat application[1].

1.1 Topic-based messaging model

The MQTT messaging model is organised along the principles of publish-subscribe, as illustrated in the figure below.

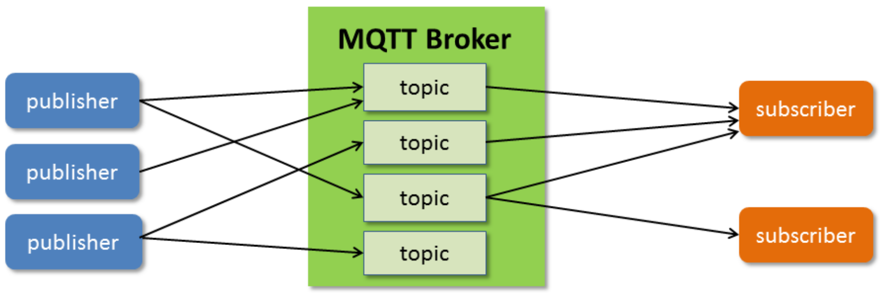

Figure 1: MQTT follows a pub-sub pattern structured along topics.

The MQTT protocol is based on the principle of publishing messages and subscribing to **topics**. Multiple clients connect to a broker and subscribe to topics that they are interested in. Clients also connect to the

[1]https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920

broker to publish messages on specific topics. Many clients may subscribe to the same topics and do with the information as they please. The broker and MQTT act as a simple, common interface for everything to connect to.

There is no need to configure a topic, publishing on it is enough. Topics are treated as a hierarchy, using a slash (/) as a separator. Some examples of topics: /home/livingroom/bulb1/status, /home/door/sensor/battery, /home/door/sensor/battery/units.

Messages are always published to a specific topic, but subscriptions can use various forms of wild cards.

Two wildcards are available:

- the plus sign (+) can be used as a wildcard for a single level of hierarchy. If a message is published to a topic "a/b/c/d", then the following example subscriptions will match: a/b/c/d, +/b/c/d, a/+/c/d, a/+/+/d, +/+/+/+. The following subscriptions will not match: a/b/c, b/+/c/d, +/+/+
- the hash sign (#) can be used as a wildcard for all remaining levels of hierarchy. This means that it must be the final character in a subscription. With a topic of "a/b/c/d", the following example subscriptions will match: a/b/c/d, #, a/#, a/b/#, a/b/c/#, +/b/c/#.

Zero length topic levels are valid, which can lead to some slightly non-obvious behaviour. For example, a topic of "a//topic" would correctly match against a subscription of "a/+/topic". Likewise, zero length topic levels can exist at both the beginning and the end of a topic string, so "/a/topic" would match against a subscription of "+/a/topic", "#" or "/#", and a topic "a/topic/" would match against a subscription of "a/topic/+" or "a/topic/#".

1.2 Quality of Service levels

MQTT defines three levels of Quality of Service (QoS). The QoS defines how hard the broker/client will try to ensure that a message is received. Messages may be sent at any QoS level, and clients may attempt to subscribe to topics at any QoS level. This means that the client chooses the maximum QoS it will receive. For example, if a message is published at QoS 2 and a client is subscribed with QoS 0, the message will be delivered to that client with QoS 0. If a second client is also subscribed to the same topic, but with QoS 2, then it will receive the same message but with QoS 2. For a second example, if a client is subscribed with QoS 2 and a message is published on QoS 0, the client will receive it on QoS 0.

Higher levels of QoS are more reliable, but involve higher latency and have higher bandwidth requirements.

- 0: The broker/client will deliver the message once, with no confirmation.
- 1: The broker/client will deliver the message at least once, with confirmation required.
- 2: The broker/client will deliver the message exactly once by using a four step handshake.

1.3 Last Will and Testament (LWT) message

Last Will and Testament or in short LWT is a setting of the MQTT client which helps detecting failures to other clients. If the client has set a LWT topic and message on connect, the broker will send that to the specified topic, which allows the client to notify others about its failure.

1.4 Retained messages

All messages may be set to be retained. This means that the broker will keep the message even after sending it to all current subscribers. If a new subscription is made that matches the topic of the retained message, then the message will be sent to the client. This is useful as a "last known good" mechanism. If a topic is only updated infrequently, then without a retained message, a newly subscribed client may have to wait a long time to receive an update. With a retained message, the client will receive an instant update.

2 Updated Requirements and topic definition

The Twitter-like application must meet the following requirements:

- Show online/offline state of all app users.
- Subscribe to a user feed.
- Post messages to your feed.
- Save messages received from broker to `ContentProvider`. Messages which are send before subscribing can not be received and unsaved messages can not be reacquired from the broker.

We provide you with a MQTT broker and `ContentProvider`, so you can focus on the development of the mobile app. Before we start coding, we need to agree on a common topic structure so you can test the communication of your app with the apps build by other students.

Discussion: What is a good topic structure? How can we accomplish user feeds and online/offline state?

3 Twitter-like application: part II

Start from the code provided on Minerva (solution of lab 3 with TODOs for this session). Download source code and **import project** into Android Studio. The `MainActivity` will show all users connected or disconnected from the MQTT broker. The `DetailActivity` will show the message feed of the selected person. The start code contains two extra dependencies configured in the `gradle.build` file. These libraries will be automatically downloaded and cashed by Android Studio.

org.eclipse.paho.client.mqttv3 This library is an implementation of a MQTT client which can be used to connect to a broker, to subscribe to a topic and to publish messages. You can find sample code on `http://www.eclipse.org/paho/clients/java#getting-started`.

org.eclipse.paho.android.service The Paho Android Service is an interface to the Paho Java MQTT client library that provides a long running service for handling sending and receiving messages on behalf of Android client applications when their main activity may not be running.

The Paho Android Service provides an asynchronous API (`http://www.eclipse.org/paho/files/android-javadoc/index.html`). To use the asynchronous Android service you need to use the `MqttAndroidClient` instead of the `MqttClient` class from the paho sample. The major difference is the requirement of an asynchronous callback method for the client. Subscribing and publishing to a topic will fail if the `ImqttActionListener` is not passed. There are three callback interfaces which can be interesting: `IMqttActionListener` (connection succeeded or failed), `MqttCallback` (connection lost, message arrived and message delivered) and `MqttTraceHandler` (trace information).

Question: We need to access the MQTT client application wide. How can we accomplish this? What is the best approach?

- Update the provided `MqttHandler` class which handles the Mqtt communication and implement the Mqtt callback interfaces. Create and connect the client when the app is started. The `MqttAndroidClient` requires a `Context` which is not available in a normal Java class. Find a way to get the `ApplicationContext` which is always available when your app process is running. This is important because an activity's `Context` is cleaned up when the activity is destroyed.

  Hint: Look at the provided MyApplication class.

  - Add `ACCESS_NETWORK_STATE`, `INTERNET` and `WAKE_LOCK` permissions to the manifest file. This is needed because we want to access a server on the network and an intent service will receive message even when the application is in the background.
  - Create `MqttAndroidClient` with `ApplicationContext`, a unique client id (your UGent mail address) and the server url (`tcp://<server-ip>:1883`)
  - Set callbacks and connection options. When a client disconnects we want to remember the client on the server so do not clean up the session (`setCleanSession`). To inform other clients of your offline state you can set a LWT message before connecting to the broker. This LWT message should post to the state topic of your user.
  - Connect to the server and on success send your online state to the state topic and subscribe to the state topic of all users.

Question: Can we subscribe to all topics and if this is possible why is this a bad idea?

- To receive messages from other users we use the paho android library. We need to add the intent service from this library to the app manifest file.
- Implement the `messageArrived` method of the `MqttCallback` interface to handle state updates (client online or offline messages) and new messages. Insert the new data into the content provider using the context's content resolver. The contact and message table url is accessible through the `MessageProvider` class. Use the tables column names as the `ContentValues` keys (`DatabaseContract.Message|Contact`). Inserting new data in the content provider will update the list in `MainFragment`.

  Hint: To differentiate between state message or normal message you can inspect the topic name.
- Update the `DetailFragment` so when the users feed is selected in the list subscribe to the contacts topic (otherwise you won't receive messages to this topic).
- Update sending messages to your feed so the message is forwarded to the correct topic and other users can receive your messages.

Question: Why is this application not ready for production? Can we fix this with MQTT?