Lab 4: Preparation

Design and Development of Mobile Applications
E. De Coninck, S. Leroux, P. Simoens
2014-2015

1 Primer on the 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.

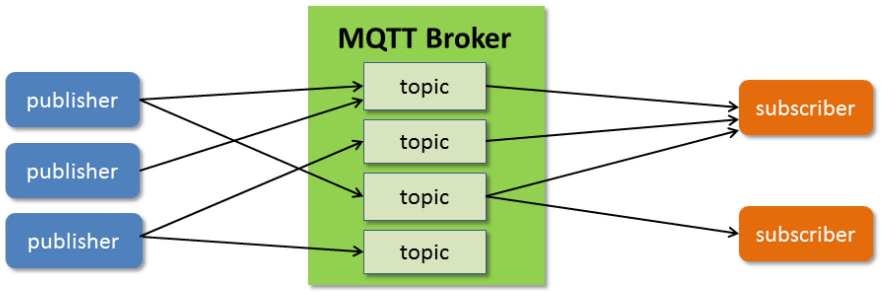

Figure 1: MQTT follows a pub-sub pattern structured along topics.

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 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.

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

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 (LWT) allow MQTT to declare what message should be sent on it's behalf by the broker, after it has gone offline. The analogy is that of a real last will: If a person dies, she can formulate a testament, in which she declares what actions should be taken after she has passed away. An executor will heed those wishes and execute them on her behalf. Last will statements are in the form of [topic: '/some/topic', message: 'some message'] and set by the client.

During normal operation, a client will keep the connection to the MQTT-broker open by sending periodic *keepAlive* messages interspersed with the actual messages. If the client goes offline, the connection to the broker will time out, due to the lack of *keepAlive* messages. This is where LWT comes in: if no LWT is specified, the broker doesn't care and just closes the connection. With a LWT however, the broker will execute the client's last will and publish the LWT message on the LWT topic that was configured by the client.

1.4 Retained messages

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 this 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 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 will provide you with a MQTT broker and `ContentProvider`, so you can focus on the development of the mobile app. First 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?