KV6006 practical session - 3 - MQTT

Arguably, the core of Internet of Things systems is the message-passing protocol MQTT. The letters used to stand for something like 'Message Queue Telemetry Transport' but there was never really a 'queue'. Officially it's no longer even pretending to be an acronym.

Topics and Payloads

An MQTT message originates from a data source, which publishes it to a 'broker' – server – which in turn retransmits the message to anything that's subscribed to listen to it. Messages are sent on specific 'topics', which are named like:

/Northumbria/City_Campus/Ellison_Building/E_Block/305

The topic doesn't need to already exist before a client publishes or subscribes to it. Subscribers can specify wildcards:

/Northumbria/City_Campus/+/E_Block/+

The + is a single-level wildcard, so this would receive messages about any building on City Campus that has an 'E_Block', and for any room in that building.

/Northumbria/City_Campus/#

matches any number of levels, so this would match *everything* on City Campus. On a busy MQTT broker, /# might involve quite a lot of messages.

Subscribing to MQTT messages

We're going to use the Paho MQTT client library in Python, and we've a little bit of setup to do. In a fresh Thonny document:

```
# examples/mqtt_subscribe/mqtt_subscribe.py
import paho.mqtt.client as mqtt
import config
import json

def on_connect(client, userdata, flags, rc):
    """Callback for when mqtt client connects."""
    print("Connected with result code: " + str(rc))
    # For now, subscribe to the firehose
```

```python
    client.subscribe("/KV6006/#")

def on_message(client, userdata, msg):
    """Callback for when mqtt client receives a message."""
    topic = msg.topic
    # Decode the payload: convert it from bytes to a string
    payload = str(msg.payload.decode("utf-8"))
    # For now, output the message
    print(topic + "  " + payload)

def on_log(mqttc, obj, level, string):
    print(string)

# Now we have our callback functions for connection and message receipt,
# we can go ahead and make a connection.
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.username_pw_set(config.mqtt_username, config.mqtt_password)

try:
    client.connect("connect.nustem.uk", 1883, 60)
    print("MQTT connection successful.")
    client.loop_forever()
except:
    print("MQTT connection failed.")
    exit(1)
```

You'll notice the import from `config.py`: this file should already be in your `student_work` directory. It contains the username and password for the MQTT broker we're using.

If all goes well, the `client.loop_forever()` line will keep the program running. When a message is received, `on_message()` is called to handle it - you should see messages being output.

If the connection drops the program will likely crash; in a real-world situation you'd want to be a bit more robust in your error handling. That said, I've had very similar code to the above running on a server for almost a year without issue.

Parsing data received over MQTT

You'll notice a steady stream of data on the `/KV6006/Sensors` topic. This is coming from a device at the front of the room: a Raspberry Pi Pico W microcontroller connected to a bunch of different sensors. A MicroPython script on the Pico polls data from the sensors, packages it as JSON, and broadcasts it over MQTT every second.

The Pico is connected to a WiFi network from a 4G mobile router. The MQTT broker is physically in London.

Since the sensors data feed is formatted as JSON, you can expand your `on_message()` function to do something like:

```
# [...]
payload = msg.payload.decode("utf-8")
# Parse  the string as JSON
data = json.loads(payload)

# Now we can iterate over the sensor data:
for sensor in data['sensors']:
    print("Sensor: " + sensor['name'])
    print("Value: " + str(sensor['value']))
```

The sensor `value` is of type `float` or `int`: in some case you may need to explicitly convert it to a string with `str()`.

> *NOTE: the Paho MQTT package we're using suppresses error messages in the `on_message()` callback. Which makes debugging infuriatingly difficult. You can work around this by adding `client.on_log = on_log` to the configuration after the `client.on_connect` and `client.on_message` lines. However, you'll see a lot of diagnostics.*

Getting just some of the data

Suppose you wanted just the temperature sensor data. Unfortunately, there isn't a useful key you can access: something like `data['sensors']['temperature']` won't work. It's possible my Python-fu just isn't strong enough, but I think you'd have to do something like:

```
for sensor in data ['sensors']:
    if sensor['name'] == 'temperature':
        print("Temperature is: " + str(sensor['value']))
```

> *See `examples/mqtt_subscribe/mqtt_subscribe-parse.py` if you can't get this working.*

Having to scan the entire JSON tree to extract a single piece of data probably illustrates that I should have thought more carefully about the data structure. Think about (and discuss) how the sensor data could be better structured. Perhaps I should have broken the individual sensors out into different MQTT topics? How might you allow selectors like `data['sensors']['temperature']`?

Sending MQTT messages

This is even simpler:

```python
# examples/mqtt_send/basic_send.py
import json
import paho.mqtt.client as mqtt
import config

client = mqtt.Client()

payload = {"name:" "Bob",
           "favourite_colour": "blue",
           "data_value": 328}

mqtt.connect(config.mqtt_server, config.mqtt_port, 60)
mqtt.publish("/KV6006/test", json.dumps(payload))
```

You could wrap the last two lines in a `try/except` structure, but for our purposes today it's not terribly important. The `json.dump()` method handles converting the Python dictionary `payload` into JSON format for us.

You won't see output from this unless someting is running that's subscribed to `/KV6006/test`. Get somebody to run `mqtt_subscribe/mqtt_subscribe_all.py` and watch the output there?

Controlling a device over MQTT

Finally, we get to the odd little device which has been sitting next to your Raspberry Pi all along. This is an even simpler microcontroller than the Pico – an ESP8266, which is about as cheap as WiFi-enabled boards get. It's sitting on a chunk of breadboard (the white slab with holes in it), wired up to two output devices:

- An RGB LED, which can display a range of colours at variable brightness.
- A servo motor, which can turn its white plastic 'horn' through about 180 degrees.

Each device listens for commands via its own MQTT topic, `/KV6006/output/[device_code]`. The device code is printed on a sticker on the device.

You can command the device using something like this. **A couple of lines need changing to reflect the label on your device**:

```python
# examples/mqtt_send/control_device.py
import paho.mqtt.client as mqtt
import config
import json
from time import sleep

def on_publish(client, userdata, result):
    print("Data published")
    pass

# TODO: Change the client name here to reflect your device code
```

```python
client = mqtt.Client("controlD31")
client.on_publish = on_publish
client.username_pw_set(config.mqtt_username, config.mqtt_password)

try:
    client.connect("connect.nustem.uk", 1883, 60)
    print("MQTT connection successful")
except:
    print("MQTT connection failed")
    exit(1)

# TODO: Again, change the target device here
target = "/KV6006/output/D31"
# Set up a dictionary structure for command and value to pass
message = {"command": "LEDhue",
            "value": 60}
# Send message, converting payload to JSON
client.publish(target, json.dumps(message))

message = {"command": "servoAngle",
            "value": 180}
client.publish(target, json.dumps(message))

# 2-second pause
sleep(2)

message = {"command": "servoAngle",
            "value": 0}
client.publish(target, json.dumps(message))
```

Work through what this is doing, and discuss what's happening at each stage. It'd probably be good if you called one of us over and talked us through your understanding of it.

A note about colour

Colour representation is one of *those* subjects. You'd think it would be easy, but… no. I've just deleted 300 words from the worksheet here. A lucky escape. The important bits:

The ESP8266 devices use an HSV colour model. You pass them a 'hue angle' to describe a colour, rather than trying to mix the colour you want by varying red, green and blue componets. That conversion is handled by the device. Only, 'hue angle' in this case is an 8-bit value, so we're mapping `0..360` degrees into a `0..255` numeric range.

Upshot: pick a colour from the scale below, take a broad guess at the number associated with it, and try that.

Things to discuss at this point

- `LEDhue` and `servoAngle` aren't great names for commands. More common would be `setLEDhue` and `setServoAngle`. Why?
- The code the ESP8266s are running was written years ago, for an installation art project. For obscure reasons they got called 'Skutters,' and the code is… mmm, *not good*. If you need a laugh (or to recoil in horror), take a look in `output_devices/skutter`. However, the code still works after all these years, so I don't feel too terrible about it. Skutters have a much more extensive API than covered above, and can in theory handle looping animation between two different states.
- If you find the command to change the LED brightness, be warned that full brightness is eye-searingly horrid. Don't say I didn't warn you.