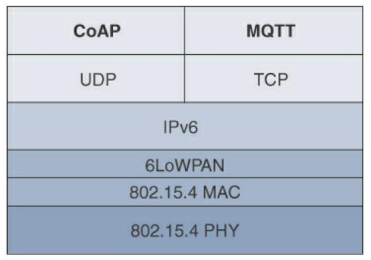
MQTT – Message Queuing Telemetry Transport

Ryan Bates

MQTT is a network level 5 application protocol built on top of TCP often used with many internet-connected devices. It can be compared to other protocols such as as HTTP, a frequently web-based protocol also built on TCP, and CoAP, an alternative built on UDP. As a reminder, TCP messages carry a better garuntee of delivery compared to UDP, but are often higher-weight, and may cause delays for all messages if a packet is missed, while UDP is appropriate for streaming frequent smaller pieces of data where a single loss is not a critical failure. As MQTT a level 5 implementation, it will often not require specialized hardware beyond ‘conventional’ networking adapters that provide HTTP – just new software, so retrofitting might well be viable.



(slide 63)

Unlike HTTP, MQTT does not create direct connections between each client (see slide 62). Instead, it utilizes a broker that takes all incoming data from clients on different topics, as publications. Independently of these publications each client can also optionally subscribe to ‘hear’ and receive data from any of these publishes topics

Imagine MQTT being used in a factory where many manufacturing processes are taking place. Each process or machine might broadcast a success/failure signal, or a signal when an iteration of a process on a product is complete.

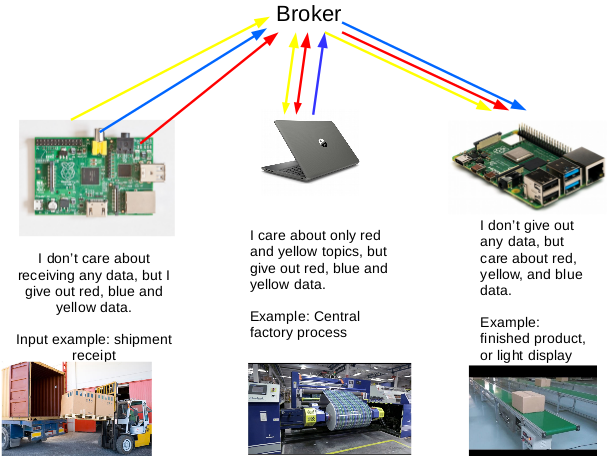
* Advantages
  + Can set high or low reliability depending on needs via QoS levels
  + Less overhead than HTTP
  + Publish/Subscriber model is easily scalable for more machines being added
  + Each new machine can selectively choose what other machines it needs to know about, and each other machine can decide if they need to know about the new machine’s publications (many-to-many relationship)
* Disadvantages
  + High power requirements (easily supplied in immobile factory)
  + Requires reliable broker (easy to maintain in a factory environment)
  + Relatively high processing requirements (versus CoAP)

I have implemented an example of this protocol on a few heterogeneous devices – specifically, two Raspberry Pis and a PC running Linux.

One advantage of MQTT is that each client can independently decide what type data it cares to provide (publish) and receive (subscribe to). No device is obligated to deal with all of the network traffic, aside from the broker. As such, we can avoid overwhelming any specific device, and can make sure the broker is on a strong and secure machine – in this case, our PC.



In my practical implementation, the Raspberry Pi on the left is fitted with three push buttons, red, yellow, and blue. The Raspberry Pi on the right has three LEDs on independent GPIO pins, one each of red, yellow, and blue. Each of these buttons and LEDs are tied to a corresponding red, yellow, or blue channel where they are published or subscribed to. ‘Virtual’ versions of the publisher and subscriber client are also run on the PC, where push buttons are replaced with user terminal input and LEDs are replaced with terminal alerts.

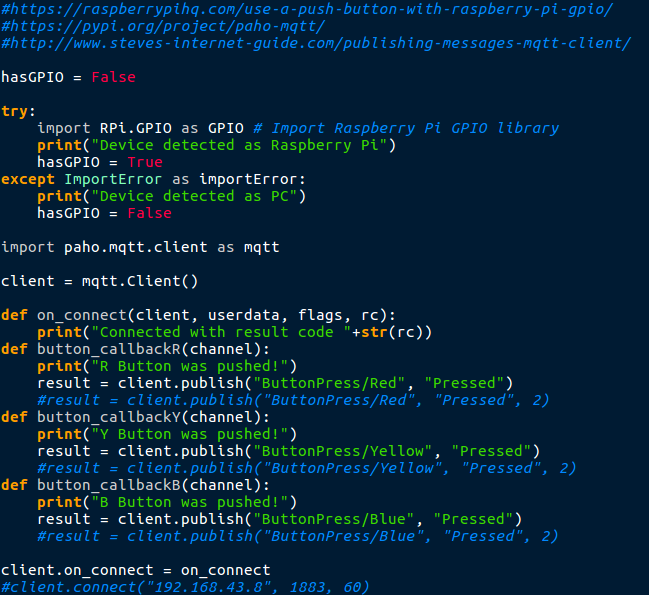


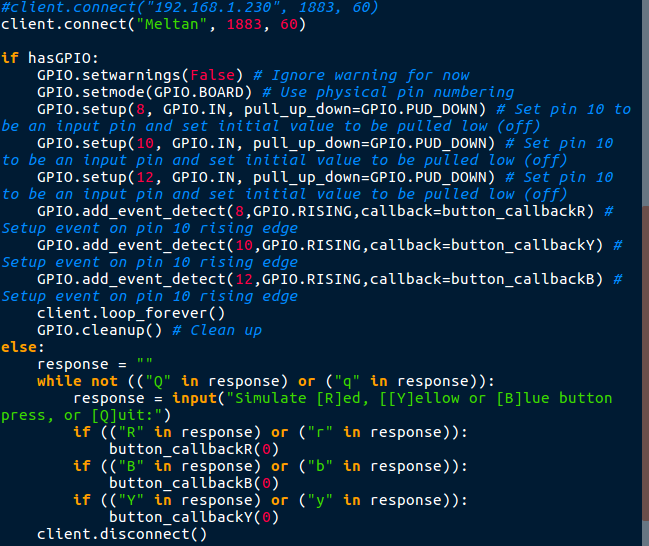
The colored lines here show the topics and how the messages move throughout the network. The left device with the push buttons only publishes all types of data but never subscribes, so all three color topics are monodirectional. Similarly the right device with the LED output only subscribes (though to all three topics) but never publishes, so the arrows are in the reverse direction. In this example the virtual subscribing client on the PC is set to only subscribe to red and yellow topics. This client with therefore never receive any blue topic data, shown as an example of how clients can be selective to reduce traffic, complexity, and privacy concerns. The broker must by necessity handle all traffic, taking all published data and passing it to subscribers by the topic they have requested.

There are easy practical analogues to each of these devices. Places where we have inputs to the system but do not care about an response from the system, such as our button inputs on the left, can be compared to receiving raw materials at a factory. Mirroring those might be some clients as that on the right that create some sort of report or output of final products, where data from the system is needed but no new data is provided back to the network. Devices that both care about some responses from the network and provide some feedback themselves might be machines on an assembly line, where they care about some information from previous machines (but not all), and likewise provide information that machines directly after them would be interested in. MQTT lets them see receive and disseminate this data without spamming the messages to every machine in the factory.

Let’s break it down.

First, an example of a publisher:





Some of these lines bear explaining.

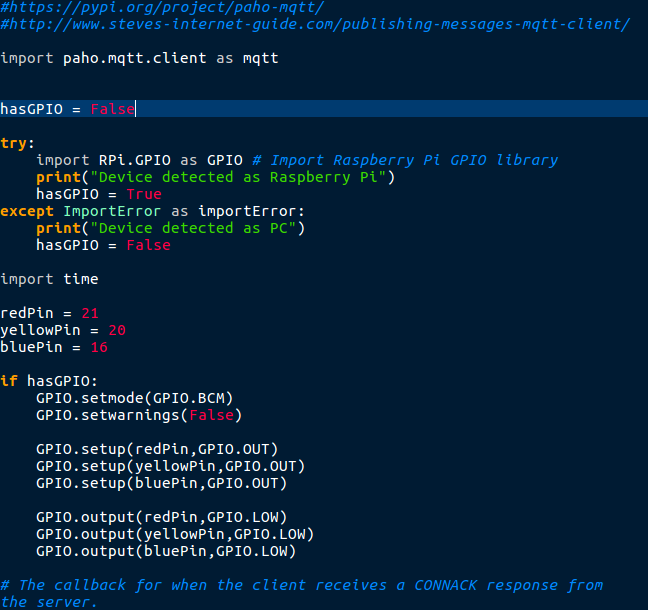
client.connect("Meltan", 1883, 60)

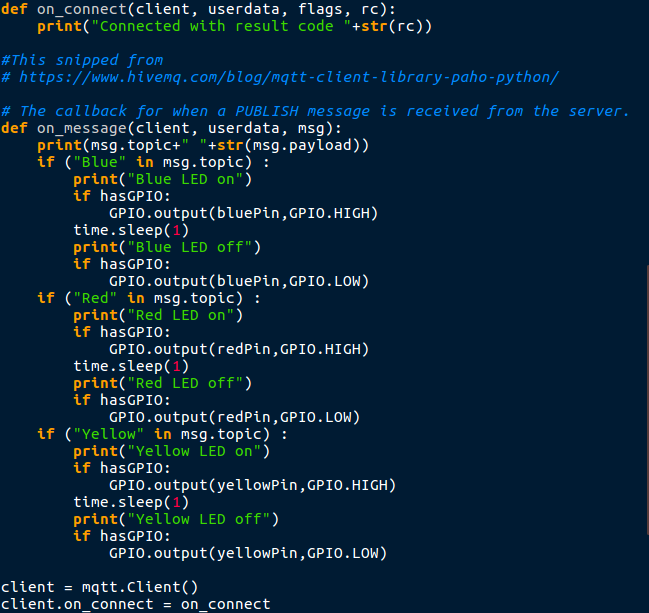
This line connects a client to the host (“Meltan”, in this case) at the specified port. The host here must be the broker. The client does not need to know the locations of any other clients.

result = client.publish("ButtonPress/Blue", "Pressed")

On a button press we publish as a discrete event. Here the topic is "ButtonPress/Blue", where often the slashes can denote subtopics. The ‘payload’ of this message is "Pressed", but could be anything we want to say to subscribers of this topic. Optionally a third argument can be added, the Quality of Service (QoS) level, such as to ask for no repeats or guaranteed delivery of the message.

An example of a subscriber client is similar.







client.connect("Meltan", 1883, 60)

Again, we connect to the same broker address.

client.subscribe("ButtonPress/Blue", 2)

The subscriber tells the broker that it wants to listen to the topic is "ButtonPress/Blue". The other argument again denotes the quality of service level.

def on\_message(client, userdata, msg):

…

...

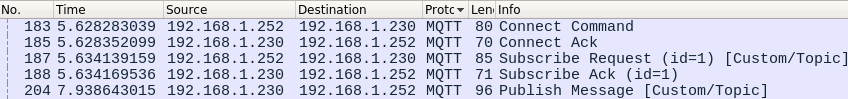
client.on\_message = on\_message

These two lines are separate in the code but are connected. The first is the definition of a function we define that can take in the content of a message, and variously goes on to turns on LEDs depending on the details. The second is where we assign this function as a callback, to be called with our input data every time we receive that subscribed data.

Of course, a client could also both subscribe and publish.

Network Breakdown

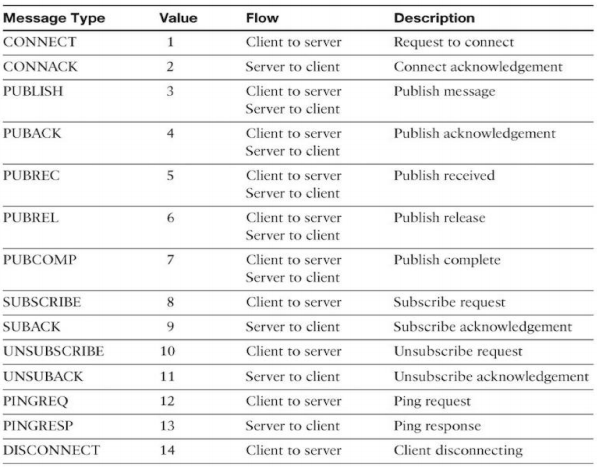
We can see what MQTT looks like in detail by using Wireshark to analyze packets being transmitted.



In this example, 192.168.1.230 is a Dell desktop running Ubuntu Linux, connected via Ethernet. It is running the Mosquitto broker server and also our publisher client.

192.168.1.252 is a Raspberry Pi running Raspbian Linux connected via wifi. It is running the subscriber client.

Since these packets are sorted by time, we can read the Info section and check it against the message types from slide 83:



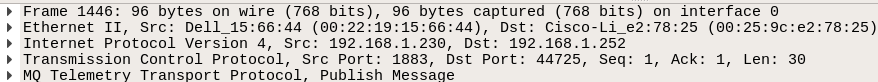
1. We start with the CONNECT command as the Raspberry Pi client connects to the desktop.
2. This is is acknowledged with a CONNACK/Connect Ack message back from the desktop.
3. The Raspberry Pi then tells the broker on the server the subscription topic it wants to hear about. In a SUBSCRIBE message In this case the topic is literally “Custom/Topic”.
4. The desktop again sends an acknowledgement back to the Pi as a SUBACK.
5. Finally, a message is published.  
     
   Note: there are likely many pings and ping responses going on under the hood, but a typical use case will be focusing on the sub/pub model and not simple pings.

Notice that the broker 192.168.1.230 is always involved in any MQTT transaction. Neither of the separate clients ever talk directly to each other – they always go through the broker.

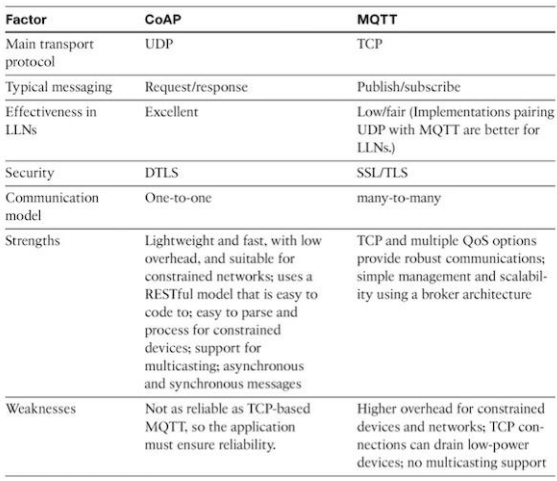


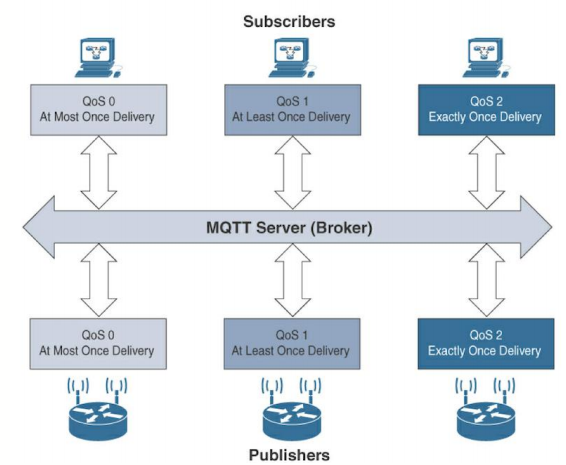
The Source is





Expanding these we get:

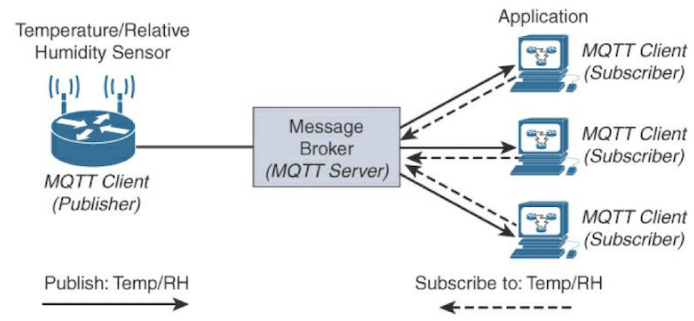
Source: Prof. Nof Abuzainab, ENPM 809F: Internet of Things Lecture 5: RPL Security + Application Protocols for IoT, slide 91.



Source: Prof. Nof Abuzainab, ENPM 809F: Internet of Things Lecture 5: RPL Security + Application Protocols for IoT, slide 90.

Pub/sub is better for iot because lower overhead

Source: Prof. Nof Abuzainab, ENPM 809F: Internet of Things Lecture 5: RPL Security + Application Protocols for IoT, slide 92.



TODO: QoS

Show size difference with http

What was MQTT originally named for

Source: Prof. Nof Abuzainab, ENPM 809F: Internet of Things Lecture 5: RPL Security + Application Protocols for IoT, slide 80.

TODO:

QoS

Mention that payload is up to 256 megs 82

LEDs/Buttons

Disconnect command

Discuss size

github

Install wireshark, mosquitto on laptop

Make sure things auto connect to phone hotspot

virtual pub client

virtual sub client

selective sub

Does changing the QoS change the timing?

Change repo name to mqtt

References

* Slide number references are from Abuzainab, “ENPM 809F: Internet of Things Lecture 5: RPL Security + Application Protocols for IoT”
* MQTT Broker used: <https://mosquitto.org/>
* Code snippets provided from:
  + https://raspberrypihq.com/use-a-push-button-with-raspberry-pi-gpio/
  + https://pypi.org/project/paho-mqtt/
  + <http://www.steves-internet-guide.com/publishing-messages-mqtt-client/>
* Image sources:
  + https://www.raspberrypi.org/forums/viewtopic.php?t=4751
  + https://www.currys.co.uk/gbuk/computing/laptops/laptops/hp-15-db0521sa-15-6-amd-a6-laptop-1-tb-hdd-grey-10180991-pdt.html
  + <https://en.wikipedia.org/wiki/Raspberry_Pi>
  + http://www.forkliftamerica.com/blog/improving-material-handling-efficiency/
  + https://www.youtube.com/watch?v=xNePYj2GydM
  + https://www.youtube.com/watch?v=Ixb4W1tvvKM