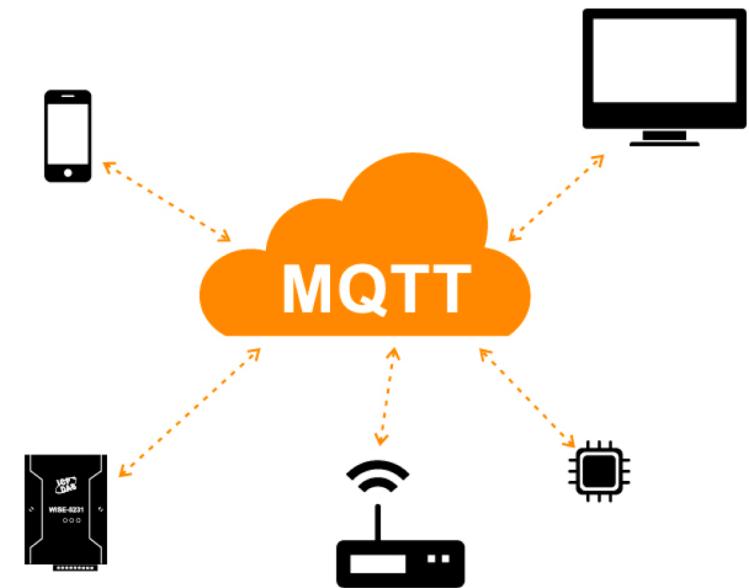


Intro to MQTT

Pietro Manzoni

Universitat Politecnica de Valencia (UPV)
Valencia - SPAIN

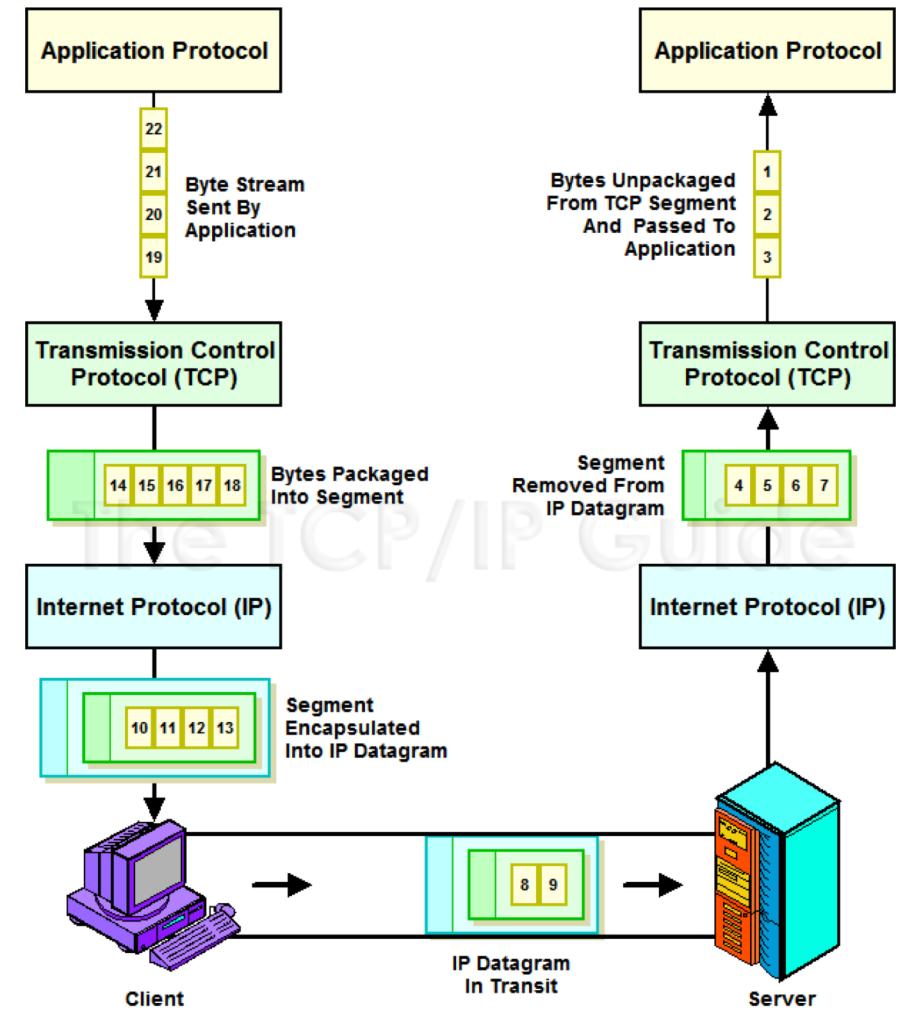
pmanzoni@disca.upv.es

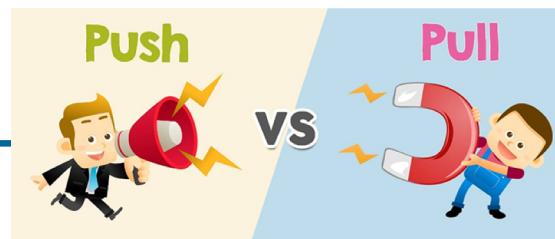


<https://github.com/pmanzoni/KIC2019>

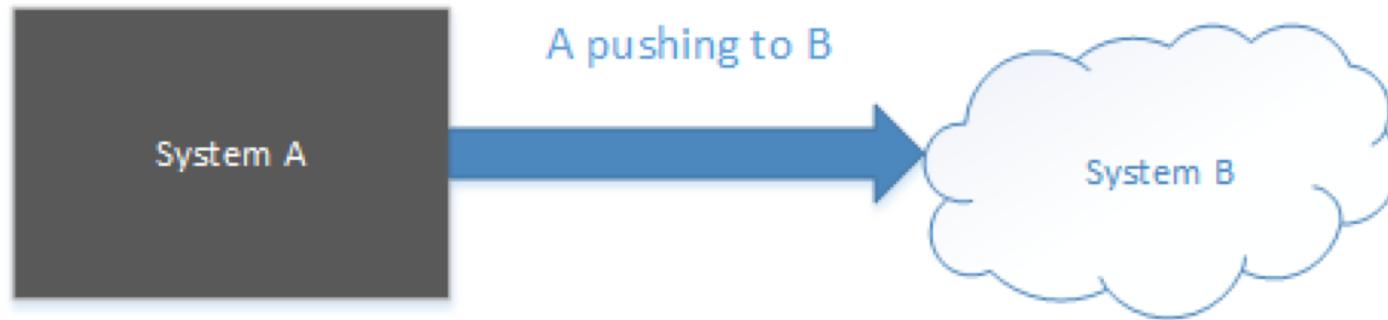
From “byte streams” to “messages”

- The “old” vision of data communication was based on **reliable byte streams**, i.e., TCP
 - Nowadays **messages interchange** is becoming more common
 - E.g., Twitter, Whatsapp, Instagram, Snapchat, Facebook,...
 - Actually is not that new...
 - emails: SMTP+MIME,
 - FTP,



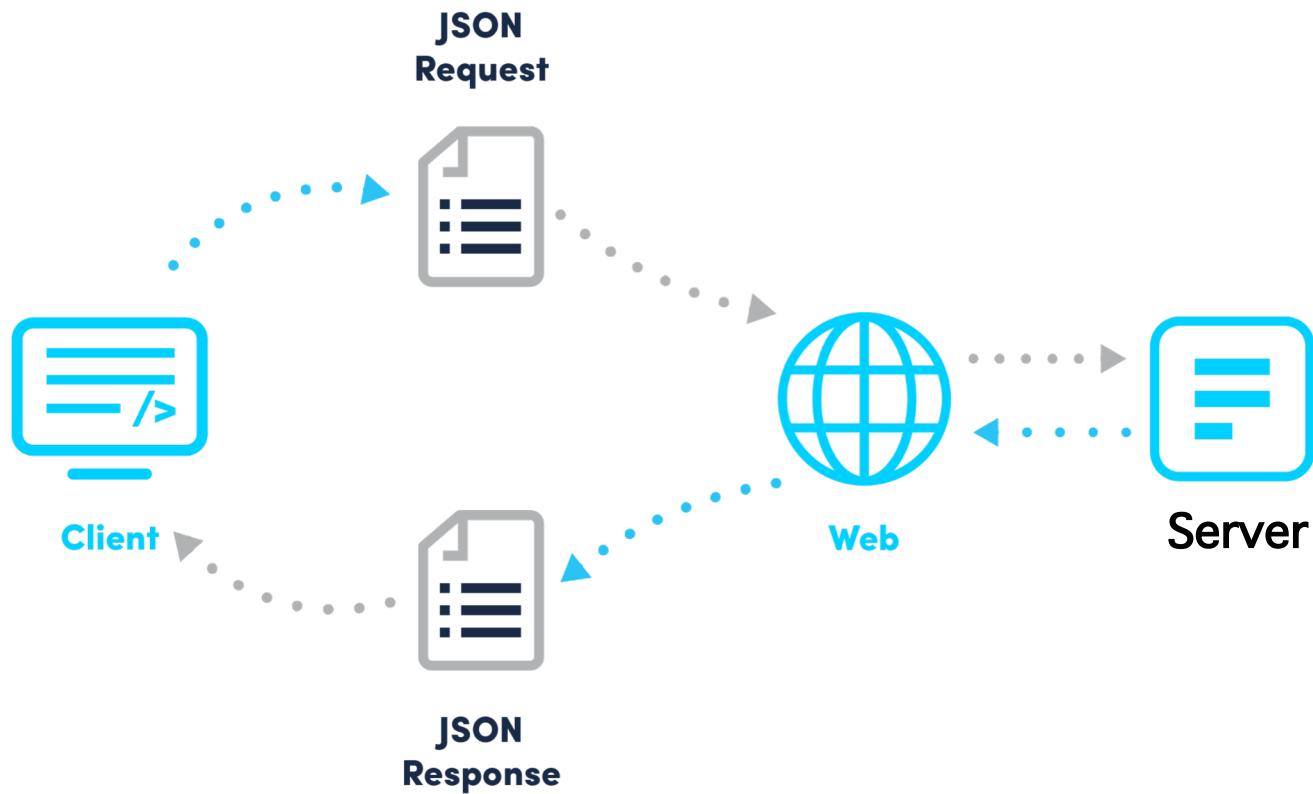


Ways to interchange “messages”



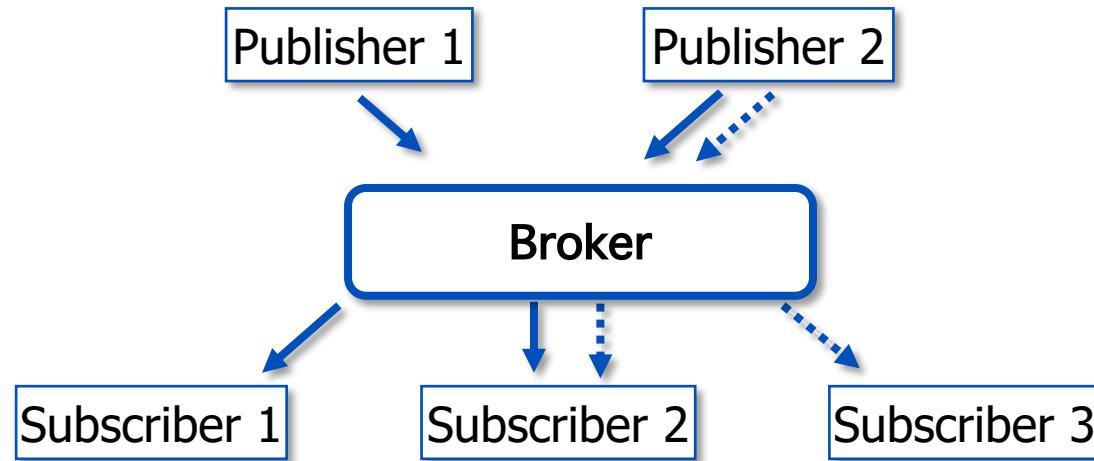
Request/response approach

- REST: Representational State Transfer
- Widely used; based on HTTP
- *Lighter version: CoAP (Constrained Application Protocol)*



- Publish/Subscriber

- aka: producer/consumer



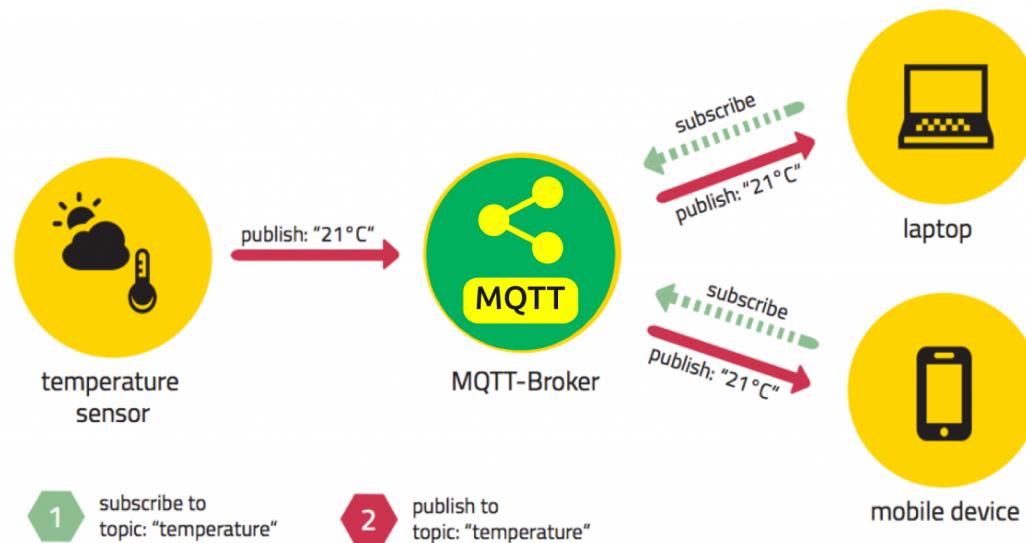
- Various protocols:

- MQTT, AMQP, XMPP (was Jabber)

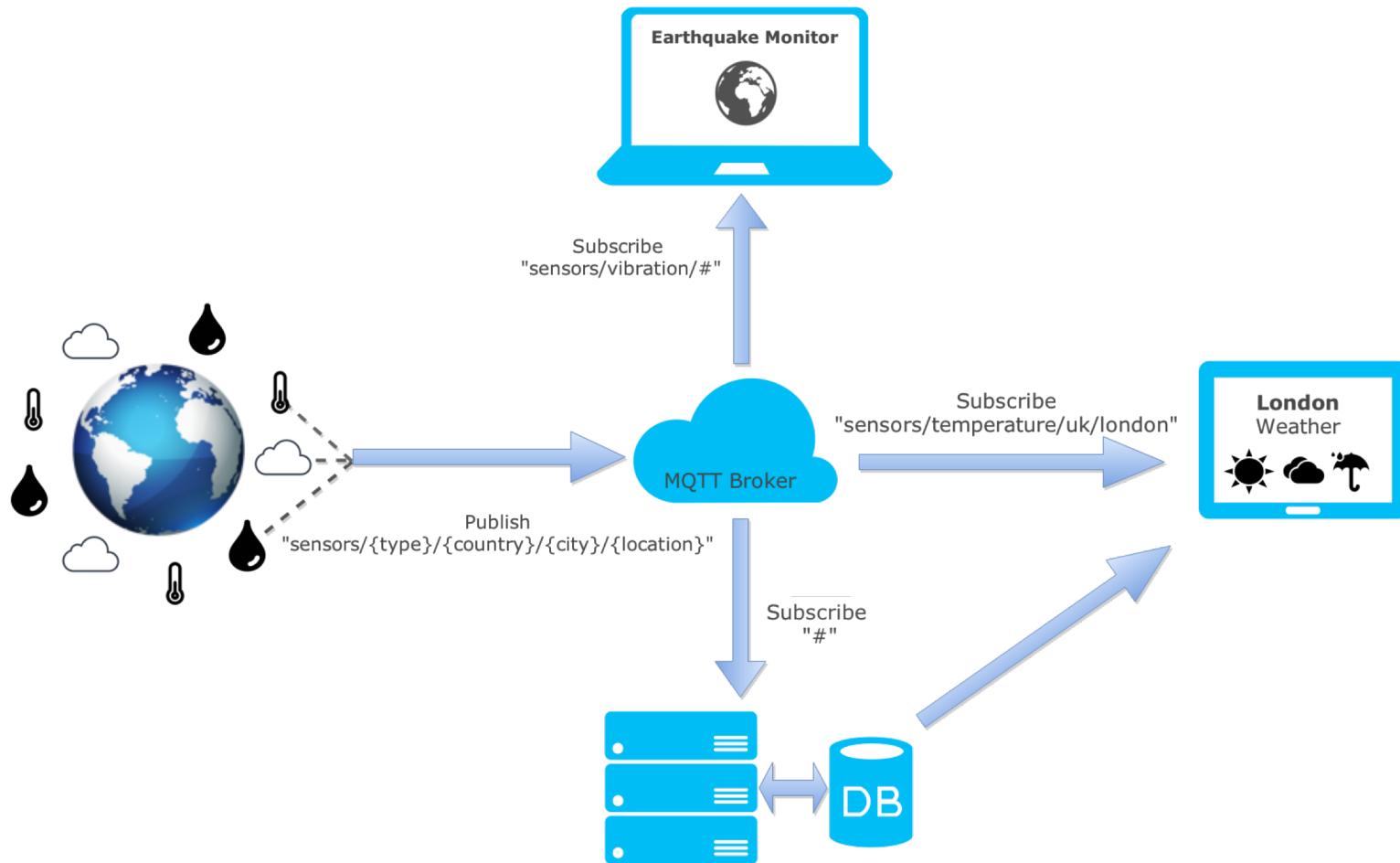
- Growing technique

- E.g., <https://cloud.google.com/iot/docs/how-tos/mqtt-bridge>

- Pub/Sub separates a client, who is sending a message about a specific **topic**, called **publisher**, from another client (or more clients), who is receiving the message, called **subscriber**.
- There is a third component, called **broker**, which is known by both the publisher and subscriber, which filters all incoming messages and distributes them accordingly.



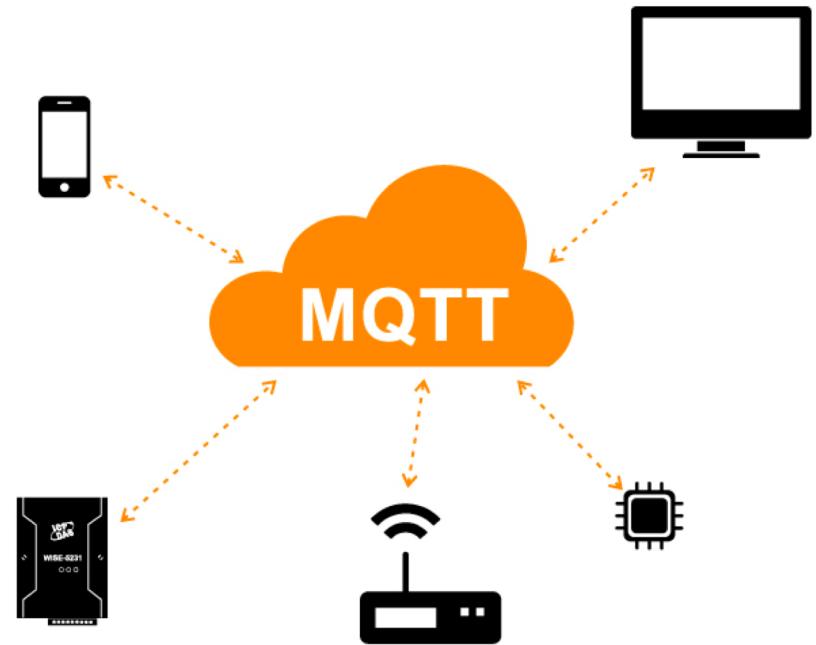
HiveMQ®



Source: <https://zoetrope.io/tech-blog/brief-practical-introduction-mqtt-protocol-and-its-application-iot>

Intro to MQTT

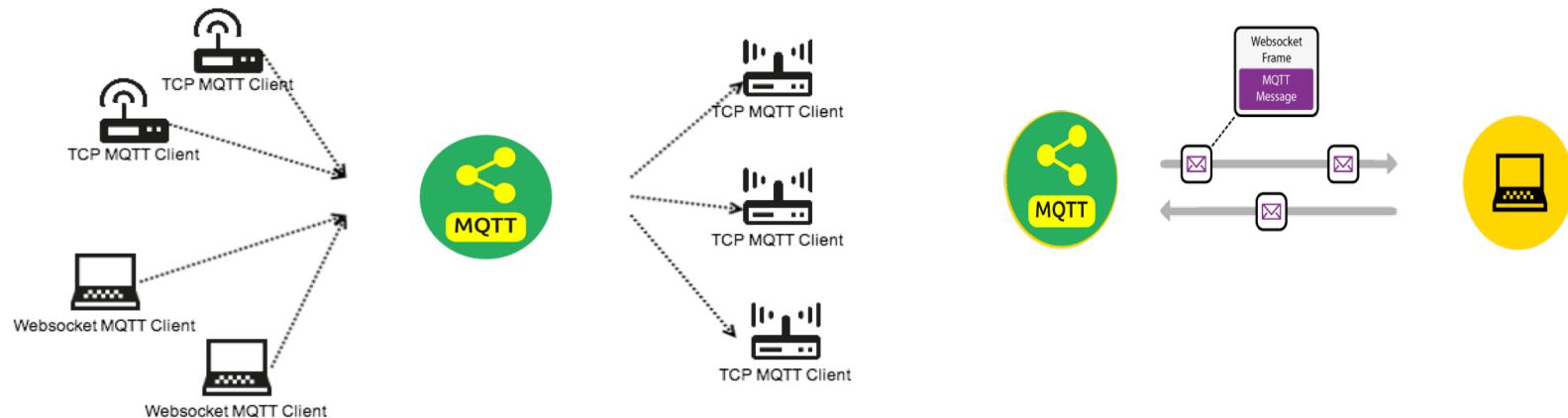
○ Fundamental concepts



- A **lightweight publish-subscribe protocol** that can run on embedded devices and mobile platforms → <http://mqtt.org/>
 - Low power usage.
 - Binary compressed headers
 - Maximum message size of 256MB
 - not really designed for sending large amounts of data
 - better at a high volume of low size messages.
- Documentation sources:
 - The MQTT community wiki:
 - <https://github.com/mqtt/mqtt.github.io/wiki>
 - A very good tutorial:
 - <http://www.hivemq.com/mqtt-essentials/>

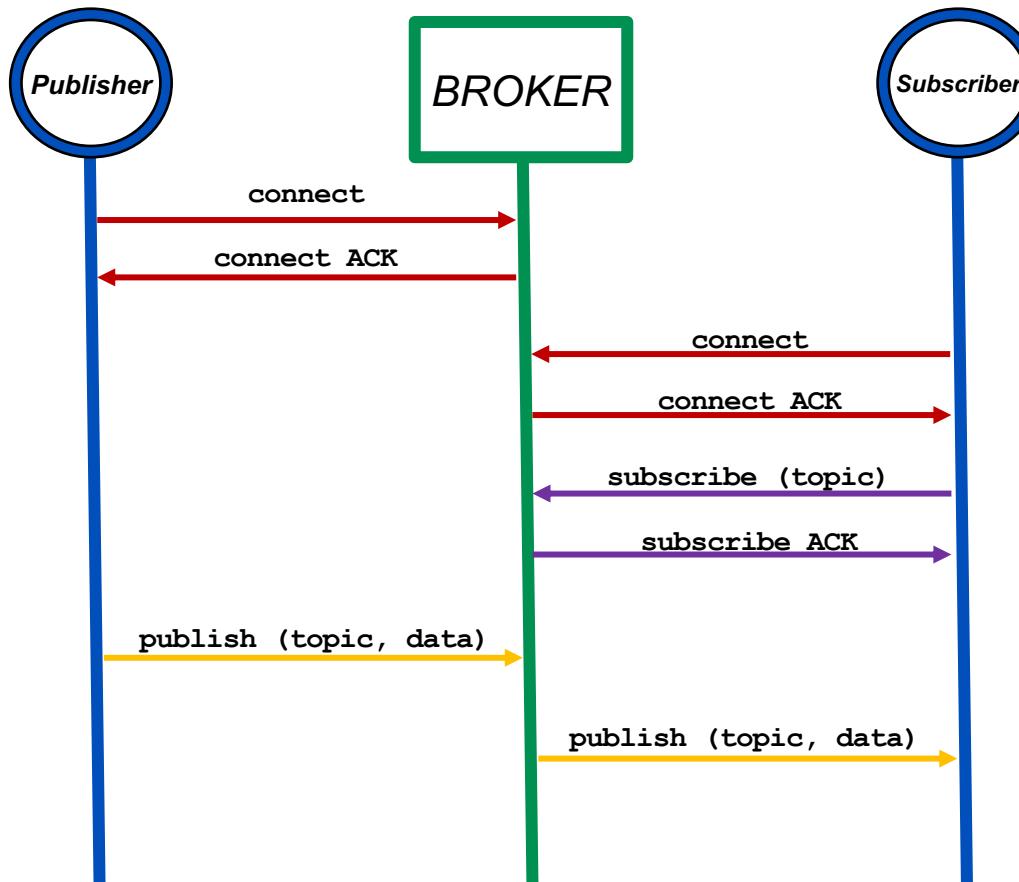
- **MQTT 3.1.1 is the current version of the protocol.**
 - Standard document here:
 - <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>
 - October 29th 2014: MQTT was officially approved as OASIS Standard.
 - https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt
- MQTT v5.0 is the successor of MQTT 3.1.1
 - Current status: Committee Specification 02 (15 May 2018)
 - <http://docs.oasis-open.org/mqtt/mqtt/v5.0/cs02/mqtt-v5.0-cs02.html>
 - **Not backward compatible**; too many new things are introduced so existing implementations have to be revisited, for example:
 - **Enhancements for scalability** and large scale systems in respect to setups with 1000s and millions of devices.
 - **Improved error reporting** (Reason Code & Reason String)
 - Performance improvements and improved support for small clients
 - https://www.youtube.com/watch?time_continue=3&v=YIpesv_bJgU

- mainly of TCP
 - There is also the closely related [MQTT for Sensor Networks \(MQTT-SN\)](#) where TCP is replaced by UDP → TCP stack is too complex for WSN
- websockets can be used, too!
 - Websockets allows you to receive MQTT data directly into a web browser.



- Both, TCP & websockets can work on top of “Transport Layer Security (TLS)” (and its predecessor, Secure Sockets Layer (SSL))

Publish/subscribe interactions sequence



- MQTT Topics are structured in a hierarchy similar to folders and files in a file system using the forward slash (/) as a delimiter.
- Allow to create a user friendly and self descriptive **naming structures**
- Topic names are:
 - Case sensitive
 - use UTF-8 strings.
 - Must consist of at least one character to be valid.
- Except for the \$SYS topic **there is no default or standard topic structure.**

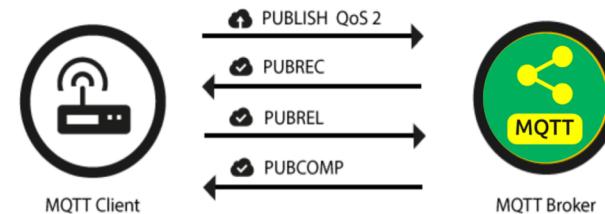


Special \$SYS/ topics

- \$SYS/broker/clients/connected
- \$SYS/broker/clients/disconnected
- \$SYS/broker/clients/total
- \$SYS/broker/messages/sent
- \$SYS/broker/uptime

- Topic subscriptions can have wildcards. These enable nodes to subscribe to groups of topics that don't exist yet, allowing greater flexibility in the network's messaging structure.
 - '+' matches anything at a given tree level
 - '#' matches a whole sub-tree
- Examples:
 - Subscribing to topic **house/#** covers:
 - ✓ house/room1/main-light
 - ✓ house/room1/alarm
 - ✓ house/garage/main-light
 - ✓ house/main-door
 - Subscribing to topic **house/+/main-light** covers:
 - ✓ house/room1/main-light
 - ✓ house/room2/main-light
 - ✓ house/garage/main-light
 - but doesn't cover
 - ✓ house/room1/side-light
 - ✓ house/room2/side-light

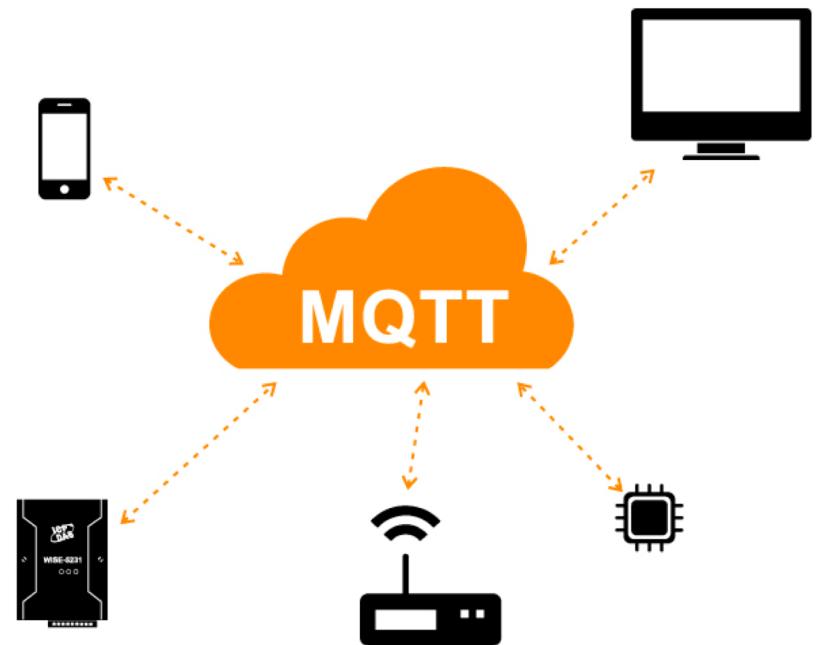
- Messages are published with a **Quality of Service (QoS)** level, which specifies delivery requirements.
- A **QoS 0 ("at most once")** message is fire-and-forget.
 - For example, a notification from a doorbell may only matter when immediately delivered.
- With **QoS 1 ("at least once")**, the broker stores messages on disk and retries until clients have acknowledged their delivery.
 - (Possibly with duplicates.) It's usually worth ensuring error messages are delivered, even with a delay.
- **QoS 2 ("exactly once")** messages have a second acknowledgement round-trip, to ensure that **non-idempotent messages** can be delivered exactly once.



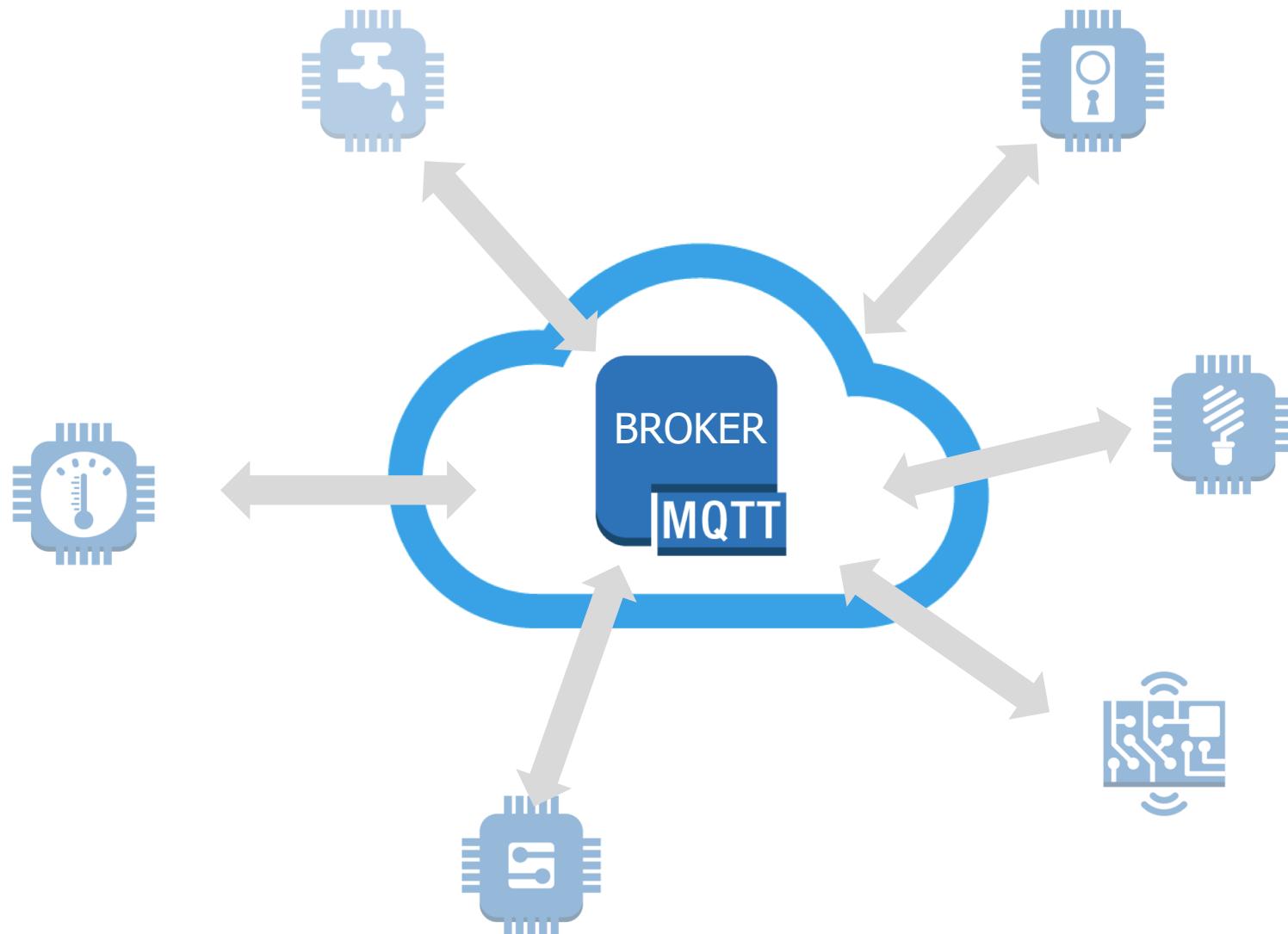
- A retained message is a normal MQTT message with the **retained flag set to true**. The broker will store the last retained message and the corresponding QoS for that topic
 - Each client that subscribes to a topic pattern, which matches the topic of the retained message, will receive the message immediately after subscribing.
 - For each topic **only one retained message** will be stored by the broker.
- Retained messages can help newly subscribed clients to get a status update immediately after subscribing to a topic and don't have to wait until a publishing clients send the next update.
 - In other words a retained message on a topic is the last known good value, because it doesn't have to be the last value, but it certainly is the last message with the retained flag set to true.

Intro to MQTT

- Brokers and clients



Creating a broker



- The most widely used are:
 - <http://mosquitto.org/>
 - man page: <https://mosquitto.org/man/mosquitto-8.html>
 - <http://www.hivemq.com/>
 - The standard trial version only supports 25 connections.
- And also:
 - <https://www.rabbitmq.com/mqtt.html>
 - <http://activemq.apache.org/mqtt.html>
- A quite complete list can be found here:
 - <https://github.com/mqtt/mqtt.github.io/wiki/servers>

Installing Mosquitto on a Raspberry Pi

- It takes only a few seconds to install a Mosquitto broker on a Raspberry. You need to execute the following steps:

```
sudo apt-get update
```

```
sudo apt-get install mosquitto mosquitto-clients
```

- Installation guidelines with websockets

<https://gist.github.com/smoofit/dafa493aec8d41ea057370dbfde3f3fc>

- Managing the broker:

- To start and stop its execution use:

```
sudo /etc/init.d/mosquitto start/stop
```

- Verbose mode:

```
sudo mosquitto -v
```

- To check if the broker is running you can use the command:

```
sudo netstat -tanlp | grep 1883
```

- note: "-tanlp" stands for: *tcp, all, numeric, listening, program*

Cloud based MQTT brokers: CloudMQTT

<https://www.cloudmqtt.com/>

→ based on Mosquitto

CloudMQTT

Pricing

Documentation

Support

Blog

Hosted message broker for the Internet of Things



Power Pug

- Up to 10 000 connections
- No artificial limitations
- Support by e-mail
- Support by phone

\$ 299

PER MONTH

Get Now

mized message queues for IoT, ready in seconds.



Cute Cat

- 5 users/acl rules/connections
- 10 Kbit/s

FREE

Get Now



<https://flespi.com/mqtt-broker>

MQTT broker

Fast, secure, and free public MQTT broker with MQTT 5.0 support, private namespace, WSS, ACLs, and rich API.

- flespi MQTT broker architecure
- MQTT as a remote distributed storage system
- MQTT as the foundation for event-driven web-application design

Also check out [MQTT Board](#) - our MQTT 5.0 client tool for debugging and testing.

Cloud based brokers: flespi

Terms of use

Free \$0/mo

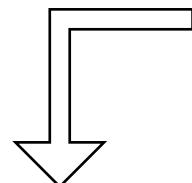
MQTT

100 active MQTT sessions

<https://flespi.io/#/panel/mqttboard>

The screenshot shows the flespi MQTT Board interface. On the left is a sidebar with user info (Pietro manzoni@ieee.org), tokens, MQTT (selected), and links to MQTT Board, Toolbox, and MQTT Broker API. The main area has a header 'mqtt-board-panel-3de8eb91'. It's divided into two sections: 'Subscriber' (orange background) and 'Publisher' (blue background). The Subscriber section has a topic field set to '#', QoS options (0, 1, 2), and checkboxes for 'No local' and 'Retain as Published'. The Publisher section has a topic field set to 'my/topic', a message field containing '{"hello": "world"}', and checkboxes for 'Retain' and 'Duplicate flag'.

<https://flespi.com/mqtt-api>



flespi MQTT broker connection details

- **Host** — mqqt.flespi.io.
- **Port** — [8883 \(SSL\)](#) or [1883 \(non-SSL\)](#); for MQTT over WebSockets: [443 \(SSL\)](#) or [80 \(non-SSL\)](#).
- **Authorization** — use a [flespi platform token](#) as MQTT session username; no password.
- **Client ID** — use any unique identifier within your flespi user session.
- **Topic** — you can publish messages to any topic except **flespi/**.
- **ACL** — both **flespi/** and **MQTT pub/sub** restrictions [determined by the token](#).

I1RKMMIUJppLd1QoSgAQ8MvJPyNV9R2HIJgij01S1gt5rajaeIOaiaKWwlHt2z1z

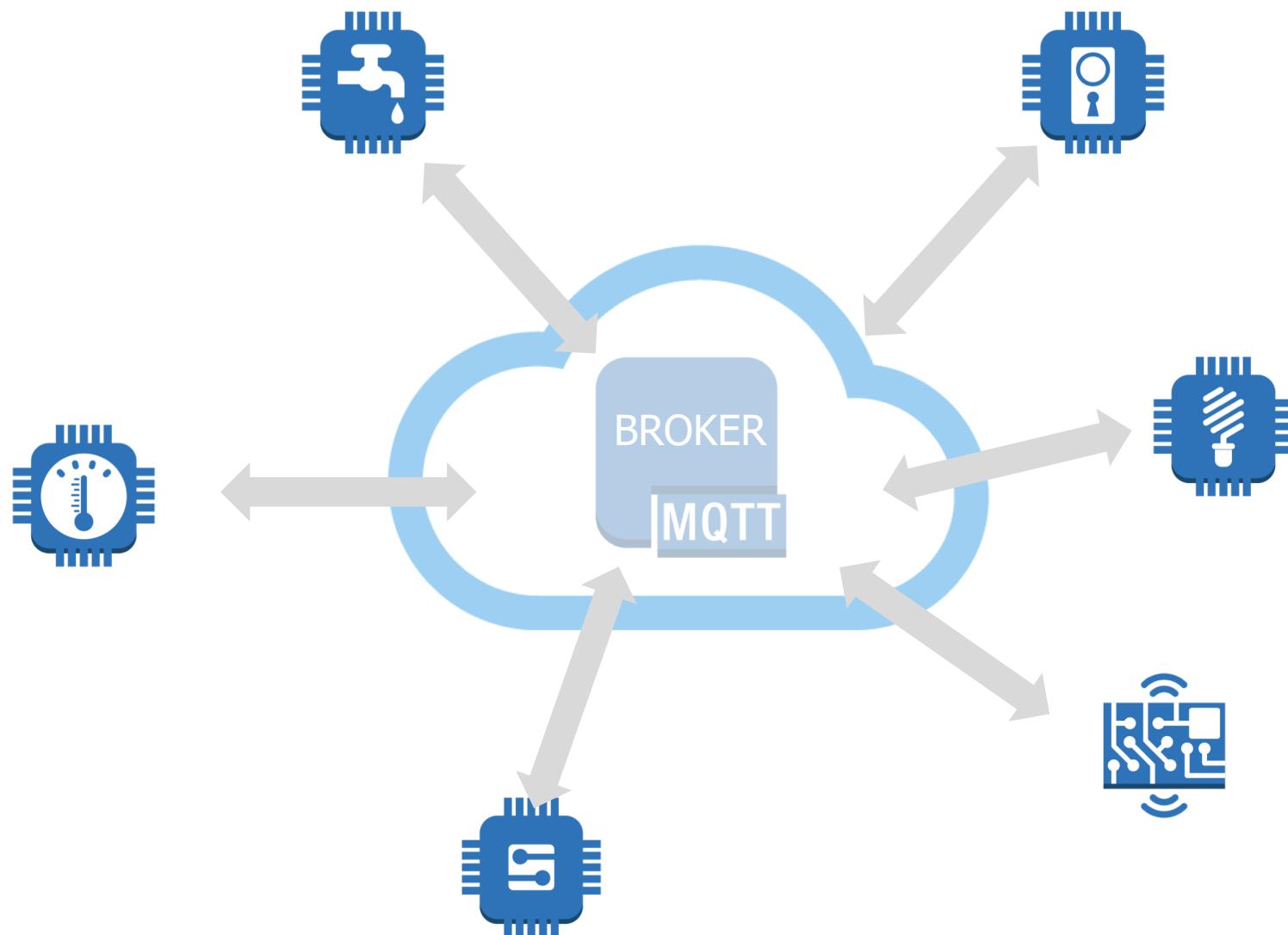
○ TCP based:

- <https://iot.eclipse.org/getting-started/#sandboxes>
 - Hostname: **iot.eclipse.org**
- <http://test.mosquitto.org/>
 - Hostname: **test.mosquitto.org**
- <https://www.hivemq.com/mqtt-demo/>
 - Hostname: **broker.hivemq.com**
 - <http://www.mqtt-dashboard.com/>
- Ports:
 - standard: 1883
 - encrypted: 8883 (*TLS v1.2, v1.1 or v1.0 with x509 certificates*)

○ Websockets based:

- broker.mqttdashboard.com port: 8000
- test.mosquitto.org port: 8080
- broker.hivemq.com port: 8000

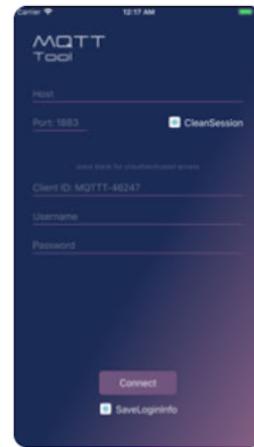
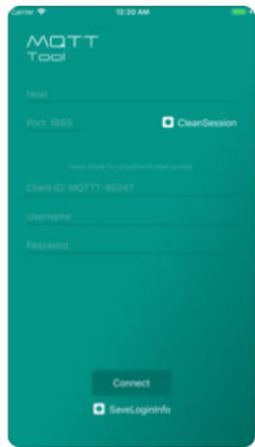
○ https://github.com/mqtt/mqtt.github.io/wiki/public_brokers



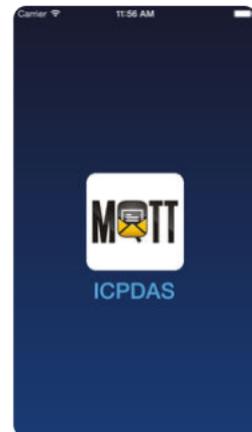
- The Mosquitto broker comes with a couple of useful commands to quickly publish and subscribe to some topic.
- Their basic syntax is the following.
 - `mosquitto_sub -h HOSTNAME -t TOPIC`
 - `mosquitto_pub -h HOSTNAME -t TOPIC -m MSG`
- More information can be found:
 - https://mosquitto.org/man/mosquitto_sub-1.html
 - https://mosquitto.org/man/mosquitto_pub-1.html

MQTT clients: iOS

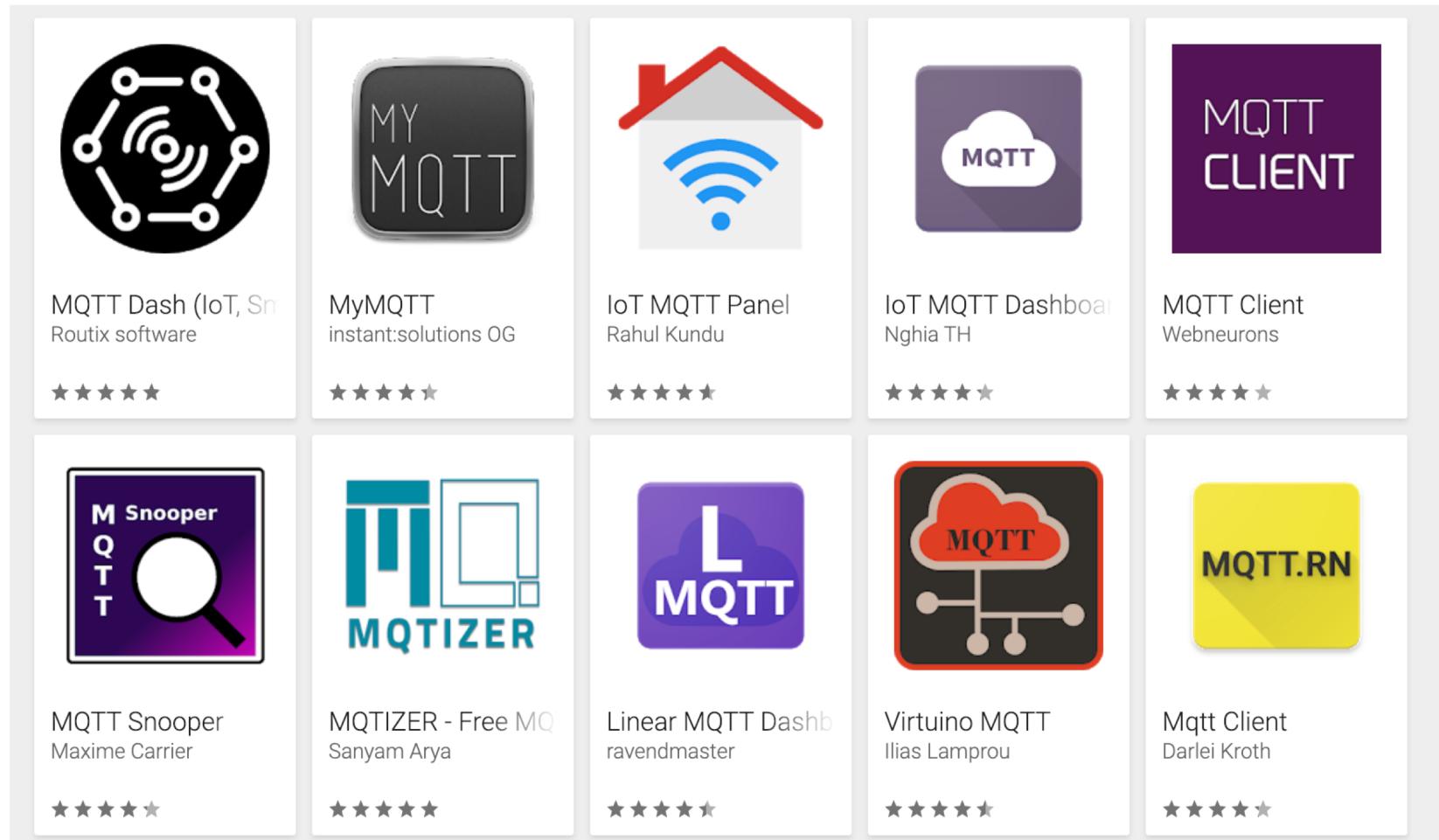
Mqttt
Utilidades



ICPDAS MQ...
Utilidades



MQTT clients: Android



<http://test.mosquitto.org/ws.html>

MQTT over WebSockets

This is a very early/incomplete/broken example of MQTT over Websockets for test.mosquitto.org. Play around with the buttons below, but don't be surprised if it breaks or isn't very pretty. If you want to develop your own websockets/mqtt app, use the url ws://test.mosquitto.org/mqtt , use subprotocol "mqtt" (preferred) or "mqqtv3.1" (legacy) and binary data. Then just treat the websocket as a normal socket connection and read/write MQTT packets.

Usage

Click Connect, then use the Publish and/or Subscribe buttons. You should see text appear below. If you've got another mqtt client available, try subscribe to a topic here then use your other client to send a message to that topic.

Broker

[Connect](#) [Disconnect](#)

Publish

Topic:

Payload:

[Publish](#)

Subscribe



Websockets Client Showcase

Connection

Host: broker.mqttdashboard.com

Port: 8000

ClientID:

clientId-EVU0qAkr8g

[Connect](#)

Username:

Password:

Keep Alive: 60

Clean Session:

Last-Will Topic:

Last-Will QoS: 0

Last-Will Retain:

Last-Will Message:

[Publish](#)

[Messages](#)

[Subscriptions](#)

<http://www.hivemq.com/demos/websocket-client/>

<http://mitsuruog.github.io/what-mqtt/>

MQTT on Websocket sample

message clear

Connect / Disconnect

[connect](#) [disconnect](#)

MQTT broker on websocket

Address: ws://broker.hivemq.com:8000/mqtt

Subscribe / Unsubscribe

Topic: mitsuruog

[subscribe](#) [unsubscribe](#)

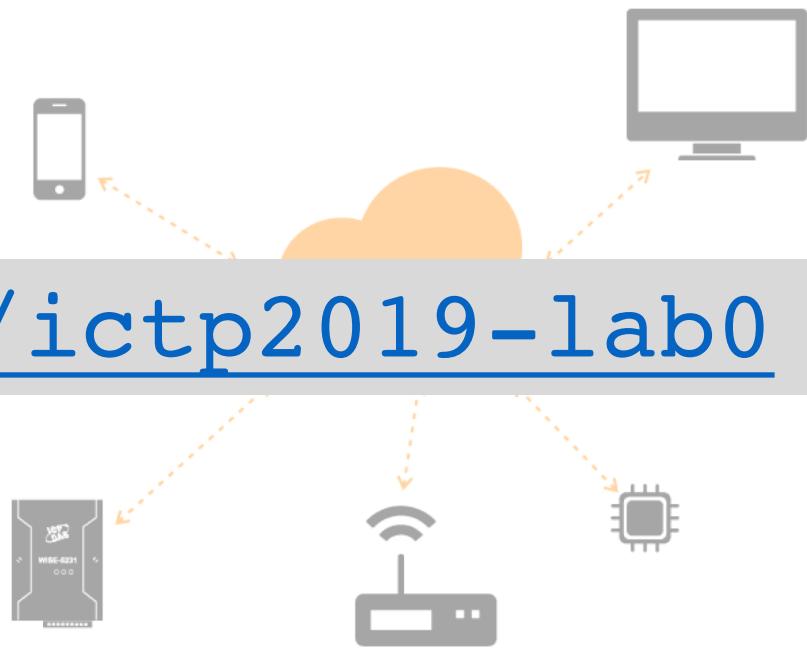
Publish

Topic: mitsuruog

Intro to MQTT

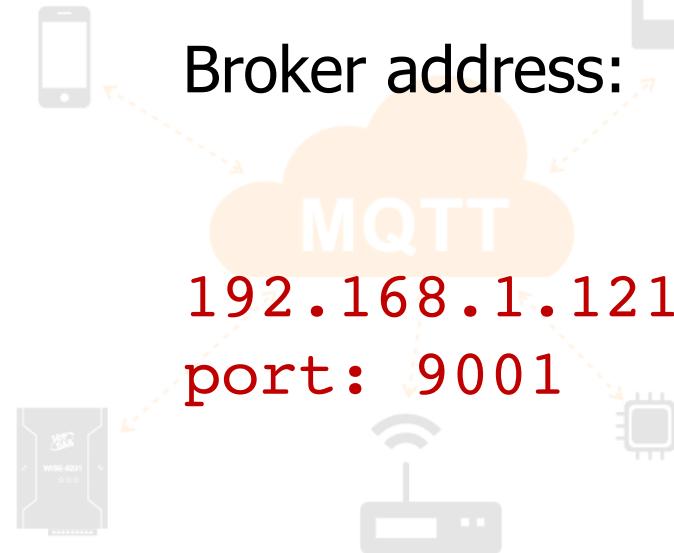
- Time for some exercise: Lab 0

<https://bit.ly/ictp2019-lab0>



Intro to MQTT

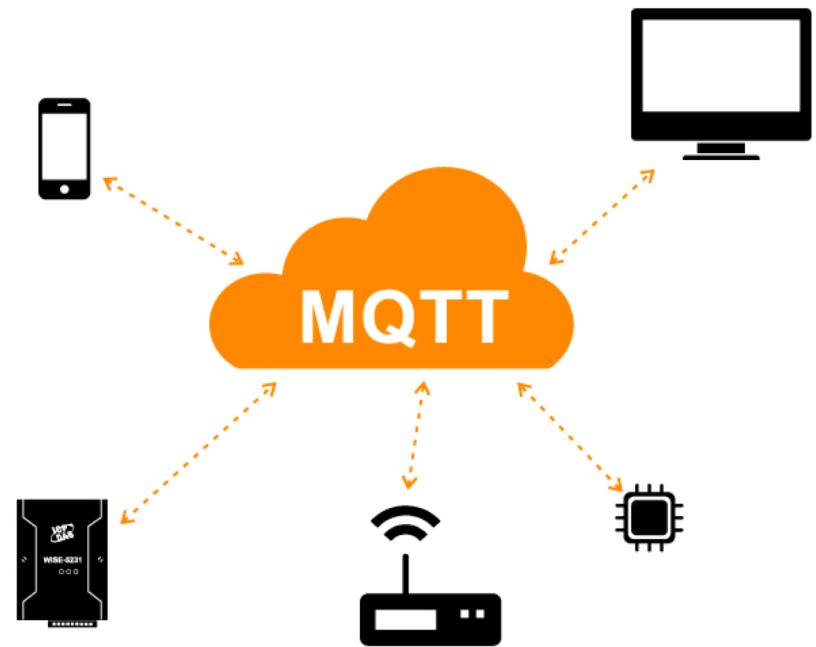
- More time for some demo/exercise: Lab 0.1



<https://www.raspberrypi.org/products/sense-hat/>

Intro to MQTT

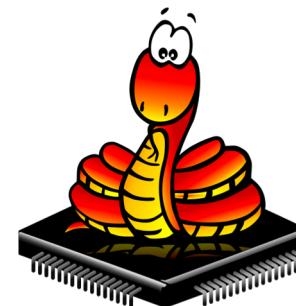
- Clients in Python



- The MQTT available versions for Python and MicroPython are slightly different.
- MicroPython is intended for constrained environments, in particular, microcontrollers, which have orders of magnitude less performance and memory than "desktop" systems on which Python3
- Basically remember that, when using the LoPy you have to use the MicroPython version of MQTT
- In the following we will see information about both cases.



vs.





- Eclipse Paho Python (originally the mosquitto Python client)
 - <http://www.eclipse.org/paho/>
- Documentation: <https://pypi.org/project/paho-mqtt/>
 - or: <http://www.eclipse.org/paho/clients/python/docs/>
- Source: <https://github.com/eclipse/paho.mqtt.python>

Client	MQTT 3.1	MQTT 3.1.1	MQTT 5.0	LWT	SSL / TLS	Automatic Reconnect	Offline Buffering	Message Persistence	WebSocket Support	Standard MQTT Support	Blocking API	Non-Blocking API	High Availability
Java	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Python	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗
JavaScript	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
GoLang	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
C	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C++	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Rust	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
.Net (C#)	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✗	✓	✗
Android Service	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Embedded C/C++	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗



Paho MQTT Python client: general usage flow

The general usage flow is as follows:

- Create a client instance
- Connect to a broker using one of the `connect*()` functions
- Call one of the `loop*()` functions to maintain network traffic flow with the broker
- Use `subscribe()` to subscribe to a topic and receive messages
- Use `publish()` to publish messages to the broker
- Use `disconnect()` to disconnect from the broker



Example 1: a simple subscriber

```
# File: sisub.py

import paho.mqtt.client as mqtt

THE_BROKER = "iot.eclipse.org"
THE_TOPIC = "$SYS/#"
CLIENT_ID = ""

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected to ", client._host, "port: ", client._port)
    print("Flags: ", flags, "returned code: ", rc)
    client.subscribe(THE_TOPIC, qos=0)

# The callback for when a message is received from the server.
def on_message(client, userdata, msg):
    print("sisub: msg received with topic: {} and payload: {}".format(msg.topic, str(msg.payload)))

client = mqtt.Client(client_id=CLIENT_ID,
                     clean_session=True,
                     userdata=None,
                     protocol=mqtt.MQTTv311,
                     transport="tcp")

client.on_connect = on_connect
client.on_message = on_message

client.username_pw_set(None, password=None)
client.connect(THE_BROKER, port=1883, keepalive=60)

# Blocking call that processes network traffic, dispatches callbacks and handles reconnecting.
client.loop_forever()
```

More on this later



Example 1: output

```
Connected to iot.eclipse.org port: 1883
Flags: {'session present': 0} returned code: 0
sisub: msg received with topic: $SYS/broker/version and payload: b'mosquitto version 1.4.15'
sisub: msg received with topic: $SYS/broker/timestamp and payload: b'2018-04-11 '
sisub: msg received with topic: $SYS/broker/clients/total and payload: b'162523'
sisub: msg received with topic: $SYS/broker/clients/active and payload: b'4103'
sisub: msg received with topic: $SYS/broker/clients/inactive and payload: b'158420'
sisub: msg received with topic: $SYS/broker/clients/maximum and payload: b'162524'
sisub: msg received with topic: $SYS/broker/clients/disconnected and payload: b'158420'
sisub: msg received with topic: $SYS/broker/clients/connected and payload: b'4103'
sisub: msg received with topic: $SYS/broker/clients/expired and payload: b'0'
sisub: msg received with topic: $SYS/broker/messages/received and payload: b'1171291305'
sisub: msg received with topic: $SYS/broker/messages/sent and payload: b'6271921352'
sisub: msg received with topic: $SYS/broker/messages/stored and payload: b'1380714' ...
...
...
```



- `connect(host, port=1883, keepalive=60, bind_address='')`
- The broker acknowledgement will generate a callback (`on_connect`).
- Return Codes:
 - 0: Connection successful
 - 1: Connection refused – incorrect protocol version
 - 2: Connection refused – invalid client identifier
 - 3: Connection refused – server unavailable
 - 4: Connection refused – bad username or password
 - 5: Connection refused – not authorised
 - 6-255: Currently unused.



subscribe(topic, qos=0)

- e.g., subscribe("my/topic", 2)
- E.g., subscribe([("my/topic", 0), ("another/topic", 2)])

- on_message(client, userdata, message) Called when a message has been received on a topic that the client subscribes to.

publish(topic, payload=None, qos=0, retain=False)



Paho MQTT Python client: Network loop

```
client.username_pw_set(None, password=None)
client.connect(THE_BROKER, port=1883, keepalive=60)

# Blocking call that processes network traffic, dispatches callbacks and handles reconnecting.
client.loop_forever()
```

More on this later

What are network loops for?



Paho MQTT Python client: Network loop

`loop_forever()`

- This is a blocking form of the network loop and will not return until the client calls `disconnect()`. It automatically handles reconnecting.

`loop_start() / loop_stop()`

- These functions implement a threaded interface to the network loop.
 - Calling `loop_start()` once, before or after `connect()`, runs a thread in the background to call `loop()` automatically. This frees up the main thread for other work that may be blocking.
 - Call `loop_stop()` to stop the background thread.

`loop(timeout=1.0)`

- Call regularly to process network events.
 - This call waits in `select()` until the network socket is available for reading or writing, if appropriate, then handles the incoming/outgoing data.
 - This function blocks for up to `timeout` seconds.



Example 2: subscriber with `loop_start`/`loop_stop`

```
import sys
import time

import paho.mqtt.client as mqtt

THE_BROKER = "test.mosquitto.org"
THE_TOPIC = "$SYS/broker/load/bytes/#"

def on_connect(mqttc, obj, flags, rc):
    print("Connected to ", mqttc._host, "port: ", mqttc._port)
    mqttc.subscribe(THE_TOPIC, 0)

def on_message(mqttc, obj, msg):
    global msg_counter
    print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))
    msg_counter+=1

def on_subscribe(mqttc, obj, mid, granted_qos):
    print("Subscribed: ", mid, "granted QoS: ", granted_qos)

mqttc = mqtt.Client()
mqttc.on_message = on_message
mqttc.on_connect = on_connect
mqttc.on_subscribe = on_subscribe

mqttc.connect(THE_BROKER, keepalive=60)

msg_counter = 0
mqttc.loop_start()
while msg_counter < 10:
    time.sleep(0.1)
mqttc.loop_stop()
print msg_counter
```

```
paho-code:pietro$ python example3.py
('Connected to ', 'test.mosquitto.org', 'port: ', 1883)
('Subscribed: ', 1, 'granted QoS: ', (0,))
$SYS/broker/load/bytes/received/1min 0 489527.05
$SYS/broker/load/bytes/received/5min 0 491792.65
$SYS/broker/load/bytes/received/15min 0 495387.48
$SYS/broker/load/bytes/sent/1min 0 4133472.81
$SYS/broker/load/bytes/sent/5min 0 3515397.37
$SYS/broker/load/bytes/sent/15min 0 2885966.59
$SYS/broker/load/bytes/received/1min 0 483622.23
$SYS/broker/load/bytes/sent/1min 0 3766302.58
$SYS/broker/load/bytes/received/5min 0 490441.96
$SYS/broker/load/bytes/sent/5min 0 3458734.24
$SYS/broker/load/bytes/received/15min 0 494888.07
$SYS/broker/load/bytes/sent/15min 0 2874493.79
12
```



Example 3: very basic periodic producer

```
import random
import time

import paho.mqtt.client as mqtt

THE_BROKER = "test.mosquitto.org"
THE_TOPIC = "PMtest/rndvalue"
CLIENT_ID = ""

# The callback for when the client receives a CONNACK response
# from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected to ", client._host, "port: ", client._port)
    print("Flags: ", flags, "returned code: ", rc)

# The callback for when a message is published.
def on_publish(client, userdata, mid):
    print("sipub: msg published (mid={})".format(mid))

client = mqtt.Client(client_id=CLIENT_ID,
                      clean_session=True,
                      userdata=None,
                      protocol=mqtt.MQTTv311,
                      transport="tcp")

client.on_connect = on_connect
client.on_publish = on_publish

client.username_pw_set(None, password=None)
client.connect(THE_BROKER, port=1883, keepalive=60)

client.loop_start()

while True:

    msg_to_be_sent = random.randint(0, 100)
    client.publish(THE_TOPIC,
                  payload=msg_to_be_sent,
                  qos=0,
                  retain=False)

    time.sleep(5)

    client.loop_stop()
```

Generates a new data every 5 secs



Example 3: output

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
sipub: msg published (mid=1)
Connected to test.mosquitto.org port: 1883
Flags: {'session present': 0} returned code: 0
sipub: msg published (mid=2)
sipub: msg published (mid=3)
sipub: msg published (mid=4)
sipub: msg published (mid=5)
sipub: msg published (mid=6)
sipub: msg published (mid=7)
```

Producer output

Output obtained with a modified version of Example1.
Which parts of that code had to be modified?

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
Connected to test.mosquitto.org port: 1883
Flags: {'session present': 0} returned code: 0
sisub: msg received with topic: PMtest/rndvalue and payload: b'11'
sisub: msg received with topic: PMtest/rndvalue and payload: b'14'
sisub: msg received with topic: PMtest/rndvalue and payload: b'31'
sisub: msg received with topic: PMtest/rndvalue and payload: b'27'
sisub: msg received with topic: PMtest/rndvalue and payload: b'60'
sisub: msg received with topic: PMtest/rndvalue and payload: b'70'
sisub: msg received with topic: PMtest/rndvalue and payload: b'60'
sisub: msg received with topic: PMtest/rndvalue and payload: b'66'
sisub: msg received with topic: PMtest/rndvalue and payload: b'45'
sisub: msg received with topic: PMtest/rndvalue and payload: b'56'
sisub: msg received with topic: PMtest/rndvalue and payload: b'37'
...
```



Example 4: Pub/Sub with JSON

Producer

```
...
mqttc.loop_start()

while True:
    # Getting the data
    the_time = time.strftime("%H:%M:%S")
    the_value = random.randint(1,100)
    the_msg={'Sensor': 1, 'C_F': 'C',
              'Value': the_value, 'Time': the_time}

    the_msg_str = json.dumps(the_msg)

    mqttc.publish(THE_TOPIC, the_msg_str)
    time.sleep(5)

mqttc.loop_stop()
```

```
...
# The callback for when a PUBLISH message is received
# from the server.

def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

    themsg = json.loads(str(msg.payload))

    print("Sensor "+str(themsg['Sensor'])+" got value "+
          str(themsg['Value'])+" "+themsg['C_F']+"
          at time "+str(themsg['Time']))

...
...
```

Consumer

```
paho-code:pietro$ python example4-cons.py
Connected with result code 0
PMtest/jsonvalue {"Time": "12:19:30", "Sensor": 1, "Value": 33, "C_F": "C"}
Sensor 1 got value 33 C at time 12:19:30
PMtest/jsonvalue {"Time": "12:19:35", "Sensor": 1, "Value": 11, "C_F": "C"}
Sensor 1 got value 11 C at time 12:19:35
```



- Import the library

```
from mqtt import MQTTClient
```

- Creating a client:

```
MQTTClient(client_id, server, port=0, user=None, password=None,  
           keepalive=0, ssl=False, ssl_params={})  
e.g., client = MQTTClient("dev_id", "10.1.1.101", 1883)
```

- The various calls:

- `connect(clean_session=True):`
- `publish(topic, msg, retain=False, qos=0):`
- `subscribe(topic, qos=0):`
- `set_callback(self, f):`

- `wait_msg()`:

- Wait for a single incoming MQTT message and process it. Subscribed messages are delivered to a callback previously set by `.set_callback()` method. Other (internal) MQTT messages processed internally.

- `check_msg()`:

- Checks whether a pending message from server is available. If not, returns immediately with `None`. Otherwise, does the same processing as `wait_msg`.



MicroPython: a simple publisher

```
# file: mp_sipub.py
```

```
from mqtt import MQTTClient
```

```
import pycom
```

```
import sys
```

```
import time
```

```
import ufun
```

```
wifi_ssid = 'THE_NAME_OF_THE_AP'
```

```
wifi_passwd = ''
```

```
THE_BROKER = "iot.eclipse.org"
```

```
THE_TOPIC = "test/SRM2018"
```

```
CLIENT_ID = ""
```

```
def settimeout(duration):
```

```
    pass
```

```
def get_data_from_sensor(sensor_id="RAND"):
```

```
    if sensor_id == "RAND":
```

```
        return ufun.random_in_range()
```



```
### if __name__ == "__main__":  
  
ufun.connect_to_wifi(wifi_ssid, wifi_passwd)  
  
client = MQTTClient(CLIENT_ID, THE_BROKER, 1883)
```

```
print ("Connecting to broker: " + THE_BROKER)
```

```
try:
```

```
    client.connect()
```

```
except OSError:
```

```
    print ("Cannot connect to broker: " + THE_BROKER)  
    sys.exit()
```

```
print ("Connected to broker: " + THE_BROKER)
```

```
print('Sending messages...')
```

```
while True:
```

```
    # creating the data
```

```
the_data = get_data_from_sensor()
```

```
# publishing the data
```

```
client.publish(THE_TOPIC, str(the_data))
```

```
print("Published message with value: {}".format(the_data))
```

```
time.sleep(1)
```





MicroPython: a simple subscriber

```
# file: mp_sisub.py
```

```
from mqtt import MQTTClient
import pycom
import sys
import time

import ufun

wifi_ssid = 'THE_NAME_OF_THE_AP'
wifi_passwd = ''

THE_BROKER = "iot.eclipse.org"
THE_TOPIC = "test/SRM2018"
CLIENT_ID = ""

def settimeout(duration):
    pass

def on_message(topic, msg):
    print("Received msg: ", str(msg),
          "with topic: ", str(topic))
```

```
## if __name__ == "__main__":
ufun.connect_to_wifi(wifi_ssid, wifi_passwd)

client = MQTTClient(CLIENT_ID, THE_BROKER, 1883)
client.set_callback(on_message)

print ("Connecting to broker: " + THE_BROKER)
try:
    client.connect()
except OSError:
    print ("Cannot connect to broker: " + THE_BROKER)
    sys.exit()
print ("Connected to broker: " + THE_BROKER)

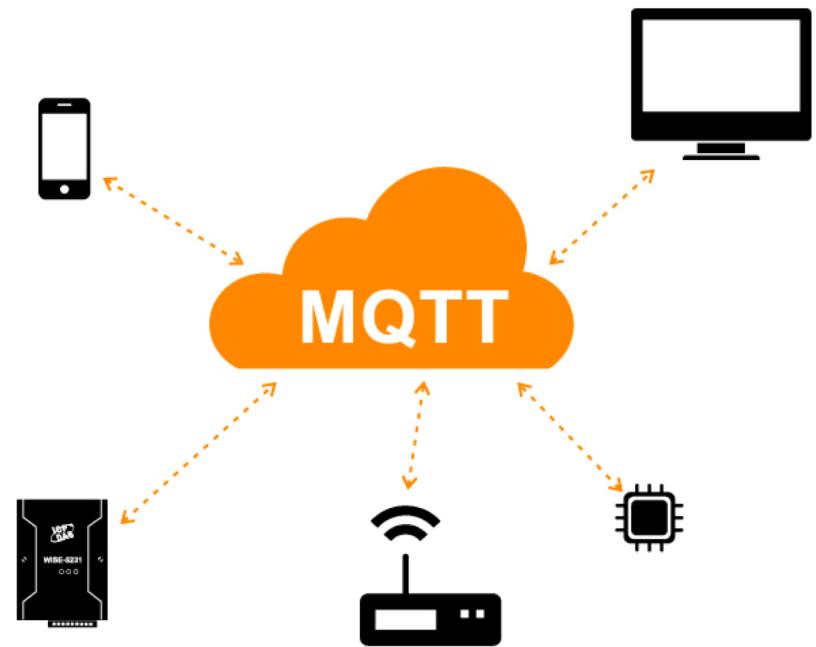
client.subscribe(THE_TOPIC)

print('Waiting messages...')
while 1:
    client.check_msg()
```



Intro to MQTT

- Some final details



- The keep alive functionality assures that the connection is still open and both broker and client are connected to one another.
- The client specifies a time interval in seconds and communicates it to the broker during the establishment of the connection.
 - The interval is the longest possible period of time which broker and client can endure without sending a message.
 - If the broker doesn't receive a PINGREQ or any other packet from a particular client, it will close the connection and send out the [last will and testament message](#) (if the client had specified one).
- Good to Know
 - The MQTT client is responsible of setting the right keep alive value.
 - The maximum keep alive is 18h 12min 15 sec.
 - If the keep alive interval is set to 0, the keep alive mechanism is deactivated.

- When clients connect, they can specify an optional “will” message, to be delivered if they are unexpectedly disconnected from the network.
 - (In the absence of other activity, a 2-byte ping message is sent to clients at a configurable interval.)
- This “last will and testament” can be used to notify other parts of the system that a node has gone down.

MQTT-Packet:	
CONNECT	
contains:	Example
clientId	“client-1”
cleanSession	true
username (optional)	“hans”
password (optional)	“letmein”
lastWillTopic (optional)	“/hans/will”
lastWillQos (optional)	2
lastWillMessage (optional)	“unexpected exit”
lastWillRetain (optional)	false
keepAlive	60

WHEN?

- A persistent session saves all information relevant for the client on the broker. The session is identified by the **clientId** provided by the client on connection establishment
- So what will be stored in the session?
 - Existence of a session, even if there are no subscriptions
 - All subscriptions
 - All messages in a Quality of Service (QoS) 1 or 2 flow, which are not confirmed by the client
 - All new QoS 1 or 2 messages, which the client missed while it was offline
 - All received QoS 2 messages, which are not yet confirmed to the client
 - That means even if the client is offline all the above will be stored by the broker and are available right after the client reconnects.
- Persistent session on the client side
 - Similar to the broker, each MQTT client must store a persistent session too. So when a client requests the server to hold session data, it also has the responsibility to hold some information by itself:
 - All messages in a QoS 1 or 2 flow, which are not confirmed by the broker
 - All received QoS 2 messages, which are not yet confirmed to the broker

- First of all:
 - Don't use a leading forward slash
 - Don't use spaces in a topic
 - Use only ASCII characters, avoid non printable characters
- Then, try to..
 - Keep the topic short and concise
 - Use specific topics, instead of general ones
 - Don't forget extensibility
- Finally, be careful and don't subscribe to #

Why?

- MQTT has the option for Transport Layer Security (TLS) encryption.
- MQTT also provides username/password authentication with the broker.
 - Note that the password is transmitted in clear text. Thus, be sure to use TLS encryption if you are using authentication.



Smart homes can be easily hacked via unsecured MQTT servers

<https://www.helpnetsecurity.com/2018/08/20/unsecured-mqtt-servers/>

In fact, by using the Shodan IoT search engine, Avast researchers found over 49,000 MQTT servers exposed on the Internet and, of these, nearly 33,000 servers have no password protection, allowing attackers to access them and all the messages flowing through it.

TOTAL RESULTS

49,197

TOP COUNTRIES



China

12,151

United States

8,257

Germany

3,092

Korea, Republic of

2,003

Hong Kong

2,002

TOTAL RESULTS

32,888

TOP COUNTRIES



China

8,446

United States

4,733

Germany

1,719

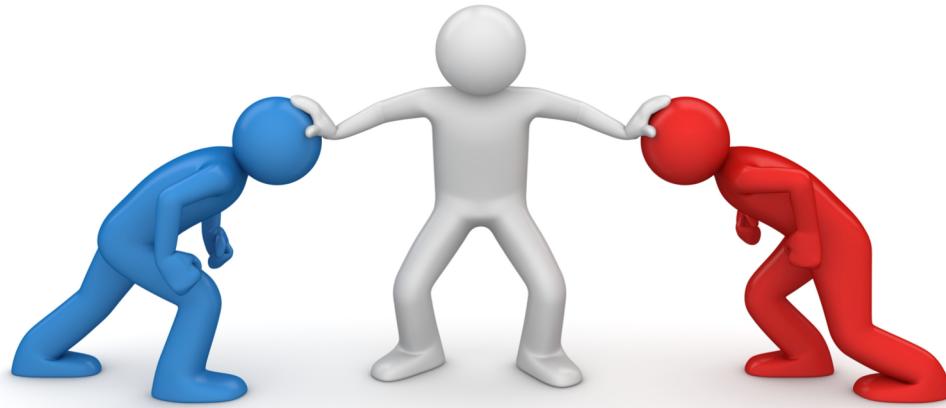
Hong Kong

1,614

Taiwan

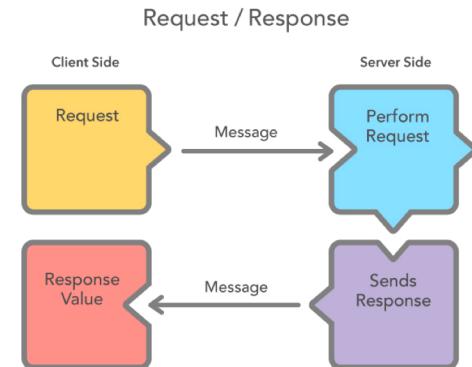
1,565

- Can they really be compared?!?!
- MQTT was created basically as a lightweight messaging protocol for lightweight communication between devices and computer systems
- REST stands on the shoulders of the almighty HTTP
- So it's better to understand their weak and strong points and build a system taking the best of both worlds... if required



- It is always independent of the type of platform or languages
 - The only thing is that it is indispensable that the responses to the requests should always take place in the language used for the information exchange, normally XML or JSON.
- It is stateless → This allows for scalability, by adding additional server nodes behind a load balancer
 - No state can be stored on servers: “[keep the application state on the client.](#)”
 - All messages exchanged between client and server have all the context needed to know what to do with the message.

- Today's real world embedded devices for IoT usually lacks the ability to handle high-level protocols like HTTP and they may be served better by lightweight binary protocols.
- It is **PULL based**. This poses a problem when services depend on being up to date with data they don't own and manage.
 - Being up to date requires polling, which quickly add up in a system with enough interconnected services.
 - Pull style can produce heavy unnecessary workloads and bandwidth consumption due to for example a request/response polling-based monitoring & control systems
- It is based on one-to-one interaction:



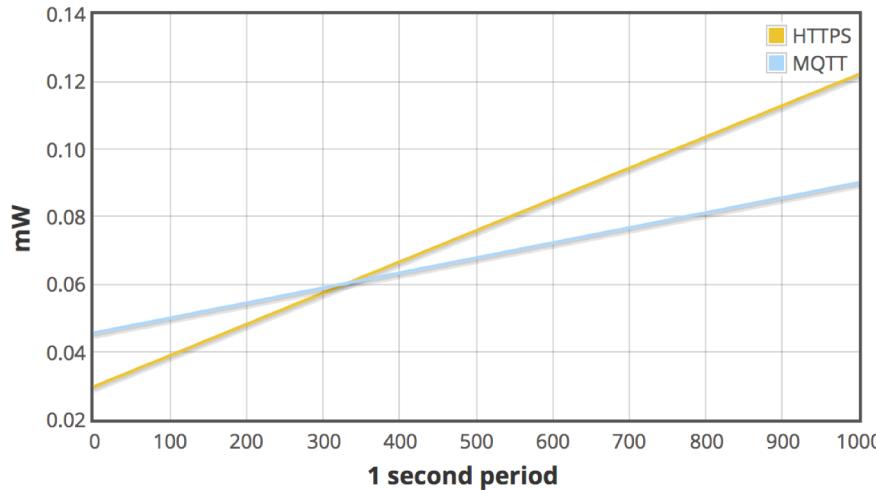
- **Push based:** no need to continuously look for updates
- It has built-in function useful for reliable behavior in an unreliable or intermittently connected wireless environments.
 1. “last will & testament” so all apps know immediately if a client disconnects ungracefully,
 2. “retained message” so any user re-connecting immediately gets the very latest information, etc.
- Useful for one-to-many, many-to-many applications
- Small memory footprint protocol, with reduced use of battery

Energy usage: some number

amount of power taken to establish the initial connection to the server:

% Battery Used			
3G		Wifi	
HTTPS	MQTT	HTTPS	MQTT
0.02972	0.04563	0.00228	0.00276

3G – 240s Keep Alive – % Battery Used Creating and Maintaining a Connection



cost of 'maintaining' that connection (in % Battery / Hour):

	% Battery / Hour			
	3G		Wifi	
Keep Alive (Seconds)	HTTPS	MQTT	HTTPS	MQTT
60	1.11553	0.72465	0.15839	0.01055
120	0.48697	0.32041	0.08774	0.00478
240	0.33277	0.16027	0.02897	0.00230
480	0.08263	0.07991	0.00824	0.00112

you'd save ~4.1% battery per day just by using MQTT over HTTPS to maintain an open stable connection.

<http://stephendnicholas.com/posts/power-profiling-mqtt-vs-https>

- If the broker fails...
- Does not define a standard client API, so application developers have to select the best fit.
- Does not include many features that are common in Enterprise Messaging Systems like:
 - expiration, timestamp, priority, custom message headers, ...
- Does not have a **point-to-point** (aka queues) messaging pattern
 - Point to Point or One to One means that there can be more than one consumer listening on a queue but only one of them will be get the message
- Maximum message size 256MB

A large, stylized word cloud centered around the words "Thank You" in multiple languages. The word "Thank" is at the top left, "You" is at the bottom right, and "You" is in the center. The surrounding text is in various colors (red, orange, yellow, green, blue, purple) and includes many other words related to gratitude and thanks in different languages.

