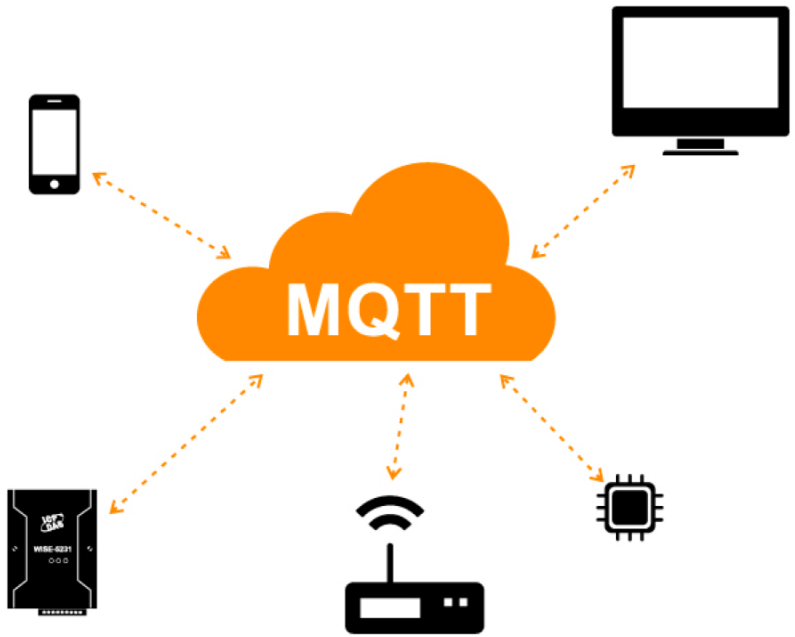
Intro to MQTT… with something of REST

Pietro Manzoni

Universitat Politecnica de Valencia (UPV)

Valencia - SPAIN

pmanzoni@disca.upv.es

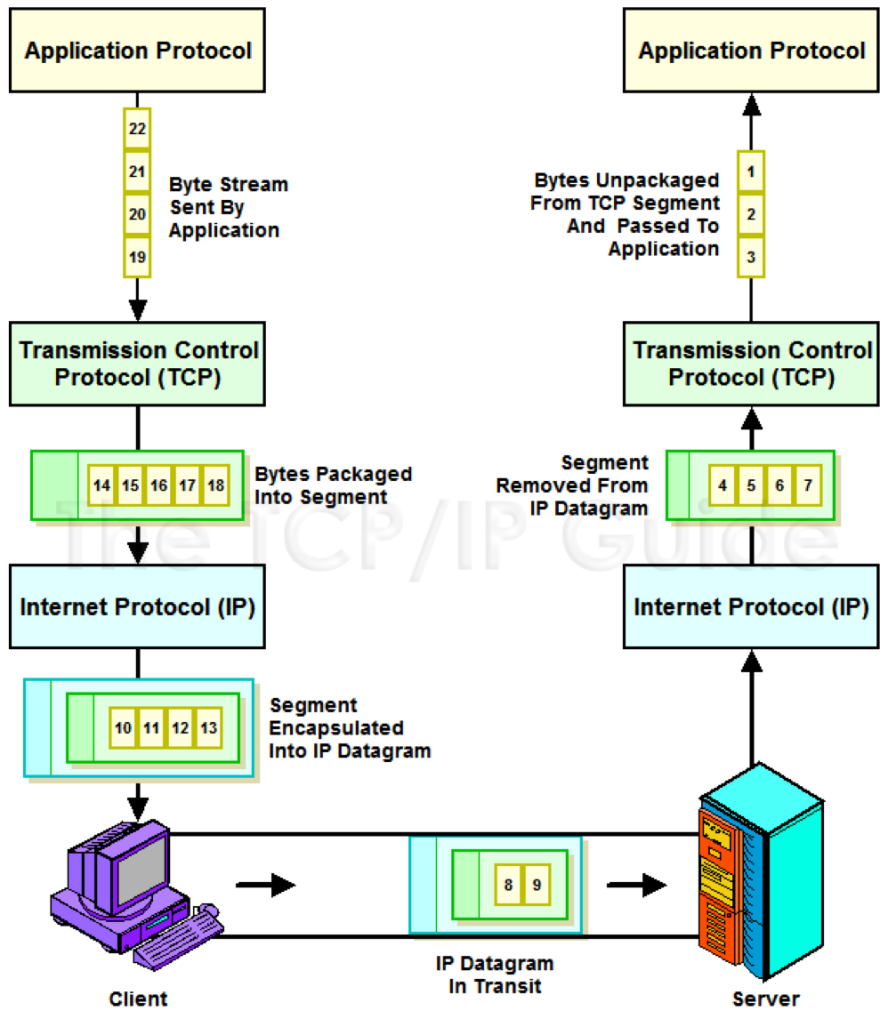


Universitat Politècnica de València

- The *Universitat Politècnica de València* (UPV) is a Spanish public educational institution founded in 1968.
- Its academic community comprises 36.823 students, almost 2.661 lecturers and researchers, and 1.422 administration and services professionals.
- It has three campuses: the main one in the city of Valencia and the other two in Alcoy and Gandia.
- The Vera Campus covers around 840.000 m² and is almost 2 km long. It is a pedestrian campus with over 123.000 m² of green areas.
- UPV is composed of 10 schools, 3 faculties and 2 higher polytechnic schools.

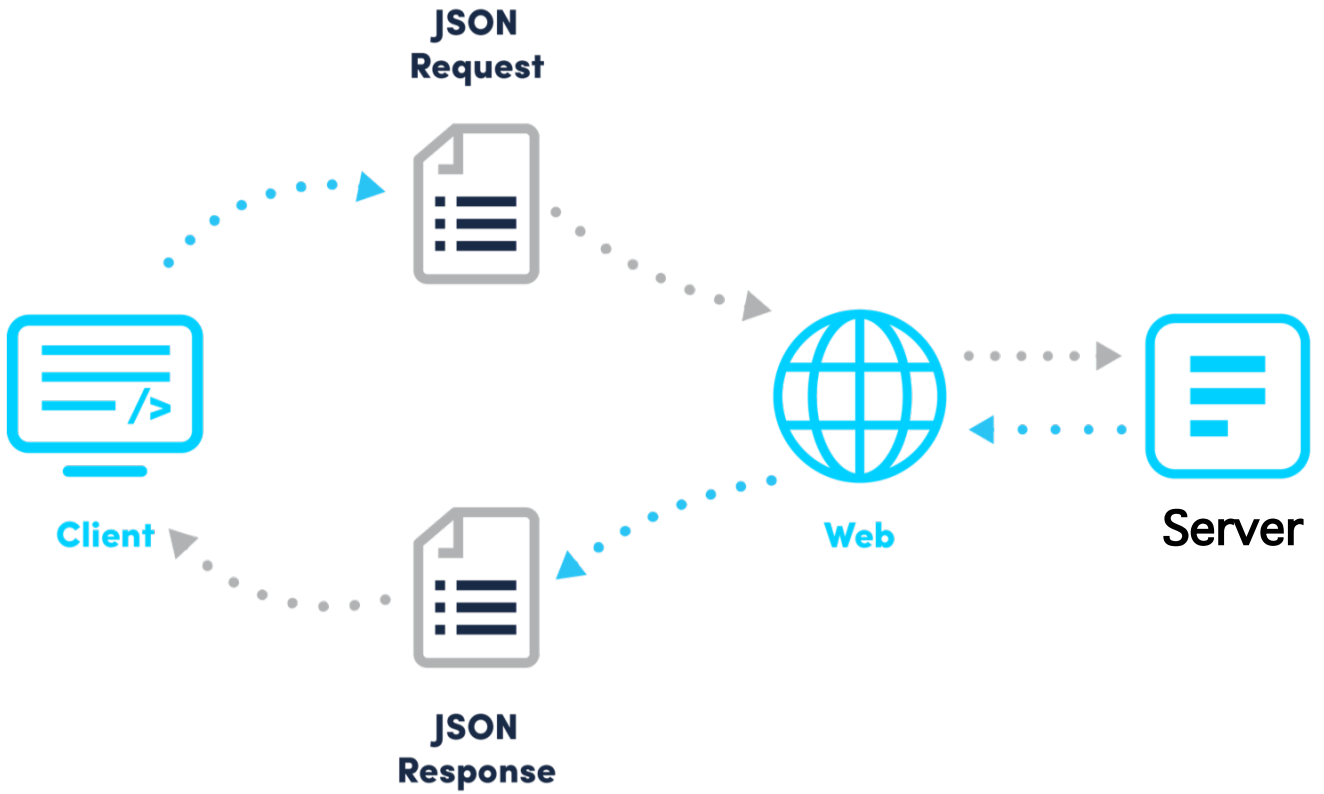
From "byte streams" to "messages"

- 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,
  - uucp



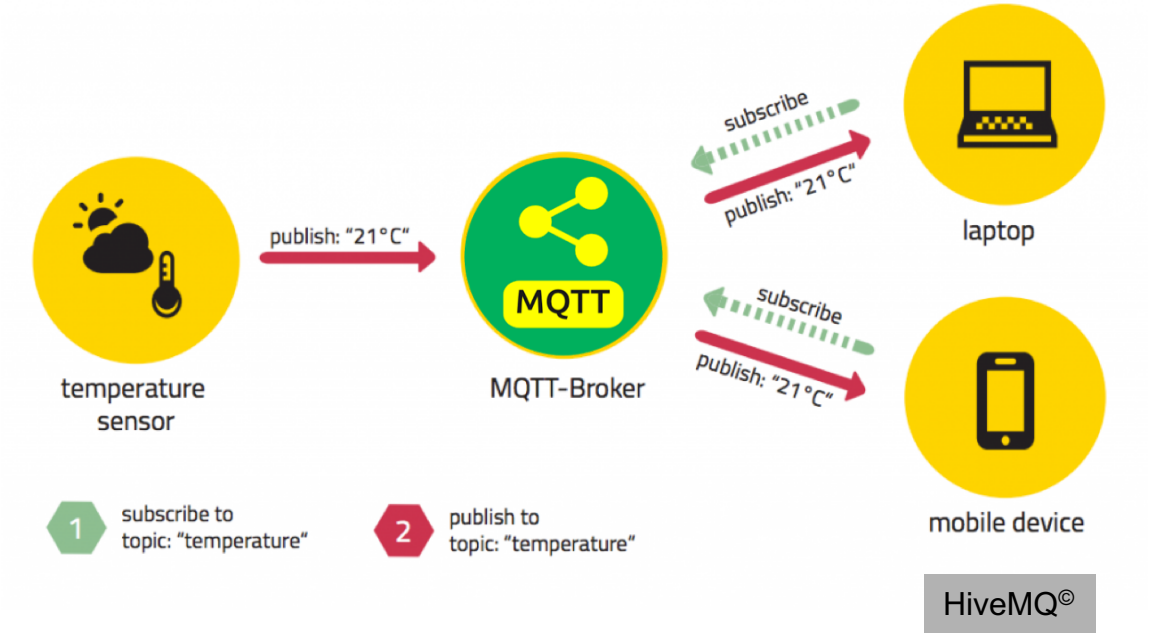
Interchanging messages: request/response paradigm

- **REST**: Representational State Transfer
- Widely used; based on HTTP
- Lighter version: **CoAP**



Interchanging message: pub/sub paradigm

- Publish/Subscriber
  - aka: producer/consumer
- Growing paradigm
  - E.g., https://cloud.google.com/iot/docs/how-tos/mqtt-bridge
- Various solutions
  - **MQTT**, AMQP, XMPP (was Jabber)



Data Representation

{“value": 237} vs. <value>237</value>

- Data-interchange format: (1) should be easy for humans to read and write, and (2) should be easy for machines to parse and generate
- Two main formats:

JavaScript Object Notation (JSON) [http://www.json.org/]

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
      "menuitem": [
         {"value": "New", "onclick": ”NewDoc()"},
         {"value": "Open", "onclick": "OpenDoc()"},
         {"value": "Close", "onclick": "CloseDoc()"}
      ]
    }
  }
}
```

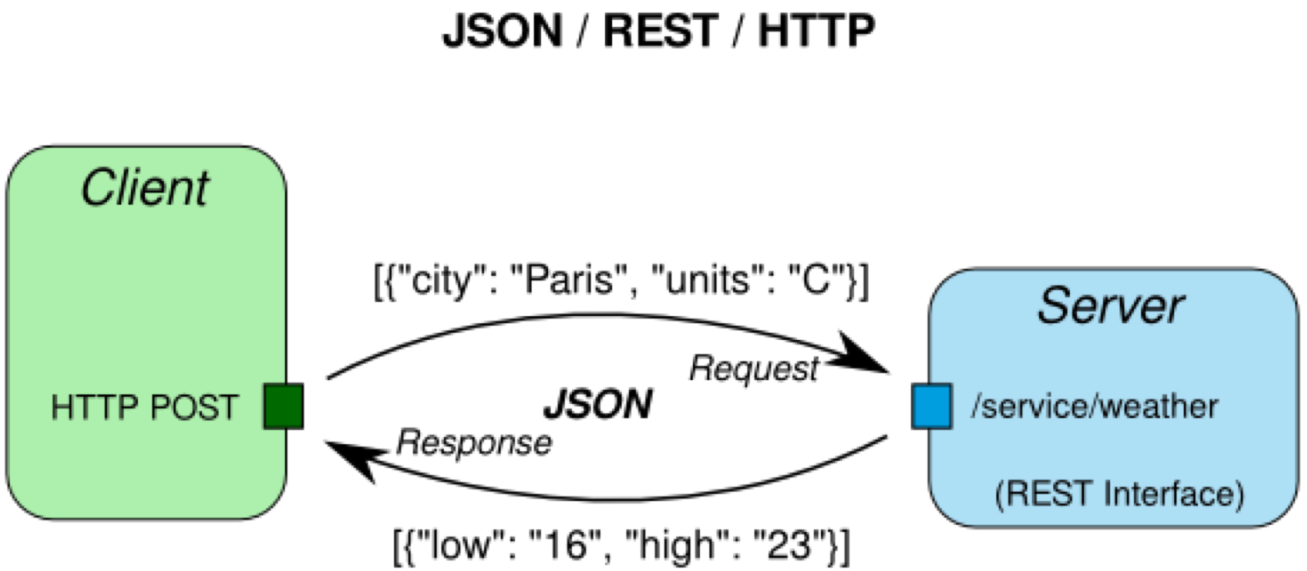
XML

```
<menu>
  <id>file</id>
  <value>File</value>
  <popup>
    <menuitem>
      <value>New</value>
      <onclick>NewDoc()</onclick>
    </menuitem>
    <menuitem>
      <value>Open</value>
      <onclick>OpenDoc()</onclick>
    </menuitem>
    <menuitem>
      <value>Close</value>
      <onclick>CloseDoc()</onclick>
    </menuitem>
  </popup>
</menu>
```

JSON and Python

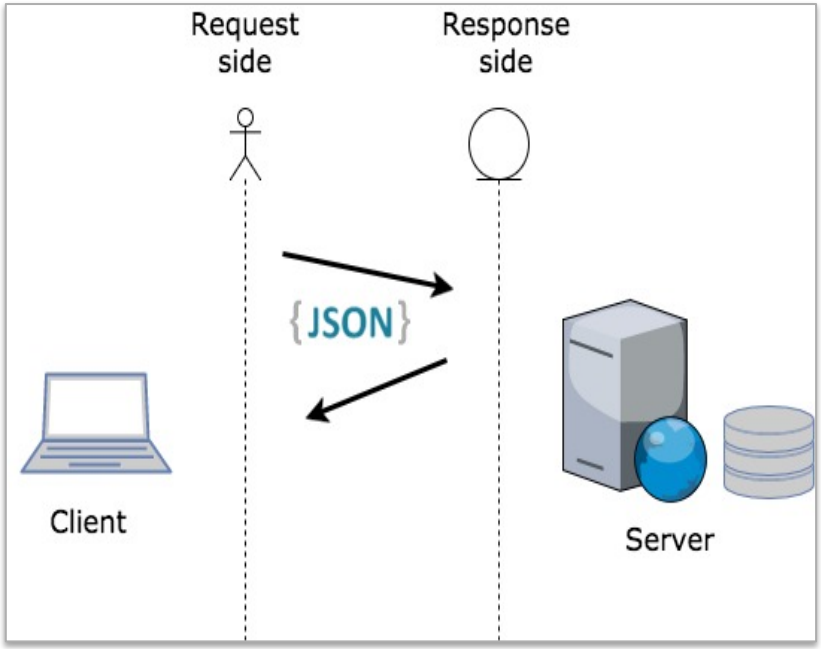
```
>>> import json

>>> d = {'sensorId': 'temp1', 'Value': 25}

>>> d
{'sensorId': 'temp1', 'Value': 25}
>>> d['sensorId']
'temp1'

>>> dj = json.dumps(d)
>>> dj
'{"sensorId": "temp1", "Value": 25}'

>>> nd = json.loads(dj)
>>> nd
{u'sensorId': u'temp1', u'Value': 25}
>>> nd['sensorId']
u'temp1'
```

REST: Representational State Transfer



REST and HTTP

- REST stands for Representational State Transfer.
  - It basically leverages the HTTP protocol and its related frameworks to provide data services.
- The motivation for REST was to capture the characteristics of the Web which made the Web successful.
  - Make a Request – Receive Response – Display Response
- REST is not a standard… but it uses several standards:
  - HTTP
  - URL
  - Resource Representations: XML/HTML/GIF/JPEG/etc
  - Resource Types, MIME Types: text/xml, text/html, image/gif, image/jpeg, etc



REST is widely used

- Twitter:
  - https://developer.twitter.com/en/docs/basics/getting-started
- Facebook:
  - https://developers.facebook.com/docs/graph-api
- Amazon offers several REST services, e.g., for their S3 storage solution
  - https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
- Tesla Model S uses an (undocumented) REST API between the car systems and its Android/iOS apps.
  - https://timdorr.docs.apiary.io/#
- Google Maps:
  - https://developers.google.com/maps/web-services/
  - Try: `http://maps.googleapis.com/maps/api/geocode/json?address=lecco`

The Wireshark view

```
171 3.765959 192.168.1.102  216.58.211.234  HTTP  455 GET /maps/api/geocode/json?address=lecco HTTP/1.1
173 3.822725 216.58.211.234 192.168.1.102   HTTP  899 HTTP/1.1 200 OK  (application/json)
```

```
▶ Frame 171: 455 bytes on wire (3640 bits), 455 bytes captured (3640 bits) on interface 0
▶ Ethernet II, Src: 98:5a:eb:d7:6c:6f, Dst: c0:c1:c0:7e:05:b1
▶ Internet Protocol Version 4, Src: 192.168.1.102, Dst: 216.58.211.234
▶ Transmission Control Protocol, Src Port: 50629 (50629), Dst Port: http (80), Seq: 1, Ack: 1, Len: 389
▼ Hypertext Transfer Protocol
  ▶ GET /maps/api/geocode/json?address=lecco HTTP/1.1\r\n
    Host: maps.googleapis.com\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:54.0) Gecko/20100101 Firefox/54.0\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3\r\n
    Accept-Encoding: gzip, deflate\r\n
    Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    \r\n
```

```
▼ Hypertext Transfer Protocol
  ▼ HTTP/1.1 200 OK\r\n
    ▶ [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      Request Version: HTTP/1.1
      Status Code: 200
      Response Phrase: OK
    Content-Type: application/json; charset=UTF-8\r\n
    Date: Thu, 29 Jun 2017 07:54:23 GMT\r\n
    Expires: Fri, 30 Jun 2017 07:54:23 GMT\r\n
    Cache-Control: public, max-age=86400\r\n
    Vary: Accept-Language\r\n
    Access-Control-Allow-Origin: *\r\n
    Content-Encoding: gzip\r\n
    Server: mafe\r\n
  ▼ Content-Length: 476\r\n
      [Content length: 476]
    X-XSS-Protection: 1; mode=block\r\n
    X-Frame-Options: SAMEORIGIN\r\n
    \r\n
```



```
{
   "results" : [
      {
         "address_components" : [
            {
               "long_name" : "Lecco",
               "short_name" : "Lecco",
               "types" : [ "locality", "political" ]
            },
            {
               "long_name" : "Lecco",
               "short_name" : "Lecco",
               "types" : [ "administrative_area_level_3", "political" ]
            },
            {
               "long_name" : "Provincia di Lecco",
               "short_name" : "LC",
               "types" : [ "administrative_area_level_2", "political" ]
            },
            {
               "long_name" : "Lombardia",
               "short_name" : "Lombardia",
               "types" : [ "administrative_area_level_1", "political" ]
            },
            {
               "long_name" : "Italia",
               "short_name" : "IT",
               "types" : [ "country", "political" ]
            },
            {
               "long_name" : "23900",
               "short_name" : "23900",
               "types" : [ "postal_code" ]
            }
         ],
         "formatted_address" : "23900 Lecco LC, Italia",
         "geometry" : {
            "bounds" : {
               "northeast" : {
                  "lat" : 45.8870552,
                  "lng" : 9.424340899999999
```

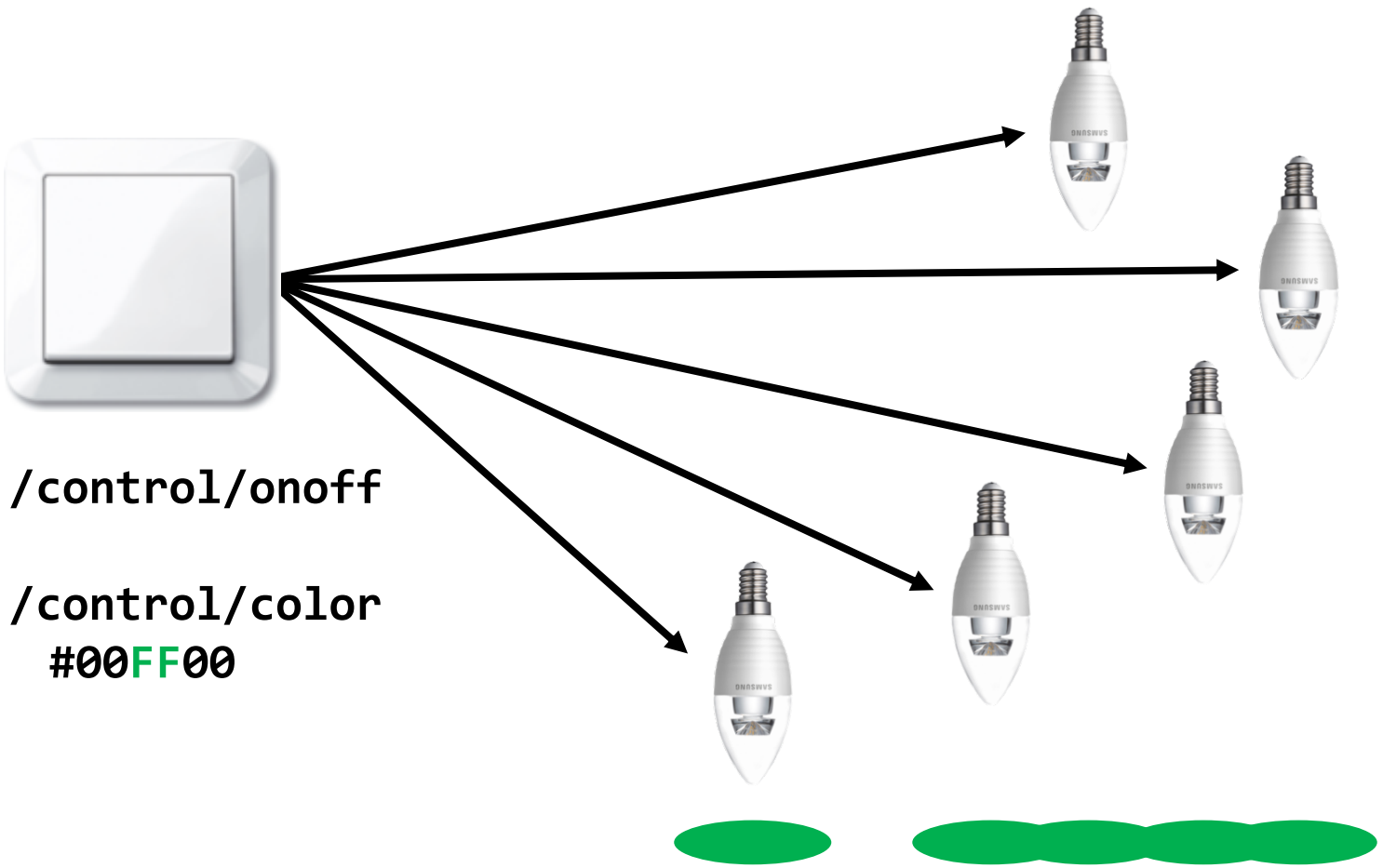
REST: the resources

- The key abstraction of information in REST is a resource.
- A resource is a conceptual mapping to a set of entities
  - Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person or a device), and so on
- Represented with a global identifier (URI in HTTP).
  For example:
  - http://www.acme.com/device-management/managed-devices/{device-id}
    - http://myhome.com/api/states/livingroom/sensor1/temperature
  - http://www.potus.org/user-management/users/{id}
  - http://www.library.edu/books/ISBN-0011/authors
- As you traverse the path from more generic to more specific, you are navigating the data

REST for devices control

```
GET /status/power
```

```
all-lights.floor-d.example.com
```

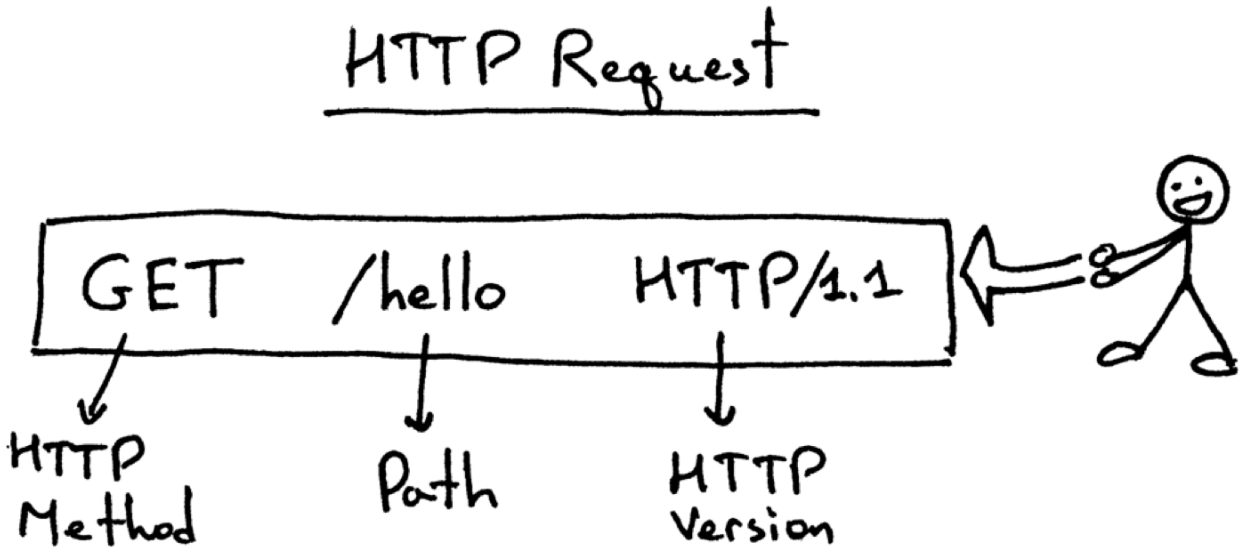


```
PUT /control/onoff

PUT /control/color
      #00FF00
```

Actions

- Represent the **operations** that can be performed on the **resources**

- HTTP GET
- HTTP POST
- HTTP PUT
- HTTP DELETE



Running example of a resource: a 'todo' list

```
>>> tasks
[
{'id': 2345, 'summary': 'recipe for tiramisu', 'description':
'call mom and ask...'},
{'id': 3657, 'summary': 'what to buy today', 'description': '6
eggs, carrots, spaghetti'}
]

>>> tasks[1]['id']
2345

>>> tasks[1]
{'id': 2345, 'summary': 'recipe for tiramisu', 'description':
'call mom and ask...'}
```

HTTP GET

- How clients ask for the information they seek.
- Issuing a GET request transfers the data from the server to the client in some representation (JSON, XML, …)

- **`GET /tasks/`**
  - Return a **list** of items on a *todo list*, in the format {"id": <item_id>, "summary": <one-line summary>}

- **`GET /tasks/<item_id>/`**
  - Fetch all available information for **a specific todo item**, in the format {"id": <item_id>, "summary": <one-line summary>, "description" : <free-form text field>}

HTTP PUT, HTTP POST

- HTTP POST creates a resource
- HTTP PUT updates a resource

- POST /tasks/
  - Create a new todo item. The **POST body is a JSON object** with two fields: "summary" (must be under 120 characters, no newline), and "description" (free-form text field).
  - On success, the status code is 201, and the response body is an object with one field: the id created by the server (e.g., { "id": 3792 }).

- PUT /tasks/<item_id>/
  - Modify an existing task. The **PUT body is a JSON object** with two fields: "summary" (must be under 120 characters, no newline), and "description" (free-form text field).

HTTP PATCH

- PATCH is defined in RFC 5789. It requests that **a set of changes** described in the request entity be applied to the resource identified by the Request- URI.
  Let's look at an example:

```
{
    'id': 3657,
    'summary': 'what to buy today',
    'description': '6 eggs, carrots, spaghetti'
}
```

- If you want to modify this entry, you choose between PUT and PATCH. A PUT might look like this:

```
PUT /tasks/3657/
{
    'id': 3657,
    'summary': 'what to buy today',
    'description': '6 eggs, carrots, spaghetti, bread'
}
```

- You can accomplish the same using PATCH:

```
PATCH /tasks/3657/
{
    'description': '6 eggs, carrots, spaghetti, bread'
}
```

- The PUT included all of the parameters on this user, while PATCH only included the one that was being modified.

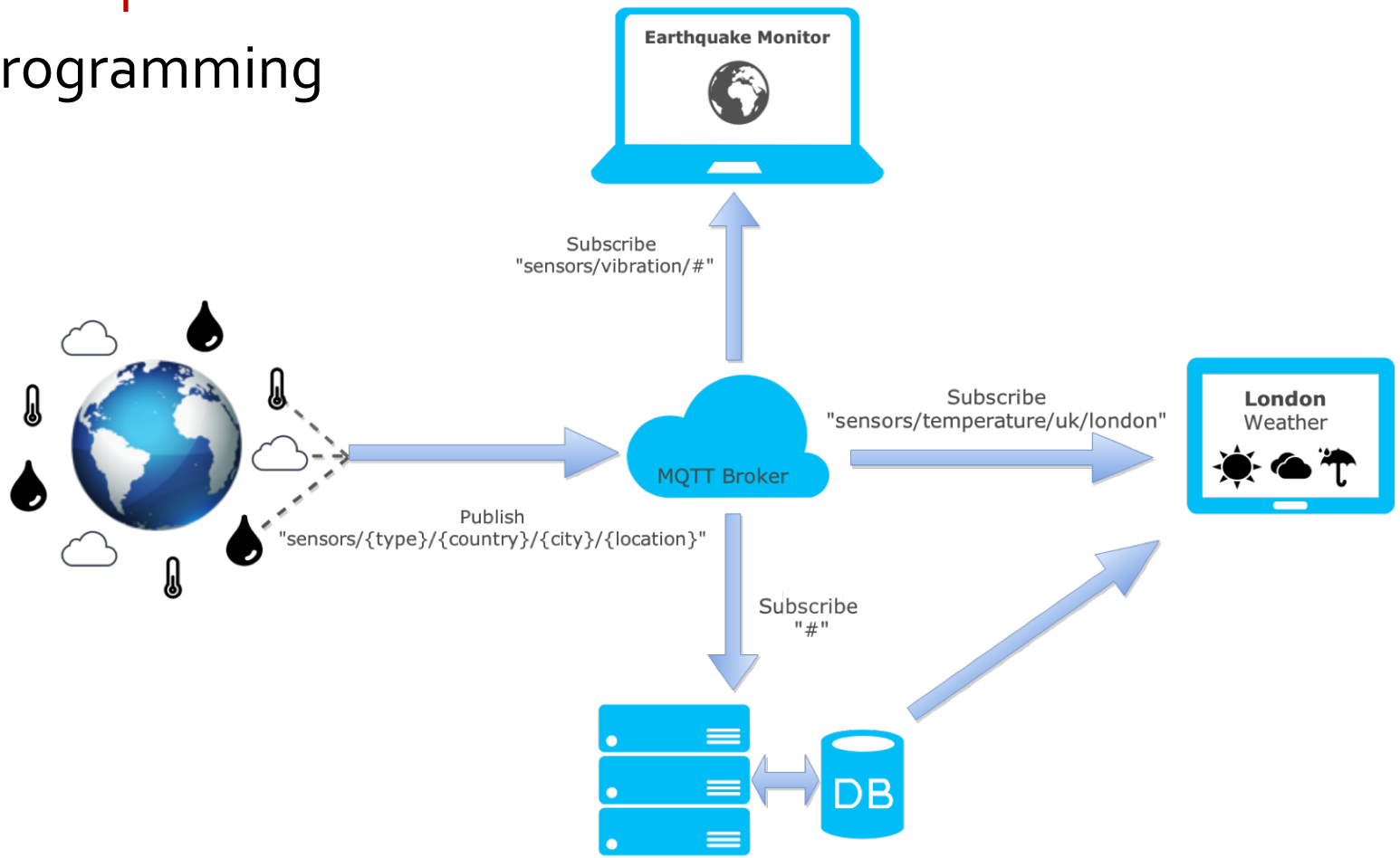
HTTP DELETE

- Removes the resource identified by the URI

- DELETE /tasks/<item_id>/
  - Mark the item as done. (I.e., strike it off the list, so GET /tasks/ will not show it.)
  - The response body is empty.

MQTT

- Basic concepts
- Basic programming



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

Message Queuing Telemetry Transport

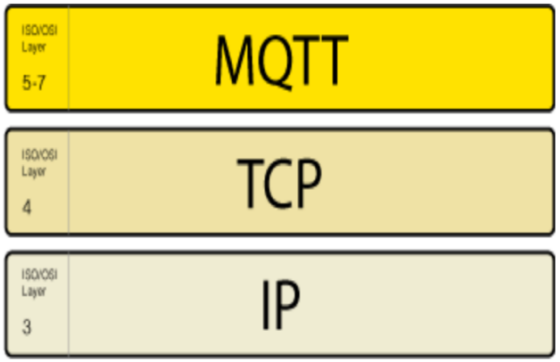
MQTT

- A lightweight publish-subscribe protocol that can run on embedded devices and mobile platforms ➔ http://mqtt.org/
  - The MQTT community wiki: https://github.com/mqtt/mqtt.github.io/wiki
  - A very good tutorial: http://www.hivemq.com/mqtt-essentials/
- Designed to provide a low latency two-way communication channel and efficient distribution to one or many receivers.
  - Assured messaging over fragile networks
  - Maximum message size of 256MB
    - not really designed for sending large amounts of data
    - better at a high volume of low size messages.
  - Binary compress headers
  - Low power usage.

MQTT overview

- MQTT is an open standard with a short and readable protocol specification.
  - October 29th 2014: MQTT was officially approved as OASIS Standard.
- **MQTT 3.1.1** is the current version of the protocol.
  - OASIS MQTT TC: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt
  - Standard document here:

    `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`
- Works on top of the TCP protocol stack
  - 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*



MQTT Version 5

- The next version of MQTT will be version 5
  - Currently (10 April 2018) in Draft 19
  - You can find the working drafts here:

`https://www.oasis-open.org/committees/documents.php?wg_abbrev=mqtt&show_descriptions=yes`

- The key ideas in version 5 can be found here:

`https://www.oasis-open.org/committees/download.php/57616/Big%20Ideas%20for%20MQTT%20v5.pdf`

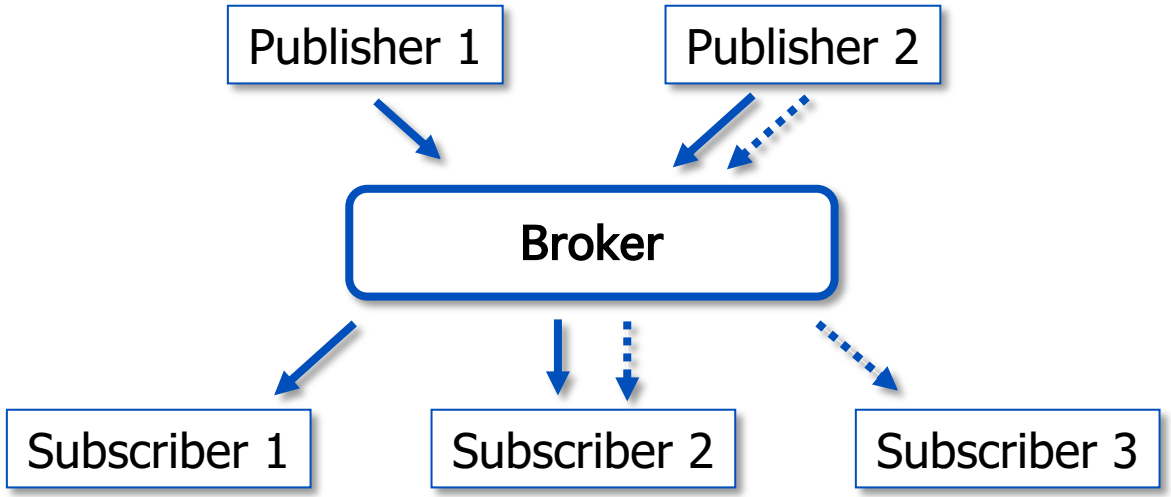
  - Basically:
    - Enhancements for **Scalability** and Large Scale Systems
    - **Improved Error Reporting**
    - Extensible Metadata
    - Added Support for Resource Constrained Clients and PerformanceImprovements

- If you are wondering what happened to version 4 then see:

`http://www.eclipse.org/community/eclipse_newsletter/2016/september/article3.php`

Publish/subscribe pattern

- Pub/Sub decouples 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.
  - This means that the publisher and subscriber don't know about the existence of one another.
- 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.



Topics

- 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

Topics wildcards

- 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
```

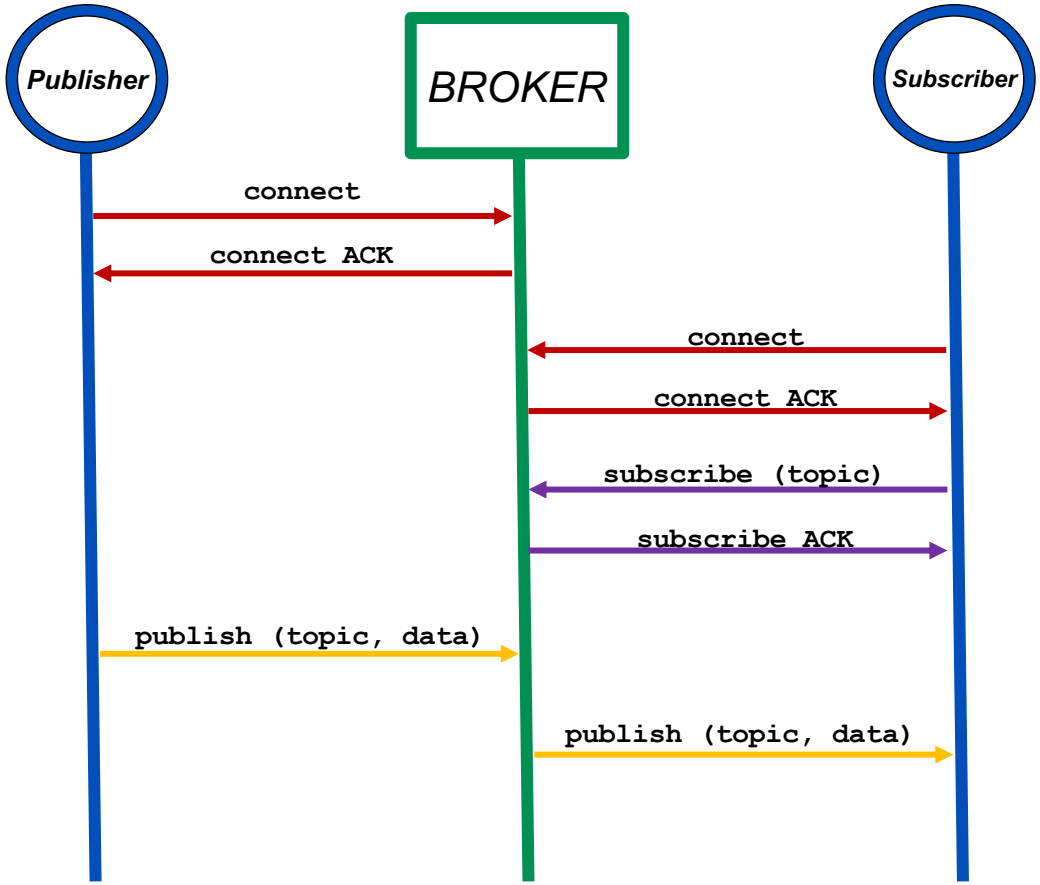
Topics best practices

- 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 #

Publish/subscribe interactions sequence



Connecting to the broker

MQTT-Packet:

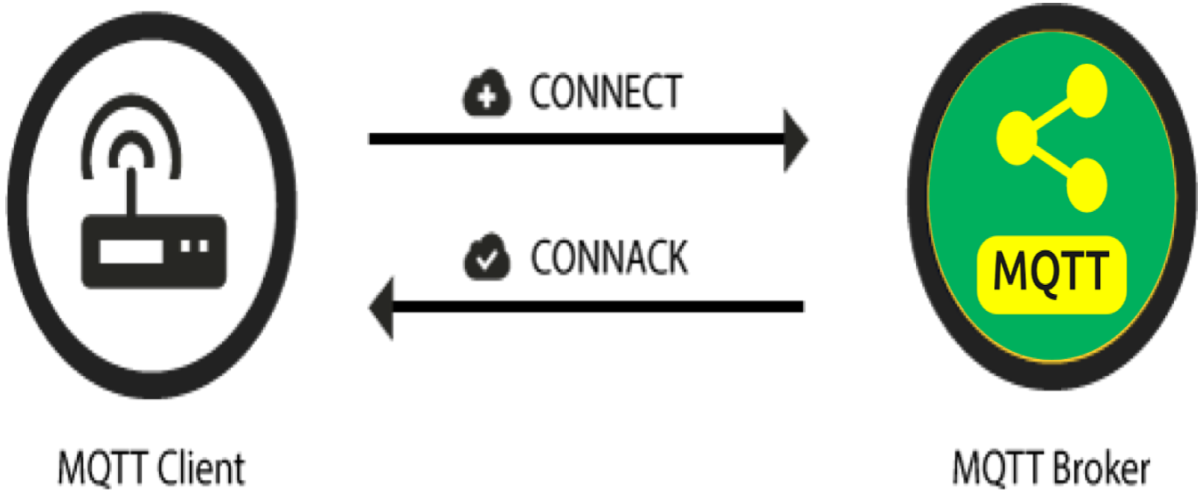
CONNECT

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

MQTT-Packet:

CONNACK

contains:	Example
sessionPresent	true
returnCode	0

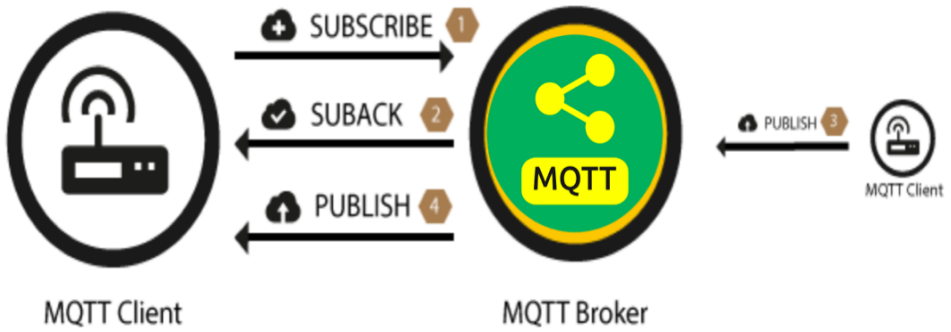


Subscribing

MQTT-Packet:

SUBSCRIBE

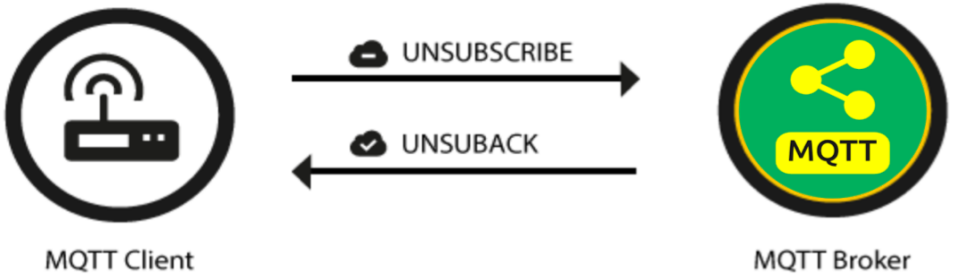
contains:		Example
packetId		4312
qos1	} (list of topic + qos)	1
topic1		"topic/1"
qos2	}	0
topic2		"topic/2"
...		...



MQTT-Packet:

SUBACK

contains:		Example
packetId		4313
returnCode 1	(one returnCode for each	2
returnCode 2	topic from SUBSCRIBE,	0
...	in the same order)	...



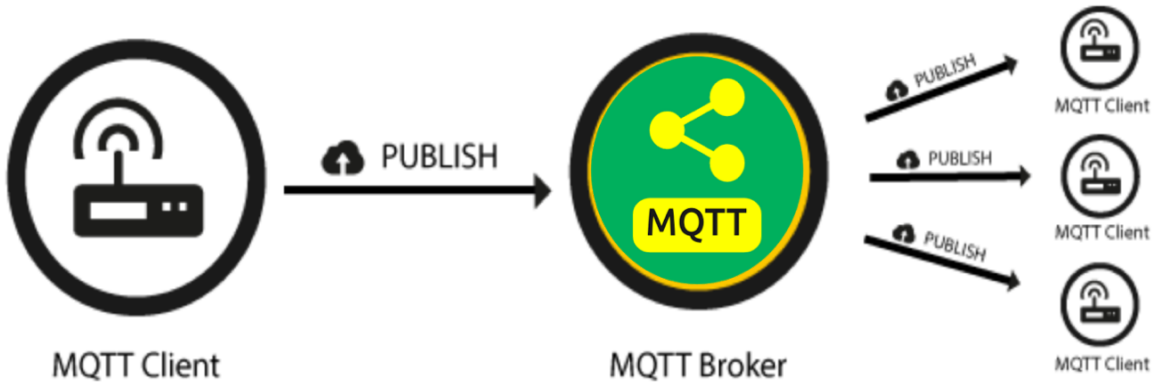
Publishing

- MQTT is data-agnostic and it totally depends on the use case how the payload is structured. It's completely up to the sender if it wants to send binary data, textual data or even full-fledged XML or JSON.

MQTT-Packet:

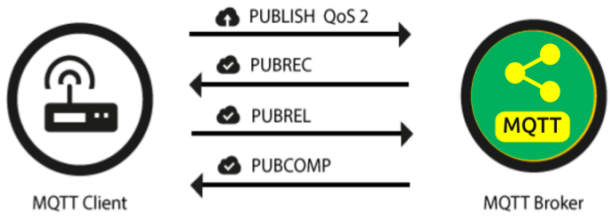
PUBLISH

contains:	Example
packetId (always 0 for qos 0)	4314
topicName	"topic/1"
qos	1
retainFlag	false
payload	"temperature:32.5"
dupFlag	false



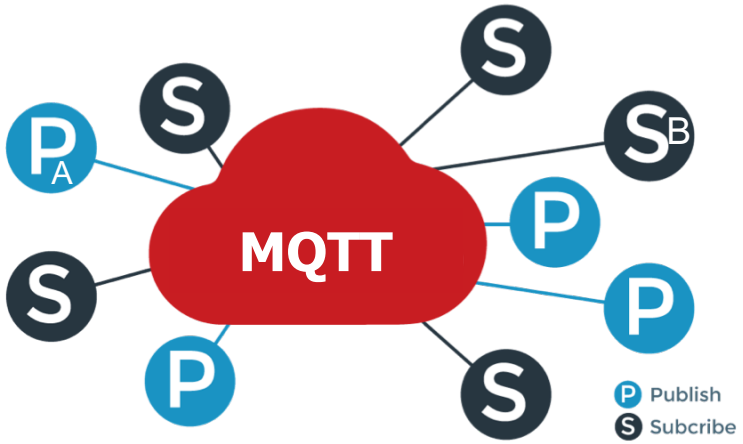
Quality of Service (QoS)

- 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.



QoS: good to know

- The QoS flows between a publishing and subscribing client are two different things and QoS can be different.
  - That means the QoS level can be different from client A, who publishes a message, and client B, who receives the published message.
  - If client B has subscribed to the broker with QoS 1 and client A sends a QoS 2 message, it will be received by client B with QoS 1. And of course it could be delivered more than once to client B, because QoS 1 only guarantees to deliver the message at least once.
- **Between the sender and the broker the QoS is defined by the sender.**



QoS: Best Practice

- Use QoS 0 when …
  - **You have a complete or almost stable connection between sender and receiver.** A classic use case is when connecting a test client or a front end application to a MQTT broker over a wired connection.
  - **You don't care if one or more messages are lost once a while.** That is sometimes the case if the data is not that important or will be send at short intervals, where it is okay that messages might get lost.
  - You don't need any message queuing. Messages are only queued for disconnected clients if they have QoS 1 or 2 and a persistent session.
- Use QoS 1 when …
  - **You need to get every message and your use case can handle duplicates.** The most often used QoS is level 1, because it guarantees the message arrives at least once. Of course your application must be tolerating duplicates and process them accordingly.
  - **You can't bear the overhead of QoS 2.** Of course QoS 1 is a lot fast in delivering messages without the guarantee of level 2.
- Use QoS 2 when …
  - **It is critical to your application to receive all messages exactly once.** This is often the case if a duplicate delivery would do harm to application users or subscribing clients. You should be aware of the overhead and that it takes a bit longer to complete the QoS 2 flow.
- Queuing of QoS 1 and 2 messages
  - All messages sent with QoS 1 and 2 will also be queued for offline clients, until they are available again. But queuing is only happening if the client has a persistent session.

Retained Messages!!!

- 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.
- The subscribing client doesn't have to match the exact topic, it will also receive a retained message if it subscribes to a topic pattern including wildcards.
  - For example client A publishes a retained message to myhome/livingroom/temperature and client B subscribes to myhome/# later on. **Client B will receive this retained message directly after subscribing.**
  - 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.
- 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.
- Warning: a retained message has nothing to do with a persistent session of any client

Persistent session

- 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

"Will" message

- 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.

WHEN?

MQTT-Packet:

CONNECT

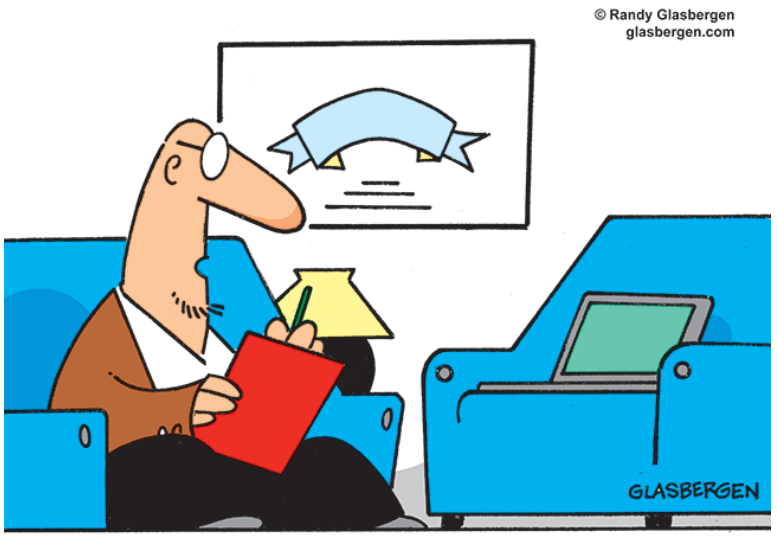
contains:	Example
clientId	"client-1"
cleanSession	true
username (optional)	"hans"
password (optional)	"letmein"
lastWillTopic (optional)	"/hans/will"
lastWillQos (optional)	2
lastWillMessage (optional)	"unexpected exit"
lastWillRetain (optional)	false
keepAlive	60

MQTT Keep alive

- The keep alive functionality assures that the connection is still open and both broker and client are connected to one another. Therefore 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.

A few word on security

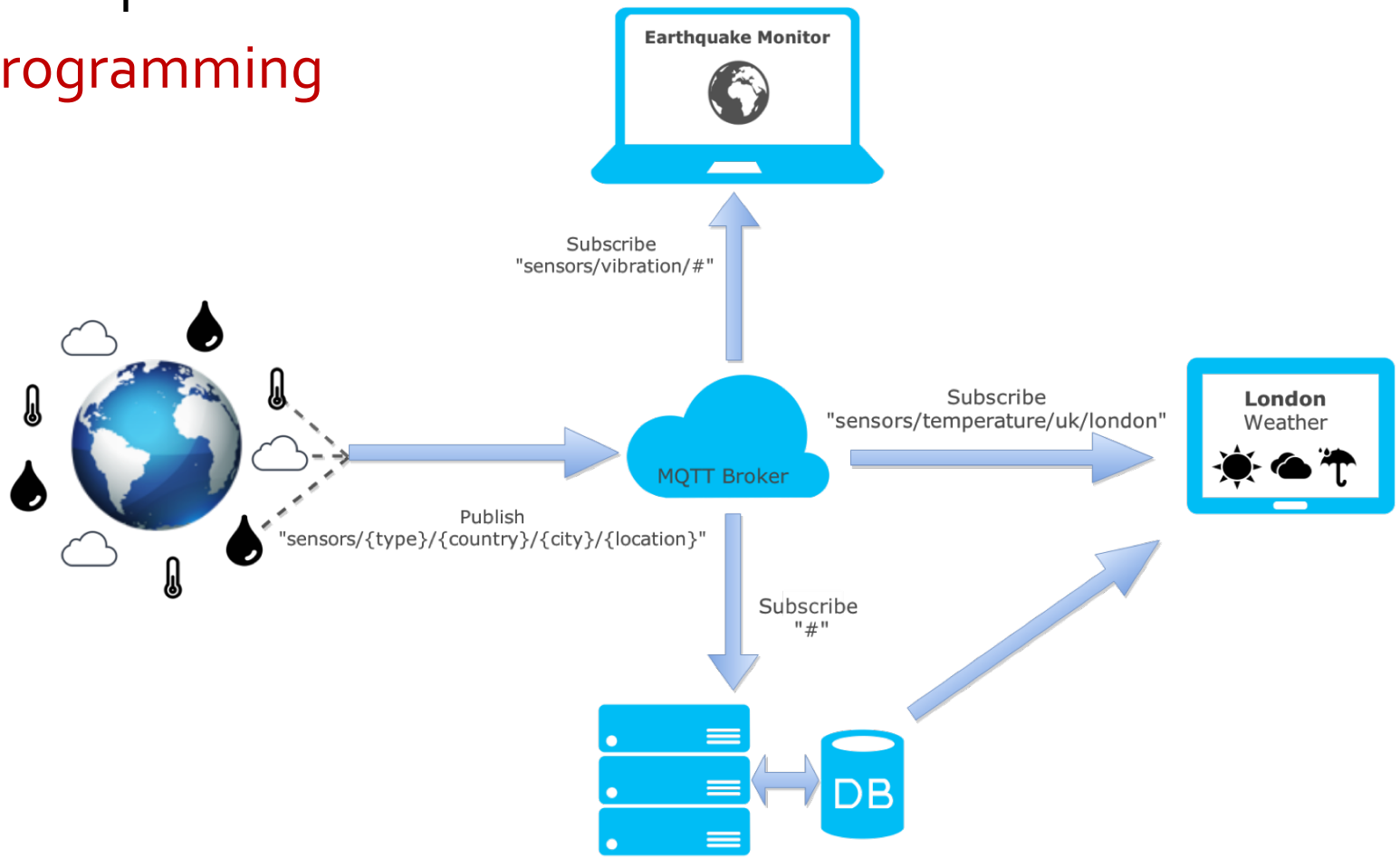
- MQTT has the option for Transport Layer Security (TLS) encryption.
- MQTT also provides username/password authentication with the broker.
  - Note that the password is transmitted in clear text. Thus, be sure to use TLS encryption if you are using authentication.



"It's not just you. We're all insecure in one way or another."

MQTT

- Basic concepts
- Basic programming



Software available

- Brokers:
  - **http://mosquitto.org/**
  - http://www.hivemq.com/
- A complete list here:
  https://github.com/mqtt/mqtt.github.io/wiki/servers
- Tools
  - https://github.com/mqtt/mqtt.github.io/wiki/tools

"Sandboxes" for brokers

- test.mosquitto.org
  - http://test.mosquitto.org/
- iot.eclipse.org
  - https://iot.eclipse.org/getting-started#sandboxes
- broker.hivemq.com
  - http://www.hivemq.com/try-out/
    - http://www.mqtt-dashboard.com/

- Ports:
  - standard: 1883
  - encrypted: 8883

Server	Broker	Port	Websocket
iot.eclipse.org	Mosquitto	1883 / 8883	n/a
broker.hivemq.com	HiveMQ	1883	8000
test.mosquitto.org	Mosquitto	1883 / 8883 / 8884	8080 / 8081
test.mosca.io	mosca	1883	80
broker.mqttdashboard.com	HiveMQ	1883	

Software available: clients

- Clients
  - **Eclipse Paho Python** (originally the mosquitto Python client)
    `http://www.eclipse.org/paho/`
    - Documentation: https://pypi.python.org/pypi/paho-mqtt
    - *or: http://www.eclipse.org/paho/clients/python/docs/*
    - Source: https://github.com/eclipse/paho.mqtt.python
- Web Clients
  - https://www.hivemq.com/blog/seven-best-mqtt-client-tools
  - http://www.hivemq.com/demos/websocket-client/
- A complete list:
  - https://github.com/mqtt/mqtt.github.io/wiki/libraries

Paho MQTT Python client: Main Methods

- The client class has several methods. The main ones are:
  - `connect()` and `disconnect()`
  - `subscribe()` and `unsubscribe()`
  - `publish()`

- Each of these methods is associated with a callback.
  - The callback is triggered by a broker response to a client command or message.

- So we have:
  - `connect()` generates `on_connect()` callback
  - `disconnect()` generates `on_disconnect()` callback
  - `subscribe()` generates `on_subscribe()` callback
  - `unsubscribe()` generates `on_unsubscribe()` callback
  - `publish()` generates `on_publish()` callback

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: the simplest subscriber

```
# File: example1.py

import paho.mqtt.client as mqtt

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

# 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, "return code: ", rc)

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe(THE_TOPIC)

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(THE_BROKER, 1883, 60)

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

Example 1: output

```
paho-code:pietro$ python example1.py
('Connected to ', 'test.mosquitto.org', 'port: ', 1883)
('Flags: ', {'session present': 0}, 'return code: ', 0)
$SYS/broker/connection/ks.ral.me.rnic/state 0
$SYS/broker/connection/Salem2.Public_Bridge/state 1
$SYS/broker/connection/RPi_MQTT_GESRV.bridgeTestMosquittoOrg/state 1
$SYS/broker/connection/br-john-jane/state 1
$SYS/broker/connection/cell_controller.bridge-01/state 1
$SYS/broker/connection/OpenWrt1504793280.test-mosquitto-org/state 1
$SYS/broker/connection/(none).test/state 0
$SYS/broker/connection/jrojoo-All-Series.test-mqtt-org/state 0
$SYS/broker/connection/archer.hive-archer/state 1
$SYS/broker/connection/MD-FelipeCoutto.test_mosquitto/state 1
$SYS/broker/connection/raspberrypi.snr-mqtt-bridge/state 0
$SYS/broker/connection/LAPTOP-TCMO862P.bridge-test-GSR01/state 0
…
```

Paho MQTT Python client

```
Client(client_id="", clean_session=True, userdata=None,
       protocol=MQTTv311, transport="tcp")
```

- Client id [string]: can be unique or assigned
- Clean Session Flag [boolean]:
  - This flag tells the broker to either…
    - Remember subscriptions and Store messages that the client has missed because it was offline value =False, or
    - Not to Remember subscriptions and not to Store messages that the client has missed because it was offline, value =True
- Userdata:
  - user defined data of any type that is passed as the userdata parameter to callbacks
- Transport:
  - 'tcp' or `websockets'

Paho MQTT Python client: connect

```
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.

Paho MQTT Python client: pub/sub

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.

```
def on_message(client, userdata, message):
    print("Received message '" + str(message.payload)
    + "' on topic '" + message.topic + "' with QoS " +
    str(message.qos))
```

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

Example 1: the simplest subscriber… modified

```
# File: example1.py

import paho.mqtt.client as mqtt

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

# 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, "return code: ", rc)

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe(THE_TOPIC)

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(THE_BROKER, 1883, 60)

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

Example 1: output

```
paho-code:pietro$ python example1.py
paho-code:pietro$
```

What happened??

Paho MQTT Python client: Network loop

`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.
- `timeout` must not exceed the `keepalive` value for the client or your client will be regularly disconnected by the broker.
- **Better to use the following two methods**

`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.
- This call also handles reconnecting to the broker. For example:

```
mqttc.connect("iot.eclipse.org")
mqttc.loop_start()
while True:
    temperature = sensor.blocking_read()
    mqttc.publish("paho/temperature", temperature)
```

- Call loop_stop() to stop the background thread.

`loop_forever()`

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

Example 2: subscriber with loop

```
# File: example2.py

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):
    print("Received ", msg.payload, "with topic ", msg.topic)

def on_publish(mqttc, obj, mid):
    print("mid: ", mid)

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_publish = on_publish
mqttc.on_subscribe = on_subscribe

mqttc.connect(THE_BROKER, keepalive=60)

while True:
    mqttc.loop()
```

paho-code:pietro$ python example2.py

```
('Connected to ', 'test.mosquitto.org', 'port: ', 1883)
('Subscribed: ', 1, 'granted QoS: ', (0,))
('Received ', '715859.78', 'with topic ', u'$SYS/broker/load/bytes/received/1min')
('Received ', '549040.48', 'with topic ', u'$SYS/broker/load/bytes/received/5min')
('Received ', '510392.06', 'with topic ', u'$SYS/broker/load/bytes/received/15min')
('Received ', '2398616.64', 'with topic ', u'$SYS/broker/load/bytes/sent/1min')
('Received ', '2188642.61', 'with topic ', u'$SYS/broker/load/bytes/sent/5min')
('Received ', '2352494.70', 'with topic ', u'$SYS/broker/load/bytes/sent/15min')
('Received ', '667690.21', 'with topic ', u'$SYS/broker/load/bytes/received/1min')
('Received ', '2674456.74', 'with topic ', u'$SYS/broker/load/bytes/sent/1min')
('Received ', '544693.38', 'with topic ', u'$SYS/broker/load/bytes/received/5min')
('Received ', '2255488.19', 'with topic ', u'$SYS/broker/load/bytes/sent/5min')
('Received ', '509394.77', 'with topic ', u'$SYS/broker/load/bytes/received/15min')
('Received ', '2373058.99', 'with topic ', u'$SYS/broker/load/bytes/sent/15min')
```

Example 3: subscriber with `loop_start/loop_stop`

```
# File: example3.py

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 4: very basic periodic producer

```
# File: example4.py

import sys
import time
import random

import paho.mqtt.client as mqtt

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

mqttc=mqtt.Client()
mqttc.connect(THE_BROKER, 1883, 60)

mqttc.loop_start()

while True:
    mqttc.publish(THE_TOPIC, random.randint(0, 100))
    time.sleep(5)

mqttc.loop_stop()
```

Generates a new data every 5 secs

```
paho-code:pietro$ python example2_4.py
('Connected to ', 'test.mosquitto.org', 'port: ', 1883)
('Subscribed: ', 1, 'granted QoS: ', (0,))
('Received ', '61', 'with topic ', u'PMtest/rndvalue')
('Received ', '62', 'with topic ', u'PMtest/rndvalue')
('Received ', '11', 'with topic ', u'PMtest/rndvalue')
('Received ', '79', 'with topic ', u'PMtest/rndvalue')
...
```

Output obtained with a modified version of example2.
Which parts of that code had to be modified?

Example 5: Pub/Sub with JSON

Producer

```
# File: example5_prod.py

import json
import random
import sys
import time

import paho.mqtt.client as mqtt

THE_BROKER = "test.mosquitto.org"
THE_TOPIC = "PMtest/jsonvalue"

mqttc=mqtt.Client()
mqttc.connect(THE_BROKER, 1883, 60)

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)

    print(the_msg_str)

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

mqttc.loop_stop()
```

paho-code:pietro$ python example5-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

Consumer

```
# File: example5_cons.py

import json
import sys
import time

import paho.mqtt.client as mqtt

THE_BROKER = "test.mosquitto.org"
THE_TOPIC = "PMtest/jsonvalue"

# The callback for when the client receives a CONNACK response from the
server.
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))

# 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']))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(THE_BROKER, 1883, 60)
client.subscribe(THE_TOPIC)

client.loop_forever()
```

MQTT with MicroPython

- 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.

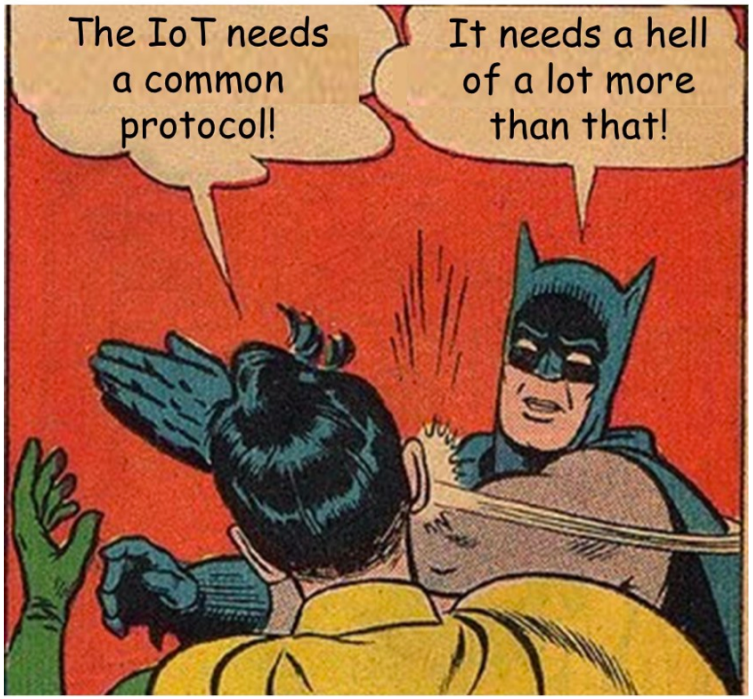
MicroPython: a simple subscriber

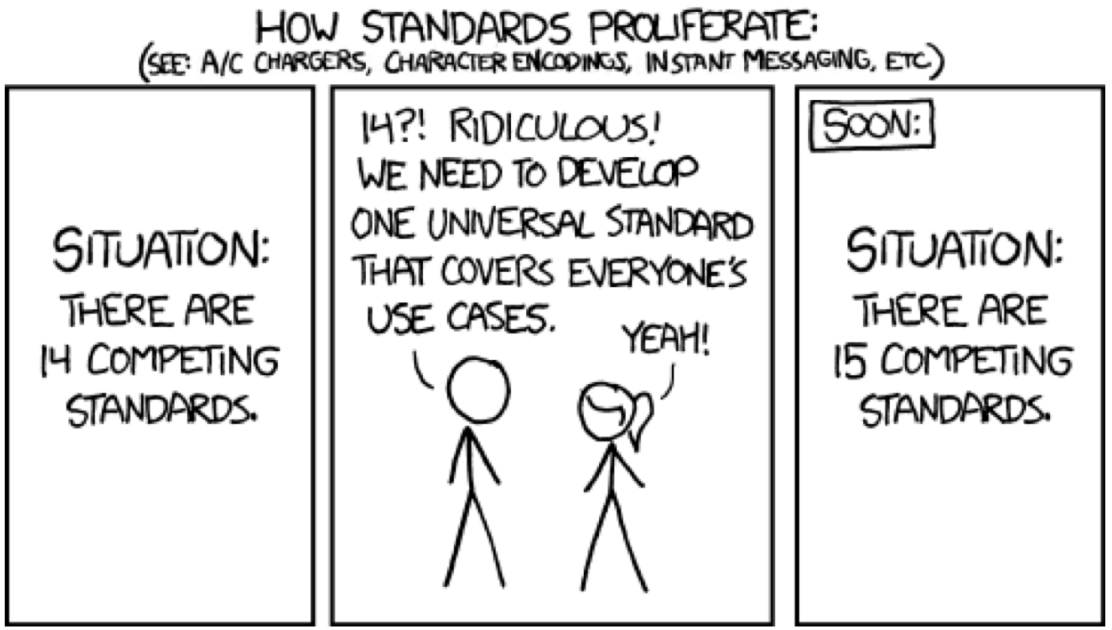
```
# file: a_simple_sub.py

from mqtt import MQTTClient
import time
import sys
import pycom

import ufun

wifi_ssid = 'LOCAL_AP'
wifi_passwd = ''
broker_addr = 'test.mosquitto.org'
dev_id = 'PMtest'

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(dev_id, broker_addr, 1883)
client.set_callback(on_message)

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

client.subscribe('lopy/lights')

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

```

MicroPython: a simple publisher

```
# file: a_simple_pub.py

from mqtt import MQTTClient
import pycom
import sys
import time

import ufun

wifi_ssid = 'LOCAL_AP'
wifi_passwd = ''
broker_addr = 'test.mosquitto.org'
dev_id = 'PMtest'

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(dev_id, broker_addr, 1883)

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

print('Sending messages...')
while True:
    # creating the data
    the_data = get_data_from_sensor()
    # publishing the data
    client.publish(dev_id+'/sdata', str(the_data))
    time.sleep(1)

```

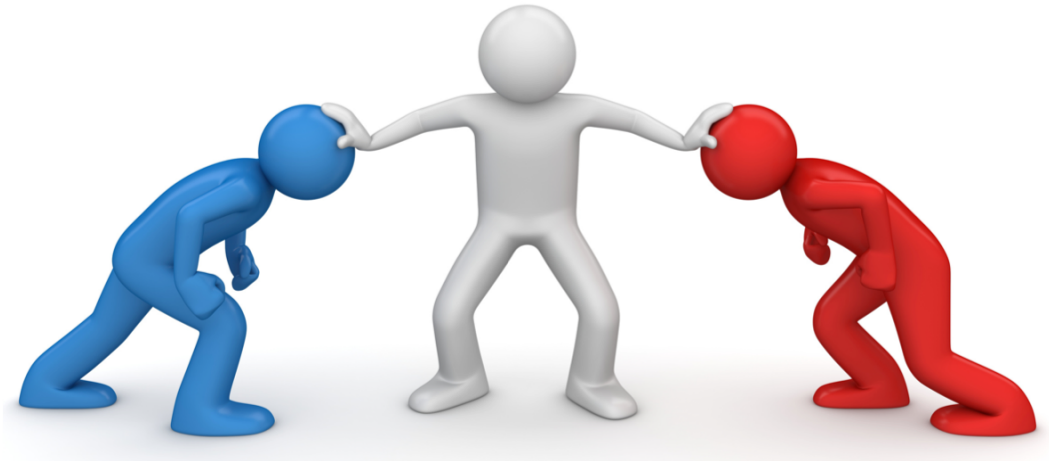
REST vs MQTT


The IoT needs a common protocol!
It needs a hell of a lot more than that!


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

MQTT vs REST

- 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



REST advantages

- The *business logic* is decoupled from the presentation.
  - So you can change one without impacting the other.
- The uniform interface means that we don't have to document on a per-resource or per-server basis, the basic operations of the API.
- The universal identifiers embodied by URIs mean again that there is no resource or server specific usage that has to be known to refer to our resources
  - This assures that any tool that can work with HTTP can use the service.

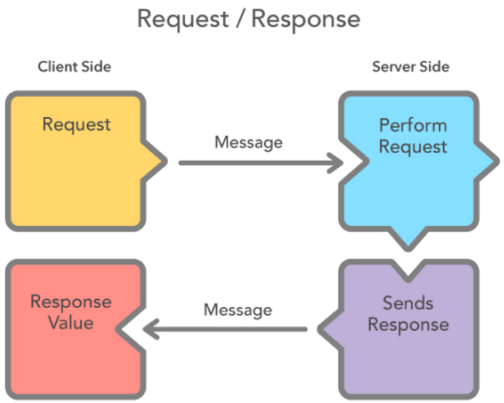
REST advantages

- 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
  - Forbids conversational state. 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.

REST disadvantages

- 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 interactions



Advantages of MQTT

- **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
- It's binary, so lower use of bandwidth… but well, depends on the payload

Energy usage: some number

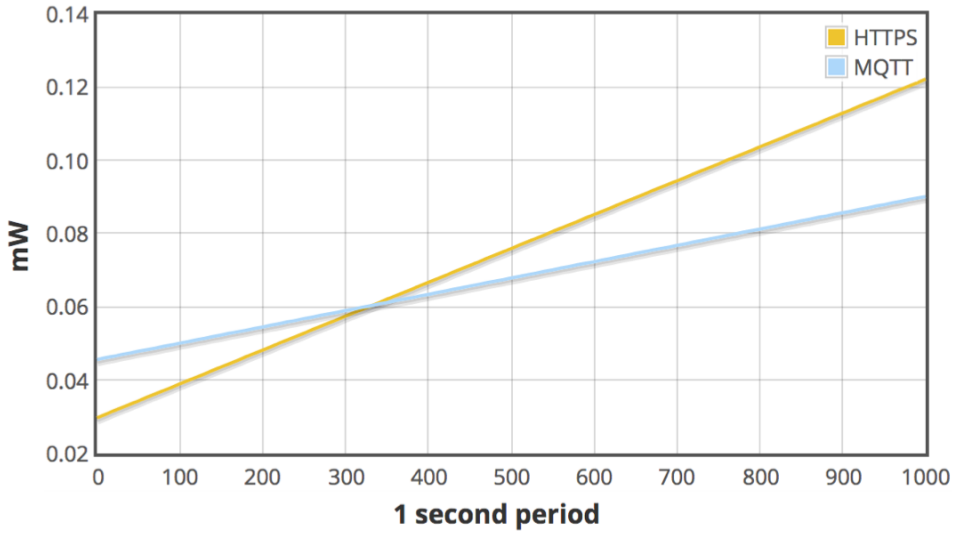
amount of power taken to establish the initial connection to the server:

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

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**

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

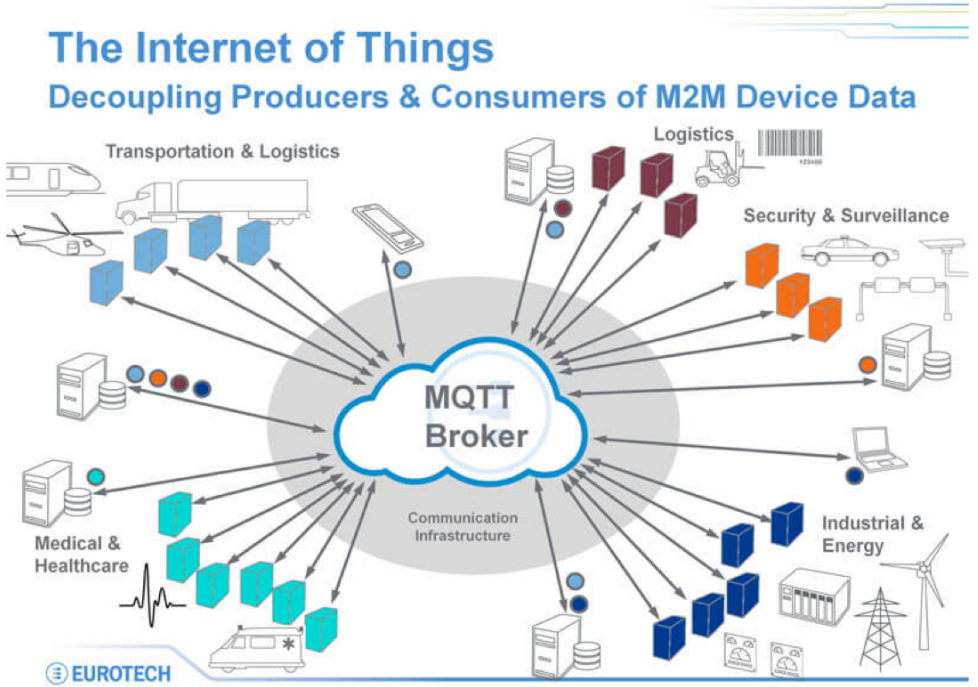


you'd save ~4.1% battery per day just by using MQTT over HTTPS to maintain an open stable connection.

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

MQTT advantages: decoupling and filtering

- Decoupling of publisher and receiver can be differentiated in more dimensions:
  - **Space decoupling**: Publisher and subscriber don't need to know each other (by IP address and port for example)
  - **Time decoupling**: Publisher and subscriber do not need to run at the same time
  - **Synchronization decoupling**: Operations on both components are not halted during publish or receiving
- **Filtering** of the messages makes possible that only certain clients receive certain messages.
  - MQTT uses subject-based filtering of messages. So each message contains a topic, which the broker uses to find out, if a subscribing client will receive the message or not.



MQTT disadvantages

- 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
- Does not define a standard client API, so application developers have to select the best fit.
- Does not include message headers and other features common to messaging platforms.
  - developers frequently have to implement these capabilities in the message payload to meet application requirements thus increasing the size of the message and increasing the BW requirements.
- Does not include many features that are common in Enterprise Messaging Systems like:
  - expiration, timestamp, priority, custom message headers, …
- Maximum message size 256MB
- If the broker fails…

So, trying to summarize

- REST
  - Advantages
    - Uniform and very widely adopted interface
    - The business logic is decoupled from the presentation
    - Allows to retrieve any historical data
  - Disadvantages
    - It's pull based
    - One-to-one interaction
    - Too heavy for small devices (*see CoAP*)
- MQTT
  - Advantages
    - Push based: no need to continuously look for updates
    - Useful for one-to-many, many-to-many applications
    - Small memory footprint protocol, with reduced use of battery
    - Built-in function useful for reliable behavior in an unreliable or intermittently connected wireless environments
  - Disadvantages
    - Does not define a standard client API
    - Does not have a point-to-point messaging pattern
    - Not much sense of history…
    - If the broker fails…