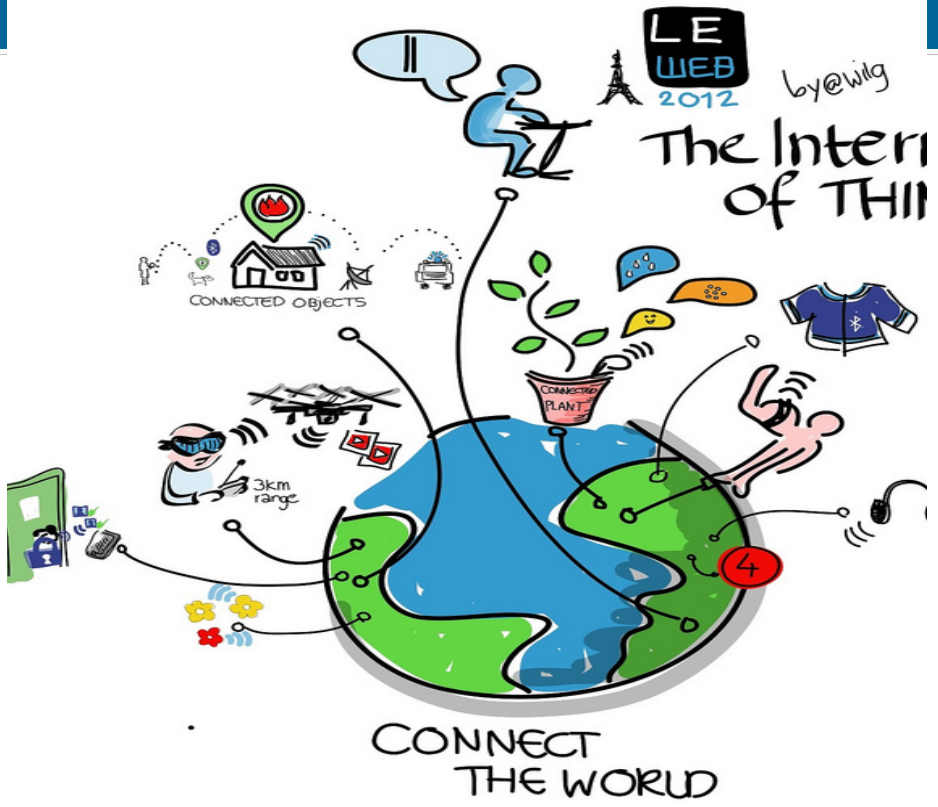
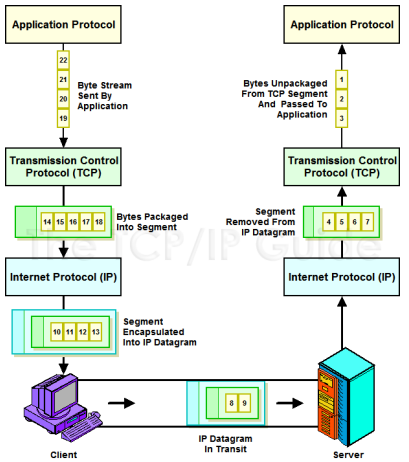
Intro to REST and MQTT

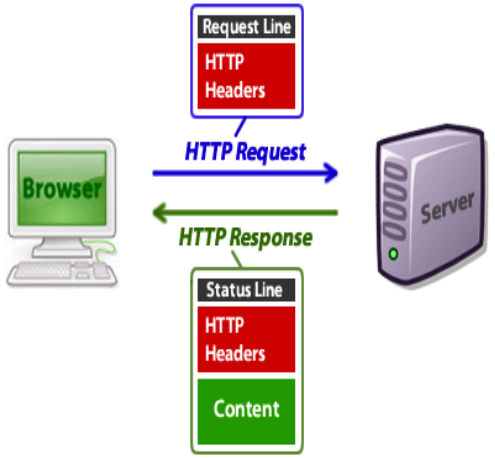
- A IoT oriented presentation

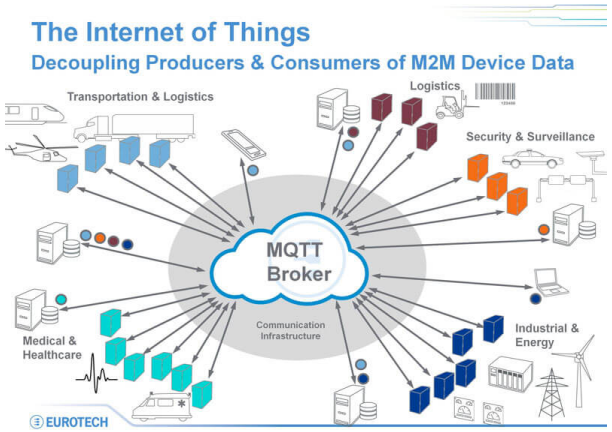


From "byte streams" to "messages"

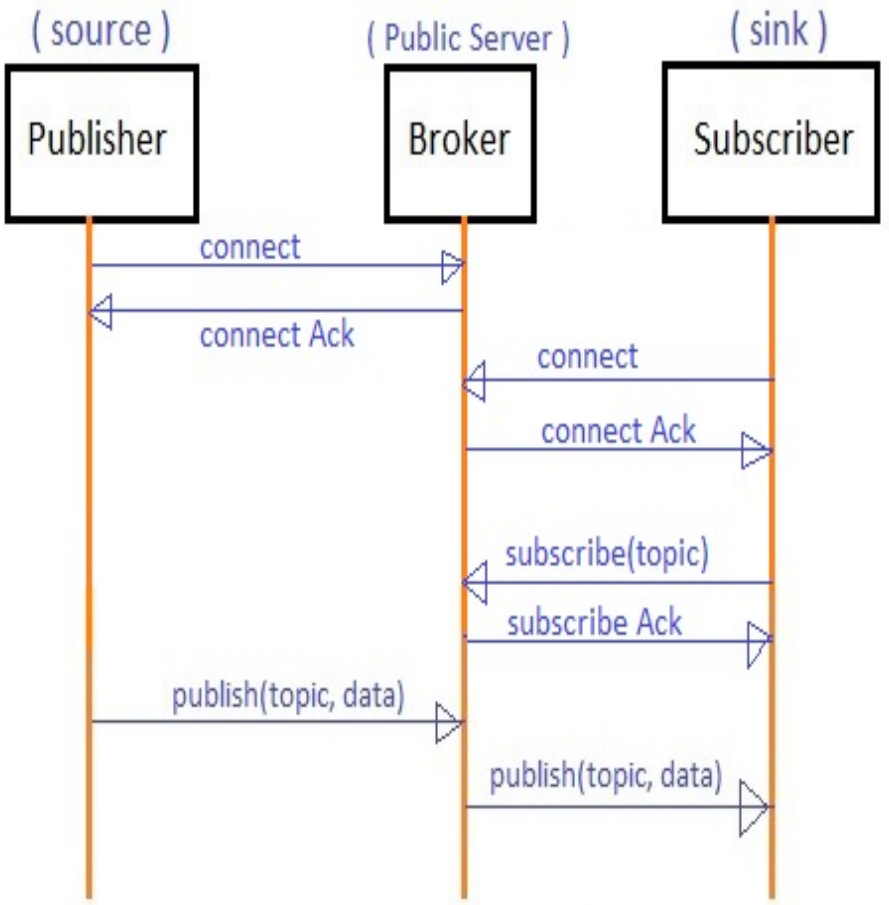
- The "old" vision of data communication was based on **reliable byte streams**, i.e., TCP
- Nowadays **messages interchange** (aka data bundles) is becoming more common
  - E.g., Twitter, Whatsapp, Instagram, Snapchat, Facebook,…
  - Actually is not that new…
    - SMTP+MIME, FTP, uucp, …
- The **request/response paradigm** is the reference:
  - HTTP ➔ **REST**, CoAP
- But also the **producer/consumer paradigm** (aka: pub/sub) is growing:
  - **MQTT**, AMQP, XMPP (was Jabber)



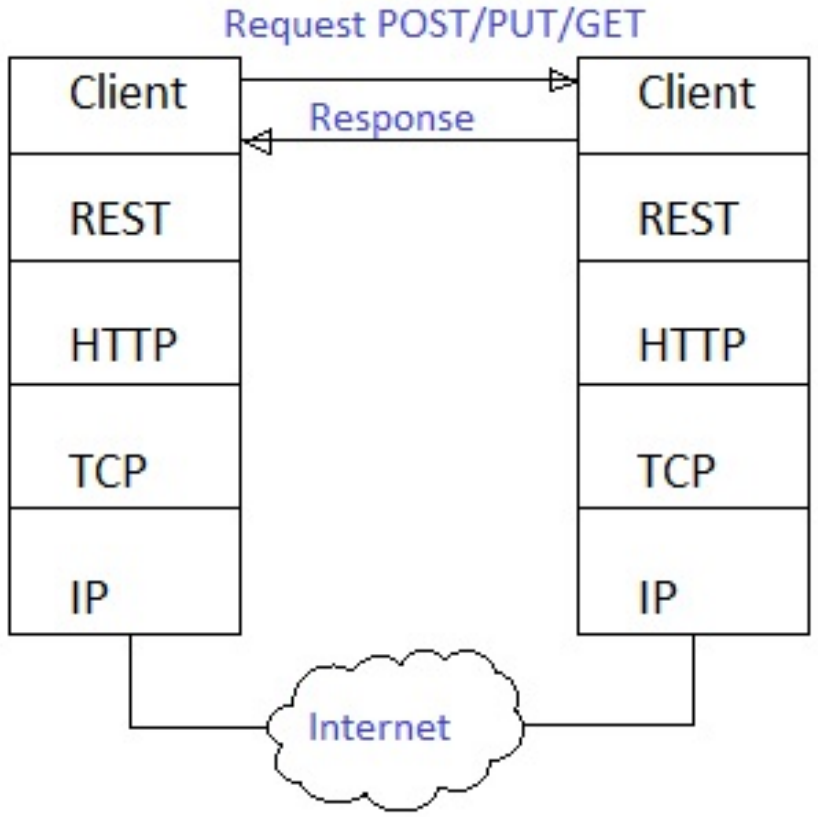




MQTT vs REST



Broker based MQTT Protocol



REST PROTOCOL

Data Representation

{“value": 237} vs. <value>237</value>

- Data-interchange format. Should be easy for humans to read and write. 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

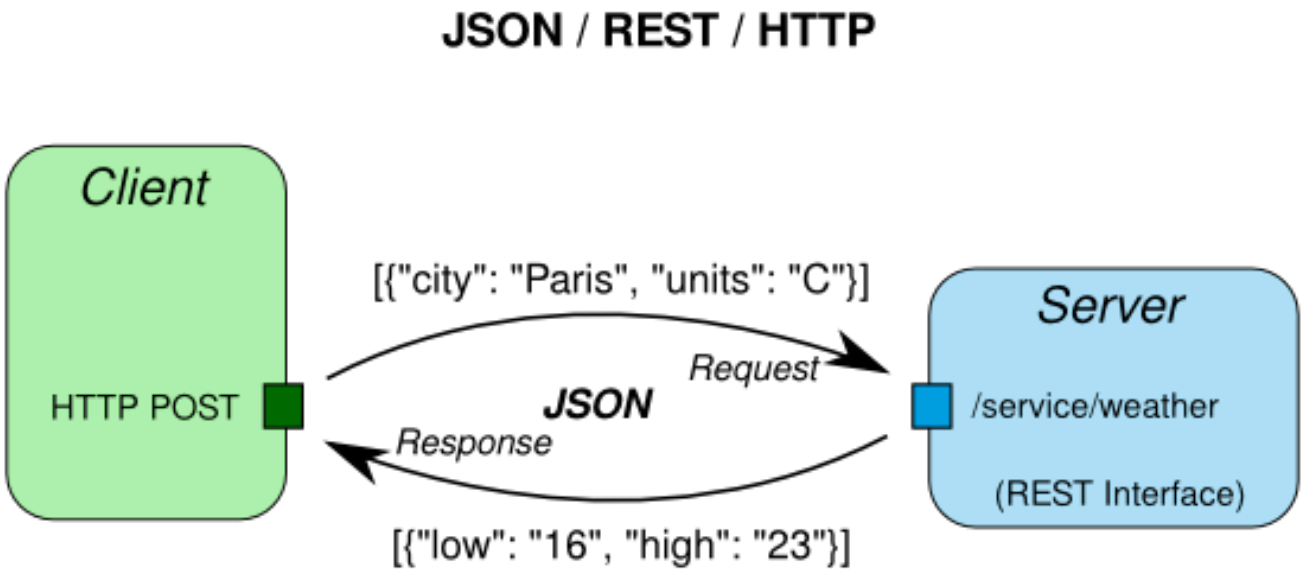
```
<menu>
  <id>file</id>
  <value>File</value>
  <popup>
    <menuitem>
      <value>New</value>
      <onclick>NewDoc()</onclick>
    </menuitem>
    <menuitem>
      <value>Open</value>
      <onclick>OpenDoc()</onclick>
    </menuitem>
    <menuitem>
      <value>Close</value>
      <onclick>CloseDoc()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

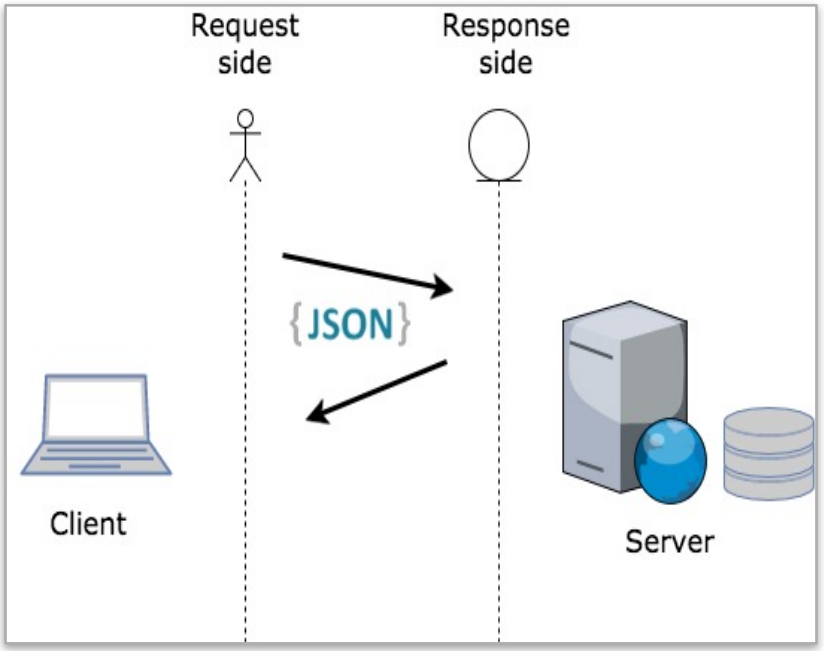
REST

- Basic concepts
- Basic programming



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
    - URI Addressable resources
- 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://dev.twitter.com/rest/public
- Facebook:
  - https://developers.facebook.com/docs/atlas-apis
- Amazon offers several REST services, e.g., for their S3 storage solution
  - http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
- The Google Glass API, known as "Mirror API", is a pure REST API.
  - Here is (https://youtu.be/JpWmGX55a40) a video talk about this API. (The actual API discussion starts after 16 minutes or so.)
- Tesla Model S uses an (undocumented) REST API between the car systems and its Android/iOS apps.
  - http://docs.timdorr.apiary.io/#reference/vehicles/state-and-settings
- 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
```

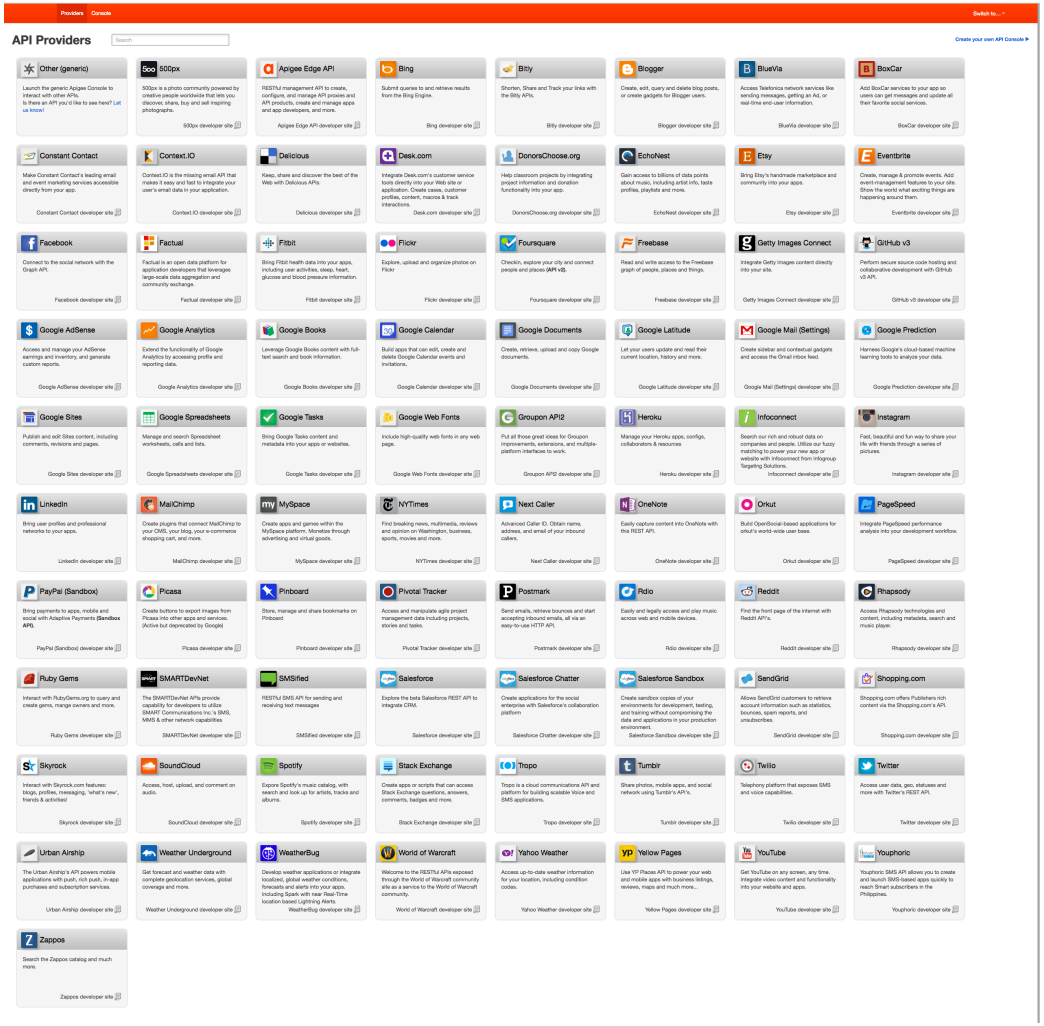
```
▼ Hypertext Transfer Protocol
  ▼ HTTP/1.1 200 OK\r\n
    ▶ [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      Request Version: HTTP/1.1
      Status Code: 200
      Response Phrase: OK
    Content-Type: application/json; charset=UTF-8\r\n
    Date: Thu, 29 Jun 2017 07:54:23 GMT\r\n
    Expires: 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
```

Very widely…

- https://apigee.com/providers



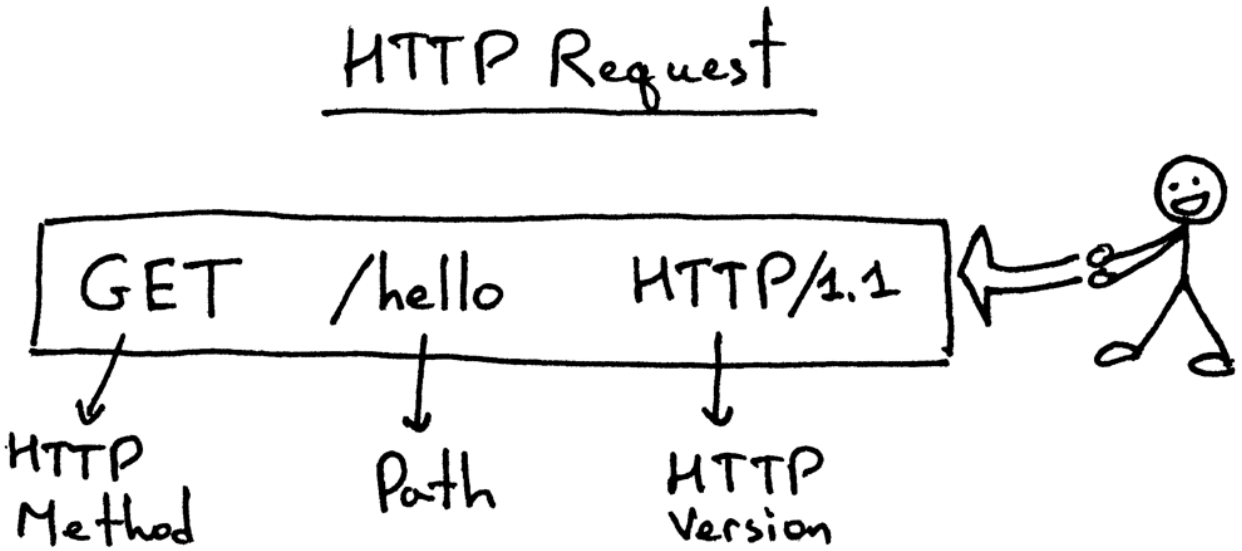
REST: the resources

- 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://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

Verbs

- Represent the actions to be performed on 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:

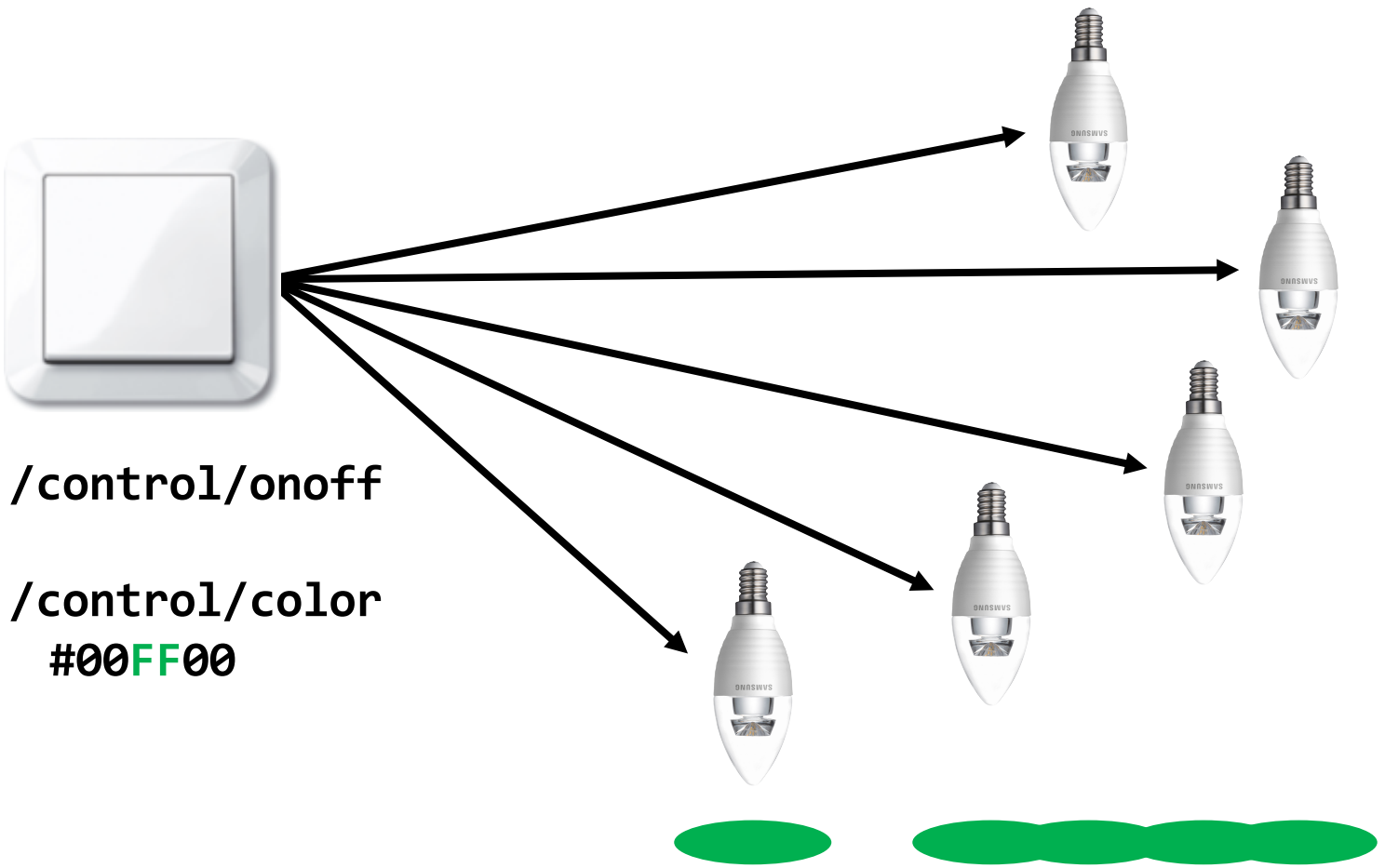
```
PATCH /tasks/3657/
{
    'description': '6 eggs, carrots, spaghetti, bread'
}
```

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

REST for devices control communication

```
GET /status/power
```

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



```
PUT /control/onoff
```

```
PUT /control/color
    #00FF00
```

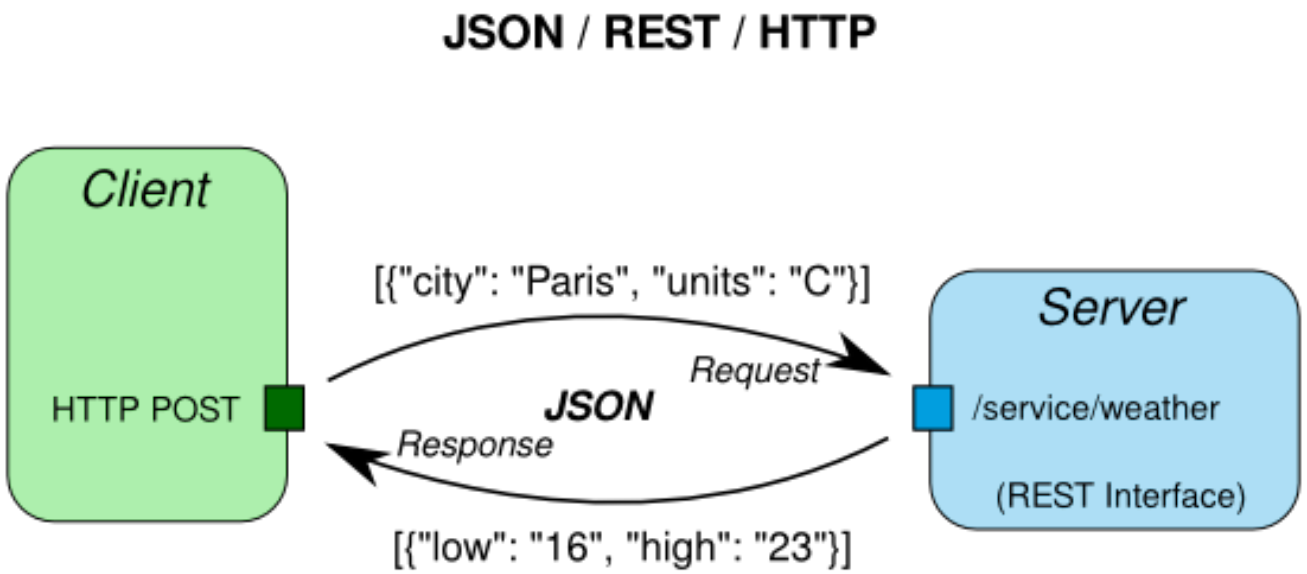
HTTP DELETE

- Removes the resource identified by the URI

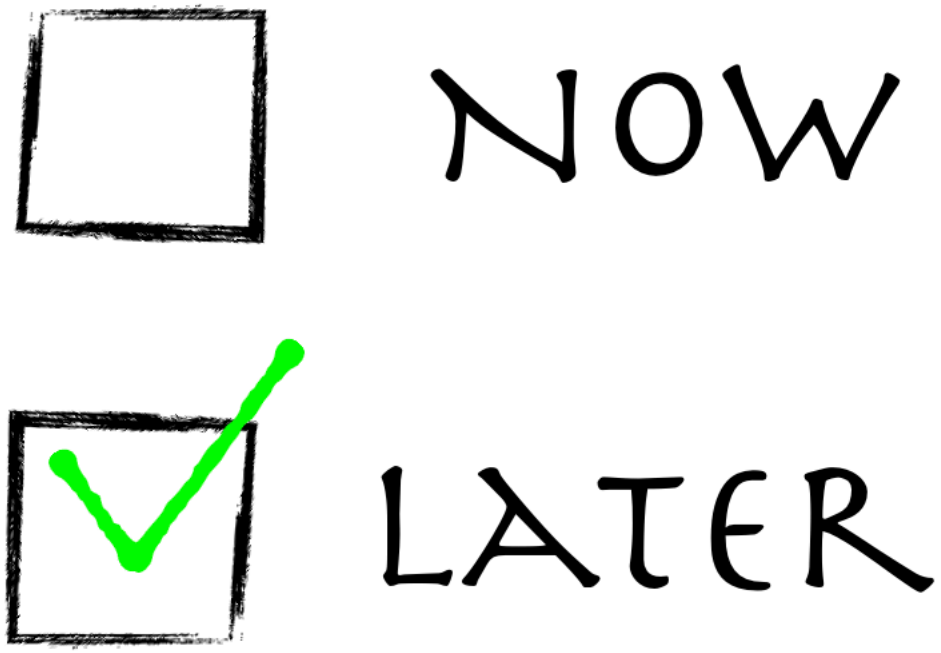
- DELETE /tasks/<item_id>/
  - Mark the item as done. (I.e., strike it off the list, so GET /tasks/ will not show it.)
  - The response body is empty.

REST

- Basic concepts
- Basic programming

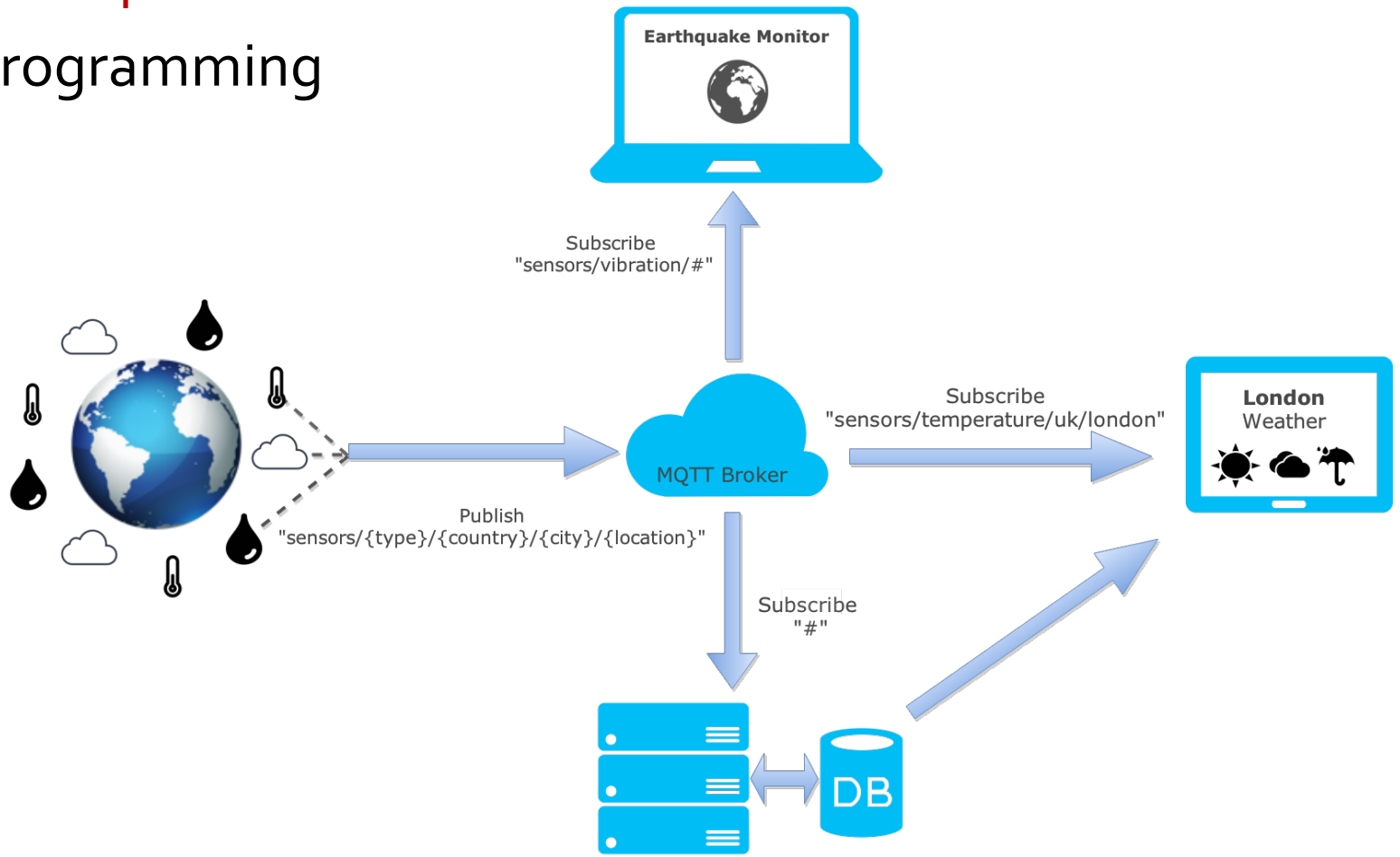


well…. later…. if we will still have time ☺



MQTT

- Basic concepts
- Basic programming



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

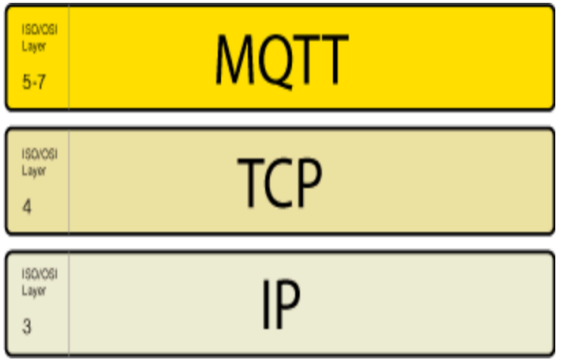
Message Queuing Telemetry Transport

MQTT

- A lightweight publish-subscribe protocol that runs on embedded devices and mobile platforms designed to connect the physical world devices with applications and middleware.
  - http://mqtt.org/
  - the MQTT community wiki: https://github.com/mqtt/mqtt.github.io/wiki
  - http://www.hivemq.com/mqtt-essentials/
- Designed to provide a low latency two-way communication channel and efficient distribution to one or many receivers.
  - Minimizes the amount of bytes flowing over the wire
  - Maximum message size of 256MB, but not really designed for sending large amounts of data; better at a high volume of low size messages.
- Assured messaging over fragile networks
- Low power usage.

MQTT overview

- MQTT is an open standard with a short and readable protocol specification.
  - `http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html`
- Works on top of the TCP protocol stack
- There is also the closely related MQTT for Sensor Networks (MQTT-SN)
  - TCP is replaced by UDP: to transfer small data pieces (measurements, remote commands, or user data) TCP stack is too complex for WSN



- 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`

MQTT Version 5

- The next version of MQTT (autumn 2017) will be version 5.
  - If you are wondering what happened to 4 then see:

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

- You can find a working draft:

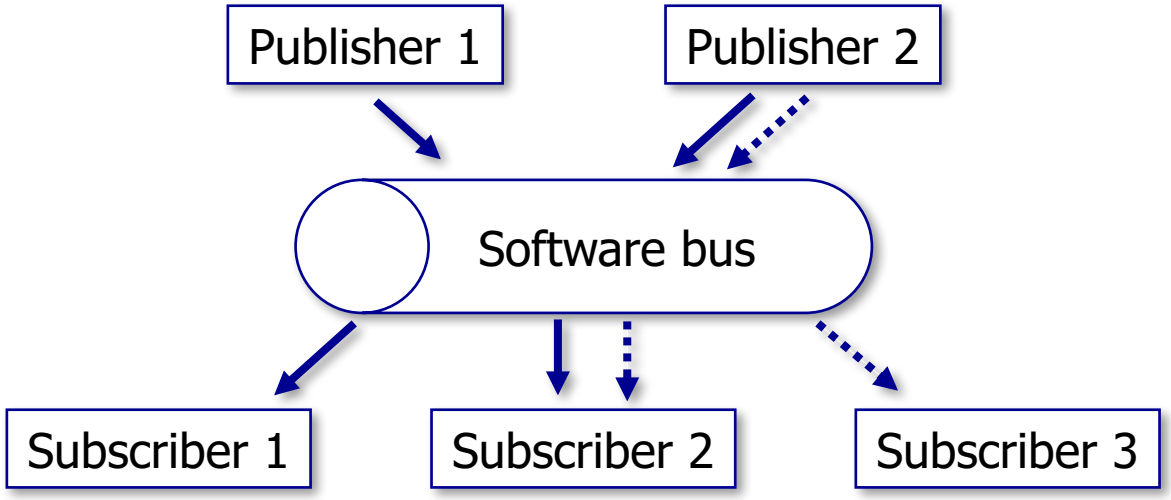
https://www.oasis-open.org/committees/download.php/59482/mqtt-v5.0-wd09.pdf

- And the key ideas in version 5 here.

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

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.



MQTT Connection

MQTT-Packet:

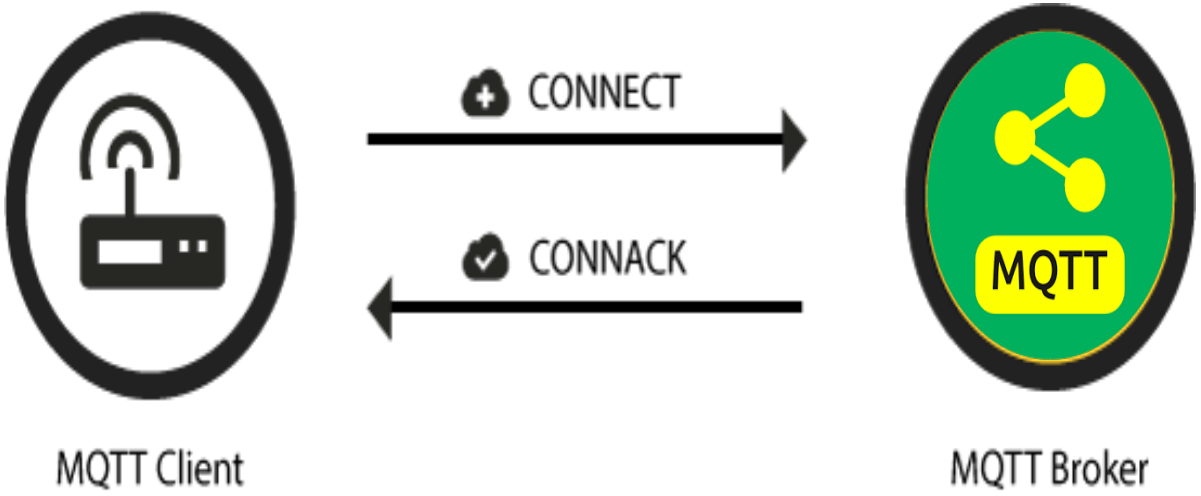
CONNECT

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



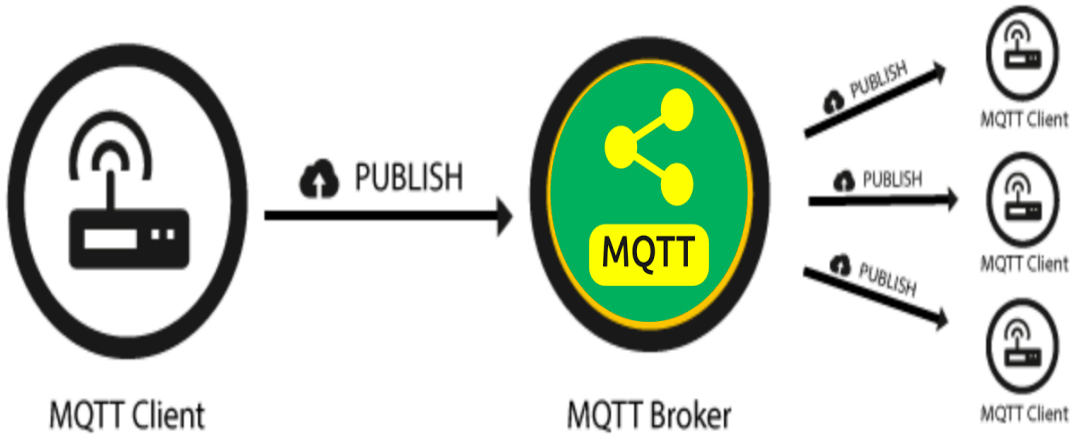
Publish

- 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

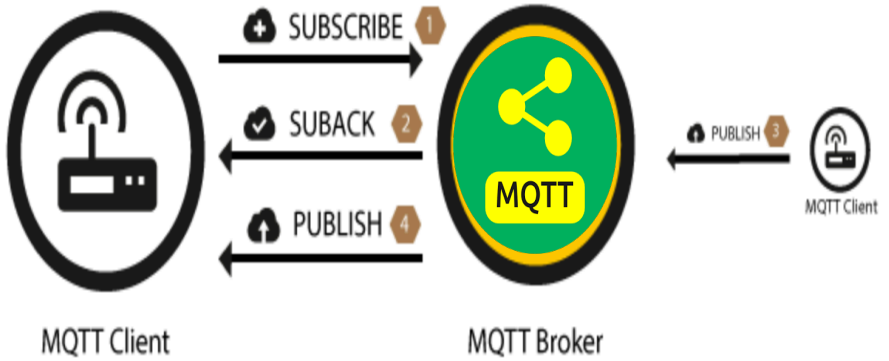


Subscribe

MQTT-Packet:

SUBSCRIBE

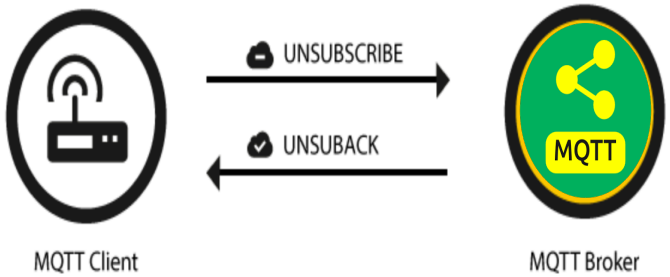
contains:		Example
packetId		4312
qos1	} (list of topic + qos)	1
topic1		"topic/1"
qos2	}	0
topic2		"topic/2"
...		...



MQTT-Packet:

SUBACK

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



Topics

- Topic subscriptions can have wildcards
  - '+' matches anything at a given tree level so the topic "sensor/+/temp" would match "sensor/dev1/temp", "sensor/dev2/temp", etc.
  - '#' matches a whole sub-tree, so "sensor/#" would match all topics under "sensor/".
- These enable nodes to subscribe to groups of topics that don't exist yet, allowing greater flexibility in the network's messaging structure.

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

Topics best practices

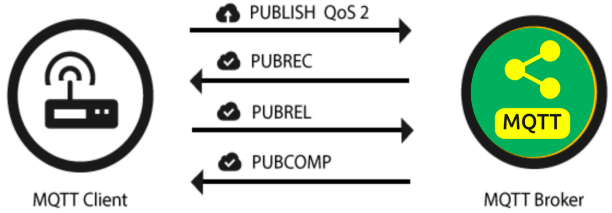
- Keep the topic short and concise
- Use specific topics, instead of general ones
- Don’t forget extensibility

- Don’t use a leading forward slash
- Don’t use spaces in a topic
- Use only ASCII characters, avoid non printable characters

- Don’t subscribe to #

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

- Downgrade of QoS
  - The QoS flows between a publishing and subscribing client are two different things as well as the 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.
  - Between the sender and the broker the QoS is defined by the sender.
  - 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.
- Packet identifiers are unique per client
  - Also important to know is that each packet identifier (used for QoS 1 and QoS 2) is unique between one client and a broker and not between all clients.
  - If a flow is completed the same packet identifier can be reused anytime. That's also the reason why the packet identifier doesn't need to be bigger than 65535, because it is unrealistic that a client sends a such large number of message, without completing the flow.

QoS: Best Practice

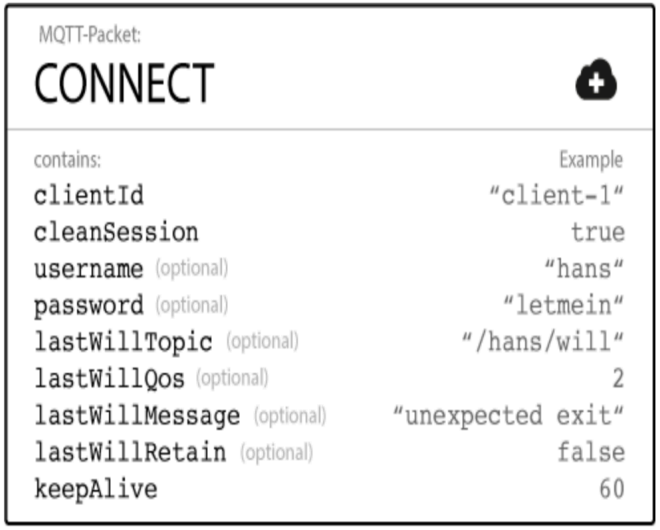
- Use QoS 0 when …
  - You have a complete or almost stable connection between sender and receiver. A classic use case is when connecting a test client or a front end application to a MQTT broker over a wired connection.
  - You don't care if one or more messages are lost once a while. That is sometimes 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.
- 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.
- 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.
- It is important to understand that a retained message has nothing to do with a persistent session of any client.

"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.



MQTT Keep alive

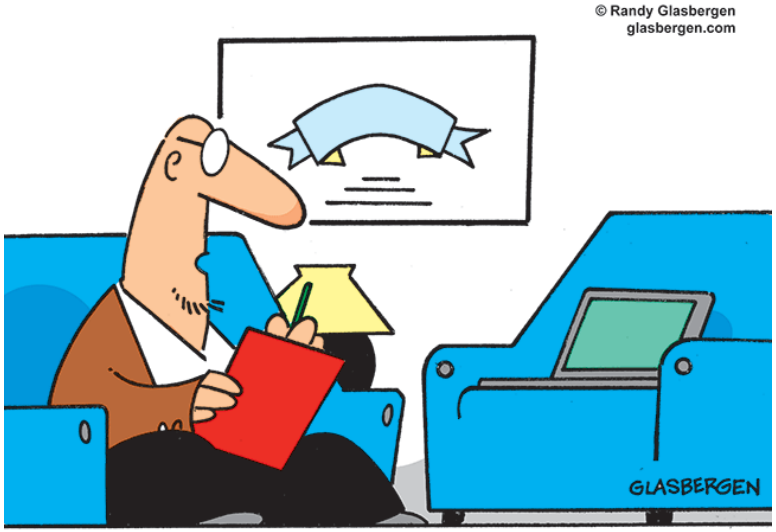
- The keep alive functionality assures that the connection is still open and both broker and client are connected to one another. Therefore the client specifies a time interval in seconds and communicates it to the broker during the establishment of the connection. The interval is the longest possible period of time, which broker and client can endure without sending a message.
  - Problem of half-open TCP connections
- Good to Know
  - 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).
  - The MQTT client is responsible of setting the right keep alive value. For example, it can adapt the interval to its current signal strength.
  - The maximum keep alive is 18h 12min 15 sec.
  - If the keep alive interval is set to 0, the keep alive mechanism is deactivated.

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 offlne
  - 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

Security

- MQTT provides security, but it is not enabled by default.
  - As a basic solution we can rely on the encrypted WiFi connection to provide a basic level of security.
- MQTT has the option for Transport Layer Security (TLS) encryption.
- MQTT also provides username/password authentication.
  - 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."

Common Questions

Q- What happens to messages that get published to topics that no one subscribes to?

A- They are discarded by the broker.

Q-How can I find out what topics have been published?

A- You can't do this easily as the broker doesn't seem to keep a list of published topics as they aren't permanent.

Q- Can I subscribe to a topic that no one is publishing to?

A- Yes

Q- Are messages stored on the broker?

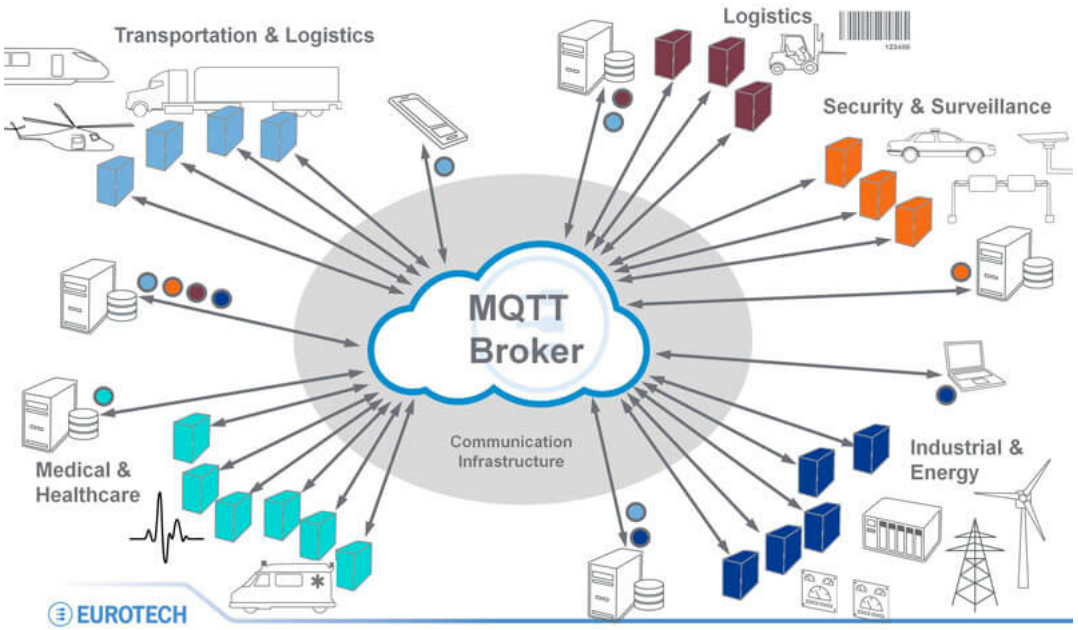
A- Yes but only temporarily. Once they have been sent to all subscribers they are then discarded. But see next question.

Q- What are retained messages?

A- When you publish a message you can have the broker store the last published message. This message will be the first message that new subscribers see when they subscribe to that topic. MQTT only retains 1 message.

MQTT

- Basic concepts
- Basic programming



Software available

- Brokers (https://github.com/mqtt/mqtt.github.io/wiki/servers):
  - **http://mosquitto.org/**
  - http://www.hivemq.com/
  - https://www.rabbitmq.com/mqtt.html
  - http://activemq.apache.org/mqtt.html
  - https://github.com/mcollina/mosca
- Clients
  - **http://www.eclipse.org/paho/**
    - Source: https://github.com/eclipse/paho.mqtt.python
    - Documentation: https://pypi.python.org/pypi/paho-mqtt
    - More docs: http://www.eclipse.org/paho/clients/python/docs/
  - http://www.hivemq.com/demos/websocket-client/
- Tools
  - https://github.com/mqtt/mqtt.github.io/wiki/tools
  - For example:
    - http://www.hivemq.com/demos/websocket-client/

"Sandboxes" for brokers

- https://iot.eclipse.org/getting-started#sandboxes
  - hostname iot.eclipse.org and port 1883.
  - encrypted port 8883
  - http://test.mosquitto.org/
  - http://www.hivemq.com/try-out/
  - http://www.mqtt-dashboard.com/

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	

- Google CLOUD PUB/SUB A global service for real-time and reliable messaging and streaming data
  - https://cloud.google.com/pubsub/

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

Example 1: subscriber with `loop_forever`

```
import paho.mqtt.client as mqtt

# 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))

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

# 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("iot.eclipse.org", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()
```

Example 1: output

```
…
$SYS/broker/load/publish/dropped/5min 4080.10
$SYS/broker/load/publish/received/5min 2113.46
$SYS/broker/load/publish/sent/5min 8473.40
$SYS/broker/load/bytes/received/5min 384447.30
$SYS/broker/load/bytes/sent/5min 1114325.30
$SYS/broker/load/sockets/5min 929.54
$SYS/broker/load/connections/5min 727.84
$SYS/broker/load/messages/received/15min 20789.77
$SYS/broker/load/messages/sent/15min 27349.78
$SYS/broker/load/publish/dropped/15min 3932.77
$SYS/broker/load/publish/received/15min 2152.81
$SYS/broker/load/publish/sent/15min 8699.76
$SYS/broker/load/bytes/received/15min 398937.27
$SYS/broker/load/bytes/sent/15min 1170197.30
$SYS/broker/load/sockets/15min 897.06
$SYS/broker/load/connections/15min 686.45
$SYS/broker/messages/stored 2018275
$SYS/broker/subscriptions/count 15864
…
```

Paho MQTT Python client

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

- Client id: can be unique or assigned
- Clean Session Flag: 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
- Transport: 'tcp' / '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)

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

```
import sys
import paho.mqtt.client as mqtt

def on_connect(mqttc, obj, flags, rc):
    print "Connected to %s:%s" % (mqttc._host, mqttc._port)

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

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

def on_subscribe(mqttc, obj, mid, granted_qos):
    print("Subscribed: "+str(mid)+" "+str(granted_qos))

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

# If you want to use a specific client id, use
# mqttc = mqtt.Client("client-id")
# but note that the client id must be unique on the broker.
# Leaving the client id parameter empty will generate a random id.
mqttc = mqtt.Client()
mqttc.on_message = on_message
mqttc.on_connect = on_connect
mqttc.on_publish = on_publish
mqttc.on_subscribe = on_subscribe

# Uncomment to enable debug messages
# mqttc.on_log = on_log
mqttc.connect("test.mosquitto.org", keepalive=60)
mqttc.subscribe("$SYS/broker/load/bytes/#", 0)

rc = 0
while rc == 0:
    rc = mqttc.loop()

print("rc: "+str(rc))
```

```
$>: python code4.py
Connected to test.mosquitto.org:1883
Subscribed: 1 (0,)
$SYS/broker/load/bytes/received/1min 0 2895441.54
$SYS/broker/load/bytes/received/5min 0 2886222.81
$SYS/broker/load/bytes/received/15min 0 2904064.82
$SYS/broker/load/bytes/sent/1min 0 6326013.29
$SYS/broker/load/bytes/sent/5min 0 5450954.94
$SYS/broker/load/bytes/sent/15min 0 4253739.61
$SYS/broker/load/bytes/received/1min 0 2907633.84
$SYS/broker/load/bytes/sent/1min 0 6260546.57
$SYS/broker/load/bytes/received/5min 0 2889175.18
$SYS/broker/load/bytes/sent/5min 0 5468388.62
$SYS/broker/load/bytes/received/15min 0 2904844.26
$SYS/broker/load/bytes/sent/15min 0 4274165.58
…
```

Example 3: subscriber with `loop_start/loop_stop`

```
import sys, time
import paho.mqtt.client as mqtt

def on_connect(mqttc, obj, flags, rc):
    print "Connected to %s:%s" % (mqttc._host, mqttc._port)

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

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

def on_subscribe(mqttc, obj, mid, granted_qos):
    print("Subscribed: "+str(mid)+" "+str(granted_qos))

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

msg_counter = 0

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("test.mosquitto.org", keepalive=60)
mqttc.subscribe("$SYS/broker/load/bytes/#", 0)

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

Example 4: very basic periodic producer

```
import paho.mqtt.client as mqtt
import time

mqttc=mqtt.Client()
mqttc.connect("localhost", 1883, 60)

mqttc.loop_start()
while True:
    mqttc.publish("pietro/test","Hello")
    time.sleep(10)  # sleep for 10 seconds before next call
mqttc.loop_stop()
```

```
1482241224: New client connected from 127.0.0.1 as paho/D00DA652C1C18AA67D (c1, k60).
1482241224: Sending CONNACK to paho/D00DA652C1C18AA67D (0, 0)
1482241224: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
1482241234: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
1482241245: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
1482241255: Received PUBLISH from paho/D00DA652C1C18AA67D (d0, q0, r0, m0, 'pietro/test', ... (5 bytes))
```

Example 5: Pub/Sub with JSON

Producer

```
import paho.mqtt.client as mqtt
import time, random, json

mqttc=mqtt.Client()
mqttc.connect("localhost", 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("pietro/test",the_msg_str)
    time.sleep(5)# sleep for 5 seconds before next call

mqttc.loop_stop()
```

Consumer

```
import paho.mqtt.client as mqtt
import json

# 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 ",
    print str(themsg['Value'])+" "+themsg['C_F'],
    print " at time "+str(themsg['Time'])

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

client.connect("localhost", 1883, 60)
client.subscribe("pietro/test")

client.loop_forever()
```

```
pietro/test {"Time": "22:31:11", "Sensor": 1, "Value": 86, "C_F": "C"}
Sensor 1 got value  86 C  at time 22:31:11
pietro/test {"Time": "22:31:16", "Sensor": 1, "Value": 90, "C_F": "C"}
Sensor 1 got value  90 C  at time 22:31:16
pietro/test {"Time": "22:31:21", "Sensor": 1, "Value": 24, "C_F": "C"}
Sensor 1 got value  24 C  at time 22:31:21
```

MQTT with MicroPython

- Import the library

```
from mqtt import MQTTClient
```

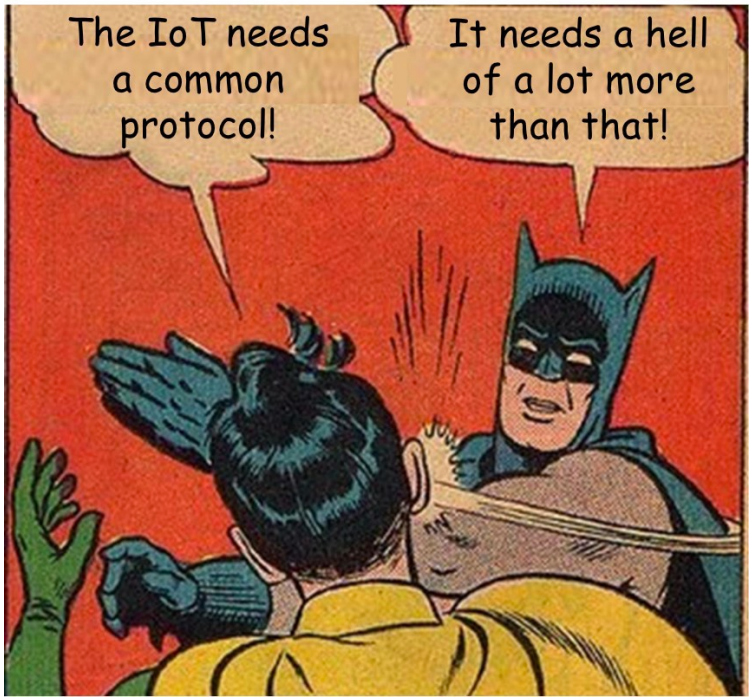
- Creating a client:

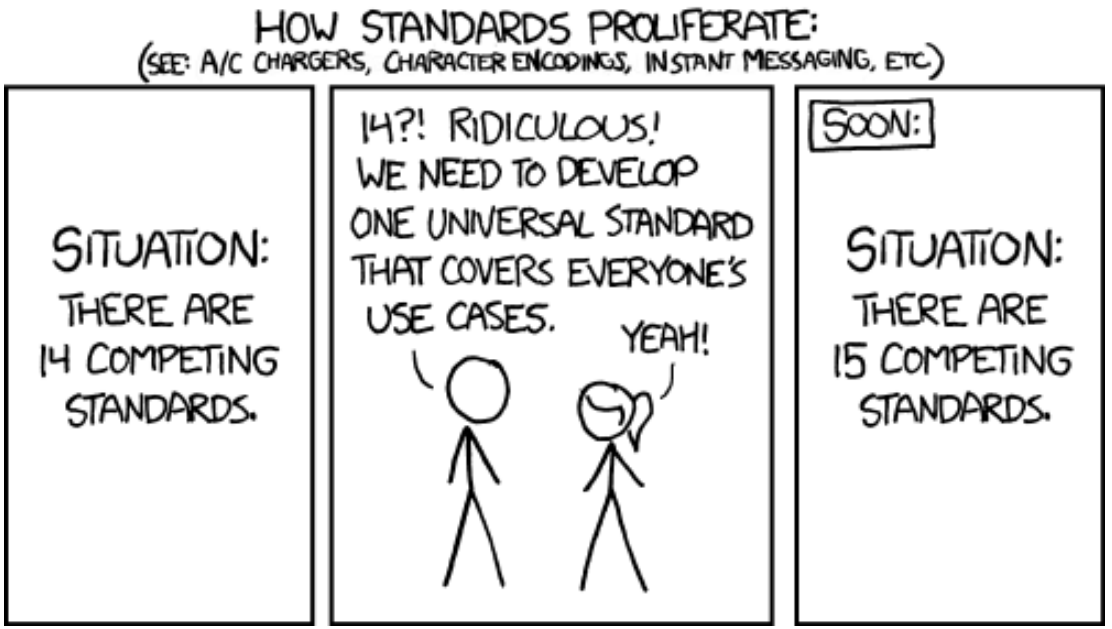
```
MQTTclient(client_id, server, port=0, user=None,
password=None, keepalive=0,  ssl=False, ssl_params={})
e.g., client = MQTTClient("dev_id", "10.1.1.101", 1883)
```

- 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.

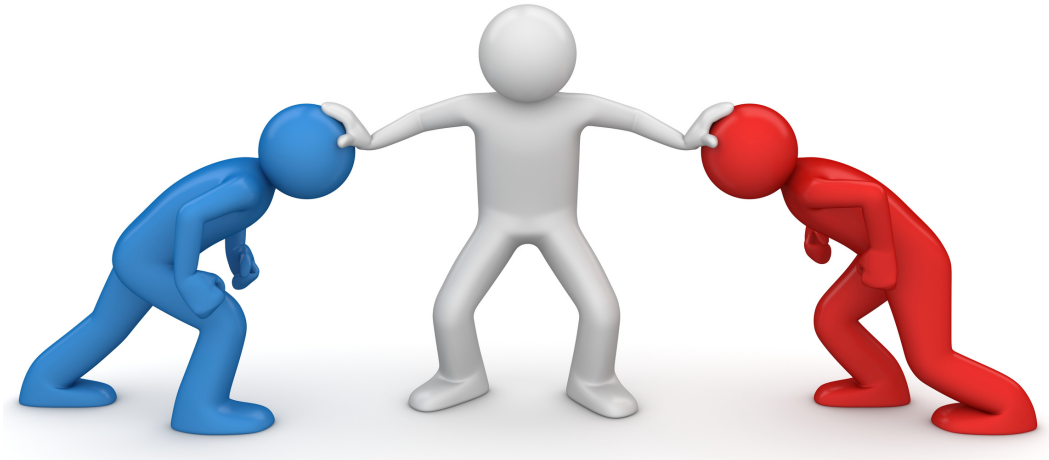
REST vs MQTT


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.
- 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.

REST advantages

- 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.
- It is cacheable ➔ you save bandwidth by caching responses from the server.
  - By using either expiry or validation (`Etag`). Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
  - This also provides an optimistic locking paradigm (also known as *Optimistic concurrency control - OCC*) through the use use of conditional requests. The `GET` method returns an `ETag` for a resource and subsequent `PUTs` use the `ETag` value in the `If-Match` headers; while the first PUT will succeed, the second will not, as the value in If-Match is based on the first version of the resource.

REST: HATEOAS

- HATEOAS, an abbreviation for Hypermedia As The Engine Of Application State, is a constraint of the REST application architecture that distinguishes it from most other network application architectures.
- The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers.
- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.
- A REST client enters a REST application through a simple fixed URL. All future actions the client may take are discovered within resource representations returned from the server.
  - The media types used for these representations, and the link relations they may contain, are standardized.
  - The client transitions through application states by selecting from the links within a representation or by manipulating the representation in other ways afforded by its media type.
  - In this way, RESTful interaction is driven by hypermedia, rather than out-of-band information.

REST: HATEOAS, an example with JSON

- For example, the following code represents a `Customer` object.

```
class Customer {
  String name;
}
```

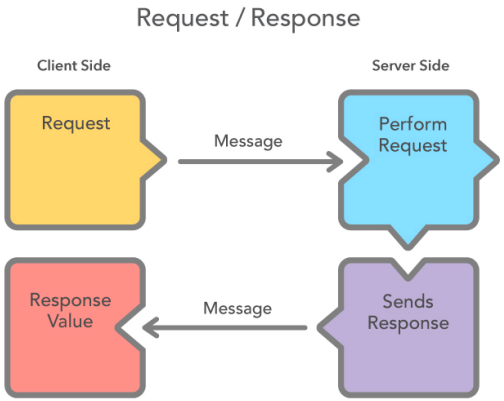
- A HATEOAS-based response would look like this:

```
{
    "name": "Alice",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/customer/1"
    } ]
}
```

- This response not only has the person's name, but includes the self-linking URL where that person is located.
  - `rel` means relationship. In this case, it's a self-referencing hyperlink. More complex systems might include other relationships. For example, an order might have a `"rel":"customer"` relationship, linking the order to its customer.
  - `href` is a complete URL that uniquely defines the resource.

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

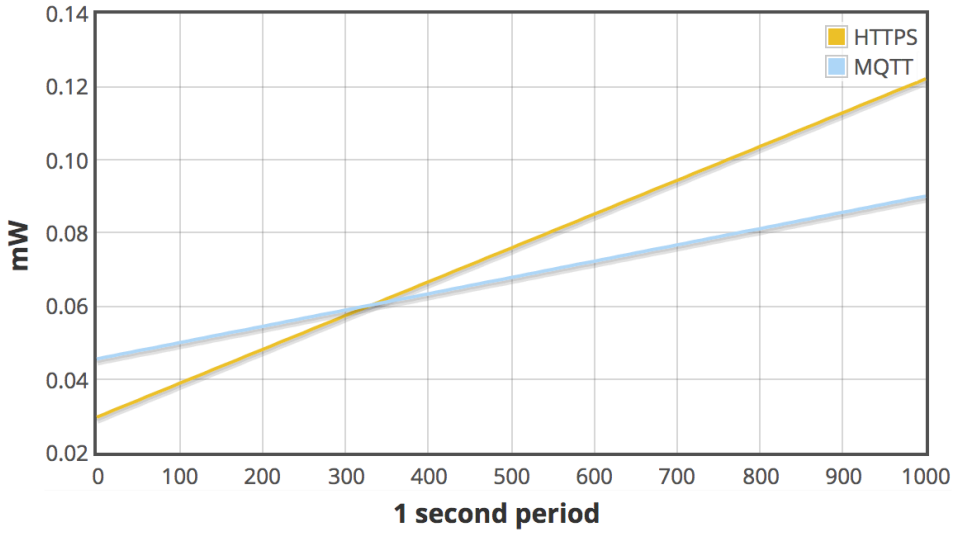
amount of power taken to establish the initial connection to the server:

% 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

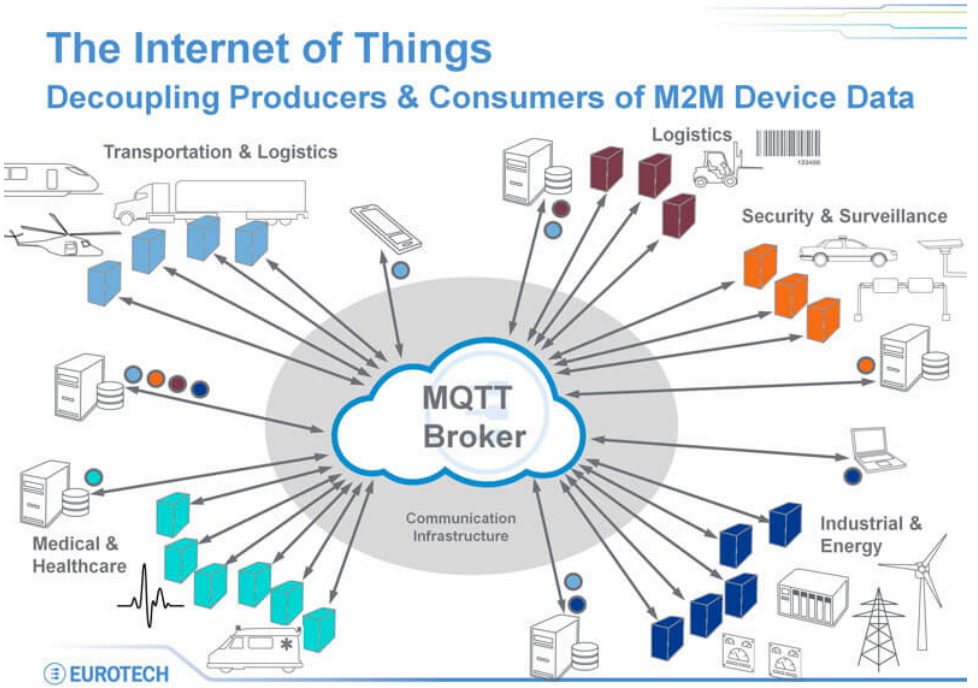


you'd save ~4.1% battery per day just by using MQTT over HTTPS to maintain an open stable connection.

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

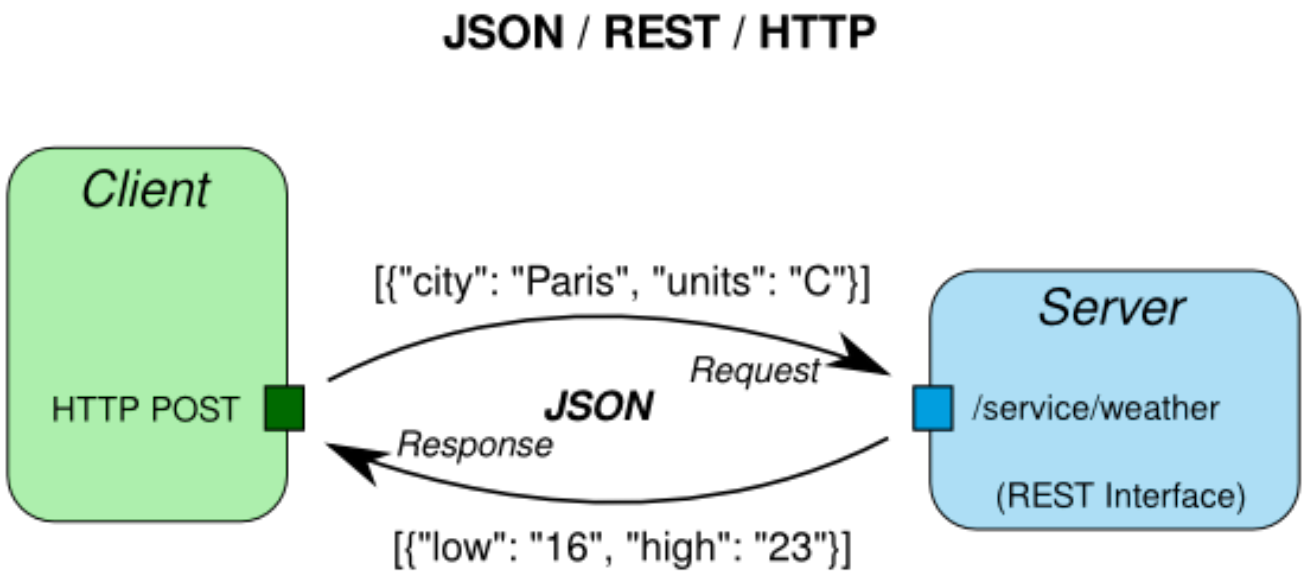
- Does not have a point-to-point (aka queues) messaging pattern
  - Point to Point or One to One means that there can be more than one consumer listening on a queue but only one of them will be get the message
- Does not define a standard client API, so application developers have to select the best 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…

REST

- Basic concepts
- Basic programming



Talking about standards: the Java case

- JSR 311: JAX-RS: The Java API for RESTful Web Services
  - https://jcp.org/en/jsr/detail?id=311
- It's a Java programming language API spec that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.
- JAX-RS uses annotations (see later "decorators" with Python), introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints.
- From version 1.1 on, JAX-RS is an official part of Java EE 6. A notable feature of being an official part of Java EE is that no configuration is necessary to start using JAX-RS.

Let's talk Python

- big web frameworks
  - Django: http://www.django-rest-framework.org/
  - Pyramid: https://trypyramid.com/
  - Falcon: http://falconframework.org/
- minimalist web frameworks
  - **Bottle: http://bottlepy.org/docs/dev/**
  - Flask: http://flask.pocoo.org/
  - Pycnic: http://pycnic.nullism.com/

WSGI (Web Server Gateway Interface)

- It's a Python specification (standards) that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.
  - http://wsgi.readthedocs.io/en/latest/
  - WSGI is not a server, a python module, a framework, an API or any kind of software. It is just an interface specification by which server and application communicate. Both server and application interface sides are specified in the PEP 3333.
  - If an application (or framework or toolkit) is written to the WSGI spec then it will run on any server written to that spec.
- A WSGI server (meaning WSGI compliant) only receives the request from the client, pass it to the application and then send the response returned by the application to the client. It does nothing else. All the gory details must be supplied by the application or middleware.
- Frameworks that run on WSGI:
  - http://wsgi.readthedocs.io/en/latest/frameworks.html
  - https://wiki.python.org/moin/WebFrameworks
- WSGI-compliant servers
  - uWSGI, Tornado, Gunicorn, Apache, Amazon Beanstalk, Google App Engine, and others.

Bottle: Python Web Framework

- The entire library is distributed as a one-file module
- The built-in default server is based on WSGIServer.
  - This non-threading HTTP server is perfectly fine for development and early production, but may become a performance bottleneck when server load increases.
  - There are three ways to eliminate this bottleneck: http://bottlepy.org/docs/0.12/deployment.html
- Bottle is database-agnostic and doesn't care where the data is coming from.
  - If you'd like to use a database in your app, the Python Package Index has several interesting options, like SQLAlchemy, PyMongo, MongoEngine, CouchDB…

- `sudo apt-get install python-bottle`

The server structure

```
import bottle
from bottle import request, response, route
from bottle import post, get, put, delete
import json
import time

_names = set()   # the set of sensors names

@post('/sensors')
def creation_handler():
  ...

@get('/sensors')
def listing_handler():
 ...

@get('/sensors/<sensor_name>')
def listing_with_name_handler(sensor_name):
 ...

@put('/sensors/<sensor_name>')
def update_handler(sensor_name):
 ...

@delete('/sensors/<sensor_name>')
def delete_handler(sensor_name):
 ...

@route('/')
def hello():
    return 'Hello World'

if __name__ == '__main__':
    bottle.run(host = '127.0.0.1', port = 8000)
```

The server side: POST

```
@post('/sensors')
def creation_handler():
    try:
        # parse input data
        try:
            data = json.loads(request.body.read())
        except:
            raise ValueError

        # extract the sensor name
        try:
            sensor_name = data['name']
        except (TypeError, KeyError):
            raise ValueError

        # check for the existence of the sensor name
        if len( _names ) > 0:
            if [ x for x in _names if x[0] == sensor_name]:
                raise KeyError

    except ValueError:
        response.status = 400
        return "Bad Request - input data with errors in POST"

    except KeyError:
        response.status = 409
        return "Bad Request - sensor name already exist in POST"

    # add element with default value set to 0 and current time
    new_sensor_t = (sensor_name, 0, time.ctime())
    _names.add( new_sensor_t )

    # return 200 Success
    response.status = 200
    response.headers['Content-Type'] = 'application/json'
    return json.dumps(new_sensor_t)
```

The server side: GET

```
@get('/sensors')
def listing_handler():
    response.headers['Content-Type'] = 'application/json'
    return json.dumps(list(_names))

@get('/sensors/<sensor_name>')
def listing_with_name_handler(sensor_name):
    response.headers['Content-Type'] = 'application/json'

    try:
        # check for the existence of sensor with name "sensor_name"
        the_sensor = [ x for x in _names if x[0] == sensor_name]
        if the_sensor:
            return json.dumps(the_sensor)
        else:
            raise KeyError

    except KeyError:
        response.status = 409
        return "Bad Request - sensor name does not exist in GET"
```

The server side: PUT

```
@put('/sensors/<sensor_name>')
def update_handler(sensor_name):
    try:
        # parse input data
        try:
            data = json.loads(request.body.read())
        except:
            raise ValueError
        # extract and validate new value
        try:
            if not data["value"].isdigit():
                raise ValueError
            newdata = int(data["value"])

        except (TypeError, KeyError):
            raise ValueError

        # check for the existence of sensor with name "sensor_name"
        the_sensor = [ x for x in _names if x[0] == sensor_name]
        if not the_sensor:
            raise KeyError



    except ValueError:
        response.status = 400
        return
    except KeyError:
        response.status = 409
        return "Bad Request - sensor name does not exist in GET”
```

```
# add new name and remove old name
 _names.remove(the_sensor[0])
 newdata_t = (sensor_name, newdata, time.ctime())
 _names.add(newdata_t)

 # return 200 Success
 response.headers['Content-Type'] = 'application/json'
 return json.dumps(newdata_t)
```

The server side: DELETE

```
@delete('/sensors/<sensor_name>')
def delete_handler(sensor_name):
    try:
        # check for the existence of sensor with name "sensor_name"
        the_sensor = [ x for x in _names if x[0] == sensor_name]
        if not the_sensor:
            raise KeyError

    except KeyError:
        response.status = 409
        return "Bad Request - sensor name does not exist in DELETE"

    # Remove name
    _names.remove(the_sensor[0])
    return
```

Client side tools

- Postman:

  https://www.getpostman.com/

- Advanced REST Client:

  https://advancedrestclient.com/

- Curl:

  https://curl.haxx.se/

  - Examples:

```
curl -X POST -H "Content-type: application/json" -d '{"name":"sensor1"}'\
      localhost:8000/sensors
curl -X POST -H "Content-type: application/json" -d '{"name":"sensor2"}'\
      localhost:8000/sensors

curl -X GET localhost:8000/sensors
curl -X GET localhost:8000/sensors/sensor2

curl -X PUT -H "Content-type: application/json" -d '{"value":"237"}'\
      localhost:8000/sensors/sensor2

curl -X DELETE localhost:8000/sensors/sensor2
```

Client side via Python

- https://pypi.python.org/pypi/requests
- http://docs.python-requests.org/en/master/
- Cite: "*Requests is the only Non-GMO HTTP library for Python, safe for human consumption.*" ☺

Periodic GET

```
import requests
import time

rest_s_url = 'http://localhost:8000'

while True:
    try:
        resp = requests.get( rest_s_url+'/sensors')
        if resp.status_code != 200:
            # This means something went wrong.
            raise ValueError

        for sdata in resp.json():
            print sdata[0]+' has value '+str(sdata[1])+' at '+sdata[2]

        time.sleep(3)

    except ValueError:
        print "Bad reply from GET"
```

Periodic PUT

```
import requests, time, random, json

rest_s_url = 'http://localhost:8000/'
sensor_name = 'sensor1'

while True:
    try:

        # get the new data value
        new_s_value = random.randint(1, 1000)
        new_sensor_t = {'value': str(new_s_value)}

        resp = requests.put(rest_s_url+'sensors/'+sensor_name,\
            data=json.dumps(new_sensor_t),\
            headers={'Content-Type':'application/json'})

        if resp.status_code != 200:
            # This means something went wrong.
            raise ValueError

        time.sleep(3)

    except ValueError:
        print "Bad reply from PUT"
```