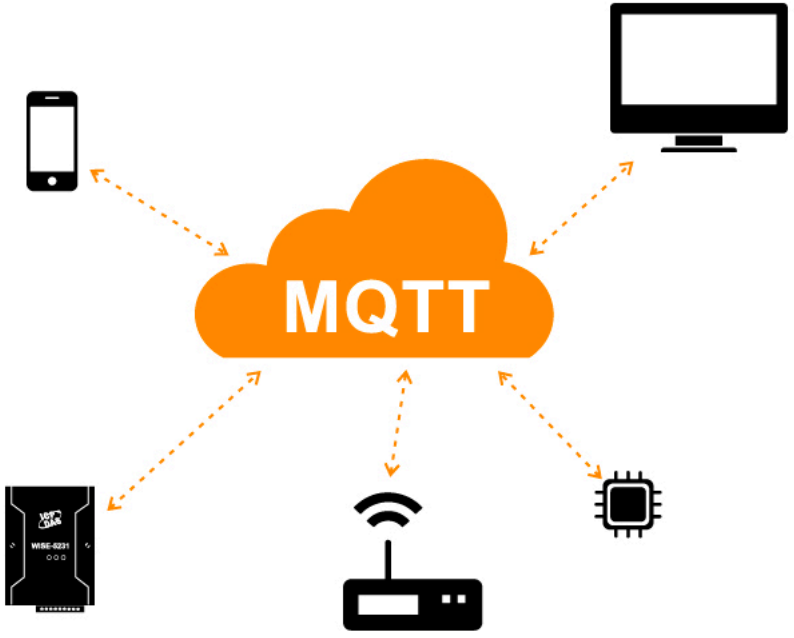
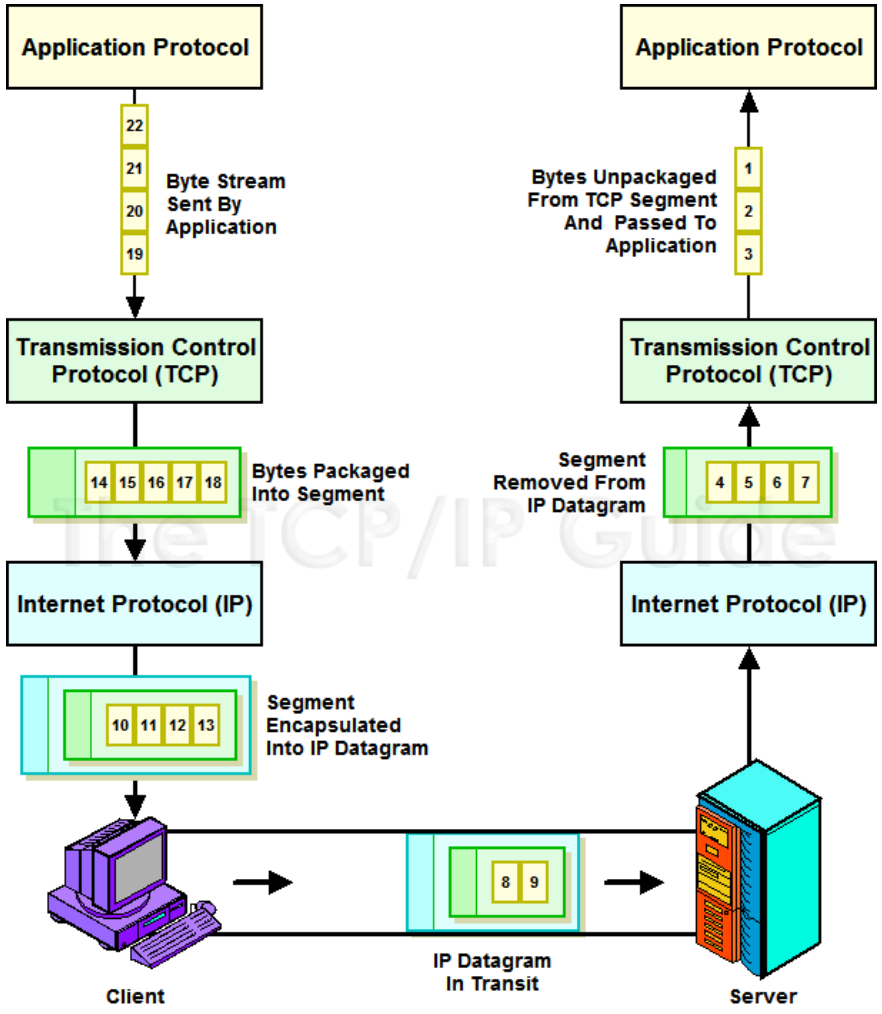
MQTT basics



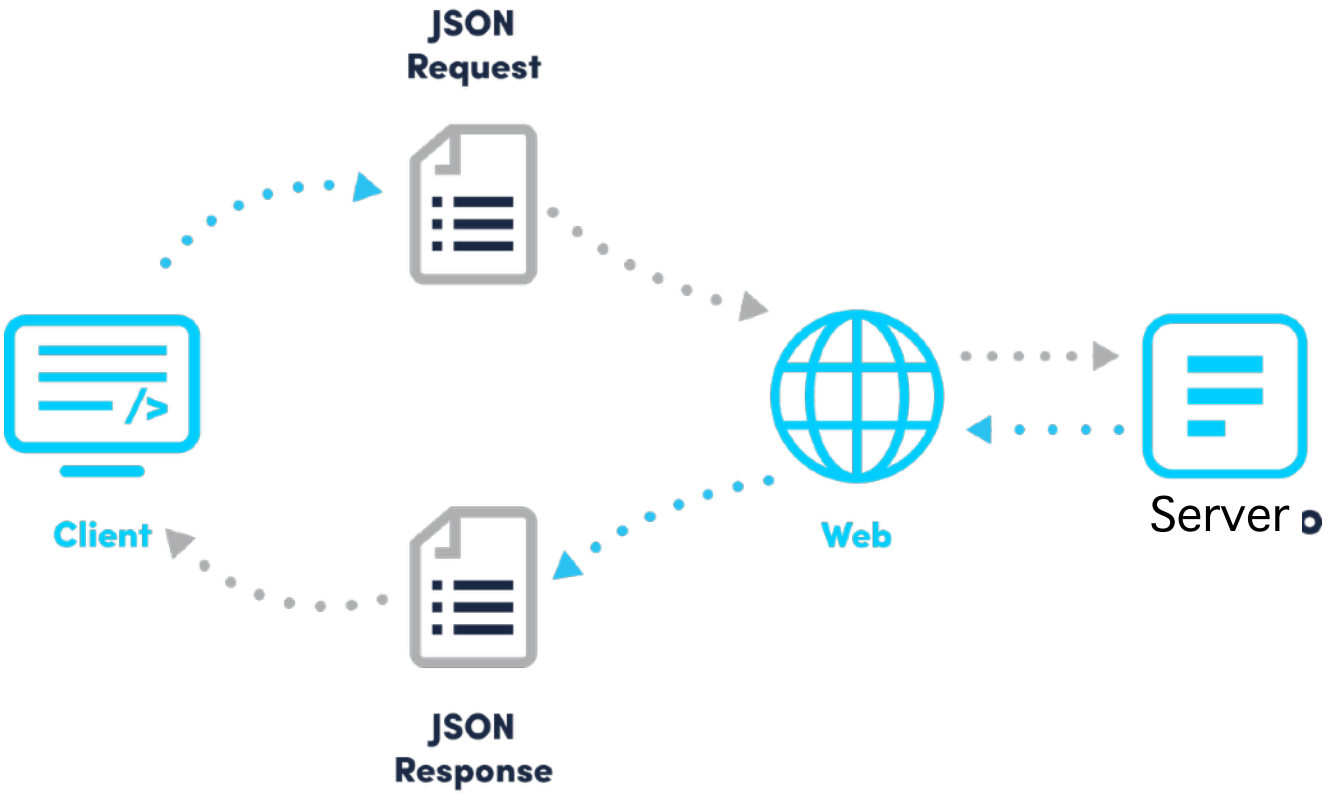
From "byte streams" to "messages"

- 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



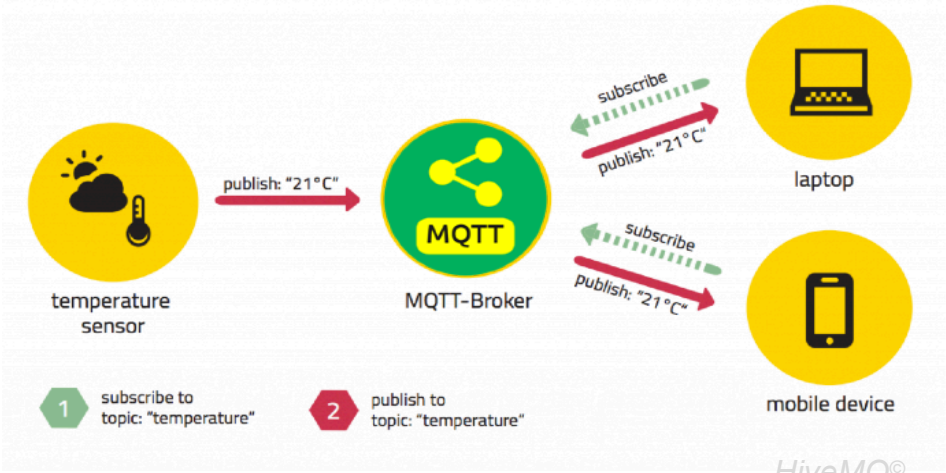
Interchanging messages: request/response paradigm

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



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)



{“value": 237} vs. <value>237</value>

Data Representation

- 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:
  - JSON
  - XML

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

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Pete", "lastName":"Jones" }
]}
```

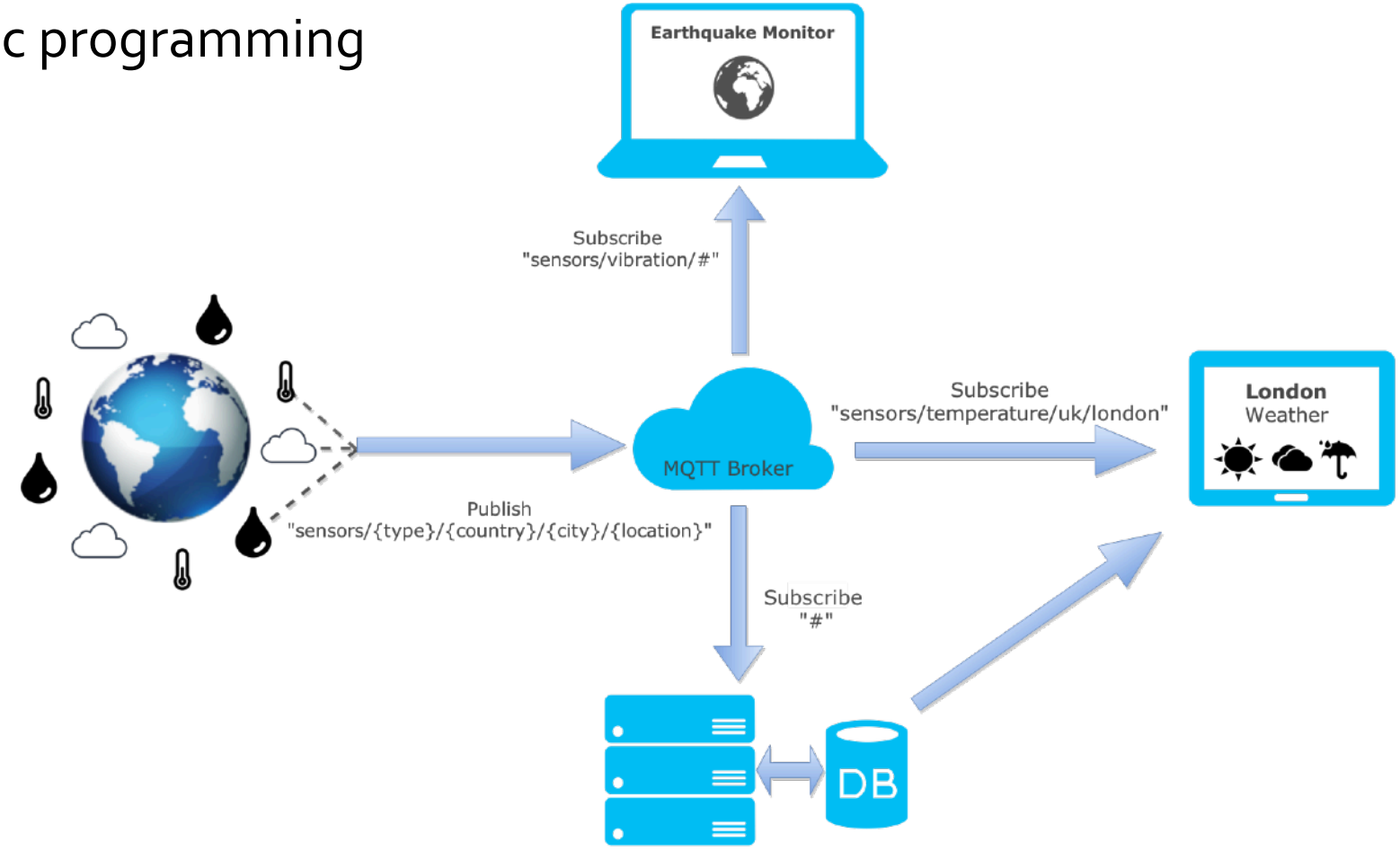
XML

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Pete</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

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'
```

- Basic concepts
- Basic programming



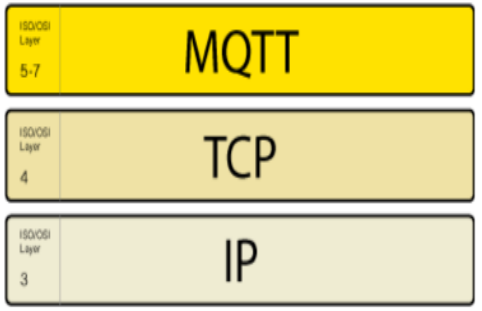
Source: https://zoetrope.io/tech-blog/brief-practical-introduction-mqtt-protocol-and-its-application-iot

MQTT Message Queuing Telemetry Transport

- 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

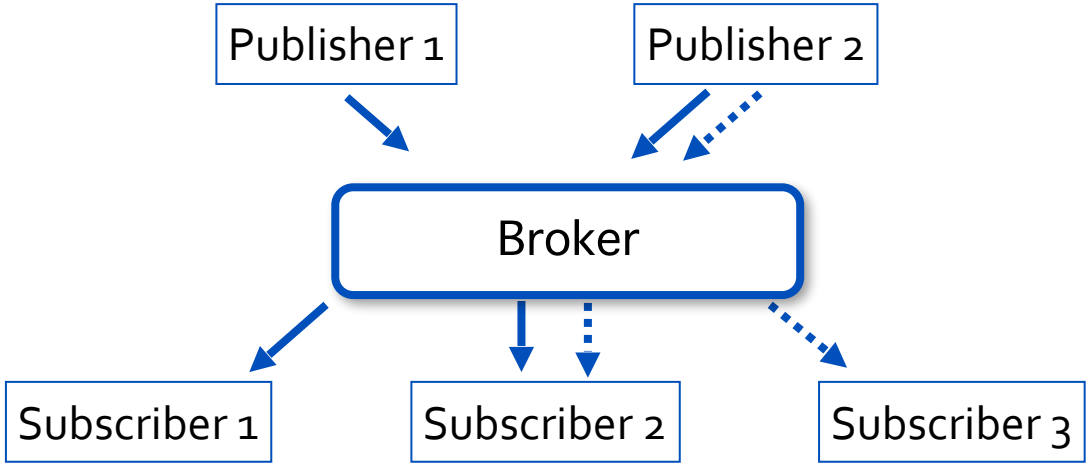
MQTT v5.0

- MQTT v5.0 is the successor of MQTT 3.1.1
  - MQTT v5.0 is not backward compatible; too many new things are introduced so existing implementations have to be revisited.
- According to the specification, MQTT v5.0 adds a significant number of new features to MQTT while keeping much of the core in place.
  - `http://docs.oasis-open.org/mqtt/mqtt/v5.0/cs02/mqtt-v5.0-cs02.html`
- The major functional objectives are:
  - Enhancements for scalability and large scale systems in respect to setups with 1000s and millions of devices.
  - Improved error reporting (Reason Code & Reason String)
  - Extensibility mechanisms including **user properties, payload format and content type**
  - Performance improvements and improved support for small clients

- `https://www.youtube.com/watch?time_continue=3&v=YIpesv_bJgU`

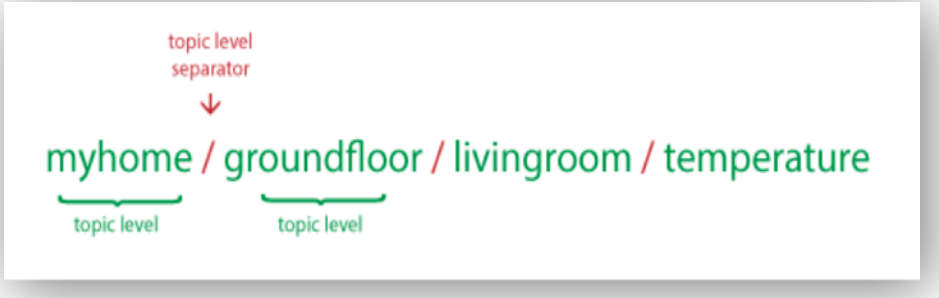
Publish/subscribe pattern

- 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

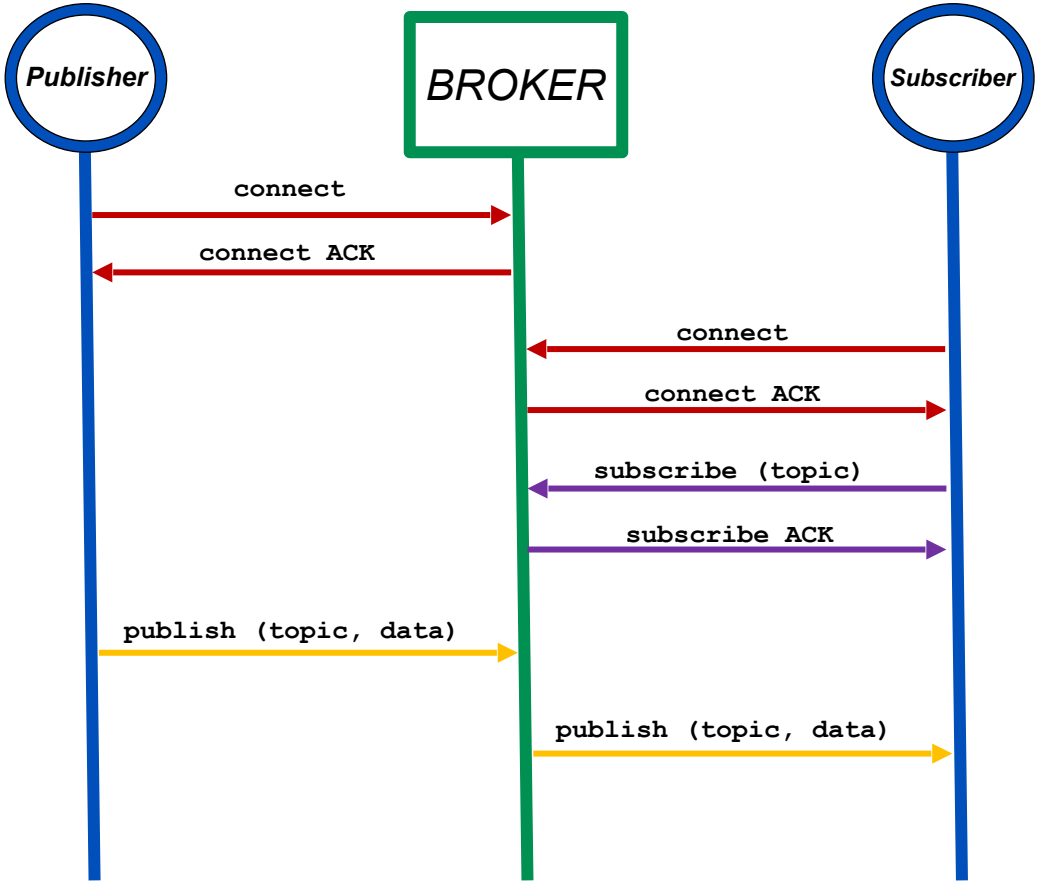
Topics wildcards

- 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

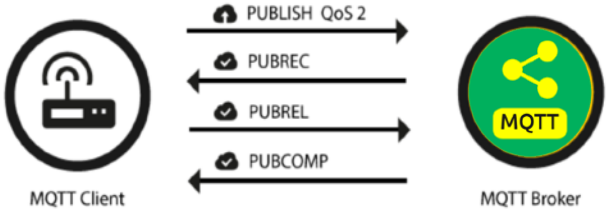
- First of all:
  - Don't use a leading forward slash
  - Don't use spaces in a topic
  - Use only ASCII characters, avoid non printable characters
- Then, try to..
  - Keep the topic short and concise
  - Use specific topics, instead of general ones
  - Don't forget extensibility
- Finally, be careful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

Publish/subscribe interactions sequence



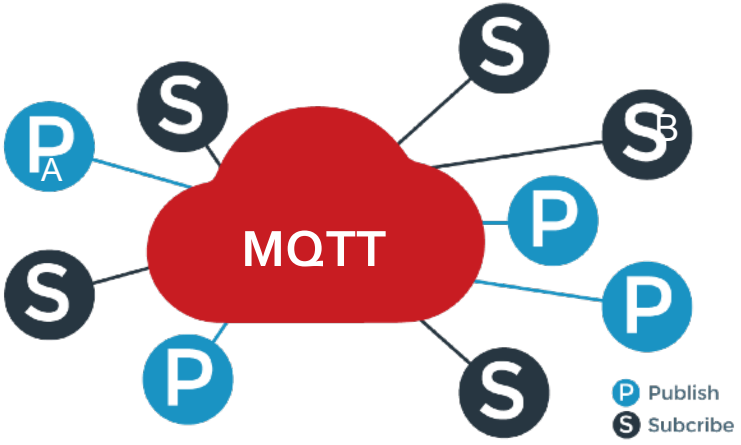
Quality of Service (QoS)

- 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.
- 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.

Retained Messages!!!

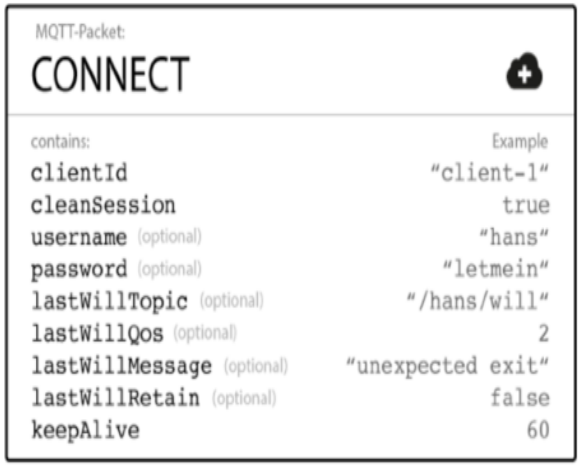
- A retained message is a normal MQTT message with the retained flag set to true. **The broker will store the last retained message and the corresponding QoS for that topic**
  - Each client that subscribes to a topic pattern, which matches the topic of the retained message, will receive the message 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.



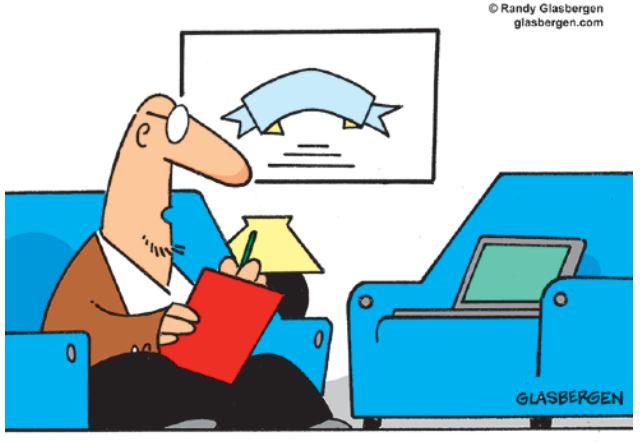
WHEN?

MQTT 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 words 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."

A few words on security

Smart homes can be easily hacked via unsecured MQTT servers

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

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

TOTAL RESULTS

49,197

TOP COUNTRIES

China	12,151
United States	8,257
Germany	3,092
Korea, Republic of	2,003
Hong Kong	2,002

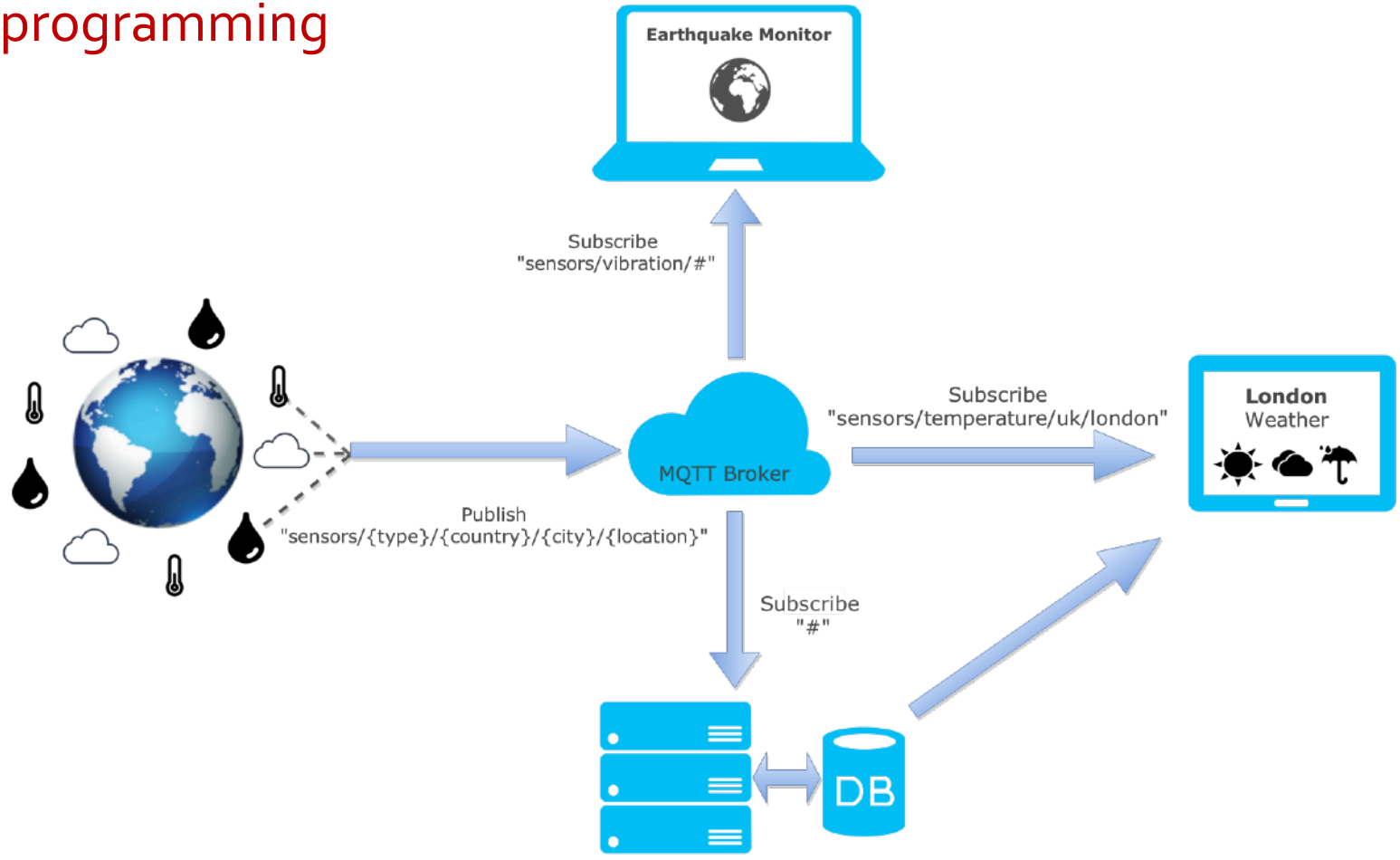
TOTAL RESULTS

32,888

TOP COUNTRIES

China	8,446
United States	4,733
Germany	1,719
Hong Kong	1,614
Taiwan	1,565

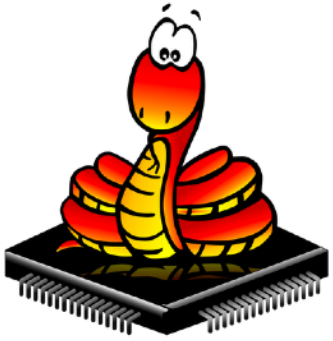
- Basic concepts
- Basic programming



Warning: Python vs Micropython

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

vs.



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

- iot.eclipse.org
  - https://iot.eclipse.org/getting-started#sandboxes
- test.mosquitto.org
  - http://test.mosquitto.org/
- 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: 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: 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.

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_start/loop_stop`

```
# 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):
    global msg_counter
    print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))
    msg_counter+=1

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

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

mqttc.connect(THE_BROKER, keepalive=60)

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

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

Example 3: very basic periodic producer

```
# File: example3.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()
```

```
('Connected to ', 'test.mosquitto.org', 'port: ', 1883)
('Flags: ', {'session present': 0}, 'return code: ', 0)
PMtest/rndvalue 56
PMtest/rndvalue 25
PMtest/rndvalue 43
PMtest/rndvalue 67
PMtest/rndvalue 0
PMtest/rndvalue 44
…
```

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

Generates a new data every 5 secs

Example 4: Pub/Sub with JSON

Producer

```
# File: example4_prod.py

…

mqttc.loop_start()

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

    the_msg_str = json.dumps(the_msg)

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

mqttc.loop_stop()
```

Consumer

```
# File: example4_cons.py
 …

# 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']))

…
```

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

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 publisher

```
# file: a_simple_pub.py
from mqtt import MQTTClient
import pycom
import sys
import time

import ufun

wifi_ssid = 'THE_NAME_OF_THE_AP'
wifi_passwd = ''
broker_addr = 'iot.eclipse.org'
dev_id = 'THE_NAME_OF_THE_DEVICE'

def get_data_from_sensor(sensor_id="RAND"):
    if sensor_id == "RAND":
        return ufun.random_in_range()

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:
    the_data = get_data_from_sensor()
    client.publish('THE_TOPIC_TO_BE_USED', str(the_data))
    time.sleep(1)
```

MicroPython: a simple subscriber

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


import ufun

wifi_ssid = 'THE_NAME_OF_THE_AP'
wifi_passwd = ''
broker_addr = 'iot.eclipse.org'
dev_id = 'THE_NAME_OF_THE_DEVICE'

def settimeout(duration):
   pass

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


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('THE_TOPIC_TO_BE_USED')

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