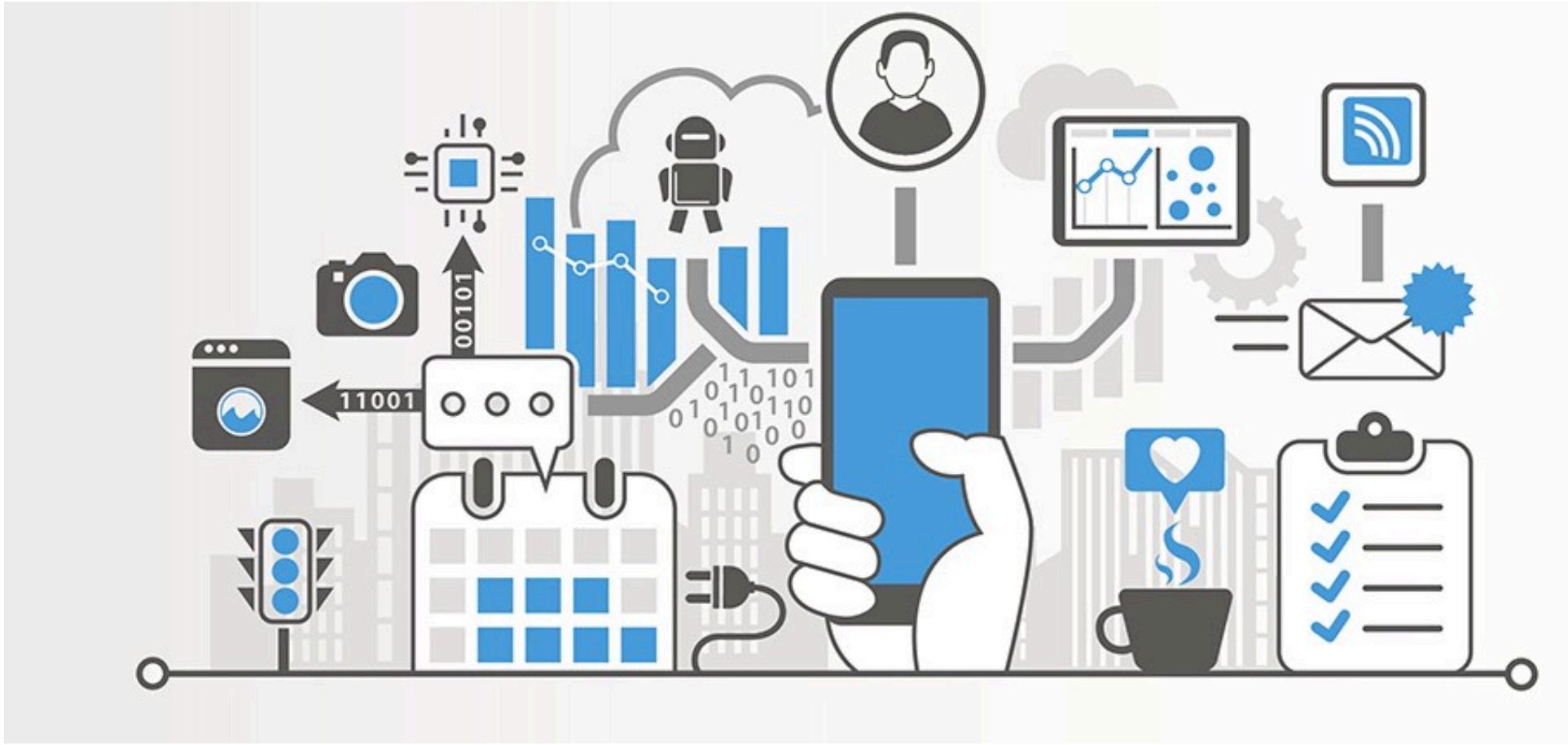
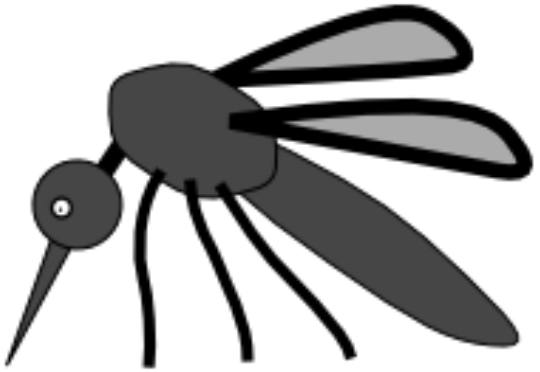
Programming for IoT Applications

Edoardo Patti

Lecture 8

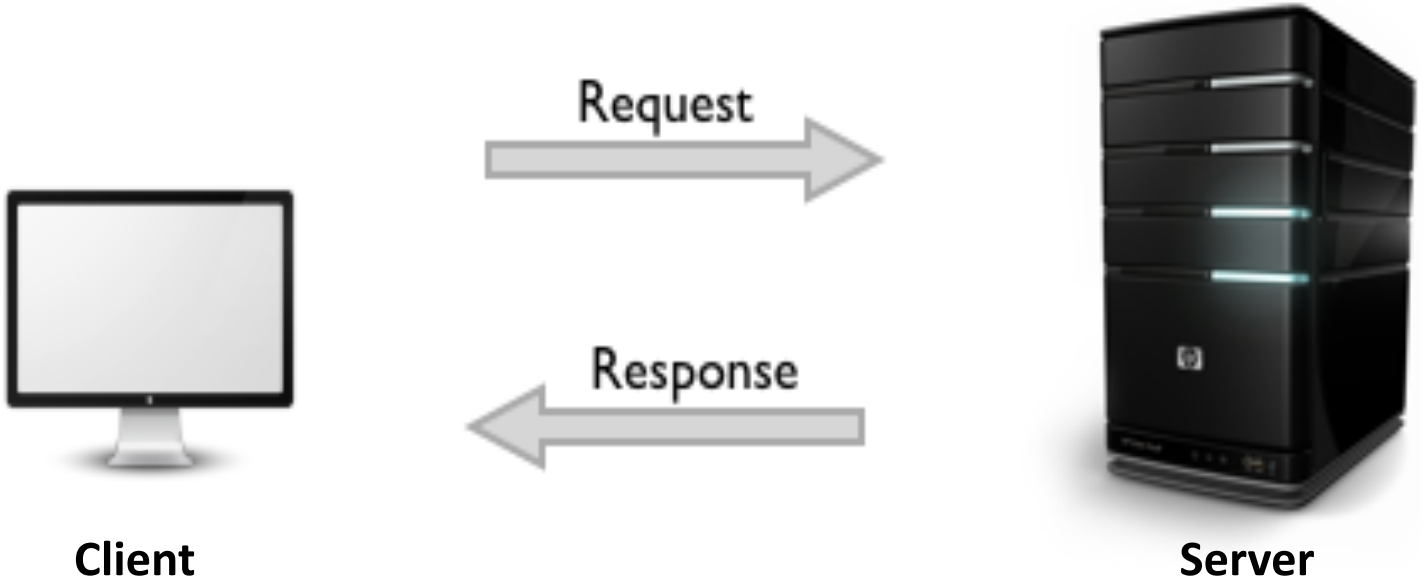




MQTT PROTOCOL AND PYTHON IMPLEMENTATION

Communication paradigms

Request/Response is a **synchronous** communication paradigm. The client requests for some data and the server responds to the request.



Communication paradigms

Publish/subscribe is an **asynchronous** communication paradigm. It allows the development of loosely-coupled event-driven architectures. It removes the dependencies between producer and consumer of information.

The communication is based on Topics

(i.e. a label to identify a communication

flow)

Communication paradigms

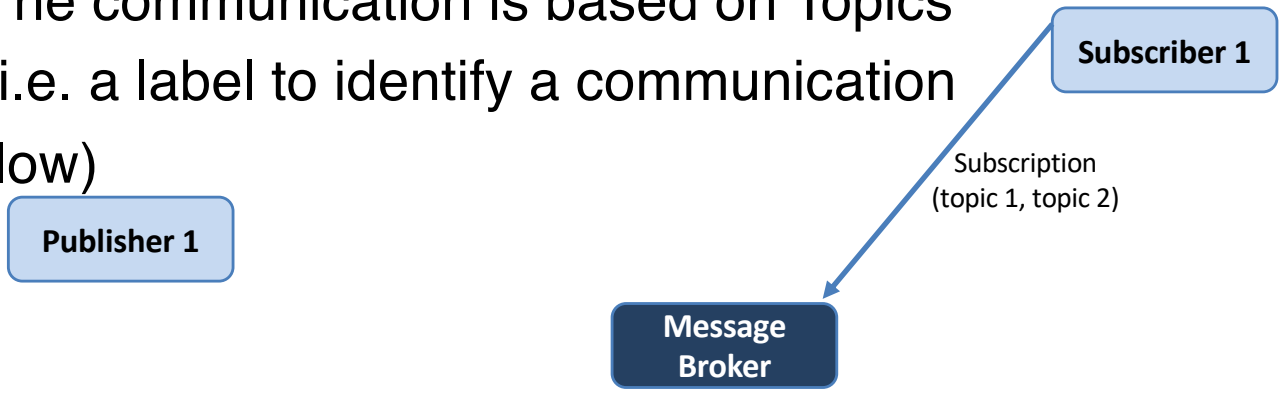
Publish/subscribe is an **asynchronous** communication paradigm. It allows the development of loosely-coupled event-driven architectures. It removes the dependencies between producer and consumer of information.

The communication is based on Topics (i.e. a label to identify a communication flow)

Communication paradigms

Publish/subscribe is an **asynchronous** communication paradigm. It allows the development of loosely-coupled event-driven architectures. It removes the dependencies between producer and consumer of information.

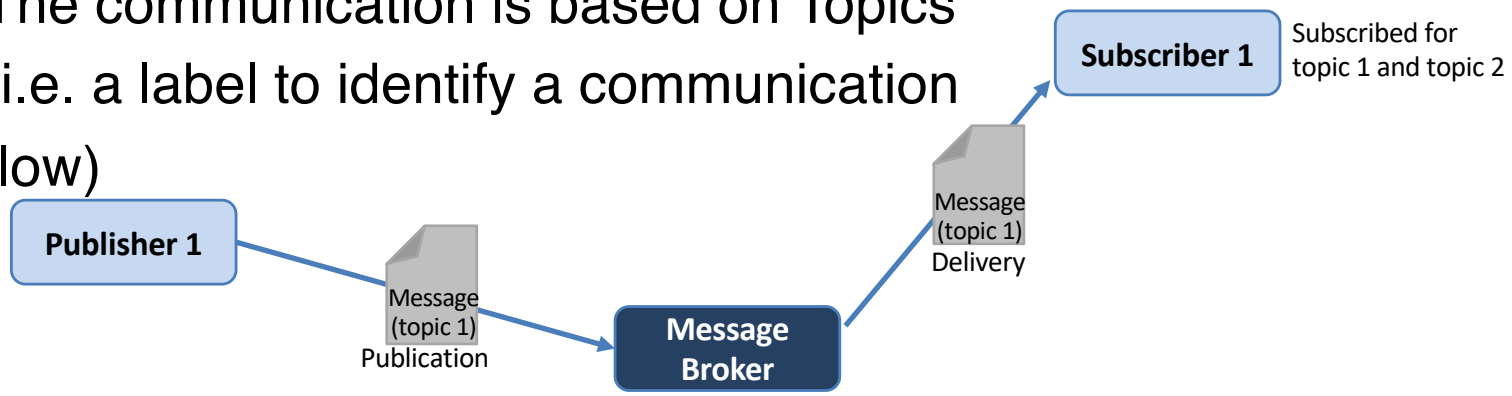
The communication is based on Topics (i.e. a label to identify a communication flow)



Communication paradigms

Publish/subscribe is an **asynchronous** communication paradigm. It allows the development of loosely-coupled event-driven architectures. It removes the dependencies between producer and consumer of information.

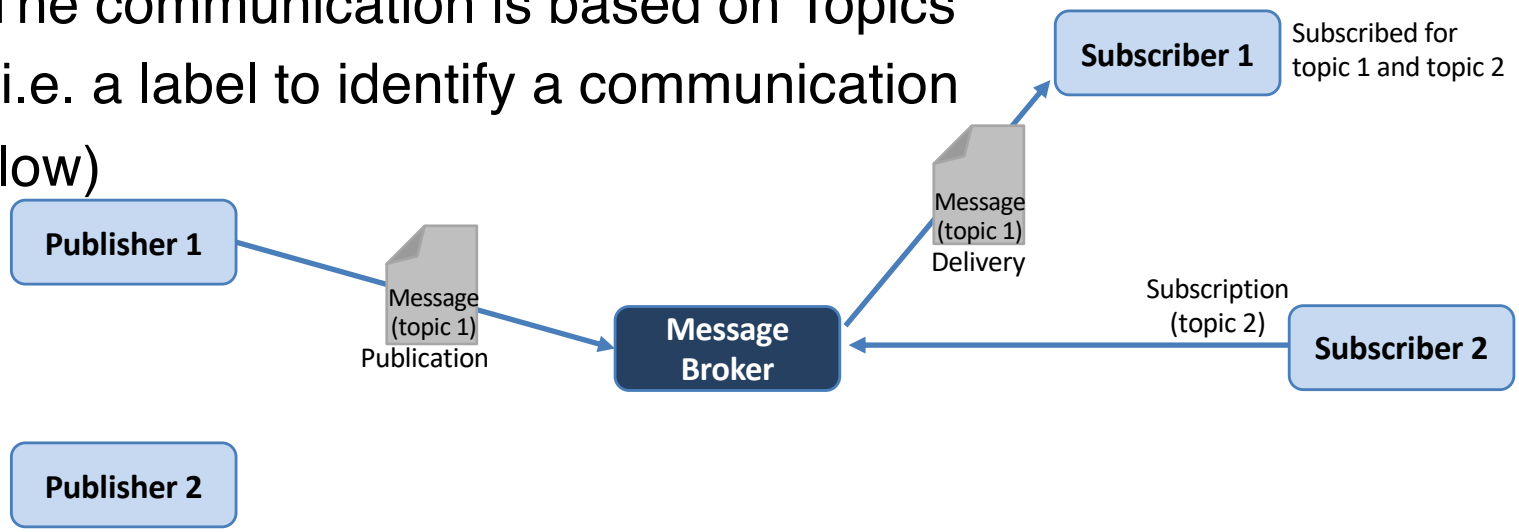
The communication is based on Topics (i.e. a label to identify a communication flow)



Communication paradigms

Publish/subscribe is an **asynchronous** communication paradigm. It allows the development of loosely-coupled event-driven architectures. It removes the dependencies between producer and consumer of information.

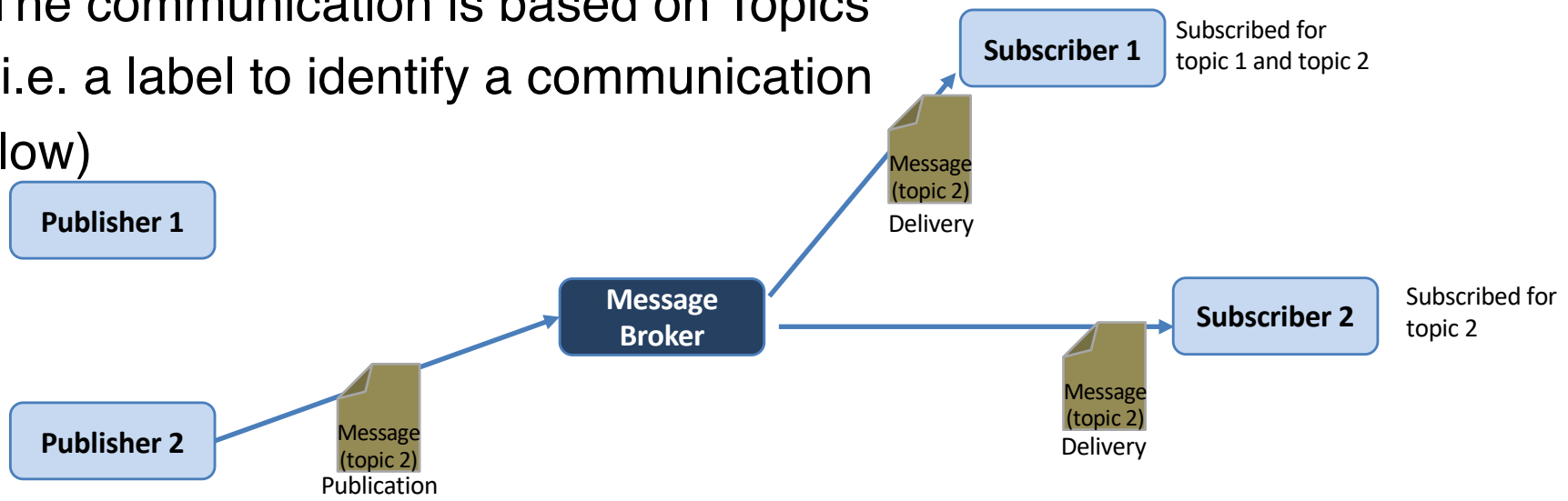
The communication is based on Topics (i.e. a label to identify a communication flow)



Communication paradigms

Publish/subscribe is an **asynchronous** communication paradigm. It allows the development of loosely-coupled event-driven architectures. It removes the dependencies between producer and consumer of information.

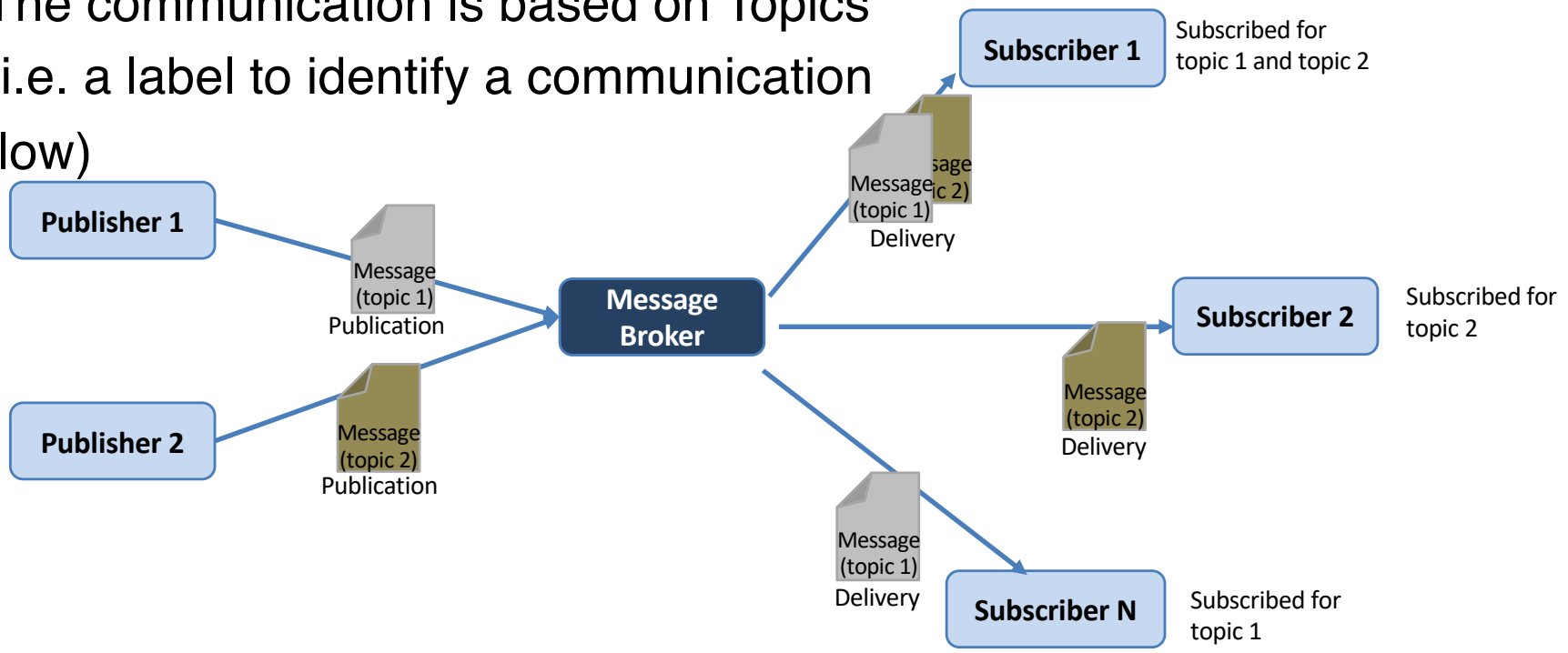
The communication is based on Topics (i.e. a label to identify a communication flow)



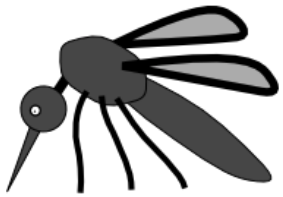
Communication paradigms

Publish/subscribe is an **asynchronous** communication paradigm. It allows the development of loosely-coupled event-driven architectures. It removes the dependencies between producer and consumer of information.

The communication is based on Topics (i.e. a label to identify a communication flow)



MQTT



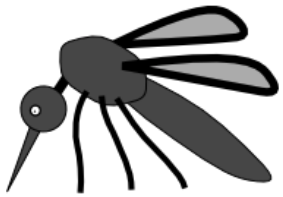
MQTT (Message Queue Telemetry Transport) is a publish-subscribe messaging protocol for use on top of the TCP/IP protocol.

It could be used for Event driven Architectures or Message Oriented Middleware

Facebook used MQTT in Facebook Messenger.

This is a **smartphone application**, not a sensor application.

MQTT: Topic



MQTT provides functionality of a topic-based publish/subscribe mechanism. Each message is published under a certain topic.

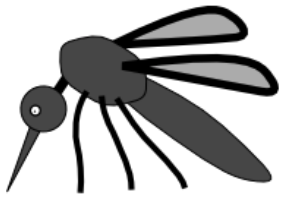
The **topic** is hierarchical with **"/"** used as separator

Example:

- /sensor/1234/temperature
- /sensor/1234/humidity
- /sensor/567/temperature
- /sensor/567/humidity

During the **Publication** a specific topic must be given

MQTT: Wildcards



A **Subscription** may be to an explicit topic, in which case only messages to that topic will be received, or it may include **wildcards**

Only subscribers can use wildcards

Two wildcards are available, '**+**' or '**#**'.

'**+**' can be used as a wildcard for a single level of hierarchy.

Examples:

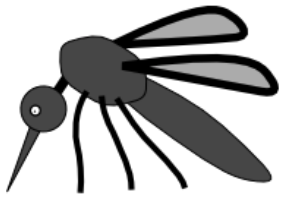
It could be used to get information on all measurements sent by sensor 1234:

- /sensor/1234/**+**

Or the humidity measurement sent by all sensors in the network

- /sensor/**+**/humidity

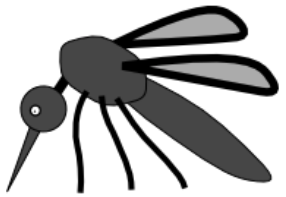
MQTT: Wildcards



For a topic of "**a/b/c/d**", the following example subscriptions will match:

- a/b/c/d
- **+**/b/c/d
- a/**+**/c/d
- a/**+**/**+**/d
- **+**/**+**/**+**/**+**

MQTT: Wildcards



'#' can be used as a wildcard for all remaining levels of hierarchy. This means that **it must be the final character in a subscription**.

Examples:

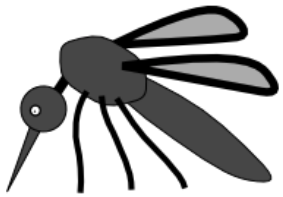
It could be used to get information on all measurements sent by sensor 1234:

- /sensor/1234/#

Or to get information on all measurements sent by all sensors in the network

- /sensor/#

MQTT: Wildcards



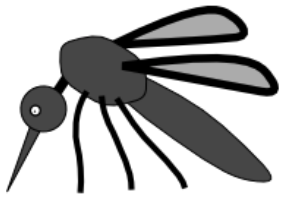
For a topic of "**a/b/c/d**", the following example subscriptions will match:

- a/b/c/d
- **#**
- a/**#**
- a/b/**#**
- a/b/c/**#**

Wildcards can also be combined together in the same subscription

- **+**/b/c/**#**

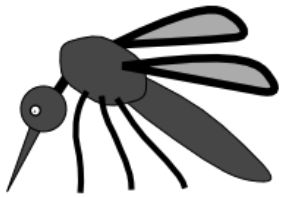
MQTT: QoS



MQTT defines three levels of **Quality of Service (QoS)**. The **QoS** defines how hard the broker/client will try to ensure that a message is received.

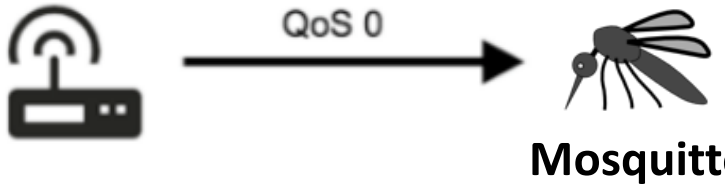
Higher levels of QoS are more reliable, but involve higher latency

MQTT: QoS



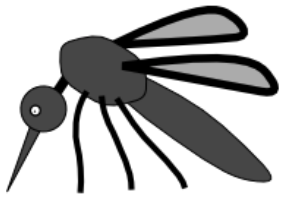
MQTT defines three levels of **Quality of Service (QoS)**. The **QoS** defines how hard the broker/client will try to ensure that a message is received.

Higher levels of QoS are more reliable, but involve higher latency



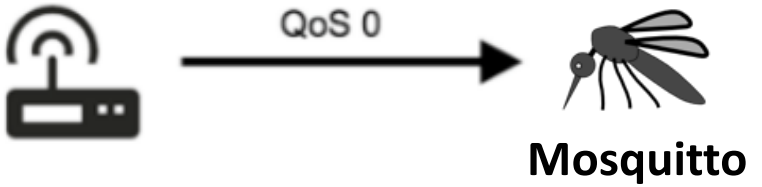
QOS 0:
The broker/client will deliver the message once, with no confirmation.

MQTT: QoS

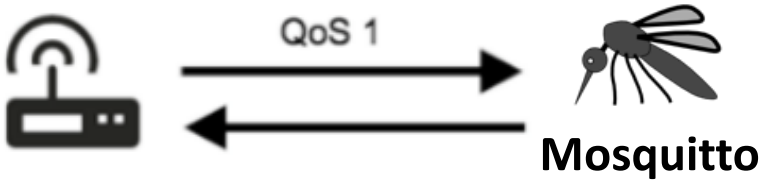


MQTT defines three levels of **Quality of Service (QoS)**. The **QoS** defines how hard the broker/client will try to ensure that a message is received.

Higher levels of QoS are more reliable, but involve higher latency

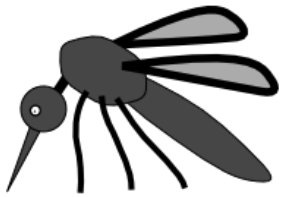


QOS 0:
The broker/client will deliver the message once, with no confirmation.



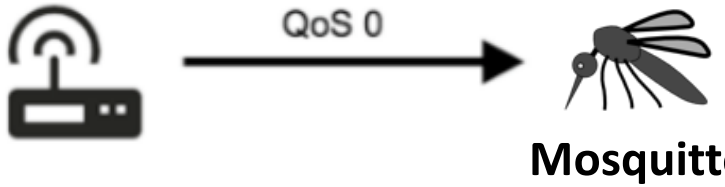
QOS 1:
The broker/client will deliver the message at least once, with confirmation required.

MQTT: QoS

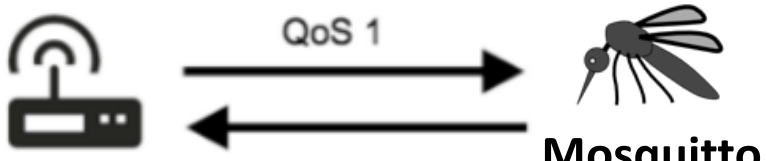


MQTT defines three levels of **Quality of Service (QoS)**. The **QoS** defines how hard the broker/client will try to ensure that a message is received.

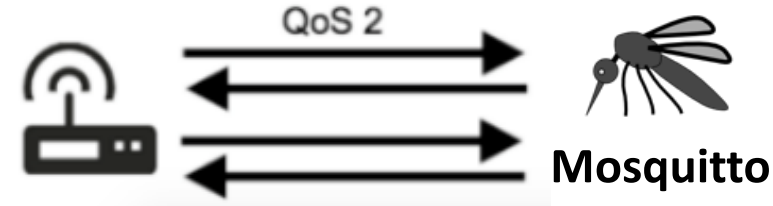
Higher levels of QoS are more reliable, but involve higher latency



QOS 0:
The broker/client will deliver the message once, with no confirmation.

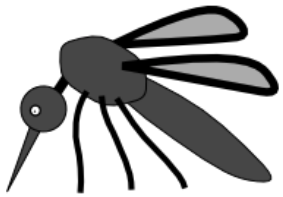


QOS 1:
The broker/client will deliver the message at least once, with confirmation required.



QOS 2:
The broker/client will deliver the message exactly once by using a four step handshake.

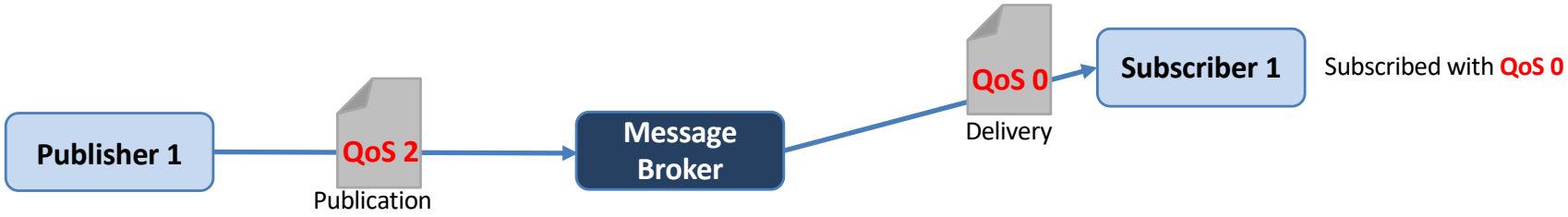
MQTT: QoS



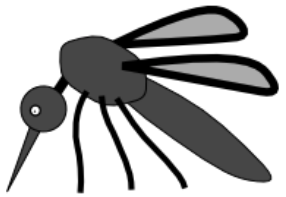
Messages may be sent at any QoS level, and clients may attempt to subscribe to topics at any QoS level.

Examples:

1) If a message is published at QoS 2 and a client is subscribed with QoS 0, the message will be delivered to that client with QoS 0.



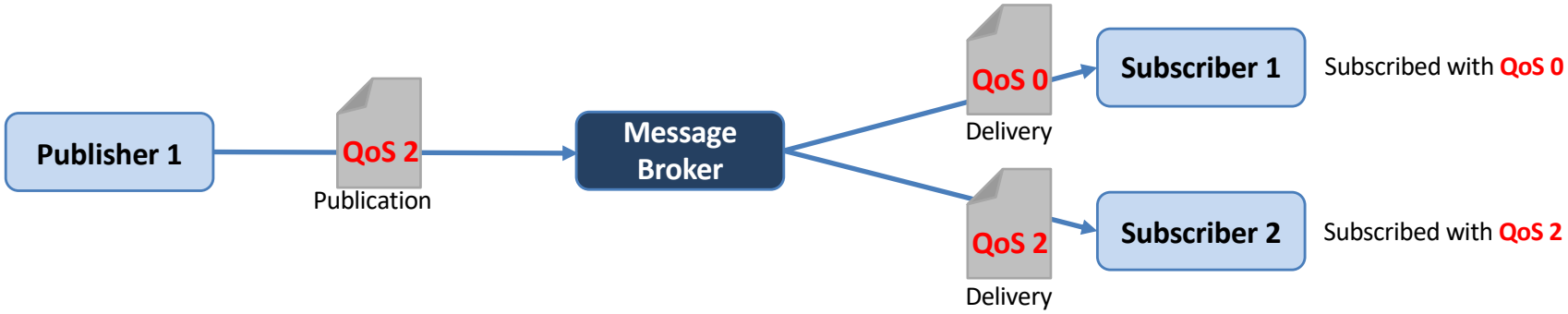
MQTT: QoS



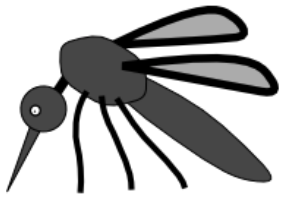
Messages may be sent at any QoS level, and clients may attempt to subscribe to topics at any QoS level.

Examples:

1) If a message is published at QoS 2 and a client is subscribed with QoS 0, the message will be delivered to that client with QoS 0. If a second client is also subscribed to the same topic, but with QoS 2, then it will receive the same message but with QoS 2.



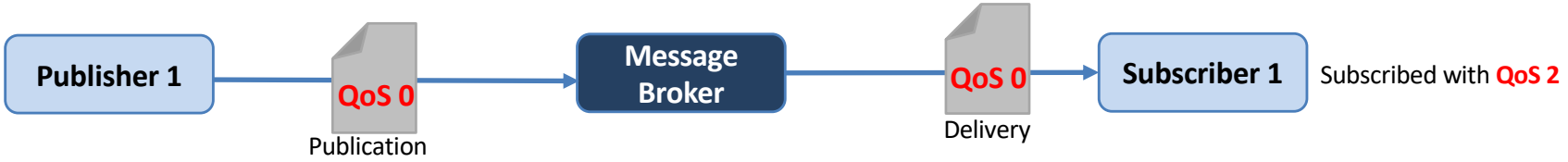
MQTT: QoS



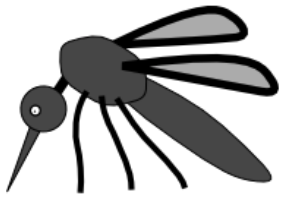
Messages may be sent at any QoS level, and clients may attempt to subscribe to topics at any QoS level.

Examples:

2) If a client is subscribed with QoS 2 and a message is published on QoS 0, the client will receive it on QoS 0.



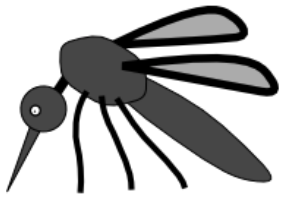
MQTT: data transmission



Session:

- A session identifies a (possibly temporary) attachment of a client to a server. All communication between client and server takes place as part of a session.

MQTT: data transmission

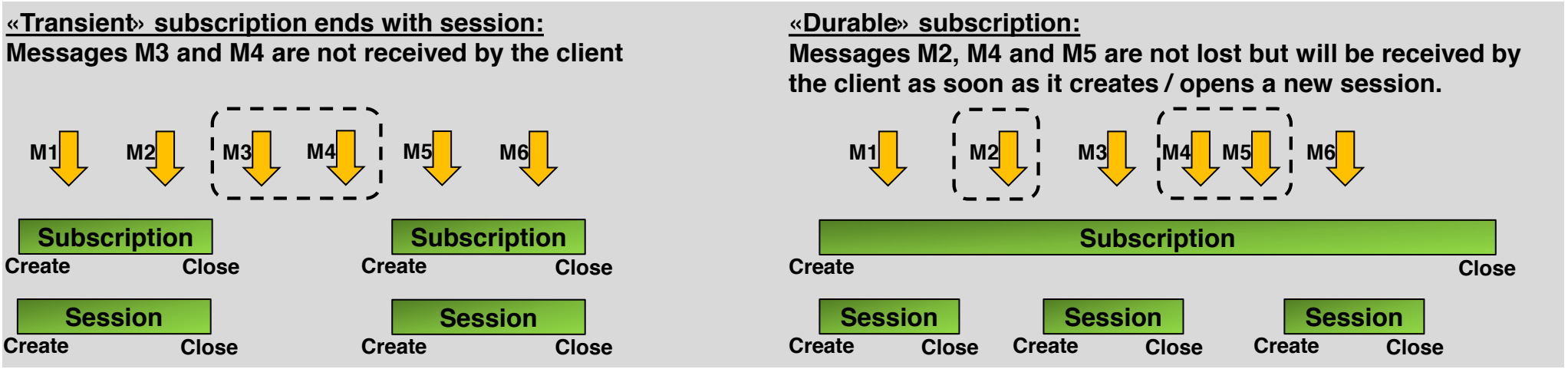


Session:

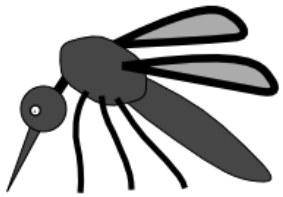
- A session identifies a (possibly temporary) attachment of a client to a server. All communication between client and server takes place as part of a session.

Subscription:

- Unlike sessions, a subscription logically attaches a client to a topic. When subscribed to a topic, a client can exchange messages with a topic. Subscriptions can be **«transient»** or **«durable»**.



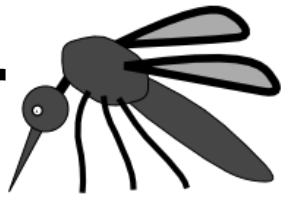
MQTT: data transmission



Message:

- Messages are the units of data exchange between topic clients. MQTT is agnostic to the internal structure of messages.

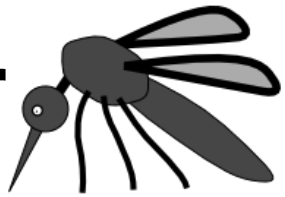
Mosquitto message broker



In this course, **Mosquitto** will be used during lab activities.
http://mosquitto.org/

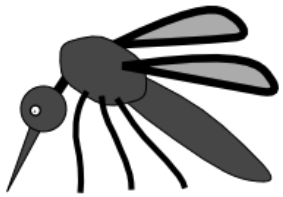
Mosquitto is an open source message broker that implements the MQTT protocol versions 3.1 and 3.1.1.

Mosquitto message broker



- Mosquitto provides a lightweight server implementation of the MQTT, written in C
- Typically, the current implementation of Roger Light's Mosquitto has an executable in the order of 120kB that consumes around 3MB RAM with 1000 clients connected. There have been reports of successful **tests with 100,000 connected clients** at modest message rates.

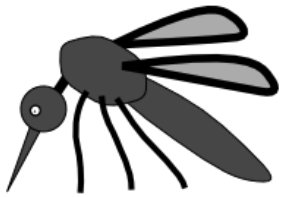
MQTT Library for Python



The Eclipse Paho MQTT Python client library implements the versions 3.1 and 3.1.1 of the MQTT protocol

It provides a client class which enable applications to connect to an MQTT broker to publish messages and to subscribe to topics and receive published messages.

Paho: programming a client

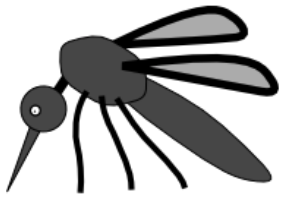


The client class can be used as an instance. The general usage flow is as follows:

1. Create a client instance
2. Connect to a broker using one of the connect() functions
3. Call one of the loop() functions to maintain network traffic flow with the broker
4. Use subscribe() to subscribe to a topic and receive messages
5. Use publish() to publish messages to the broker
6. Use unsubscribe() to unsubscribe to a topic before disconnecting (It also depends on the connection type required, *transient* or *durable*)
7. Use disconnect() to disconnect from the broker

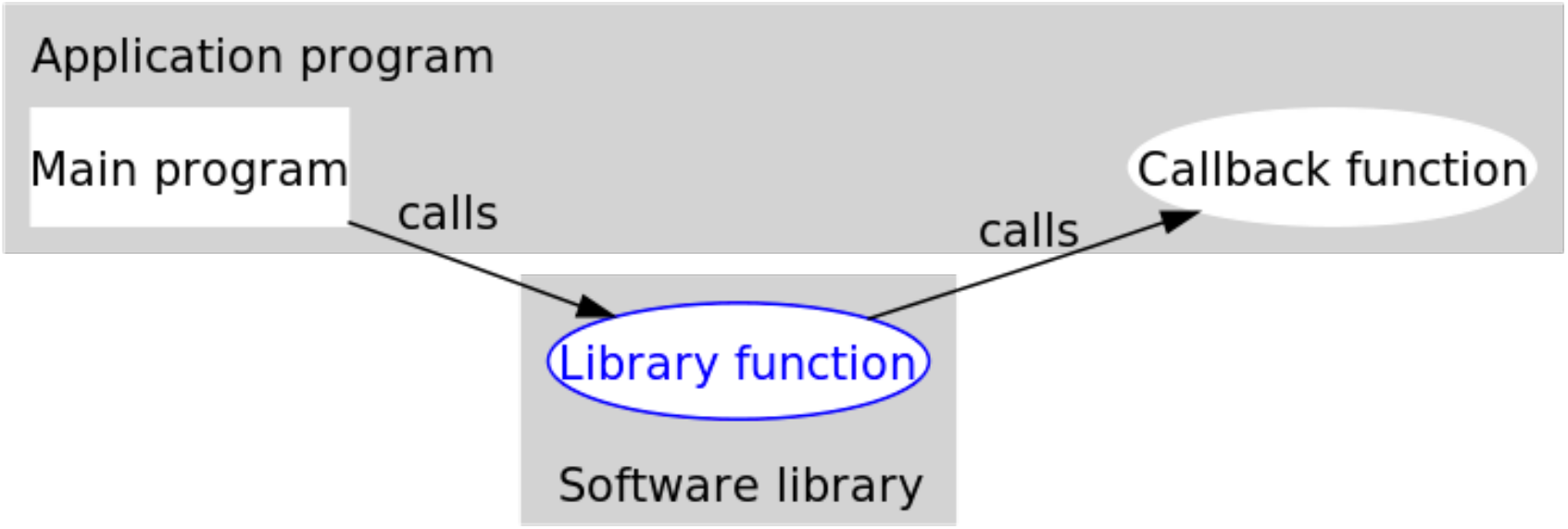
Callbacks will be called to allow the application to process events as necessary.

Definition for Callback

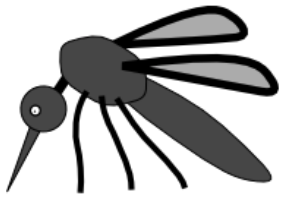


Callback is a subroutine given as argument to a function.

This subroutine is expected to be called back by the function. The invocation may be immediate as in **synchronous callback**, or it might happen at later time as in **asynchronous callback**.



Paho: client contractor Client()



Client() constructor takes the following arguments:

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

client_id
the unique client id string used when connecting to the broker.

clean_session
a boolean that determines the client type. If **True**, the broker will remove all information about this client when it disconnects (*transient connection*). If **False**, the client is a durable client and subscription information and queued messages will be retained when the client disconnects *(durable connection).*

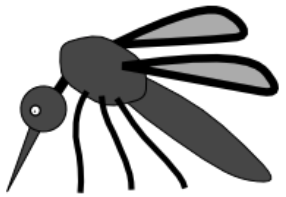
userdatauser
defined data of any type that is passed as the userdata parameter to callbacks. It may be updated at a later point with the user_data_set() function.

Protocol
the version of the MQTT protocol to use for this client. Can be either MQTTv31 or MQTTv311

Example
import paho.mqtt.client as mqtt

mqttclient = mqtt.Client()

Paho: Connection management



The connect() function connects the client to a broker. This is a blocking function. It takes the following arguments:

connect(host, port=1883, keepalive=60, bind_address="")

host
the hostname or IP address of the remote broker

port
the network port of the server host to connect to. Defaults to 1883.

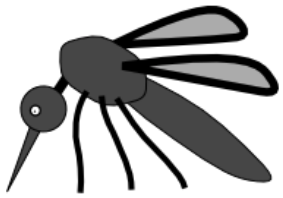
keepalive
maximum period in seconds allowed between communications with the broker. If no other messages are being exchanged, this controls the rate at which the client will send ping messages to the broker

bind_address
the IP address of a local network interface to bind this client to, assuming multiple interfaces exist

Callback
When the client receives a CONNACK message from the broker in response to the connect it generates an on_connect() callback.

Example
mqttclient.connect("iot.eclipse.org")

Paho: Connection management



The reconnect() function reconnects to a broker using the previously provided details. **You must have called connect() before calling this function.**

Callback

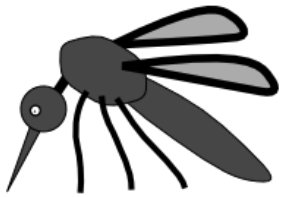
When the client receives a CONNACK message from the broker in response to the connect it generates an on_connect() callback.

The disconnect() function disconnects from the broker cleanly.

Callback

When the client has sent the disconnect message it generates an on_disconnect() callback.

Paho: Network Loop



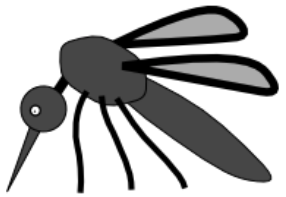
Loop functions are the driving force behind the client.

If they are not called, incoming network data will not be processed and outgoing network data may not be sent in a timely fashion. There are four options for managing the network loop:

- loop (timeout=1.0, max_packets=1)
- loop_start ()
- loop_stop ()
- loop_forever ()

Do not mix the different loop functions.

Paho: Network Loop



loop (timeout=1.0, max_packets=1)

Call regularly to process network events. This call waits until the network socket is available for reading or writing, then handles the incoming/outgoing data.

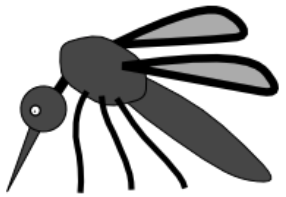
This function blocks for up to **timeout** seconds. **timeout** must not exceed the **keepalive** value for the client or your client will be regularly disconnected by the broker.

The **max_packets** argument is obsolete and should be left unset.

Example:

```
run = True
while run:
    mqttclient.loop()
```

Paho: Network Loop



loop_start ()

loop_stop(force=False)

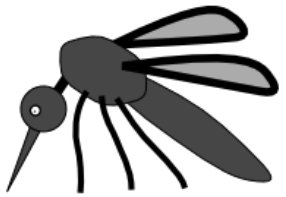
These functions implement a threaded interface to the network loop. Calling loop_start() once, before or after connect(), runs a thread in the background to call loop() automatically. This frees up the main thread for other work that may be blocking. This call also handles reconnecting to the broker.

Call loop_stop() to stop the background thread. The force argument is currently ignored.

Example

```
mqttc.connect("iot.eclipse.org")
mqttc.loop_start()
while True:
    temperature = ‘{“measurement” : “Temp”, “value”: 27.3}’
    mqttclient.publish("paho/temperature", temperature)
```

Paho: Network Loop



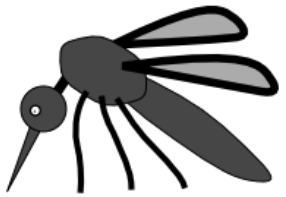
loop_forever (timeout=1.0, max_packets=1, retry_first_connection=False)

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

Warning: This might lead to situations where the client keeps connecting to an non existing host without failing.

The **timeout** and **max_packets** arguments are obsolete and should be left unset.

Paho: Publish



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

This causes a message to be sent to the broker and subsequently from the broker to any clients subscribing to matching topics. It takes the following arguments:

topic

the topic that the message should be published on

payload

the actual message to send. If not given, or set to None a zero length message will be used. Passing an *int* or *float* will result in the payload being converted to a string representing that number. If you wish to send a true *int* or *float*, use struct.pack() to create the payload

(In this course, the payload will be a JSON)

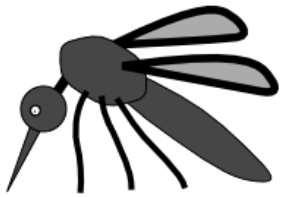
qos

the quality of service level to use

retain

if set to True, the message will be set as the “last known good”/retained message for the topic.

Paho: Publish



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

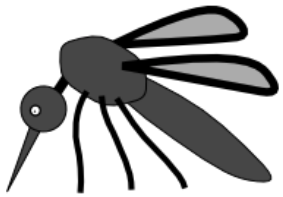
Returns a tuple *(result, mid)*, where result is **MQTT_ERR_SUCCESS** to indicate success or **MQTT_ERR_NO_CONN** if the client is not currently connected. **mid** is the message ID for the publish request.

A **ValueError Exception** will be raised if topic is None, has zero length or is invalid (i.e. contains a wildcard), if qos is not one of 0, 1 or 2, or if the length of the payload is greater than 268435455 bytes.

Callback

When the message has been sent to the broker an **on_publish()** callback will be generated.

Paho: Subscribe



subscribe(topic, qos=0)

Subscribe the client to one or more topics. This function may be called in three different ways:

- **Simple string and integer**

  e.g. subscribe("my/topic", 2)

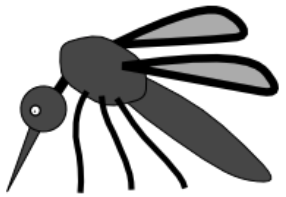
  topic

  a string specifying the subscription topic to subscribe to.

  qos

  the desired quality of service level for the subscription. Defaults to 0.

Paho: Subscribe



- **String and integer tuple**

  e.g. subscribe(("my/topic", 1))

  topic

  a tuple of (topic, qos). Both topic and qos must be present in the tuple.

- **List of string and integer tuples**

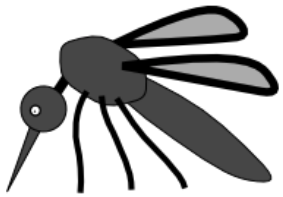
  e.g. subscribe([("my/topic", 0), ("another/topic", 2)])

  This allows multiple topic subscriptions in a single SUBSCRIPTION command, which is more efficient than using multiple calls to subscribe().

  topic

  a list of tuple of format (topic, qos). Both topic and qos must be present in all of the tuples.

Paho: Subscribe



subscribe(topic, qos=0)

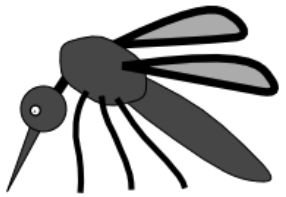
The function returns a tuple (result, mid), where result is **MQTT_ERR_SUCCESS** to indicate success or (**MQTT_ERR_NO_CONN**, None) if the client is not currently connected.
mid is the message ID for the subscribe request.

ValueError Exception will be raised if qos is not 0, 1 or 2, or if topic is None or has zero string length, or if topic is not a string, tuple or list.

Callback

When the broker has acknowledged the subscription, an **on_subscribe()** callback will be generated.

Paho: Unsubscribe



unsubscribe (topic)

Unsubscribe the client from one or more topics.

topic

a single string, or list of strings that are the subscription topics to unsubscribe from.

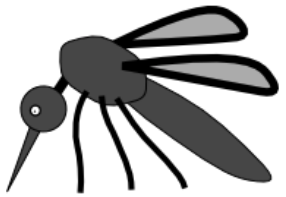
Returns a tuple (result, mid), where result is **MQTT_ERR_SUCCESS** to indicate success, or (**MQTT_ERR_NO_CONN**, None) if the client is not currently connected. mid is the message ID for the unsubscribe request.

ValueError Exception is raised if topic is None or has zero string length, or is not a string or list.

Callback

When the broker has acknowledged the unsubscribe, an **on_unsubscribe()** callback will be generated.

Paho: Callbacks

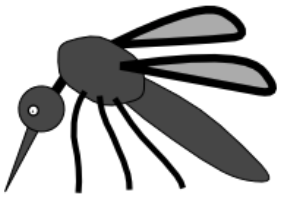


Complete list of callbacks for paho.mqtt.client:

- on_connect(client, userdata, flags, rc)
- on_disconnect(client, userdata, rc)
- on_message(client, userdata, message)
- message_callback_add(sub, callback)
- message_callback_remove(sub)
- on_publish(client, userdata, mid)
- on_subscribe(client, userdata, mid, granted_qos)
- on_unsubscribe(client, userdata, mid)
- on_log(client, userdata, level, buf)

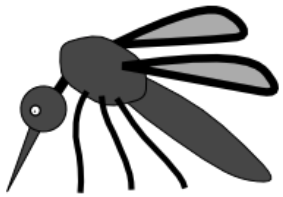
More details on: https://pypi.python.org/pypi/paho-mqtt#callbacks

Paho: Subscriber example



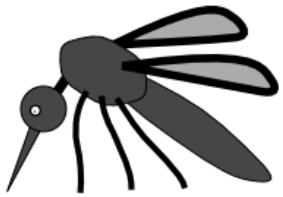
Look at simpleSubscriber.py
in mqtt_examples.zip

Paho: Publisher example



Look at simplePublisher.py in mqtt_examples.zip

Paho: example



In mqtt_examples.zip look at:

- MyMQTT.py,
- DoSometing.py,
- publisher.py
- subscriber.py