Eclipse Mosquitto™ is an open source (EPL/EDL licensed) message broker that implements the [MQTT](http://mqtt.org/) protocol versions 3.1 and 3.1.1. MQTT provides a lightweight method of carrying out messaging using a publish/subscribe model.

This makes it suitable for "Internet of Things" messaging such as with low power sensors or mobile devices such as phones, embedded computers or microcontrollers like the Arduino.

If you want to do reliable message handling with Mosquitto, the short answer is: **You can't**.

limitations:

▪ Exchanging messages in this way is obviously slower than having no consistency checks in place.

▪ Since the Mosquitto broker only writes the in-memory database to disk every X (where X is configurable) seconds, you may lose data if the broker crashes.

▪ On the consumer side, it is the MQTT library that confirms the receipt of the message. However, as far as I can tell, there is no way to manually confirm the receipt of a message. So if your client crashes while handling a message, rather than while it is receiving a message, you may still lose the message. If you wish to handle this case, you can store the message on the client as soon as possible. This is, however, not much more reliable. The only other way is to implement some manual protocol via the exchange of messages where the original publisher retains a message and resends it unless its been acknowledged by the consumer.

How it works:

+----------+ +--------+ +----------+

| producer |---->| broker |---->| consumer |

+----------+ +--------+ +—————+

The producer sends messages to a topic on the broker.

The broker maintains an internal state of topics and which consumers are interested in which topics. It also maintains a queue of messages which still need to be sent to each consumer.

How the broker decided what / when to send to which consumer depends on settings such as the QoS (Quality of Service) and what kind of session the consumer is opening.

Setting:

install by home brew

**Producer and consumer settings**

When creating a consumer or producer, ensure you set these settings properly:

**▪ quality-of-service** must be 2.

▪ The consumer must send a **client\_id**.

**▪ clean\_session** on the consumer must be False.

These are the base requirements to ensure that each consumer will receive messages exactly once, even if they've been offline for a while. The **quality-of-service** setting of 2 ensures that the broker requires acknowledgement from the consumer that a message has been received properly. Only then does the broker update its internal state to advance the consumer to the next message in the queue. If the client crashes before acknowledging the message, it'll be resent the next time.

The **client\_id** gives the broker a unique name under which to store session state information such as the last message the consumer has properly acknowledged. Without a client\_id, the broker cannot do this.

The **clean\_session** setting lets the consumer inform the broker about whether it wants its session state remembered. Without it, the broker assumes the broker assumes the consumer does not care about past messages and such. It will only receive any new messages that are produced after the consumer has connected to the broker.

Example:

import paho.mqtt.client as paho

import time

client = paho.Client(protocol=paho.MQTTv31)

client.connect("localhost", 1883)

client.loop\_start()

client.publish("mytesttopic", str("foo"), qos=2)

time.sleep(1) # Give the client loop time to proess the message

**Broker settings**

The following settings are relevant configuration options on the broker. You can generally find these settings in/etc/mosquitto/mosquitto.conf.

▪ The broker must have **persistence** set to True in the broker configuration.

▪ You may want to set **max\_inflight\_messages** to 1 in the broker configuration to ensure correct ordering of messages.

▪ Configure **max\_queued\_messsages** to the maximum number of messages to retain in a queue.

▪ Tweak **autosave\_interval** to how often you want the broker to write the in-memory database to disk.

The **persistence** setting informs the broker that you'd like session state and message queues written to disk. If the broker for some reason, the messages will (mostly) still be there.

You can ensure that messages are sent to consumers in the same order as they were sent to the broker by the producers by setting the **max\_inflight\_messages** setting to 1. This will probably severely limit the throughput speed of messages.

The **max\_queued\_messsages** determines how many unconfirmed messages should maximally be retained in queues. This should basically be the product of the maximum number of messages per second and the maximum time a consumer might be offline. Say we're processing 1 message per second and we want the consumer to be able to be offline for 2 hours (= 7200 seconds), then the max\_queued\_messsages setting should be 1 \* 7200 = 7200.

The **autosave\_interval** determines how often you want the broker to write the in-memory database to disk. I suspect that setting this to a very low level will cause severe Disk I/O activity.

Example:

import paho.mqtt.client as paho

def on\_message(client, userdata, msg):

print(msg.topic+" "+str(msg.qos)+" "+str(msg.payload))

client = paho.Client("testclient", clean\_session=False, protocol=paho.MQTTv31)

client.on\_message = on\_message

client.connect("localhost", 1883)

client.subscribe("mytesttopic", qos=2)

client.loop\_forever()

Things need to notice:

▪ If the broker or one of the clients doesn't support the MQTTv32 protocol, things will fail silently. So I specify MQTTv31 manually.

▪ The client loop needs some time to process the sending and receiving of messages. If you send a single message and exit your program right away, the loop doesn't have time to actually send the message.

▪ The subscriber must have already run once before the broker will start keeping messages for it. Otherwise, the broker has no idea that a consumer with QoS=2 is interested in messages (and would have to keep messages for ever). So register your consumer once by just running it, before the producer runs.