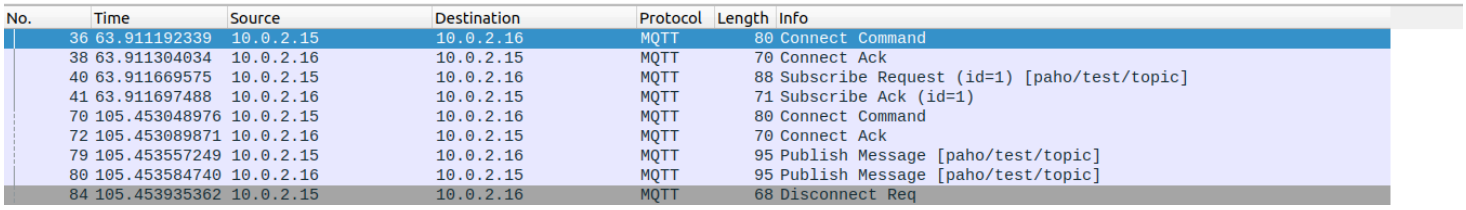
Victor MAYAUD

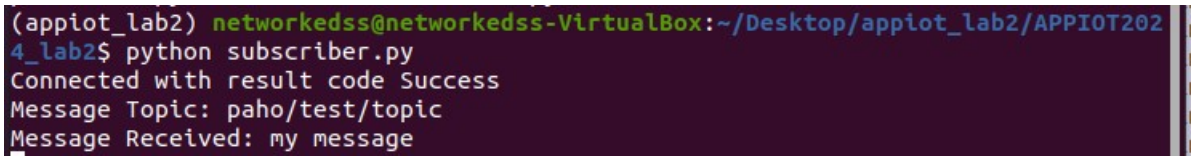
APPIOT
LAB 2 : MQTT

1/ Run the subscriber.py and publisher.py scripts and Analyze trafic with wireshark.

Screenshots:



No.	Time	Source	Destination	Protocol	Length	Info
36	63.911192339	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
38	63.911304034	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
40	63.911669575	10.0.2.15	10.0.2.16	MQTT	88	Subscribe Request (id=1) [paho/test/topic]
41	63.911697488	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
70	105.453048976	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
72	105.453089871	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
79	105.453557249	10.0.2.15	10.0.2.16	MQTT	95	Publish Message [paho/test/topic]
80	105.453584740	10.0.2.16	10.0.2.15	MQTT	95	Publish Message [paho/test/topic]
84	105.453935362	10.0.2.15	10.0.2.16	MQTT	68	Disconnect Req



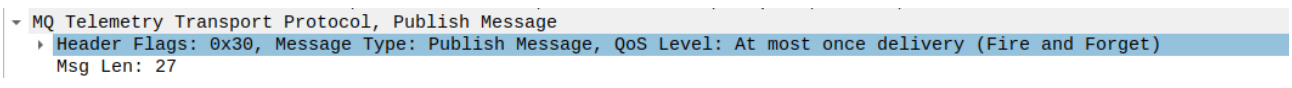
```
(appiot_lab2) networkedss@networkedss-VirtualBox:~/Desktop/appiot_lab2/APPIOT202
4_lab2$ python subscriber.py
Connected with result code Success
Message Topic: paho/test/topic
Message Received: my message
```

Here are steps of what happened between the exchange:

1. **Connect**: A client connects to the broker.
2. **Connect Ack**: The broker acknowledges the connection.
3. **Subscribe Request**: A client subscribes to a topic.
4. **Subscribe Ack**: The broker acknowledges the subscription.
5. **Connect**: A client connects to the broker.
6. **Connect Ack**: The broker acknowledges the connection.
7. **Publish Message**: A client publishes a message on a topic to which it is subscribed.
8. **Disconnect** : Disconnection of the client.

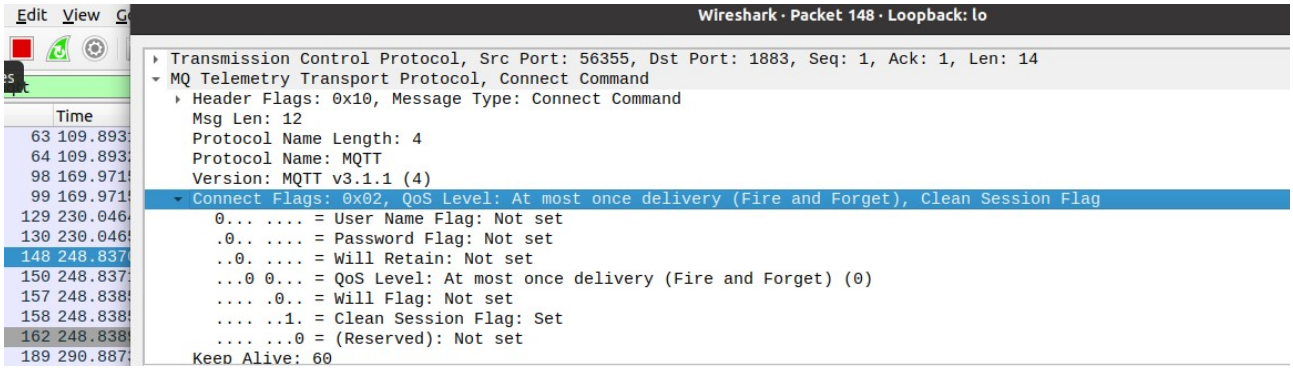
In the scenario depicted, the MQTT client is both publishing to and subscribed to the same topic, which is why we see two publish messages: one for the original publish action and another for the message being received by the client. This round-trip of messages occurs because MQTT brokers are designed to distribute messages to all subscribers of a topic, including the sender if it's subscribed to that topic.

The QoS level for both are QoS 0, we can see that in the description of the packet:



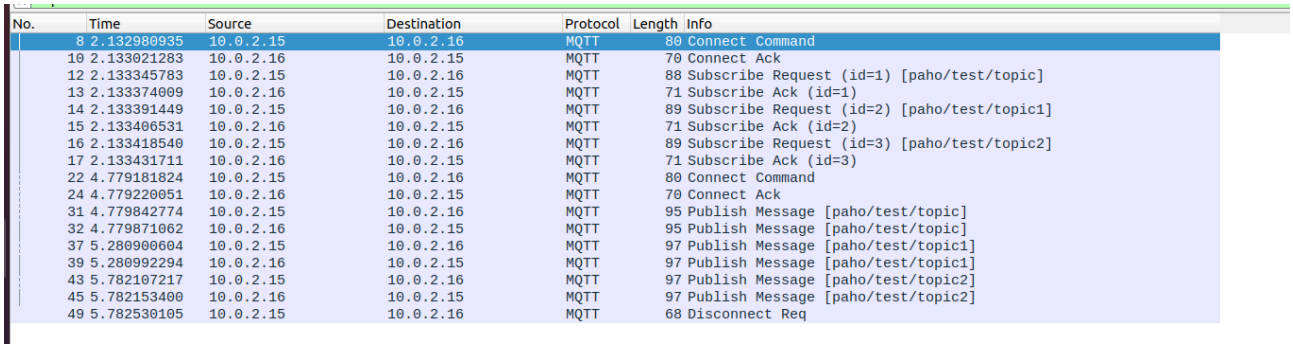
```
▾ MQ Telemetry Transport Protocol, Publish Message
  ▸ Header Flags: 0x30, Message Type: Publish Message, QoS Level: At most once delivery (Fire and Forget)
    Msg Len: 27
```

Since the Clean Session flag is set to 1 in the connection packet, we can determine that the connection is utilizing a 'clean session :



2/ Subscribing to topics and Receiving messages

I've created new topics named 'topic1' and 'topic2', and I've modified the code in 'publisher.py' to introduce a delay between publishing messages. Since the client is subscribed to all these topics, it receives published messages for each one, resulting in two 'Publish' packets for every topic. Multiple 'Connect' packets are observed because the client establishes separate connections for each topic.



No.	Time	Source	Destination	Protocol	Length	Info
8	2.132980935	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
10	2.133021283	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
12	2.133345783	10.0.2.15	10.0.2.16	MQTT	88	Subscribe Request (id=1) [paho/test/topic]
13	2.133374009	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
14	2.133391449	10.0.2.15	10.0.2.16	MQTT	89	Subscribe Request (id=2) [paho/test/topic1]
15	2.133406531	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=2)
16	2.133418540	10.0.2.15	10.0.2.16	MQTT	89	Subscribe Request (id=3) [paho/test/topic2]
17	2.133431711	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=3)
22	4.779181824	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
24	4.779220051	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
31	4.779842774	10.0.2.15	10.0.2.16	MQTT	95	Publish Message [paho/test/topic]
32	4.779871062	10.0.2.16	10.0.2.15	MQTT	95	Publish Message [paho/test/topic]
37	5.280900604	10.0.2.15	10.0.2.16	MQTT	97	Publish Message [paho/test/topic1]
39	5.280992294	10.0.2.16	10.0.2.15	MQTT	97	Publish Message [paho/test/topic1]
43	5.782107217	10.0.2.15	10.0.2.16	MQTT	97	Publish Message [paho/test/topic2]
45	5.782153400	10.0.2.16	10.0.2.15	MQTT	97	Publish Message [paho/test/topic2]
49	5.782530105	10.0.2.15	10.0.2.16	MQTT	68	Disconnect Req

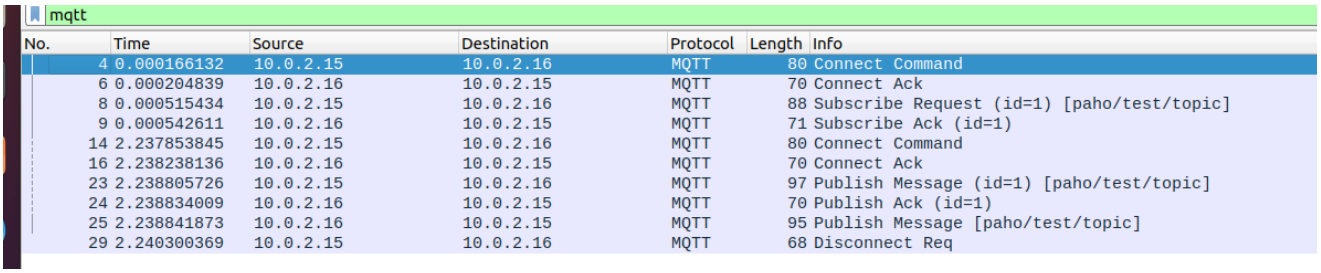
```
12
13 # Our application produce some messages
14 msg_info = mqttc.publish("paho/test/topic", "my message")
15 #we wait after each messages requested.
16 msg_info.wait_for_publish()
17 time.sleep(0.5)
18
19 msg_info = mqttc.publish("paho/test/topic1", "my message1")
20 msg_info.wait_for_publish()
21 time.sleep(0.5)
22
23 msg_info = mqttc.publish("paho/test/topic2", "my message2")
24 msg_info.wait_for_publish()
25
```

We also add the new topics in subscriber.py:

```
4         # our subscribed is persisted across reconnections
5         print("Connected with result code "+str(reason_co
6         client.subscribe("paho/test/topic")
7         client.subscribe("paho/test/topic1")
8         client.subscribe("paho/test/topic2")
9
```

3/ Quality of service.

Set the QoS to 1:

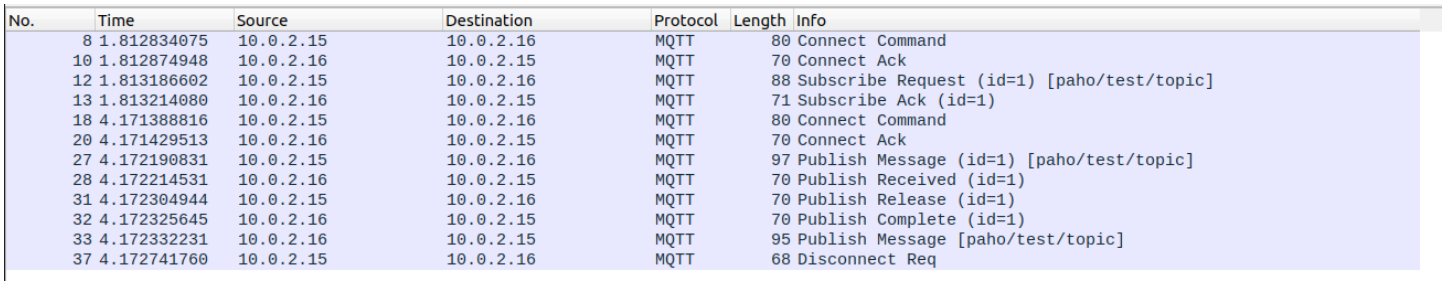


mqtt

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000166132	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
6	0.000204839	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
8	0.000515434	10.0.2.15	10.0.2.16	MQTT	88	Subscribe Request (id=1) [paho/test/topic]
9	0.000542611	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
14	2.237853845	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
16	2.238238136	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
23	2.238805726	10.0.2.15	10.0.2.16	MQTT	97	Publish Message (id=1) [paho/test/topic]
24	2.238834009	10.0.2.16	10.0.2.15	MQTT	70	Publish Ack (id=1)
25	2.238841873	10.0.2.16	10.0.2.15	MQTT	95	Publish Message [paho/test/topic]
29	2.240300369	10.0.2.15	10.0.2.16	MQTT	68	Disconnect Req

When the client publish something the broker answer with a publish ack.

Set the QoS to 2:



No.	Time	Source	Destination	Protocol	Length	Info
8	1.812834075	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
10	1.812874948	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
12	1.813186602	10.0.2.15	10.0.2.16	MQTT	88	Subscribe Request (id=1) [paho/test/topic]
13	1.813214080	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
18	4.171388816	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
20	4.171429513	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
27	4.172190831	10.0.2.15	10.0.2.16	MQTT	97	Publish Message (id=1) [paho/test/topic]
28	4.172214531	10.0.2.16	10.0.2.15	MQTT	70	Publish Received (id=1)
31	4.172304944	10.0.2.15	10.0.2.16	MQTT	70	Publish Release (id=1)
32	4.172325645	10.0.2.16	10.0.2.15	MQTT	70	Publish Complete (id=1)
33	4.172332231	10.0.2.16	10.0.2.15	MQTT	95	Publish Message [paho/test/topic]
37	4.172741760	10.0.2.15	10.0.2.16	MQTT	68	Disconnect Req

Here is the new steps in QoS2 :

1. **Publish Message**: The client publishes a message with QoS 2, which requires a four-step handshake to ensure the message is received exactly once.
2. **Publish Received (Pubrec)**: The broker acknowledges it has received the publish message and will process it.
3. **Publish Release (Pubrel)**: The client responds to the Pubrec, indicating that the broker can release or forward the message to any subscribers.
4. **Publish Complete (Pubcomp)**: The broker sends this message to confirm that the message was released to the subscribers, completing the QoS 2 protocol exchange.

In QoS 0, messages are published without any acknowledgment from the receiver, essentially a 'fire-and-forget' approach. In QoS 1, each published message is followed by an acknowledgment (PUBACK) from the receiver to confirm delivery at least once. Lastly, QoS 2 involves a four-step communication process to ensure each message is received exactly once, providing the highest level of reliability.

Higher QoS levels, increase message reliability, also introduce additional communication steps, resulting in more time-consuming exchanges. QoS 0 is the quickest with a single-step message delivery without any acknowledgment. QoS 1 adds an acknowledgment step, ensuring that messages are delivered at least once. QoS 2, the most reliable, requires a four-step process to guarantee that messages are delivered exactly once, significantly extending the communication time for each message.

4/ Define topic levels and play with wildcards (Single-level and Multi-level).

In MQTT, single-level wildcards (+) and multi-level wildcards (#) allow for flexible topic subscription.

For example, subscribing to test/+/temperature will match test/livingroom/temperature and test/kitchen/temperature, but not test2/livingroom/humidity.

Subscribing to test/# will match any topics starting with test/, including test/livingroom/temperature, test/kitchen/humidity, and even test/livingroom/light/intensity.

If we implement this in subscriber.py we have this:

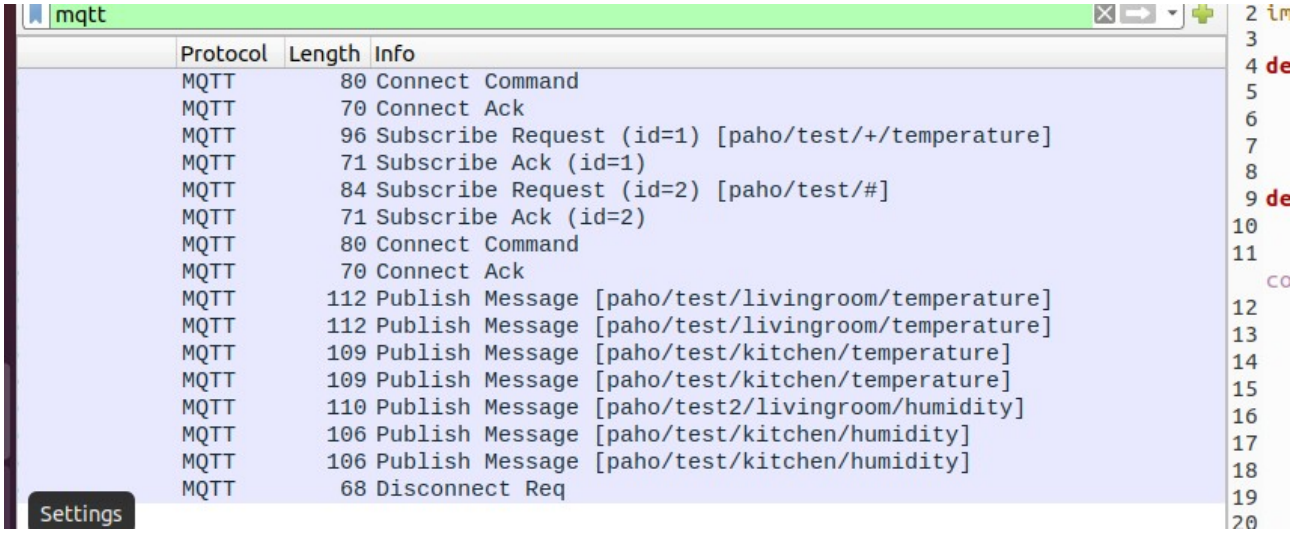
```python
#THIS IS THE SUBSCRIBER CODE
import paho.mqtt.client as mqtt

def on_message(client, userdata, message):

    print("Message Topic:", message.topic)
    print("Message Received:", message.payload.decode())

def on_connect(client, userdata, flags, reason_code, properties):
    if reason_code.is_failure:
        print(f"Failed to connect: {reason_code}. loop_forever() will retry connection")
    else:
        # we should always subscribe from on_connect callback to be sure
        # our subscribed is persisted across reconnections.
        print("Connected with result code "+str(reason_code))

        #wildcards
        client.subscribe("paho/test/+/temperature")
        client.subscribe("paho/test/#")

mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
mqttc.on_connect = on_connect
mqttc.on_message = on_message


mqttc.user_data_set([])
mqttc.connect('10.0.2.16') #HERE YOU SHOULD SPECIFY THE BROKER IP
mqttc.loop_forever()
print(f"Received the following message: {mqttc.user_data_get()}")
```

We have the two new subscriptions with the use of wildcards.

We can test the subscriptions with this code on publisher.py:

```
12
13 # Our application produce some messages
14 msg_info = mqttc.publish("paho/test/livingroom/temperature", "my message")
15 #we wait after each messages requested.
16 msg_info.wait_for_publish()
17 time.sleep(0.5)
18
19 msg_info = mqttc.publish("paho/test/kitchen/temperature", "my message")
20 msg_info.wait_for_publish()
21 time.sleep(0.5)
22
23 msg_info = mqttc.publish("paho/test2/livingroom/humidity", "my message")
24 msg_info.wait_for_publish()
25 time.sleep(0.5)
26
27 msg_info = mqttc.publish("paho/test/kitchen/humidity", "my message")
28 msg_info.wait_for_publish()
29 time.sleep(0.5)
```

In wireshark we obtain this:



Initially, we capture the packets related to connection establishment and topic subscriptions. Following that, we focus on the message publishing phase. The client publishes messages to various topics, and the broker dispatches these messages back to the client only for the topics to which it has subscribed. The third message is not relayed back to the client, indicating that the client is not subscribed to that particular topic.

5/Exploring message retention in MQTT:

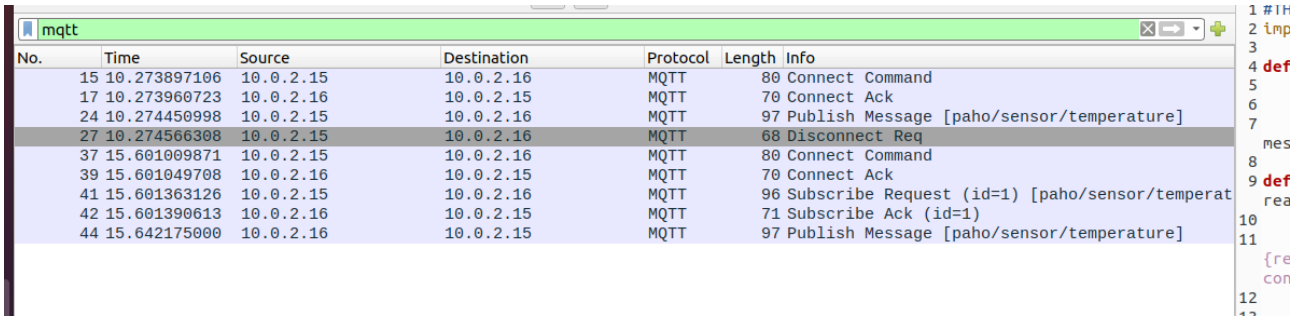
In order to create message retention, I implemented this in the code publisher.py:

```
12
13 # Our application produce some messages
14 msg_info = mqttc.publish("paho/sensor/temperature", "20°",0,True)
15 #we wait after each messages requested.
16 msg_info.wait_for_publish()
17
```

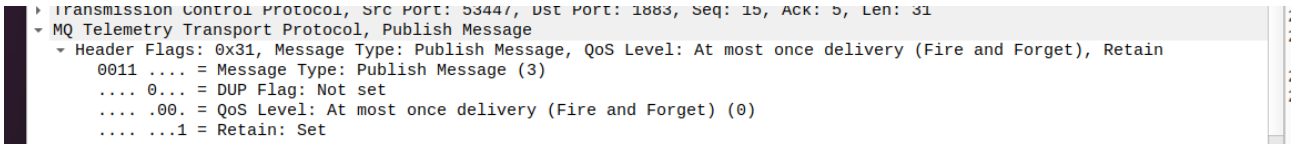
We publish in the topic “paho/sensor/temperature” with QoS=0 and the input True means that it’s a retention message. Then on the code subscriber.py I add the subscription to the topic:

```
#wildcards
client.subscribe("paho/sensor/temperature")
```

Then I recorded the packet with wireshark in taking care of launching the publisher before launching the subscriber and I obtained this:



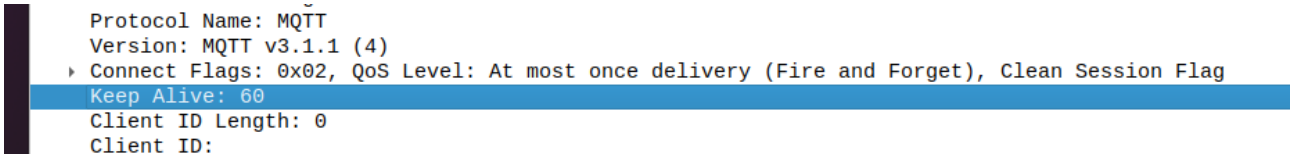
Initially, the client establishes a connection, publishes a message with the retention flag set, and then disconnects. Starting from the fifth packet, we executed subscribe.py. The connection is successfully made, and the client sends a request to subscribe to the correct topic. After the subscription is confirmed, the broker immediately sends the retained message to the client. This immediate delivery occurs because the message was published with the retain flag, meaning the client does not need to wait for a new message to be published in order to receive the last state of that topic. In the publish message packet we can see the flag retain set to 1:



So it’s a retention message.

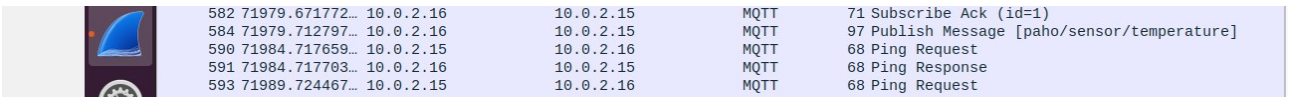
6/Understanding the MQTT Keepalive Mechanism:

Keepalive timer value :



The value is equal to 60 by default.

When the keepalive time interval set by the MQTT client expires without any other communication from the client, the client is expected to send a PINGREQ (Ping Request) to the broker. The broker, upon receiving the PINGREQ, will reply with a PINGRESP (Ping Response). This exchange of ping messages ensures both parties that the connection is still active. If the client fails to send a PINGREQ or the broker doesn't respond with a PINGRESP, the client should assume the connection has been lost and may attempt to reconnect.



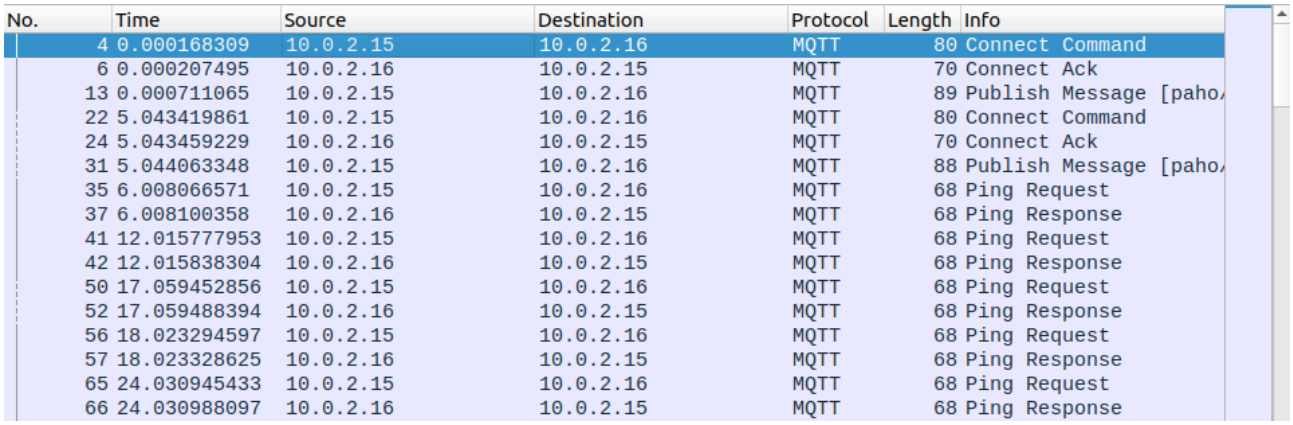
If we change the keepalive value by 5 second the client is expected to send a PINGREQ every 5 seconds.

I have crafted two separate publisher scripts that perform different types of publication, each with distinct keepalive intervals. To observe the impact of the keepalive setting, it is crucial to prevent the publishers from disconnecting immediately after sending their messages. This approach allows us to see how the keepalive mechanism maintains the connection when no further communication occurs. Below is the code for one of the publishers (publisher_part6.py):

publisher_part6.py | subscriber_part6.py | publisher2_part6.py

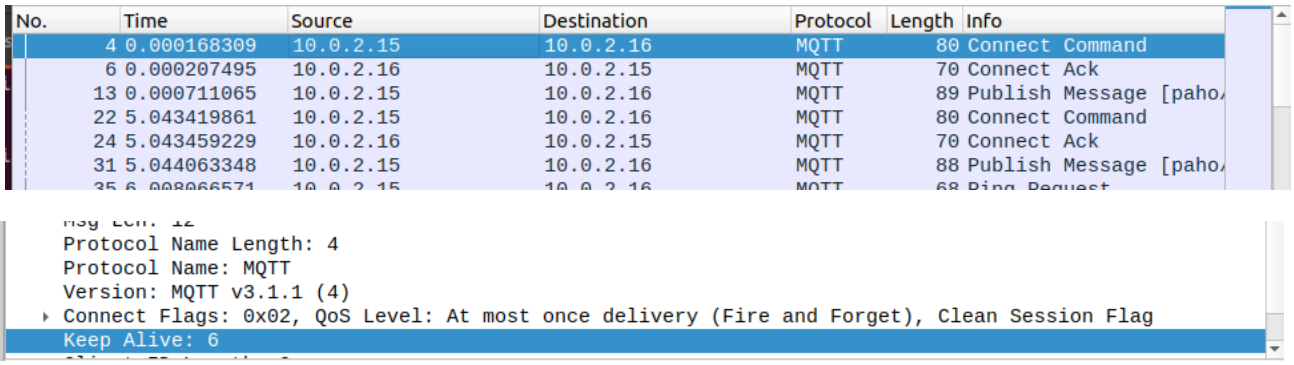
```python
#THIS IS THE PUBLISHER CODE
import time
import paho.mqtt.client as mqtt

unacked_publish = set()

mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)

mqttc.connect('10.0.2.16',keepalive=6) #HERE YOU SHOULD SPECIFY THE BROKER IP
mqttc.loop_start()


# Our application produce some messages
msg_info = mqttc.publish("paho/test", "my message",0)
#we wait after each messages requested.
msg_info.wait_for_publish()


# Wait for all message to be published
while len(unacked_publish):
    time.sleep(0.1)

# Due to race-condition described above, the following way to wait for all publish is safer
msg_info.wait_for_publish()

#mqttc.disconnect() #need to delete it in order to see the imact of keepalive value
mqttc.loop_forever()
```

I captured this with wireshark:



No.	Time	Source	Destination	Protocol	Length	Info
4	0.000168309	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
6	0.000207495	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
13	0.000711065	10.0.2.15	10.0.2.16	MQTT	89	Publish Message [paho/
22	5.043419861	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
24	5.043459229	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
31	5.044063348	10.0.2.15	10.0.2.16	MQTT	88	Publish Message [paho/
35	6.008066571	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
37	6.008100358	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
41	12.015777953	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
42	12.015838304	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
50	17.059452856	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
52	17.059488394	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
56	18.023294597	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
57	18.023328625	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
65	24.030945433	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
66	24.030988097	10.0.2.16	10.0.2.15	MQTT	68	Ping Response

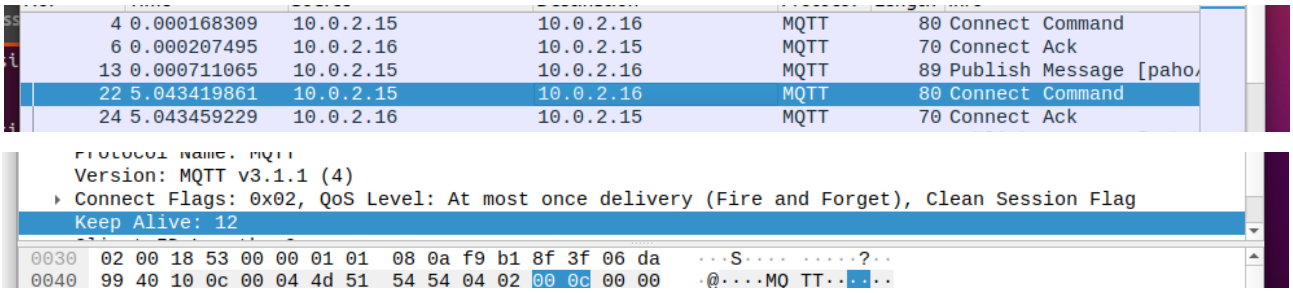
the first publisher has a keep alive value equals to 6 and it was launch at 0s:



No.	Time	Source	Destination	Protocol	Length	Info
4	0.000168309	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
6	0.000207495	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
13	0.000711065	10.0.2.15	10.0.2.16	MQTT	89	Publish Message [paho/
22	5.043419861	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
24	5.043459229	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
31	5.044063348	10.0.2.15	10.0.2.16	MQTT	88	Publish Message [paho/

```
Protocol Name Length: 4
Protocol Name: MQTT
Version: MQTT v3.1.1 (4)
▸ Connect Flags: 0x02, QoS Level: At most once delivery (Fire and Forget), Clean Session Flag
Keep Alive: 6
```

The second second publisher has a keep alive value equals to 12 and it was launch at 5s:



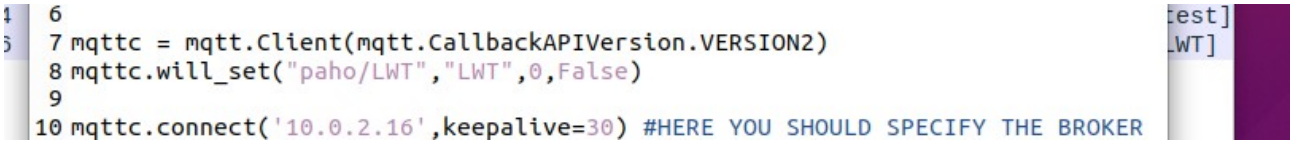
4	0.000168309	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
6	0.000207495	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
13	0.000711065	10.0.2.15	10.0.2.16	MQTT	89	Publish Message [paho/
22	5.043419861	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
24	5.043459229	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack

```
Version: MQTT v3.1.1 (4)
▸ Connect Flags: 0x02, QoS Level: At most once delivery (Fire and Forget), Clean Session Flag
Keep Alive: 12
0030  02 00 18 53 00 00 01 01  08 0a f9 b1 8f 3f 06 da   ···S···· ·····?··
0040  99 40 10 0c 00 04 4d 51  54 54 04 02 00 0c 00 00   ·@····MQ TT······
```

So at 6, 12, 18 and 24 seconds we have the Ping request from the first publisher and at 17s we have the ping request of the second publisher.

7. Exploring MQTT's last will and testament (LWT) :

In the publisher code I changed the code in order to put a last will and testament. I add this line of code :

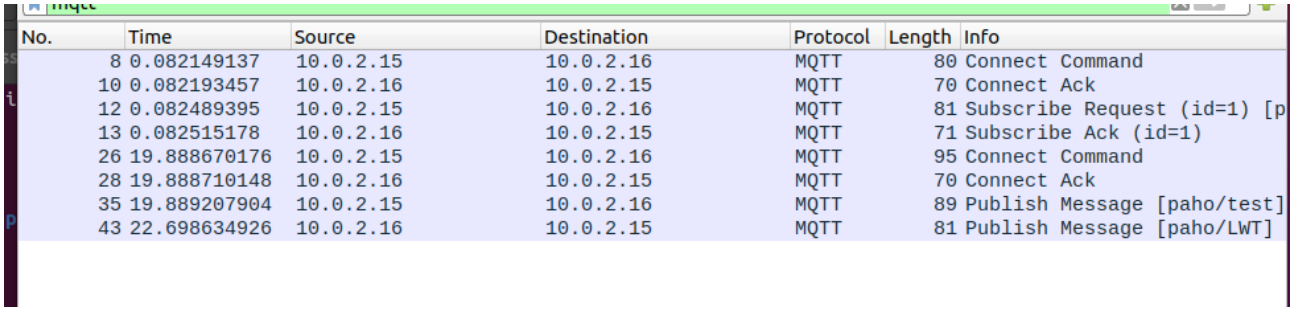


```
6
7 mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
8 mqttc.will_set("paho/LWT","LWT",0,False)
9
10 mqttc.connect('10.0.2.16',keepalive=30) #HERE YOU SHOULD SPECIFY THE BROKER
```

and in the subscriber code I subscribed to this topic:

```
#wildcards
client.subscribe("paho/LWT")
```

Finally I obtained this in wireshark when I launched the subscriber first and then the publisher:



No.	Time	Source	Destination	Protocol	Length	Info
8	0.082149137	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
10	0.082193457	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
12	0.082489395	10.0.2.15	10.0.2.16	MQTT	81	Subscribe Request (id=1) [p
13	0.082515178	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
26	19.888670176	10.0.2.15	10.0.2.16	MQTT	95	Connect Command
28	19.888710148	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
35	19.889207904	10.0.2.15	10.0.2.16	MQTT	89	Publish Message [paho/test]
43	22.698634926	10.0.2.16	10.0.2.15	MQTT	81	Publish Message [paho/LWT]

The initial four packets depict the activities of the subscriber followed by those of the publisher. Between the 19th and 22nd-second marks, I halted the publisher, which led to the transmission of the last will and testament message by the broker. This message is represented by the final 'Publish' packet observed. The subscriber successfully received this message as intended.

```
(appiot_lab2) networkedss@networkedss-VirtualBox:~/Desktop/appiot_lab2/APPIOT202
4_lab2$ python subscriber_part7.py
Connected with result code Success
Message Topic: paho/LWT
Message Received: LWT
```

8/ Effects of clean session flag and QoS on message delivery:

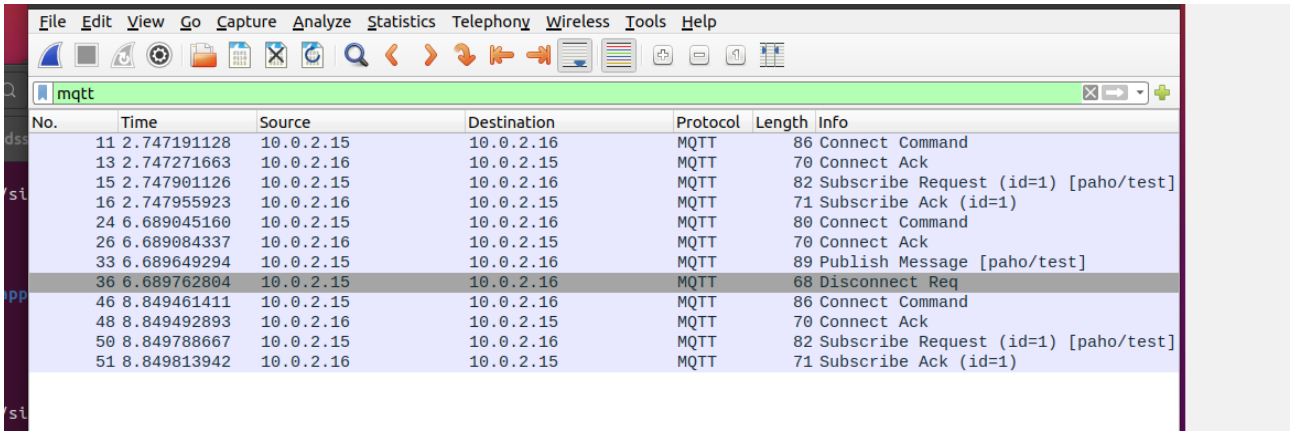
In order to make a persistent session I add this line of code in subscriber.py (subscriber_part8.py):

```
19
20 mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2,client_id="unique",clean_session=False)
```

and I put QoS equals to 1.

Then I adapted the code of the publisher in order to publish in the same topic as the subscriber and I adapted the QoS for the experience.

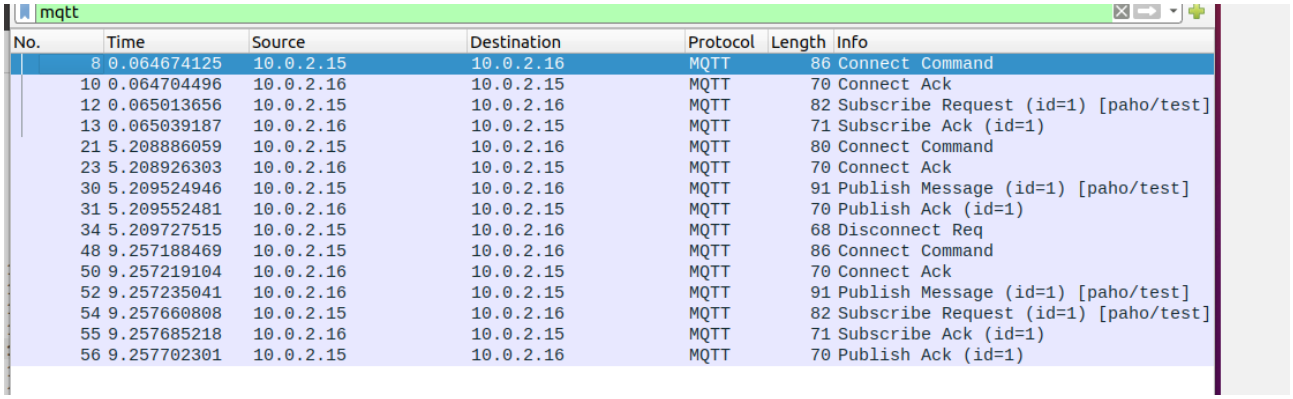
First scenario (QoS = 0):



No.	Time	Source	Destination	Protocol	Length	Info
11	2.747191128	10.0.2.15	10.0.2.16	MQTT	86	Connect Command
13	2.747271663	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
15	2.747901126	10.0.2.15	10.0.2.16	MQTT	82	Subscribe Request (id=1) [paho/test]
16	2.747955923	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
24	6.689045160	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
26	6.689084337	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
33	6.689649294	10.0.2.15	10.0.2.16	MQTT	89	Publish Message [paho/test]
36	6.689762804	10.0.2.15	10.0.2.16	MQTT	68	Disconnect Req
46	8.849461411	10.0.2.15	10.0.2.16	MQTT	86	Connect Command
48	8.849492893	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
50	8.849788667	10.0.2.15	10.0.2.16	MQTT	82	Subscribe Request (id=1) [paho/test]
51	8.849813942	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)

In the first scenario with QoS set to 0, I initiated the subscriber's session and then launched the publisher. Since QoS 0 offers 'at most once delivery,' the message published during this period does not get queued for the subscriber if it is disconnected. As a result, when I stopped the subscriber at 5 seconds and subsequently initiated the publisher, the messages sent in the subscriber's absence were not received upon reconnection. This demonstrates that with QoS 0, messages are not retained for future delivery if the subscriber is not actively connected.

Second scenario (QoS = 1):



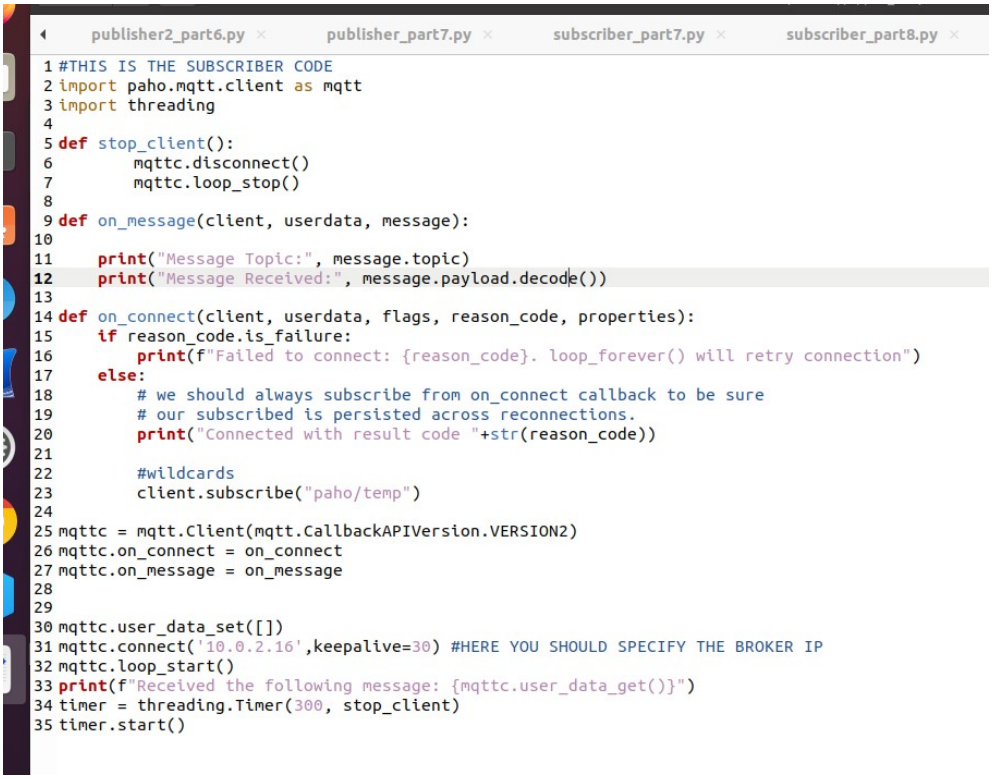
mqtt

No.	Time	Source	Destination	Protocol	Length	Info
8	0.064674125	10.0.2.15	10.0.2.16	MQTT	86	Connect Command
10	0.064704496	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
12	0.065013656	10.0.2.15	10.0.2.16	MQTT	82	Subscribe Request (id=1) [paho/test]
13	0.065039187	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
21	5.208886059	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
23	5.208926303	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
30	5.209524946	10.0.2.15	10.0.2.16	MQTT	91	Publish Message (id=1) [paho/test]
31	5.209552481	10.0.2.16	10.0.2.15	MQTT	70	Publish Ack (id=1)
34	5.209727515	10.0.2.15	10.0.2.16	MQTT	68	Disconnect Req
48	9.257188469	10.0.2.15	10.0.2.16	MQTT	86	Connect Command
50	9.257219104	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
52	9.257235041	10.0.2.16	10.0.2.15	MQTT	91	Publish Message (id=1) [paho/test]
54	9.257660808	10.0.2.15	10.0.2.16	MQTT	82	Subscribe Request (id=1) [paho/test]
55	9.257685218	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
56	9.257702301	10.0.2.15	10.0.2.16	MQTT	70	Publish Ack (id=1)

In this scenario, operating with QoS 1, upon reactivating the subscriber, it successfully received the message that was published while it was disconnected. This confirms that the persistent session feature is functioning correctly, as QoS 1 ensures message delivery at least once, retaining the message until the subscriber is able to receive it.

Second Part:

subcriber.py:



publisher2_part6.py | publisher_part7.py | subscriber_part7.py | subscriber_part8.py

```
#THIS IS THE SUBSCRIBER CODE
import paho.mqtt.client as mqtt
import threading

def stop_client():
        mqttc.disconnect()
        mqttc.loop_stop()

def on_message(client, userdata, message):

    print("Message Topic:", message.topic)
    print("Message Received:", message.payload.decode())

def on_connect(client, userdata, flags, reason_code, properties):
    if reason_code.is_failure:
        print(f"Failed to connect: {reason_code}. loop_forever() will retry connection")
    else:
        # we should always subscribe from on_connect callback to be sure
        # our subscribed is persisted across reconnections.
        print("Connected with result code "+str(reason_code))

        #wildcards
        client.subscribe("paho/temp")

mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
mqttc.on_connect = on_connect
mqttc.on_message = on_message


mqttc.user_data_set([])
mqttc.connect('10.0.2.16',keepalive=30) #HERE YOU SHOULD SPECIFY THE BROKER IP
mqttc.loop_start()
print(f"Received the following message: {mqttc.user_data_get()}")
timer = threading.Timer(300, stop_client)
timer.start()
```

This script sets up an MQTT client that subscribes to a specific topic and receives messages. It starts by importing the necessary modules and defining the stop_client function, which disconnects the client and stops the MQTT loop, effectively halting message reception. The on_message and on_connect callback functions handle incoming messages and successful connections, printing relevant information to the console. The MQTT client is configured with a specified keepalive interval and is started with loop_start() to begin non-blocking network traffic handling. To ensure the script runs for only 5 minutes, a threading timer is initiated, which schedules the stop_client function to run after 300 seconds (5 minutes). Once this timer expires, the client will disconnect and stop processing any further messages.

Publisher.py:

subscriber_part8.py | publisher_part8.py | publisher_secondPart.py | subscriber_secondPart.py

```
#THIS IS THE PUBLISHER CODE
import time
import paho.mqtt.client as mqtt
import random

unacked_publish = set()

mqttc = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)

mqttc.connect('10.0.2.16',keepalive=30) #HERE YOU SHOULD SPECIFY THE BROKER IP
mqttc.loop_start()


while True:
        temp = random.randint(10,30)
        msg_info = mqttc.publish("paho/temp", str(temp),0)
        msg_info.wait_for_publish()
        time.sleep(30)

```

I incorporated an infinite loop, within which a new temperature value ranging from 10° to 30° is published at 30-second intervals.

I obtained this on wireshark:

mqtt

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000188445	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
6	0.000232689	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
13	0.001039261	10.0.2.15	10.0.2.16	MQTT	82	Subscribe Request (id=1) [paho/temp]
14	0.001068860	10.0.2.16	10.0.2.15	MQTT	71	Subscribe Ack (id=1)
19	2.382977282	10.0.2.15	10.0.2.16	MQTT	80	Connect Command
21	2.383020101	10.0.2.16	10.0.2.15	MQTT	70	Connect Ack
28	2.383939474	10.0.2.15	10.0.2.16	MQTT	81	Publish Message [paho/temp]
29	2.383968738	10.0.2.16	10.0.2.15	MQTT	81	Publish Message [paho/temp]
50	30.427268980	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
51	30.427295045	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
55	32.412906722	10.0.2.15	10.0.2.16	MQTT	81	Publish Message [paho/temp]
57	32.412956599	10.0.2.16	10.0.2.15	MQTT	81	Publish Message [paho/temp]
61	32.413166221	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
63	32.413185279	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
79	60.457034522	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
80	60.457068221	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
88	62.427923943	10.0.2.15	10.0.2.16	MQTT	81	Publish Message [paho/temp]
89	62.427969624	10.0.2.16	10.0.2.15	MQTT	81	Publish Message [paho/temp]
94	62.470233555	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
96	62.470323622	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
112	90.468252543	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
113	90.468298133	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
117	92.437467172	10.0.2.15	10.0.2.16	MQTT	81	Publish Message [paho/temp]
118	92.437567489	10.0.2.16	10.0.2.15	MQTT	81	Publish Message [paho/temp]
123	92.480941969	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
125	92.480982010	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
145	120.481795818	10.0.2.15	10.0.2.16	MQTT	68	Ping Request
146	120.481831974	10.0.2.16	10.0.2.15	MQTT	68	Ping Response
150	122.454217518	10.0.2.15	10.0.2.16	MQTT	81	Publish Message [paho/temp]
151	122.454260127	10.0.2.16	10.0.2.15	MQTT	81	Publish Message [paho/temp]

…

```
247 212.52888962 10.0.2.16      10.0.2.15      MQTT    81 Publish Message [paho/temp]
252 212.572463573 10.0.2.15     10.0.2.16      MQTT    68 Ping Request
254 212.572505273 10.0.2.16     10.0.2.15      MQTT    68 Ping Response
274 240.570255190 10.0.2.15     10.0.2.16      MQTT    68 Ping Request
275 240.570286036 10.0.2.16     10.0.2.15      MQTT    68 Ping Response
279 242.547617130 10.0.2.15     10.0.2.16      MQTT    81 Publish Message [paho/temp]
280 242.547660938 10.0.2.16     10.0.2.15      MQTT    81 Publish Message [paho/temp]
285 242.588637369 10.0.2.15     10.0.2.16      MQTT    68 Ping Request
287 242.588723584 10.0.2.16     10.0.2.15      MQTT    68 Ping Response
303 270.602341720 10.0.2.15     10.0.2.16      MQTT    68 Ping Request
304 270.602375476 10.0.2.16     10.0.2.15      MQTT    68 Ping Response
312 272.565028660 10.0.2.15     10.0.2.16      MQTT    81 Publish Message [paho/temp]
313 272.565069398 10.0.2.16     10.0.2.15      MQTT    81 Publish Message [paho/temp]
318 272.608336909 10.0.2.15     10.0.2.16      MQTT    68 Ping Request
320 272.608373745 10.0.2.16     10.0.2.15      MQTT    68 Ping Response
336 300.001481812 10.0.2.15     10.0.2.16      MQTT    68 Disconnect Req
345 302.579426588 10.0.2.15     10.0.2.16      MQTT    81 Publish Message [paho/temp]
349 302.620517732 10.0.2.15     10.0.2.16      MQTT    68 Ping Request
351 302.620563771 10.0.2.16     10.0.2.15      MQTT    68 Ping Response
371 332.610201480 10.0.2.15     10.0.2.16      MQTT    81 Publish Message [paho/temp]
375 332.652337997 10.0.2.15     10.0.2.16      MQTT    68 Ping Request
377 332.652373353 10.0.2.16     10.0.2.15      MQTT    68 Ping Response
```

Both codes are operating correctly. The subscriber is initiated first, followed by the publisher. As a result, we see a ping check originating from both the subscriber and the publisher at 30-second intervals, accompanied by messages of published temperature readings. After 300seconds we can see that the subscriber make a disconnect requete.