MQTT Tutorial II

by chanmin.park (devcamp@gmail.com)

MQTT 응용

MQTT 응용 구현에 사용하는 MQTT API는 내부 소켓 인터페이스로 사용자를 대신해 브로커에 메시지를 발행하고 구독에 가입한 후 메시지가 배포될 때마다 이를 사용자에게 전달하므로 일반적인 소켓 프로그램보다 쉽습니다.

단방향 클라이언트

발행자와 구독자는 통신 방향에 따라 개별적으로 구현하거나 하나로 구현할 수 있습니다. 개별 클라이언트 구현은 단방향으로 메시지가 흐르므로 발행자가 메시지를 브로커에 게시하면 브로커는 이를 구독자에게 배포합니다. Soda에 탭제된 브로커는 익명 연결을 허용하므로 브로커에 연결하기 전에 `username_pw_set()`으로 사용자 ID와 비밀번호 설정하는 것은 생략합니다.

센서값 발행자

Cds 객체로 엣지에서 수집한 조도 값을 "soda/sensors/cds" 토픽에 발행하는 발행자 구현은 다음과 같습니다. 참고로 모든 콜백 함수의 첫 번째 매개변수는 항상 Client 객체입니다.

```
01: import paho.mqtt.client as mqtt
02: from pop import Cds
03: import time
04: import signal
05
06: client = mqtt.Client()
07: cds = Cds()
08
09: def __cds_publish(client):
10:     cds_value = cds.readAverage()
11:     client.publish("soda/sensors/cds", cds_value)
12:     time.sleep(1)
13:
14: def _signal_handler(signal, frame):
15:     client.disconnect()
16:
17: def _on_connect(client, userdata, flags, rc):
18:     if rc == 0:
19:         __cds_publish(client)
20:
21: def _on_publish(client, userdata, mid):
22:     __cds_publish(client)
23:
24: def main():
25:     signal.signal(signal.SIGINT, _signal_handler)
26:     client.on_connect = _on_connect
```

```
27:     client.on_publish = _on_publish
28:
29:     client.connect("192.168.10.2")
30:     client.loop_forever()
31
32: if __name__ == "__main__":
33:     main()
```

02: pop의 Cds 클래스 로드

04: signal 모듈 로드

06: MQTT Client 객체를 만들어 client에 대입

07: Cds 객체를 만들어 cds에 대입

09 ~ 12: __cds_publish()는 조도 값을 브로커에 발행

10: 측정한 조도 값을 cds_value에 대입

11: "soda/sensors/cds" 토픽에 cds_value 발행

12: 1초 동안 프로그램 대기

14 ~ 15: _signal_handler()는 키보드 인터럽트(<Ctrl>c) 시그널이 발생하면 호출되는 핸들러

15: 브로커와의 연결을 끊음

17 ~ 19: _on_connect()는 on_connect() 콜백

18 ~ 19: 연결에 성공하면 __cds_publis() 호출

21 ~ 22: _on_publish()는 on_publish() 콜백

22: __cds_publis() 호출

24 ~ 30: main()은 작업 초기화

25: 키보드 인터럽트 시그널이 발생하면 _signal_hander()를 호출하도록 설정

26: on_connect 콜백에 _on_connect() 대입

27: on_message 콜백에 _on_message() 대입

29: 엣지 A 브로커에 연결 요청

30: 네트워크 루프 실행

일반적으로 응용프로그램이 실행 중인 상태에서 <Ctrl>c 키를 누르면 INT 시그널(키보드 인터럽트)에 의해 프로그램이 강제 종료되지만, signal()을 이용하면 사용자 함수가 호출되도록 다시 정의할 수 있습니다. 현재 발행자는 INT 시그널이 발생하면 _signal_handler()를 실행하도록 재설정했으며 _signal_handler()에서는 disconnect()를 호출하므로 브로커와의 연결을 끊고 loop_forever()로 실행 중인 네트워크 루프를 탈출합니다.

만약 여러 장치에서 동시에 발행자를 실행하면 구독자는 발행자를 식별할 수 없지만 토픽을 다음과 같이 바꾸면 가능합니다.

- 엣지 A: "soda/sensors/cds/a"
- 엣지 B: "soda/sensors/cds/b"

센서값 구독자

다음은 브로커에서 "soda/sensors/cds" 토픽 구독하는 구독자 구현입니다.

```
01: import paho.mqtt.client as mqtt
02: import signal
03:
04: client = mqtt.Client()
05:
06: def _signal_handler(signal, frame):
07:     client.disconnect()
08:
09: def _on_connect(client, userdata, flags, rc):
10:     if rc == 0:
11:         client.subscribe("soda/sensors/cds")
12:
13: def _on_message(client, userdata, message):
14:     print(message.topic, int(message.payload))
15:
16: def main():
17:     signal.signal(signal.SIGINT, _signal_handler)
18:     client.on_connect = _on_connect
19:     client.on_message = _on_message
20:
21:     client.connect("192.168.10.2")
22:     client.loop_forever()
23:
24: if __name__ == "__main__":
25:     main()
```

10 ~ 11: 연결에 성공하면 "soda/sensors/cds" 구독

13 ~ 14: _on_message()는 on_message() 콜백

14: MQTTMessage 객체에서 페이로드를 int 타입으로 변환해 토픽과 함께 출력

19: on_message 콜백에 _on_message() 대입

만약 n개의 발행자가 토픽에 자신의 식별자를 포함한다면 구독자는 subscribe()로 토픽을 등록할 때 다음과 같이 대응하면 됩니다.

- 두 개의 토픽에 대해 각각 subscribe() 실행
  - subscribe("soda/sensors/cds/a")
  - subscribe("soda/sensors/cds/b")
- 두 개의 토픽을 QoS와 함께 리스트로 묶어 subscribe() 한 번 호출
  - subscribe([("soda/sensors/cds/a", 0), ("soda/sensors/cds/b", 0)])
- 단일 토픽 필터 적용
  - subscribe("soda/sensors/cds/+")

양방향 클라이언트

양방향 클라이언트는 메시지를 발행하면서 구독도 하는 클라이언트로 브로커에 A 토픽을 발생하면서 B토픽을 구독하기 때문에 메시지는 양방향으로 흐릅니다. 즉, 하나의 프로그램에 발행자와 구독자를 모두 구현합니다.

센서값 발행, 인터벌 구독 클라이언트

다음은 센서값 발행자를 수정해 "soda/sensors/interval" 토픽 구독을 추가한 것입니다. __cds_publish()에서 센서값을 발행한 후 전역 변수 interval만큼 대기하도록 수정하고, 메시지가 수신되면 이를 실수 타입으로 바꿔 전역 변수 interval에 대입하므로 수신된 값에 따라 대기하는 시간을 바꿀 수 있습니다.

```
01: import paho.mqtt.client as mqtt
02: from pop import Cds
03: import time
04: import signal
05:
06: interval = 1
07:
08: client = mqtt.Client()
09: cds = Cds()
10:
11: def __cds_publish(client):
12:     cds_value = cds.readAverage()
13:     client.publish("soda/sensors/cds", cds_value, 1)
14:     time.sleep(interval)
15:
16: def _signal_handler(signal, frame):
17:     client.disconnect()
18:
19: def _on_connect(client, userdata, flags, rc):
20:     if rc == 0:
21:         __cds_publish(client)
22:         client.subscribe("soda/sensors/interval")
23:
24: def _on_publish(client, userdata, mid):
25:     __cds_publish(client)
26:
27: def _on_message(client, userdata, message):
```

```
28:     global interval
29:     interval = float(message.payload)
30:
31: def main():
32:     signal.signal(signal.SIGINT, _signal_handler)
33:     client.on_connect = _on_connect
34:     client.on_publish = _on_publish
35:     client.on_message = _on_message
36:
37:     client.connect("192.168.10.2")
38:     client.loop_forever()
39:
40: if __name__ == "__main__":
41:     main()
```

06: 인터벌 기본값인 1초를 interval에 대입

13: on_message() 콜백을 함께 사용하므로 반드시 QoS > 0으로 메시지 발행

14: 전역 변수 interval만큼 대기

22: 연결이 성공하면 "soda/sensors/interval" 구독

28: 전역 변수 interval 쓰기 허용 설정

29: MQTTMessage 객체의 페이로드를 float 타입으로 변환해 전역 변수 interval에 대입

인터벌 발행, 센서값 구독 클라이언트

인터벌을 발행하고 센서값을 구현하는 클라이언트 구현은 다음과 같습니다. 참고로 쉘 환경에서 `input()`으로 발행할 인터벌 값을 입력 받으면 수신한 센서 값의 출력과 사용자 입력이 뒤섞이므로 별도의 처리가 필요합니다.

```
01: import paho.mqtt.client as mqtt
02: import signal
03: import sys
04:
05: client = mqtt.Client()
06:
07: def _signal_handler(signal, frame):
08:     sys.exit(0)
09:
10: def _on_connect(client, userdata, flags, rc):
11:     if rc == 0:
12:         client.subscribe("soda/sensors/cds")
13:
14: def _on_message(client, userdata, message):
15:     print(message.topic, int(message.payload))
```

```
16:
17: def main():
18:     signal.signal(signal.SIGINT, _signal_handler)
19:     client.on_connect = _on_connect
20:     client.on_message = _on_message
21:
22:     client.connect("192.168.10.2")
23:     client.loop_start()
24:
25:     while True:
26:         interval = input()
27:         client.publish("soda/sensors/interval", interval)
28:
29: if __name__ == "__main__":
30:     main()
```

03: sys 모듈 로드

08: 키보드 인터럽트 시그널이 발생하면 프로그램 종료

25 ~ 27: while 루프를 돌며 input()으로 읽어온 interval을 "soda/sensors/interval" 토픽에 발행

구독에 따른 발행

메시지 발행과 구독에 대한 처리는 비동기이기 때문에 발행 결과를 구독한 후 다시 발행할 때는 메시지 수신을 처리하는 on_message() 콜백에서 처리합니다.

```
01: client = mqtt.Client()
02:
03: def __on_publish_led_cmd(client):
04:     led_cmd = input()
05:     client.publish("soda/actions/leds", led_cmd)
06:
07: def _on_message(client, userdata, message):
08:     print(message.payload)
09:     __on_publish_led_cmd(client)
10:
11: client.on_message = _on_message
12:
13: client.connect("192.168.10.2")
14: client.subscribe("soda/actions/ack")
15: __on_publish_led_cmd(client)
16:
17: client.loop_forever()
```

LED 제어를 발행자로 구독자로 나눠 구현해 보겠습니다. 먼저 발행자는 "soda/actions/ack"를 구독하면서 표준 입력에서 LED 제어 명령을 읽어 "soda/action/leds"에 발행한 후 on_message() 콜백으로 응답 메시지를 기다립니다. on_message()가 호출되면 수신한 내용을 출력한 후 다시 LED 제어 명령을 발행합니다.

제어 명령은 LED1을 켤 때는 “1 1”, 끌 때는 “1 0”이고 LED2는 각각 “2, 1”, “2, 0”으로 정의합니다.

```
01: import paho.mqtt.client as mqtt
02: import signal
03: import sys
04:
05: client = mqtt.Client()
06:
07: def __publish_led_cmd(client):
08:     led_cmd = input("Enter of led actions (ex: 1 1):")
09:     client.publish("soda/actions/leds", led_cmd)
10:
11: def _signal_handler(signal, frame):
12:     client.disconnect()
13:     sys.exit(0)
14:
15: def _on_connect(client, userdata, flags, rc):
16:     if rc == 0:
17:         client.subscribe("soda/actions/ack")
18:         __publish_led_cmd(client)
19:     else:
20:         sys.exit(0)
21:
22: def _on_message(client, userdata, message):
23:     print(message.payload.decode())
24:     __publish_led_cmd(client)
25:
26: def main():
27:     signal.signal(signal.SIGINT, _signal_handler)
28:     client.on_connect = _on_connect
29:     client.on_message = _on_message
30:
31:     client.connect("192.168.10.2")
32:     client.loop_forever()
33:
34: if __name__ == "__main__":
35:     main()
```

07 ~ 09: __publish_led_cmd()는 표준 입력에서 읽은 LED 제어 명령 발행

08: input()으로 읽은 LED 제어 문자열을 led_cmd에 대입

09: “soda/actions/leds”에 led_cmd 발행

17 ~ 18: 브로커에 연결되면 “soda/actions/ack”를 구독한 후 __publish_led_cmd() 호출

23: on_message() 콜백이 호출되면 수신한 바이트 배열 타입의 페이로드를 문자열로 변환해 출력

24: __publish_led_cmd() 호출

구독자는 Leds 객체를 만든 후 수신한 메시지를 분석해 LED1 또는 LED2를 켜거나 끕니다. 이때 수신한 메시지가 올바른 제어 명령이면 “led<1|2> |” 메시지를 발행하고 아니면 “unknown command”를 발행합니다.

```
01: import paho.mqtt.client as mqtt
02: from pop import Leds
03: import signal
04: import sys
05:
06: client = mqtt.Client()
07: leds = Leds()
08:
09: def _signal_handler(signal, frame):
10:     client.disconnect()
11:     sys.exit(0)
12:
13: def _on_connect(client, userdata, flags, rc):
14:     if rc == 0:
15:         client.subscribe("soda/actions/leds")
16:     else:
17:         sys.exit(0)
18:
19: def _on_message(client, userdata, message):
20:     led, cmd = [int(r) for r in message.payload.decode().split()]
21:
22:     if not ((led == 1 or led == 2) and (cmd == 0 or cmd == 1)):
23:         ack = "unknown command"
24:     else:
25:         ack = "led%d "%(led)
26:
27:         if cmd:
28:             leds[led-1].on()
29:             ack += "on"
30:         else:
31:             leds[led-1].off()
32:             ack += "off"
33:
34:     client.publish("soda/actions/ack", ack)
35:
36: def main():
37:     signal.signal(signal.SIGINT, _signal_handler)
38:     client.on_connect = _on_connect
39:     client.on_message = _on_message
40:
41:     client.connect("192.168.10.2")
42:     client.loop_forever()
43:
44: if __name__ == "__main__":
45:     main()
```

13 ~ 17: _on_connect()는 브로커에 연결되면 "soda/actions/leds" 구독

19 ~ 34: 토픽에 수신되면 LED를 껴거나 끈 후 응답 게시

20: LED 위치와 명령을 분리해 led와 cmd에 대입

22 ~ 12: LED 위치와 명령이 정의된 것이 아니면 "unknown command"를 ack에 대입

24 ~ 32: led와 cmd를 이용해 LED를 켜거나 끄고 응답을 ack에 대입

34: ack를 "soda/actions/ack" 토픽에 게시

영구 세션

Client()의 기본값인 clean_session 인자가 True일 때는 항상 클린 세션으로 통신하지만 이를 False로 설정하면 영구 세션이 만들어집니다. 클린 세션은 발행자와 구독자가 실행 중일 때만 메시지가 이동되지만 영구 세션은 한쪽을 종료한 후 다시 실행해도 메시지가 이동합니다.

영구 세션으로 발행

앞서 구현한 LED 제어 발행자를 영구 세션에서 실행되도록 코드를 수정해 보겠습니다. Client 객체를 만들 때 클라이언트 ID 인자는 "soda_led_pub", clean_session 인자는 False로 설정합니다. 또한 영구 세션은 항상 subscribe()와 publish()의 qos 인자가 0보다 켜야 합니다.

```
01: client = mqtt.Client("soda_led_pub", False)
02:
03: def __publish_led_cmd(client):
04:     led_cmd = input("Enter of led actions (ex: 1 1):")
05:     client.publish("soda/actions/leds", led_cmd, 2)
06:
07: def _on_connect(client, userdata, flags, rc):
08:     if rc == 0:
09:         client.subscribe("soda/actions/ack", 1)
10:         __publish_led_cmd(client)
11:     else:
12:         sys.exit(0)
```

영구 세션으로 구독

LED 제어 구독자도 영구 세션에서 실행되도록 수정합니다. 클라이언트 ID는 "soda_led_sub"를 사용하고 subscribe()와 publish()의 qos 인자는 0보다 큰 값을 사용합니다. 브로커와 클라이언트 객체는 QoS 1보다 QoS 2에서 좀 더 메시지 게시와 수신에 노력을 기울이지만 통신 장애가 없는 환경에서는 QoS 1로도 충분합니다.

```
01: client = mqtt.Client("soda_led_pub", False)
02:
03: def __publish_led_cmd(client):
```

```
04:     led_cmd = input("Enter of led actions (ex: 1 1):")
05:     client.publish("soda/actions/leds", led_cmd, 2)
06:
07: def _on_connect(client, userdata, flags, rc):
08:     if rc == 0:
09:         client.subscribe("soda/actions/ack", 1)
10:         __publish_led_cmd(client)
11:     else:
12:         sys.exit(0)
```

이제 발행자만 실행한 후 LED 제어 명령을 발행해도 정상적으로 브로커에 게시되고, 심지어 발행자가 종료해도 세션 상태는 유지되므로 나중에 구독자를 실행해도 브로커에 게시된 메시지를 받을 수 있습니다.

콜백 이벤트

Client 객체는 항상 콜백으로 내부 상태를 응용프로그램에 전달하는데, 이벤트 상태에 따라 호출하는 콜백의 종류도 달라집니다.

연결, 해제 콜백

on_connect() 콜백은 브로커로부터 CONNACK 패킷을 수신하면 호출됩니다. CONNACK는 connect*()로 송신한 CONNECT 패킷에 대한 응답으로, 일정 시간 동안 CONNACK를 수신하지 못하면 OSError 예외가 발생합니다.

on_disconnect() 콜백은 disconnect()로 브로커에 DISCONNECT 패킷을 송신한 후 진행 중인 작업이 있다면 이를 중단하고 호출됩니다. disconnect()를 호출한 클라이언트는 reconnect()만 유효하며 reconnect_delay_set()으로 기본값이 1초인 연결 지연 시간을 조정할 수 있습니다.

연결 및 해제 콜백의 사용법을 알아보기 위해 주변 소음을 측정하는 Sound 객체를 이용해 발행자와 구독자를 구현해 봅니다.

연결이 끊어질 때마다 다시 연결 시도

연결이 끊어질 때마다 자동으로 다시 연결을 시도하도록 Sound 구독자의 on_disconnect() 콜백에서 reconnect()를 호출합니다. 사전에 reconnect_delay_set()으로 최소 지연 시간을 변경했다면 해당 시간만큼 지연한 후 다시 브로커에 연결을 요청합니다.

```
01: import paho.mqtt.client as mqtt
02: import signal
03:
04: client = mqtt.Client()
05:
06: def _signal_handler(signal, frame):
07:     client.disconnect()
08:
09: def _on_connect(client, userdata, flags, rc):
10:     if rc == 0:
11:         print("connected...")
12:         client.subscribe("soda/sensors/sound")
```

```
13:     else:
14:         print("connection fail:", rc)
15:         sys.exit(0)
16:
17: def _on_disconnect(client, userdata, rc):
18:     print("reconnected...")
19:     client.reconnect()
20:
21: def _on_message(client, userdata, message):
22:     print(message.payload.decode())
23:
24: def main():
25:     signal.signal(signal.SIGINT, _signal_handler)
26:     client.on_connect = _on_connect
27:     client.on_disconnect = _on_disconnect
28:     client.on_message = _on_message
29:
30:     client.reconnect_delay_set(5)
31:
32:     try:
33:         client.connect_async("192.168.10.2")
34:         client.loop_forever()
35:     except OSError as e:
36:         print(e)
37:
38: if __name__ == "__main__":
39:     main()
```

10 ~ 12: 브로커에 연결되면 "soda/sensors/sound" 구독

19: 응용프로그램이 브로커와의 연결을 끊으면 reconnect()를 호출해 다시 연결

30: 다시 연결할 때 최소 지연을 5초로 설정

35 ~ 36: 브로커 주소가 유효하지 않으면 OSError 예외 발생

구독자를 종료하기 위해 <Ctrl>c를 눌러도 `_on_disconnect()` 콜백에서 다시 다시 `reconnect()`를 호출하므로 <Ctrl>c를 두 번 연속으로 누르거나 <Ctrl>\를 눌러야 합니다.

QoS와 disconnect() 부효과

QoS에 따른 `disconnect()`의 부효과를 알아보기 위해 Sound 발행자는 `on_connect()` 콜백에서 for 루프로 각각 10회씩 QoS를 바꿔가며 토픽을 게시한 후 `disconnect()`로 브로커와의 연결을 끊어 봅니다.

```
01: import paho.mqtt.client as mqtt
02: from pop import Sound
03: import sys
04:
```

```
05: client = mqtt.Client()
06: sound = Sound()
07:
08: def __publish_sound(client, i, qos):
09:     sound_val = sound.readAverage()
10:     payload = "%02d: qos = %d, sound_level = %d"%(i + 1, qos, sound_val)
11:     client.publish("soda/sensors/sound", payload, qos)
12:
13: def _on_connect(client, userdata, flags, rc):
14:     if rc == 0:
15:         print("connected...")
16:         for qos in range(3):
17:             for i in range(10):
18:                 __publish_sound(client, i, qos)
19:
20:         print("completed...")
21:         client.disconnect()
22:     else:
23:         print("connection fail:", rc)
24:         sys.exit(0)
25:
26: def _on_disconnect(client, userdata, rc):
27:     print("disconnected...")
28:
29: def main():
30:     client.on_connect = _on_connect
31:     client.on_disconnect = _on_disconnect
32:
33:     try:
34:         client.connect_async("192.168.10.2")
35:         client.loop_forever()
36:     except OSError as e:
37:         print(e)
38:
39: if __name__ == "__main__":
40:     main()
```

06: Sound 객체를 만들어 sound에 대입

08 ~ 11: __publish_sound()는 Sound 객체로 주변 소음 수준을 측정해 발행

09: 측정하나 주변 소음 평균 수준을 sound_val에 대입

10: 매개변수로 전달받은 순서 번호 i와 qos 및 sound_val를 이용해 문자열을 만들어 payload에 대입

11: "soda/sensors/sound"에 payload를 qos로 게시

16 ~ 18: 0 ~ 2까지 차례로 qos에 대입하면서 내부 for 루프 실행

17 ~ 18: 0 ~ 9까지 차례로 i에 대입하면서 for 루프 실행

18: client와 i, qos를 인자로 __publish_sound() 호출

21: for 루프가 완료되면 disconnect()로 연결 끊기

publish()는 비동기 함수이므로 for 루프가 종료해도 Clinet 객체는 여전히 브로커에 메시지 전달을 처리하고 있을 수 있습니다. QoS = 0은 응답 패킷을 요구하지 않으므로 바로 다음 단계로 넘어가지만 QoS > 0은 1개 이상의 응답 패킷을 요구하므로 QoS = 0 보다는 전달 시간이 더 걸립니다. 이 상태에서 disconnect()를 호출하면, 모든 작업이 중단되므로 남은 패킷이 있다면 이 역시 더 이상 브로커로 전송되지 않습니다.

따라서 구독자를 실행한 후 발행자를 실행하면 QoS = 0인 메시지는 모두 수신되지만 QoS = 1은 일부, QoS = 2는 하나도 수신되지 않을 수 있습니다.

발행 콜백

on_publish()는 publish()로 PUBLISH 패킷을 송신하거나(QoS = 0), PUBACK 응답 패킷을 수신하거나(QoS = 1), PUBCOMP 2차 응답 패킷을 수신하면(QoS = 2) 호출되므로 publish()로 발행한 메시지는 가급적 on_publish() 콜백으로 브로커에 송신 또는 게시되었는지 확인하는 것이 좋습니다.

Sound 발행자를 다음과 같이 on_publish() 콜백을 통해 메시지가 브로커에 전달되거나 게시되면 다음 메시지를 발행하도록 수정하면 모든 메시지를 전송한 후 disconnect()를 호출할 수 있습니다.

```
01: import paho.mqtt.client as mqtt
02: from pop import Sound
03: import sys
04:
05: client = mqtt.Client()
06: sound = Sound()
07:
08: qos = 0
09: index = 0
10:
11: def __publish_sound(client, i, qos):
12:     sound_val = sound.readAverage()
13:     payload = "%02d: qos = %d, sound_level = %d"%(i + 1, qos, sound_val)
14:     client.publish("soda/sensors/sound", payload, qos)
15:
16: def _on_connect(client, userdata, flags, rc):
17:     if rc == 0:
18:         print("connected...")
19:         _on_publish(client, userdata, -1)
20:     else:
21:         print("connection fail:", rc)
22:         sys.exit(0)
23:
24: def _on_disconnect(client, userdata, rc):
25:     print("disconnected...")
26:
27: def _on_publish(client, userdata, mid):
28:     global qos
29:     global index
```

```
30:
31:     if index > 9:
32:         qos += 1
33:         index = 0
34:
35:     if qos > 2:
36:         print("completed...")
37:         client.disconnect()
38:     else:
39:         __publish_sound(client, index, qos)
40:         index += 1
41:
42: def main():
43:     client.on_connect = _on_connect
44:     client.on_disconnect = _on_disconnect
45:     client.on_publish = _on_publish
46:     try:
47:         client.connect_async("192.168.10.2")
48:         client.loop_forever()
49:     except OSError as e:
50:         print(e)
51:
52: if __name__ == "__main__":
53:     main()
```

08: 전역 변수 qos에 0 대입

09: 전역 변수 index에 0 대입

19: 브로커에 연결되면 _on_publish() 호출

27 ~ 40: _on_publish()는 on_publish() 콜백

28 ~ 29: 전역 변수 qos와 index 쓰기 허용 설정

31 ~ 33: index > 9이면 qos와 index를 각각 1씩 증가

35 ~ 37: qos > 2이면 disconnect() 호출

38 ~ 40: soq <= 2이면 __publish_sound() 호출한 후 index를 1 증가

userdata

Clinet 객체는 사용자 데이터를 콜백 함수에서 공유할 수 있도록 `Clinet()`의 userdata 인자 또는 `user_data_set()`로 설정하는 userdata 필드를 제공합니다.

앞의 수정한 Sound 발행자는 콜백 함수에서 접근하는 2개의 전역 변수가 있는데, "qos":0과 "index":0을 요소로 갖는 딕셔너리 객체를 만들어 `Client()`의 userdata 인자로 전달하면 `_on_publish()`의 userdata 매개변수를 통해 접근할 수 있으므로 과도한 전역 변수의 사용을 줄일 수 있습니다.

```
client = mqtt.Client(userdata={"qos":0, "index":0})

def _on_publish(client, userdata, mid):
    if userdata["index"] > 9:
        userdata["qos"] += 1
        userdata["index"] = 0

    if userdata["qos"] > 2:
        print("completed...")
        client.disconnect()
    else:
        __publish_sound(client, userdata["index"], userdata["qos"])
        userdata["index"] += 1
```

연속적인 메시지 발행과 수신 콜백 문제

on_publish()에서 다시 QoS = 0인 publish()를 호출하면 on_publish() 콜백과 QoS = 0인 publish() 호출이 무한히 반복되므로 브로커로부터 PUBLISH 패킷이 수신되어도 on_message() 콜백이 호출되지 않습니다.

```
def __publish_cds(client):
    cds_val = cds.readAverage()

    client.publish("soda/sensors/cds", cds_val)      #QoS = 0

def _on_connect(client, userdata, flags, rc):
    client.subscribe("soda/sensors/cds")
    __publish_cds(client)

def _on_message(client, userdata, message):  #this has no chance of being called!
    print(message.payload.decode())

def _on_publish(client, userdata, mid):      #without waiting
    __publish_cds(client)
```

QoS = 1인 publish()는 응답 패킷인 PUBACK을 받아야 다음 PUBLISH 패킷을 송신하므로 그사이 다른 이벤트를 처리할 시간이 주어집니다. 따라서 on_publish() 콜백에서 QoS = 1인 publish()를 사용하면 메시지가 수신될 때마다 on_message() 콜백도 정상적으로 호출됩니다.

```
import paho.mqtt.client as mqtt
from pop import Cds
import signal
import datetime

client = mqtt.Client()
cds = Cds()
```

```
08:
09: def __publish_cds(client):
10:     cds_val = cds.readAverage()
11:     now = datetime.datetime.now()
12:     payload = "%s %d"%(now.strftime("%S:%f"), cds_val)
13:
14:     client.publish("soda/sensors/cds", payload, 1)        #QoS = 1
15:
16: def _signal_handler(signal, frame):
17:     client.disconnect()
18:
19: def _on_connect(client, userdata, flags, rc):
20:     if rc == 0:
21:         print("connected...")
22:         client.subscribe("soda/sensors/cds")
23:         __publish_cds(client)
24:     else:
25:         print("connection fail:", rc)
26:         sys.exit(0)
27:
28: def _on_message(client, userdata, message):
29:     now = datetime.datetime.now()
30:     t, sound_val = message.payload.decode().split()
31:     t_time = datetime.datetime.strptime(t, "%S:%f")
32:     diff = now - t_time
33:
34:     print(t, now.strftime("%S:%f"), "%.4f"%(diff.microseconds / 1000000),
sound_val)
35:
36: def _on_publish(client, userdata, mid):
37:     __publish_cds(client)
38:
39: def main():
40:     signal.signal(signal.SIGINT, _signal_handler)
41:     client.on_connect = _on_connect
42:     client.on_message = _on_message
43:     client.on_publish = _on_publish
44:
45:     try:
46:         client.connect_async("192.168.10.2")
47:         client.loop_forever()
48:     except OSError as e:
49:         print(e)
50:
51: if __name__ == "__main__":
52:     main()
```

04: datetime 모듈 로드

07: Cds 객체를 만들어 cds에 대입

9 ~ 14: __publish_cds()는 조도를 측정해 측정 시각과 함께 QoS = 1로 게시

10: 측정한 조도를 cds_val에 대입

11: 현재 시각을 now에 대입

12: now의 초와 밀리초 및 cds_val를 형식으로 문자열로 만들어 payload에 대입

13: QoS = 1로 "soda/sensors/cds"에 payload 게시

22 ~ 23: 브로커에 연결되면 "soda/sensors/cds"를 구독한 후 __publish_cds() 호출

29: 현재 시각을 now에 대입

30: 문자열로 변환한 message.payload의 를 분리해 각각 t와 sound_val에 대입

31: 문자열 t를 초와 밀리초로 구성된 datetime 객체로 바꿔 t_time에 대입

32: 측정 시간과 수신 시간을 비교하기 위해 now - t_time 결과인 timedelta 객체를 diff에 대입

34: t, now의 초, 밀리초, diff의 마이크로초, sound_val 순으로 출력

37: on_publish() 콜백이 호출될 때마다 __publish_cds() 호출

하지만 QoS > 0인 `publish()`는 응답 패킷 처리로 인해 QoS = 0인 `publish()`보다 전달 시간이 더 걸리므로 전송 시간을 줄이려면 `on_publish()` 콜백 보다는 다른 곳에서 QoS = 0인 `publish()`를 사용해야 합니다.

메시지 루프와 연속적인 메시지 발행

`loop_forever()`와 `loop_start()`는 둘 다 Clinet 객체의 메시지 처리를 담당하지만 `loop_start()` 호출은 즉시 반환되므로 다음과 작업 종료를 관리하는 상태 변수와 사용자 루프가 필요합니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     client.subscribe("soda/sensors/cds")
03:     client.is_stop = False
04:
05: def _signal_handler(signal, frame):
06:     client.disconnect()
07:     client.loop_stop()
08:     client.is_stop = True
09:
10: def main():
11:     ...
12:     client.is_stop = True   #user attribute
13:
14:     client.connect_async("192.168.10.2")
15:     client.loop_start()
16:
17:     while client.is_stop: pass
18:
```

```
19:     while not client.is_stop:
20:         __publish_cds(client)
```

메시지 발행하면서 구독하는 양방향 클라이언트에 이 구조를 적용하면 메시지 전달 시간을 최대 절반 이하로 줄일 수 있는데, 다음은 이를 적용한 Cds 양방향 클라이언트 구현을 보여줍니다.

```
01: import paho.mqtt.client as mqtt
02: from pop import Cds
03: import signal
04: import datetime
05:
06: client = mqtt.Client()
07: cds = Cds()
08:
09: def __publish_cds(client):
10:     cds_val = cds.readAverage()
11:     now = datetime.datetime.now()
12:     payload = "%s %d"%(now.strftime("%S:%f"), cds_val)
13:
14:     client.publish("soda/sensors/cds", payload, 1)
15:
16: def _signal_handler(signal, frame):
17:     client.disconnect()
18:     client.loop_stop()
19:     client.is_stop = True
20:
21: def _on_connect(client, userdata, flags, rc):
22:     if rc == 0:
23:         print("connected...")
24:         client.subscribe("soda/sensors/cds")
25:         client.is_stop = False
26:     else:
27:         print("connection fail:", rc)
28:         sys.exit(0)
29:
30: def _on_message(client, userdata, message):
31:     now = datetime.datetime.now()
32:     t, sound_val = message.payload.decode().split()
33:     t_time = datetime.datetime.strptime(t, "%S:%f")
34:     diff = now - t_time
35:
36:     print(t, now.strftime("%S:%f"), "%.4f"%(diff.microseconds / 1000000),
sound_val)
37:
38: def main():
39:     signal.signal(signal.SIGINT, _signal_handler)
40:     client.on_connect = _on_connect
41:     client.on_message = _on_message
42:     client.is_stop = True
43:
```

```
44:     try:
45:         client.connect_async("192.168.10.2")
46:         client.loop_start()
47:     except OSError as e:
48:         print(e)
49:
50:     while client.is_stop: pass
51:
52:     while not client.is_stop:
53:         __publish_cds(client)
54:
55: if __name__ == "__main__":
56:     main()
```

18: Client 객체의 메시지 루프 중단

19: INT 시그널이 발행하면 client 사용자 속성 is_stop에 True 대입

25: 브로커에 연결되면 client 사용자 속성 is_stop에 False 대입

42: client 사용자 속성 is_stop의 초깃값으로 True 대입

50: client 사용자 속성 is_stop이 True인 동안 while 루프 실행

52 ~ 53: client 사용자 속성 is_stop이 False인 동안 while 루프를 실행하며 __publish_cds() 호출

---

payload

payload는 클라이언트와 브로커 사이를 이동하는 메시지로 발행자는 publish()로 브로커에 전달하고 구독자는 브로커로부터 on_message() 콜백으로 받습니다. 이때 MQTT 전송 계층은 바이트 배열만 허용하므로 publish()는 게시할 데이터가 바이트 배열이 아니면 변환한 후 전송합니다.

on_message() 콜백의 message 매개변수는 MQTTMessage 타입으로 수신한 데이터는 payload 속성에 포함되어 있으며, 항상 바이트 배열입니다. 따라서 사용자 변환이 필요한데 발행자의 메시지 타입에 맞춰 정수나 실수는 int() 또는 float(), 문자열은 decode() 메소드를 사용합니다.

구조체 메시지 발행

struct 모듈의 pack()으로 만드는 구조체는 여러 타입 또는 같은 타입의 여러 데이터를 하나의 데이터로 묶을 수 있으므로 이를 메시지 발행에 적용하면 발행 횟수를 줄일 수 있습니다.

TempHumi 객체로 측정한 온도와 습도를 브로커에 게시할 때 pack()으로 묶으면 publish()는 한 번만 호출하면 됩니다. 아래 _do_publish_temphumi()는 temp와 humi를 "2H" (unsigned short 2개)로 묶어 전송하는 것은 보여줍니다.

```
01: def _do_publish_temphumi(client):
02:     while not client.is_stop:
03:         thData = th.read()
```

```
04:         temp = round(thData.temp)
05:         humi = round(thData.humi)
06:
07:         data = struct.pack("2H", temp, humi)
08:         client.publish("soda/sensors/temphumi", data)
09:         time.sleep(client.interval / 1000)
```

또한 주기적으로 게시되는 메시지는 작업 스레드를 사용하면 편리합니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     if rc == 0:
03:         print("connected...")
04:         client.subscribe("soda/sensors/interval")
05:         threading.Thread(target=_do_publish_temphumi, args=
(client,)).start()
06:     else:
07:         print("connection fail:", rc)
08:         sys.exit(0)
```

구조체와 작업 스레드를 이용해 온도와 습도를 게시하는 전체 구현은 다음과 같습니다.

```
01: import paho.mqtt.client as mqtt
02: from pop import TempHumi
03: import signal
04: import struct
05: import threading
06: import sys
07: import time
08:
09: client = mqtt.Client()
10: th = TempHumi()
11:
12: def _do_publish_temphumi(client):
13:     while not client.is_stop:
14:         thData =  th.read()
15:         temp = round(thData.temp)
16:         humi = round(thData.humi)
17:
18:         data = struct.pack("2H", temp, humi)
19:         client.publish("soda/sensors/temphumi", data)
20:         time.sleep(client.interval / 1000)
21:
22: def _signal_handler(signal, frame):
23:     client.is_stop = True
24:     client.disconnect()
25:
26: def _on_connect(client, userdata, flags, rc):
27:     if rc == 0:
```

```
28:         print("connected...")
29:         client.subscribe("soda/sensors/interval")
30:         threading.Thread(target=_do_publish_temphumi, args=
(client,)).start()
31:     else:
32:         print("connection fail:", rc)
33:         sys.exit(0)
34:
35: def _on_message(client, userdata, message):
36:     client.interval = int(message.payload)
37:     print("SET INTERVAL:", client.interval)
38:
39: def main():
40:     signal.signal(signal.SIGINT, _signal_handler)
41:     client.on_connect = _on_connect
42:     client.on_message = _on_message
43:     client.interval = 500    #0.5sec
44:     client.is_stop = False
45:
46:     client.connect_async("192.168.10.2")
47:     client.loop_forever()
48:
49: if __name__ == "__main__":
50:     main()
```

04: struct 모듈 로드

05: threading 모듈 로드

10: 온도와 습도를 측정하는 TempHumi 객체를 th에 대입

12 ~ 19: _do_publish_temphumi()는 지속적으로 측정한 온습도를 발행하는 스레드

13 ~ 19: client 사용자 속성 is_stop이 False인 동안 while 루프 실행

15: 현재 습도를 읽어 소수점을 제거한 후 humi에 대입

16: 현재 온도를 읽어 소수점을 제거한 후 temp에 대입

18: unsigned short(2바이트) 단위로 temp와 humi를 묶어 data에 대입

19: "soda/sensors/temphumi"에 data 발행

20: client 사용자 속성 interval만큼 밀리초 단위로 대기

23: INT 시그널이 발행하면 스레드를 멈추기 위해 client 사용자 속성 is_stop에 True 대입

29: 브로커에 연결되면 "soda/sensors/interval" 구독

30: client를 인자로 _do_publish_temphumi()를 스레드로 만들어 실행

43: client 사용자 속성 interval에 초깃값으로 500(0.5초) 대입

44: client 사용자 속성 is_stop에 초깃값으로 False 대입

구조체 메시지 구독

구조체 타입 메시지를 구독할 때는 발행자가 pack()으로 묶은 메시지 구조를 알아야 unpack()으로 풀 수 있습니다. 앞서 발행자는 온도와 습도를 "2H"로 묶어 발행했으므로 구독자는 수신한 메시지를 "2H"로 풀어야 합니다.

```
01: def _on_message(client, userdata, message):
02:     temp, humi = struct.unpack("2H", message.payload)
03:     print("temp = %d, humi = %d"%(temp, humi))
```

다음은 구조체 메시지를 구독하는 전체 구현입니다.

```
01: import paho.mqtt.client as mqtt
02: import signal
03: import struct
04: import sys
05:
06: client = mqtt.Client()
07:
08: def _signal_handler(signal, frame):
09:     client.disconnect()
10:     client.loop_stop()
11:     sys.exit(0)
12:
13: def _on_connect(client, userdata, flags, rc):
14:     client.subscribe("soda/sensors/temphumi")
15:
16: def _on_message(client, userdata, message):
17:     temp, humi = struct.unpack("2H", message.payload)
18:     print("temp = %d, humi = %d"%(temp, humi))
19:
20: def main():
21:     signal.signal(signal.SIGINT, _signal_handler)
22:     client.on_connect = _on_connect
23:     client.on_message = _on_message
24:
25:     client.connect("192.168.10.2")
26:     client.loop_start()
27:
28:     while True:
29:         interval = int(input("Enter of interval: "))
30:         client.publish("soda/sensors/interval", interval)
31:
```

```
32: if __name__ == "__main__":
33:     main()
```

10 ~ 11: INT 시그널이 발생하면 스레드에서 실행 중인 네트워크 루프를 멈추고 프로그램 강제 종료

14: 브로커에 연결되면 "soda/sensors/temphumi" 구독

17: 수신한 메시지에서 unsigned short(2바이트) 단위로 2개를 풀어 temp와 humi에 대입

28 ~ 30: 메인 모듈에서 while 발행 루프 실행

29: 표준 입력에서 읽은 문자열을 정수로 바꿔 interval에 대입

30: "soda/sensors/interval"에 interval 발행

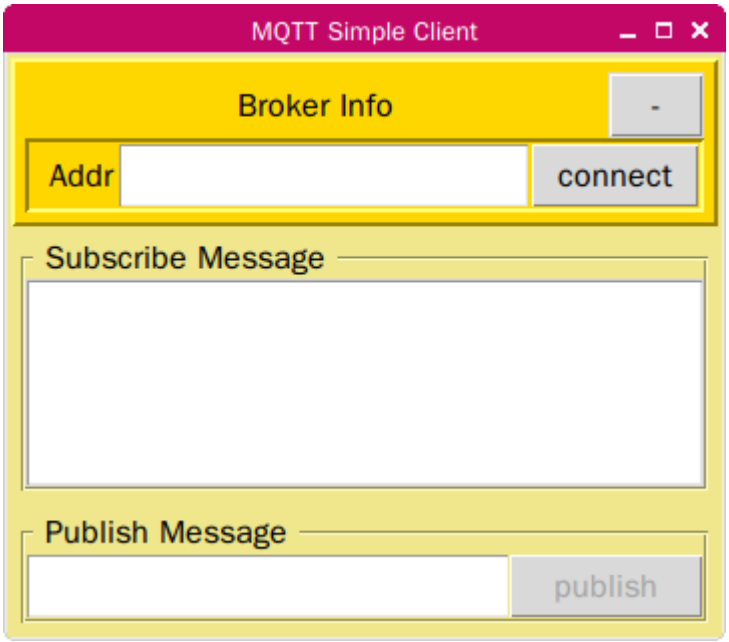
GUI 입히기

파이썬용 GUI 라이브러리는 기본 패키지인 Tkinter를 비롯해 좀더 전문적인 QT 라이브러리의 바인더(파이썬 호출 버전)인 PyQt 또는 PySide 및 교육용으로 사용하기 쉽게 Tkinter를 추상화한 AppJar가 있습니다.

Soda에는 PyQt가 미리 설치되어 있는데, GUI 프로그래밍 경험이 없다면 어렵게 느껴질 수 있으므로 AppJar를 권장합니다. AppJar의 설치 명령은 다음과 같습니다. 사용법은 http://appjar.info/ 를 참조하세요.

```
sudo pip3 install appjar
```

다음은 구현할 GUI 화면 구성입니다.



[그림 29 GUI 기반 구조체 스타일 Temp/Humi 구독, 인터벌 발행 클라이언트]

화면은 행 단위로 1개의 토글 프레임과 2개의 레이블 프레임으로 구성됩니다.

```
01: def main():
02:     app.setBg("Khaki")
03:     app.setFont(12, "은 그래픽")
04:
05:     with app.toggleFrame("Broker Info", 0, 0):
06:         ...
07:
08:     with app.labelFrame(" Subscribe Message ", 1, 0):
09:         ...
10:
11:     with app.labelFrame(" Publish Message ", 2, 0):
12:         ...
```

토글 프레임에는 브로커 주소를 입력한 후 연결을 요청하는 레이블과 addr 입력항목, connect, disconnect 버튼을 가로로 추가한 후 disconnect 버튼을 숨깁니다.

```
01:     with app.toggleFrame("Broker Info", 0, 0):
02:         app.setToggleFrameBg("Broker Info", "gold")
03:         app.toggleToggleFrame("Broker Info")
04:
05:         app.setSticky("news")
06:
07:         app.addLabel("label1", "Addr", 0, 0)
08:         app.setLabelAlign("label1", "right")
09:         app.setLabelWidth("label1", 4)
10:         app.addEntry("addr", 0, 1)
11:
12:         app.addButton("disconnect", on_bt_disconnect, 0, 2)
13:         app.hideButton("disconnect")
14:         app.addButton("connect", on_bt_connect, 0, 2)
```

첫 번째 레이블 프레임에는 수신된 센서값을 표시하는 recv 리스트박스만 추가하는데, 아이템이 5줄 이상 추가되면 스크롤이 되도록 설정합니다.

```
01:     with app.labelFrame(" Subscribe Message ", 1, 0):
02:         app.setSticky("news")
03:         app.addListBox("recv", [])
04:         app.setListBoxRows("recv", 5)
```

두 번째 레이블 프레임에는 인터벌을 발행하는데 필요한 interval 입력항목과 publish 버튼을 추가한 후 publish 버튼은 비활성화시킵니다.

```
01:     with app.labelFrame(" Publish Message ", 2, 0):
02:         app.setSticky("news")
03:         app.addEntry("interval", 0, 0)
04:         app.setEntrySubmitFunction("interval", on_bt_publish)
```

```
05:
06:         app.addButton("publish", on_bt_publish, 0, 1)
07:         app.setButtonWidth("publish", 1)
08:         app.disableButton("publish")
```

addr 입력항목에 브로커의 주소를 입력한 후 connect 버튼을 누르면 on_bt_connect() 콜백이 호출됩니다. on_bt_connect()는 클라이언트 객체를 만들어 전역 변수인 client에 대입한 후 addr 입력항목의 주소를 가져와 비동기로 브로커에 연결합니다.

이렇게 on_bt_connect()에서 Client 객체를 만드는 이유는 브로커가 실행되지 않은 상태에서 connect_async()로 연결을 요청하면 지나친 연결 대기가 발행하므로, 이를 무시하고 브로커가 실행되며 다시 연결을 시도록 하기 위함입니다.

```
01: def on_bt_connect(unuse):
02:     global client
03:
04:     if client:
05:         del client
06:
07:     client = mqtt.Client()
08:     client.on_connect = _on_connect
09:     client.on_message = _on_message
10:
11:     addr = app.getEntry("addr")
12:
13:     try:
14:         client.connect_async(addr)
15:         client.loop_start()
16:     except (ValueError, OSError):
17:         client.loop_stop()
18:         app.errorBox("Connection Error", "invalied broker address")
```

사용자가 disconnect 버튼을 누르면 on_bt_disconnect() 콜백이 호출되면서 브로커와의 연결을 끊고 네트워크 루프를 멈춘 후 connect, disconnect, publish 버튼을 원래 상태로 되돌립니다.

```
01: def on_bt_disconnect(unuse):
02:     client.disconnect()
03:     client.loop_stop()
04:
05:     change_bt_connect_ui(False)
```

change_bt_connect_ui()는 브로커에 연결되거나 연결이 끊어졌을 때 connect, disconnect, publish 버튼 상태를 바꾸는데, 인자로 True를 전달하면 연결된 상태를 반영하고 False이면 끊어졌을 때, 즉 초기 UI 상태로 되돌립니다.

```
01: def change_bt_connect_ui(is_connect):
02:     if is_connect:
03:         app.hideButton("connect")
04:         app.showButton("disconnect")
05:         app.enableButton("publish")
06:     else:
07:         app.hideButton("disconnect")
08:         app.showButton("connect")
09:         app.disableButton("publish")
```

클라이언트가 브로커에 연결되면 publish 버튼이 활성화되므로 interval 입력항목에 20 ~ 5000 사이 값을 입력할 수 있다. 입력 완료는 <Enter> 또는 publish 버튼이며, on_bt_publish() 콜백이 호출되어 "soda/sensors/interval" 토픽에 interval 입력항목에서 가져온 인터벌을 발행합니다.

```
01: def on_bt_publish(unuse):
02:     interval = int(app.getEntry("interval"))
03:     app.clearEntry("interval")
04:
05:     client.publish("soda/sensors/interval", interval)
```

브로커에 연결되면 on_connect() 콜백에서 "soda/sensors/temphumi"를 구독한 후 change_bt_connect_ui()에 True를 전달해 UI에 반영합니다. 만약 연결에 문제가 있으면 네트워크 루프를 멈추고 반환 코드를 오류 메시지박스에 표시합니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     if rc != 0:
03:         client.loop_stop()
04:         app.errorBox("Connection Error:", str(rc))
05:         return
06:
07:     client.subscribe("soda/sensors/temphumi")
08:     change_bt_connect_ui(True)
```

브로커에 연결되면 센서값이 수신될 때마다 Client 객체의 on_message() 콜백이 호출되는데, 수신된 메시지에서 unpack()으로 온도와 습도를 풀어 recv 리스트박스에 추가합니다.

```
01: def _on_message(client, userdata, message):
02:     temp, humi = struct.unpack("2H", message.payload)
03:     data = "temp = %d, humi = %d"%(temp, humi)
04:
05:     app.addListItem("recv", data)
```

지금까지 설명한 내용의 전체 코드는 다음과 같습니다. 전역 변수 clien는 on_bt_connect()에서 Client 객체로 초기화되고 app는 if **name** 구문에서 GUI 객체로 초기화됩니다.

```
01: import paho.mqtt.client as mqtt
02: import struct
03: from appJar import gui
04:
05: client = None
06: app = None
07:
08: def _on_connect(client, userdata, flags, rc):
09:     if rc != 0:
10:         client.loop_stop()
11:         app.errorBox("Connection Error:", str(rc))
12:         return
13:
14:     client.subscribe("soda/sensors/temphumi")
15:     change_bt_connect_ui(True)
16:
17: def _on_message(client, userdata, message):
18:     temp, humi = struct.unpack("2H", message.payload)
19:     data = "temp = %d, humi = %d"%(temp, humi)
20:
21:     app.addListItem("recv", data)
22:
23: def on_bt_connect(unuse):
24:     global client
25:
26:     if client:
27:         del client
28:
29:     client = mqtt.Client()
30:     client.on_connect = _on_connect
31:     client.on_message = _on_message
32:
33:     addr = app.getEntry("addr")
34:
35:     try:
36:         client.connect_async(addr)
37:         client.loop_start()
38:     except (ValueError, OSError):
39:         client.loop_stop()
40:         app.errorBox("Connection Error", "invalied broker address")
41:
42: def on_bt_disconnect(unuse):
43:     client.disconnect()
44:     client.loop_stop()
45:
46:     change_bt_connect_ui(False)
47:
48: def on_bt_publish(unuse):
49:     interval = int(app.getEntry("interval"))
50:     app.clearEntry("interval")
51:
52:     client.publish("soda/sensors/interval", interval)
```

```
53:
54: def change_bt_connect_ui(is_connect):
55:     if is_connect:
56:         app.hideButton("connect")
57:         app.showButton("disconnect")
58:         app.enableButton("publish")
59:     else:
60:         app.hideButton("disconnect")
61:         app.showButton("connect")
62:         app.disableButton("publish")
63:
64: def main():
65:     app.setBg("Khaki")
66:     app.setFont(12, "은 그래픽")
67:
68:     with app.toggleFrame("Broker Info", 0, 0):
69:         app.setToggleFrameBg("Broker Info", "gold")
70:         app.toggleToggleFrame("Broker Info")
71:
72:         app.setSticky("news")
73:
74:         app.addLabel("label1", "Addr", 0, 0)
75:         app.setLabelAlign("label1", "right")
76:         app.setLabelWidth("label1", 4)
77:         app.addEntry("addr", 0, 1)
78:
79:         app.addButton("disconnect", on_bt_disconnect, 0, 2)
80:         app.hideButton("disconnect")
81:         app.addButton("connect", on_bt_connect, 0, 2)
82:
83:     with app.labelFrame(" Subscribe Message ", 1, 0):
84:         app.setSticky("news")
85:         app.addListBox("recv", [])
86:         app.setListBoxRows("recv", 5)
87:
88:     with app.labelFrame(" Publish Message ", 2, 0):
89:         app.setSticky("news")
90:         app.addEntry("interval", 0, 0)
91:         app.setEntrySubmitFunction("interval", on_bt_publish)
92:
93:         app.addButton("publish", on_bt_publish, 0, 1)
94:         app.setButtonWidth("publish", 1)
95:         app.disableButton("publish")
96:
97: if __name__ == "__main__":
98:     app = gui("MQTT Simple Client")
99:     main()
100:         app.go()
```

적절한 토픽과 JSON 메시지

클라이언트로 발행 및 구독하는 정보가 집안의 온도, 습도, 조도, 소음 레벨일 때 이를 개별 토픽으로 분류하면 다음과 같습니다. 하지만 이 방법은 각 토픽마다 2바이트 크기의 센서값을 가지므로 발행자는 4번의 `publish()` 호출이 필요하고 구독자는 4번의 메시지 수신을 처리해야 합니다.

- house/sensors/temp
- house/sensors/humi
- house/sensors/cds
- house/sensors/sound

앞서 소개한 바와 같이 토픽을 "house/sensors"로 통일한 후 4개의 센서값을 구조체 스타일로 묶으면 발행과 구독 횟수가 1회로 줄어듭니다. 하지만 이때는 메시지의 포맷이 고정되어 특정 센서만 선택적으로 발행하거나 구독할 때 어려움이 있으므로 JSON *JavaScript Object Notation*과 같은 표준화된 데이터 포맷을 사용하는 것이 좋습니다.

JSON은 JavaScript 문법에 영향을 받은 매우 가볍고 효율적인 데이터 표현 방식으로 단순함과 유연함 때문에 네트워크 환경에서 데이터를 교환할 때 널리 사용됩니다. 일반적인 JSON 포맷은 키와 값 모음으로 파이썬은 json 패키지의 `dumps()`와 `loads()`로 이를 지원합니다. `dumps()`는 딕셔너리를 JSON 문자열로 변환*encoding*하고 `loads()`는 JSON 문자열을 다시 딕셔너리로 복원*decoding*합니다.

센서값 발행, 액션 구독 토픽

메시지에 JSON 포맷을 적용하는 방법을 알아보기 위해 "house/sensors" 토픽에 지속적으로 측정한 온도, 습도, 조도, 소음 레벨을 발행하고, "house/actions"에서 구독한 메시지를 처리해 LED, 모터 중 하나에 적용하는 발행 및 구독 클라이언트를 구현해 보겠습니다.

토픽에 대한 메시지 규칙은 다음과 같습니다.

- house/sensors
  - temp =
  - humi =
  - cds =
  - sound =
- house/actions
  - sensors
    - interval =
  - led
    - <0 | 1>
      - on = <True | False> | bright = < 1 ~ 10>
  - motor
    - direction = <"stop" | "forward" | "backword">
    - speed = <1 ~ 10>

"house/actions" 토픽의 메시지는 문자열 파싱에 대한 부담을 줄이기 위해 다음과 같이 딕셔너리로 표현된 문자열을 사용자가 입력하면 이를 `eval()`를 통해 딕셔너리 객체로 변환해 사용할 것입니다.

- interval
  - {"sensors":{"interval":1000}}
- led
  - {"led":{0:{"on":True}}}

    - {"led":{0:{"on":False}, 1:{"bright":7}}}
- motor
    - {"motor":{"direction":"forward"}}
    - {"motor":{"speed":10}}
    - {"motor":{"direction":"backward", "speed":7}}
- total
    - {"led":{0:{"on":False}, 1:{"bright":3}}, "sensors":{"interval":1000}}
    - {"led":{0:{"on":True}}, "motor":{"direction":"forward", "speed":4}, "sensors":{"interval":100}}

센서값 발행, 액션 구독 클라이언트

측정한 센서값들은 인터벌만큼 지연하면서 지속적으로 발행해야 하므로 _do_publish_sensors()를 작업 스레드로 만든 후 while 발행 루프를 실행하는데, TempHumi와 Cds, Sound 객체는 모두 이 스레드 안에서만 사용하므로 객체의 생성도 스레드 안으로 한정합니다.

```
01: def _do_publish_sensors(client):
02:     th = TempHumi(21)
03:     cds = Cds(2)
04:     sound = Sound(1)
```

while 발행 루프에서는 온도와 습도, 조도, 주변 소음 수준을 측정한 후 이를 dumps()로 묶어 "soda/sensors"에 발행한 다음 밀리초 단위로 지연하기 위해 sleep()에 client.interval / 1000을 전달합니다.

```
01: while not client.is_stop:
02:         thData = th.read()
03:         t = round(thData.temp)
04:         h = round(thData.humi)
05:         c = cds.readAverage()
06:         s = sound.readAverage()
07:
08:         data = json.dumps({"temp":t, "humi":h, "cds":c, "sound":s})
09:         client.publish("soda/sensors", data)
10:         time.sleep(client.interval / 1000)
```

인터벌을 포함해 LED와 모터는 on_message() 콜백에서 수신한 메시지를 파악해 제어하므로 Leds와 DCMotor 객체를 딕셔너리의 요소로 만들어 Client()의 userdata 인자에 딕셔너리를 전달합니다.

```
01: actions = {
02:     "leds":Leds(),
03:     "motor":DCMotor()
04: }
05:
06: client = mqtt.Client(userdata=actions)
```

on_message() 콜백은 메시지가 전달될 때마다 loads()로 JSON 포맷의 센서값들을 딕셔너리로 변환한 후 키에 따른 값을 처리합니다.

```
01: def _on_message(client, userdata, message):
02:     payload = json.loads(message.payload)
03:
04:     for key, values in payload.items():
05:         ...
```

key가 "sensors"라면 values는 "interval":<value> 타입의 딕셔너리이므로 values["interval"]로 인터벌값을 가져옵니다.

```
01:         if key == "sensors":
02:             client.interval = values["interval"]
```

key가 "led"일 때 values는 "pos":{"cmd":<value>} 타입의 딕셔너리를 값으로 갖는 딕셔너리이므로 이를 values.items()으로 풀면 action은 "cmd":<value> 타입의 딕셔너리로 추출됩니다. 마지막으로 action을 action.items()로 풀어 "cmd"에 따른 <value>를 처리합니다.

```
01:         elif key == "led":
02:             for pos, action in values.items():
03:                 for cmd, value in action.items():
04:                     pos = int(pos)
05:                     if cmd == "on":
06:                         userdata["leds"][pos].on() if value else
userdata["leds"][pos].off()
07:                     elif cmd == "bright":
08:                         userdata["leds"][pos].bright(value)
```

key가 "motor"일 때 values는 "action":<value>, ... 타입의 딕셔너리므로 values를 values.items()로 풀어 action에 따른 value를 처리합니다. 단, action이 "direction"이고 value가 "stop"이면 다음 요소를 처리할 필요가 없습니다.

```
01:         elif key == "motor":
02:             for action, value in values.items():
03:                 if action == "direction":
04:                     if value == "stop":
05:                         userdata["motor"].stop()
06:                         break
07:                     elif value == "forward":
08:                         userdata["motor"].forward()
09:                     elif value == "backward":
10:                         userdata["motor"].backward()
11:                 elif action == "speed":
12:                     userdata["motor"].setSpeed(value)
```

전체 구현은 다음과 같습니다.

```
import paho.mqtt.client as mqtt
from pop import TempHumi, Cds, Sound, Leds, DCMotor
import signal
import json
import threading
import time

actions = {
    "leds":Leds(),
    "motor":DCMotor()
}

client = mqtt.Client(userdata=actions)

def _do_publish_sensors(client):
    th = TempHumi()
    cds = Cds()
    sound = Sound()
    while not client.is_stop:
        thData = th.read()
        t = round(th.temp)
        h = round(th.humi)
        c = cds.readAverage()
        s = sound.readAverage()

        data = json.dumps({"temp":t, "humi":h, "cds":c, "sound":s})
        client.publish("soda/sensors", data)
        time.sleep(client.interval / 1000)

def _signal_handler(signal, frame):
    client.is_stop = True
    client.disconnect()

def _on_connect(client, userdata, flags, rc):
    client.subscribe("soda/actions")
    threading.Thread(target=_do_publish_sensors, args=(client,)).start()

def _on_message(client, userdata, message):
    payload = json.loads(message.payload)

    for key, values in payload.items():
        if key == "sensors":
            client.interval = values["interval"]
        elif key == "led":
            for pos, action in values.items():
                for cmd, value in action.items():
                    pos = int(pos)
                    if cmd == "on":
```

```
49:                         userdata["leds"][pos].on() if value else
userdata["leds"][pos].off()
50:                     elif cmd == "bright":
51:                         userdata["leds"][pos].bright(value)
52:         elif key == "motor":
53:             for action, value in values.items():
54:                 if action == "direction":
55:                     if value == "stop":
56:                         userdata["motor"].stop()
57:                         break
58:                     elif value == "forward":
59:                         userdata["motor"].forward()
60:                     elif value == "backward":
61:                         userdata["motor"].backward()
62:                 elif action == "speed":
63:                     userdata["motor"].setSpeed(value)
64:
65: def main():
66:     signal.signal(signal.SIGINT, _signal_handler)
67:     client.on_connect = _on_connect
68:     client.on_message = _on_message
69:     client.is_stop = False
70:     client.interval = 500
71:
72:     try:
73:         client.connect_async("192.168.10.2")
74:         client.loop_forever()
75:     except OSError as e:
76:         print(e)
77:
78: if __name__ == "__main__":
79:     main()
```

03: json 모듈 로드

08 ~ 11: Leds, DCMotor 객체를 딕셔너리의 요소로 만들어 actions에 대입

09: 키 "leds"에 대한 값으로 Leds 객체를 요소로 갖는 리스트 사용

10: 키 "motor"에 대한 값으로 DCMotor 객체 사용

13: Client()의 userdata 인자에 actions 전달

15 ~ 28: _do_publish_sensors()는 센서값을 발행하는 스레드

20 ~ 28: client.si_stop이 False인 동안 while 루프 실행

21 ~ 24: 측정한 온도, 습도, 조도, 평균 소음 레벨을 각각 t, h, c, s에 대입

26: 키 "temp", "humi", "cds", "sond"와 값 t, h, c, s의 딕셔너리를 JSON 객체로 만들어 data에 대입

29: JSON 포맷의 메시지를 딕셔너리로 변환해 payload에 대입

41 ~ 63: payload에서 key, values를 추출하면서 for 루프 실행

42 ~ 43: key == "sensors"이면 values["interval"]을 client.sensors_interval에 대입

44 ~ 51: key == "led"이면 values에서 하위 딕셔너리를 추출해 LED 제어

45 ~ 51: values에서 pos, action을 추출하면서 for 루프 실행

46 ~ 51: action에서 cmd과 value를 추출하면서 내부 for 루프 실행

47: pos를 정수로 변환해 pos에 대입

48 ~ 49: cmd == "on"이면 value에 따라 pos의 LED를 켜거나 끔

50 ~ 51: cmd == "bright"이면 value를 인자로 pos의 LED의 밝기 조절

52 ~ 63: key == "motor"이면 values에서 action, value를 추출해 모터 제어

53 ~ 63: values에서 action과 value를 추출하면서 for 루프 실행

54 ~ 61: action == "direction"이면 value에 따른 모터 동작 처리

55 ~ 57: value == "stop"이면 모터를 멈춘 후 for 루프 탈출

58 ~ 59: value == "forward"이면 시계 방향으로 회전

60 ~ 61: value == "backward"이면 시계 반대 방향으로 회전

62 ~ 63: action == "speed"이면 value를 인자로 모터 속도 설정

센서값 구독, 액션 발행 클라이언트

센서값 구독, 액션 발행 클라이언트 구현은 앞서 소개한 *GUI 입히기*를 수정하는데, struct 모듈 대신 json 모듈을 로드하고 on_connect() 콜백을 통해 subscribe()로 구독하는 토픽을 "soda/sensors/temphumi"에서 "soda/sensors"로 바꾸며, 의미에 맞춰 기존 interval 입력항목 이름도 actions로 바꿉니다.

```
01: import paho.mqtt.client as mqtt
02: import json
03: from appJar import gui
04:
05: client = None
06: app = None
07:
08: def _on_connect(client, userdata, flags, rc):
09:     ...
10:     client.subscribe("soda/sensors")
```

```
11:     change_bt_connect_ui(True)
12: ...
13: def main():
14:     ...
15:     with app.labelFrame(" Publish Message ", 2, 0):
16:         app.setSticky("news")
17:         app.addEntry("actions", 0, 0)
18:         app.setEntrySubmitFunction("actions", on_bt_publish)
19:         ...
```

메시지가 구조체 스타일에서 JSON으로 바뀌었으므로 수신한 메시지를 처리하는 on_message() 콜백과 인터벌을 발행하던 on_bt_publish()를 다음과 같이 수정합니다. 나머지 구현은 기존과 같습니다.

```
01: def _on_message(client, userdata, message):
02:     payload = json.loads(message.payload)
03:
04:     data = ""
05:     for key, item in payload.items():
06:         data += "%s:%4d, "%(key, item)
07:
08:     data = data[:-2]
09:     app.addListItem("recv", data)
10:
11: def on_bt_publish(unuse):
12:     data = eval(app.getEntry("actions"))
13:     msg = json.dumps(data)
14:     app.clearEntry("actions")
15:
16:     client.publish("soda/actions", msg)
```

테스트

엣지 A에 센서값 발행 액션 구독 클라이언트를 실행하고 엣지 B에 센서값 구독, 액션 발행 클라이언트를 실행하면 엣지 B는 엣지 A의 온도와 습도, 조도, 사운드 값을 실시간으로 모니터링하게 됩니다. 엣지 B의 클라이언트에서 Publish Message 엔트리를 통해 {"sensors":{"interval":100}}를 발행하면 엣지 A의 센서값 발행 속도가 0.1초 단위로 빨라지고 {"led":{0:{"on":True}}, "motor":{"direction":"forward", "speed":9}}를 발생하면 엣지 A의 LED1이 켜지고 모터가 빠르게 회전합니다.

디버깅

MQTT 디버깅은 이벤트가 발생할 때마다 로그 메시지를 출력하는 것이 가장 간단하나 필요에 따라 발행 또는 구독 전용 툴을 이용하거나 심지어 패킷 수준의 통신 흐름을 파악하기 위해 Wireshark와 같은 패킷 캡쳐 툴 사용하기도 합니다.

강제 종료 메시지

will_set()은 연결이 강제로 종료될 때 브로커가 구독자에 전송할 메시지를 등록하므로 이를 이용하면 강제 종료 상황을 모든 구독자에 알릴 수 있습니다. 만약 나중에 실행되는 클라이언트를 위해 이 메시지를 브로커에 계속 유지해야 합니다면 마지막 인자인 retain을 True로 설정합니다.

```
01: import paho.mqtt.client as mqtt
02:
03: def _on_message(client, userdata, message):
04:     print(message.topic, message.payload)
05:
06: client = mqtt.Client()
07:
08: client.on_message = _on_message
09: client.will_set("soda/except", "Force terminate by KeyboardInterrupt", 2,
True)
10:
11: client.connect("192.168.10.2")
12: client.subscribe("soda/except")
13:
14: try:
15:     client.loop_forever()
16: except KeyboardInterrupt:
17:     pass
```

로그 메시지

MQTT 클라이언트 프로그램을 작성할 때 on_log() 콜백을 등록한 후 enable_logger()를 호출하면 mosquitto_pub 또는 mosquitto_sub에서 -d 옵션을 사용한 것과 같이 내부에서 이벤트가 발생할 때마다 on_log() 콜백이 호출됩니다.

다음 예는 일반 세션에서 QoS = 2의 구독과 발행에 대한 모든 콜백 정보를 보여줍니다.

```
01: import paho.mqtt.client as mqtt
02: import datetime
03:
04: def _on_connect(client, userdata, flags, rc):
05:     print("\t_on_connect:", flags)
06:
07: def _on_disconnect(client, userdata, rc):
08:     print("\t_on_disconnect:", rc)
09:
10: def _on_message(client, userdata, message):
11:     print("\t_on_message:", message.topic, message.payload)
12:     client.is_message_complected = True
13:
14: def _on_publish(client, userdata, mid):
15:     print("\t_on_public:", mid)
16:
17: def _on_subscribe(client, userdata, mid, granted_qos):
18:     print("\t_on_subscribe:", mid, granted_qos)
19:
20: def _on_unsubscribe(client, userdata, mid):
21:     print("\t_on_unsubscribe:", mid)
22:
23: def _on_log(client, userdata, level, buf):
```

```
24:     now = datetime.datetime.now()
25:     print(now.strftime("[%M:%S:%f] "), end='')
26:
27:     print(level, buf)
28:
29: client = mqtt.Client("soda_pub", False)
30: client.enable_logger()
31:
32: client.on_connect = _on_connect
33: client.on_disconnect = _on_disconnect
34: client.on_message = _on_message
35: client.on_publish = _on_publish
36: client.on_subscribe = _on_subscribe
37: client.on_unsubscribe = _on_unsubscribe
38: client.on_log = _on_log
39:
40: client.connect("192.168.10.2")
41: client.subscribe("soda/something", 2)
42: client.publish("soda/something", "hello everyone!", 2)
43:
44: client.is_message_complected = False
45: client.loop_start()
46:
47: while not client.is_message_complected:
48:     pass
49:
50: client.unsubscribe("soda/something")
51:
52: try:
53:     while True:
54:         pass
55: except KeyboardInterrupt:
56:     client.disconnect()
```

프로그램 실행에 따른 이벤트 흐름은 다음과 같습니다.

- connect() 호출
  - CONNECT 송신
- subscribe() 호출
  - SUBSCRIBE 송신
- publish() 호출
  - PUBLISH 송신
- CONACK 수신
  - _on_connect() 콜백
- SUBACK 수신
  - _on_subscribe() 콜백
- PUBREC 수신
- PUBREL송신
- PUBCOMP수신
  - _on_public() 콜백

- PUBLISH 수신
    - PUBREC 송신
    - PUBREL 수신
    - _on_message() 콜백
    - PUBCOMP 송신
- unsubscribe() 호출
    - UNSUBSCRIBE 송신
- UNSUBACK 수신
- _on_unsubscribe() 콜백

> 킵얼라이브 시간이 만료될 때마다 반복
>
> - PINGREQ 송신
> - PINGREST 수신
> - ...

- 프로그램 강제 종료
    - disconnect() 호출
        - DISCONNECT 송신
            - _on_disconnect() 콜백

다음은 클린 세션에서 QoS = 0의 이벤트 흐름으로, 가장 큰 차이는 메시지 게시와 게시된 메시지의 수신 흐름입니다. 또한 `disconnect()`에 의한 콜백은 생략됩니다.

- connect() 호출
    - CONNECT 송신
- subscribe() 호출
    - SUBSCRIBE 송신
- publish() 호출
    - PUBLISH 송신
    - _on_public() 콜백
- CONACK 수신
    - _on_connect() 콜백
- SUBACK 수신
    - _on_subscribe() 콜백
- PUBLISH 수신
    - _on_message() 콜백
- unsubscribe() 호출
    - UNSUBSCRIBE 송신
- UNSUBACK 수신
    - _on_unsubscribe() 콜백

> 킵얼라이브 시간이 만료될 때마다 반복
>
> - PINGREQ 송신
> - PINGREST 수신
> - ...

- disconnect() 호출

    - DISCONNECT 송신

발행 및 구독 툴

MQTT 클라이언트인 mosquitto-clients를 설치하면 프로그램을 작성하지 않고도 브로커에 토픽을 발행해 메시지를 게시하거나 토픽 구독을 등록해 게시된 메시지를 가져올 수 있습니다. mosquitto_pub는 토픽을 발행하고 mosquitto_sub는 구독하는데, 몇 가지를 제외하면 대부분의 옵션은 동일합니다.

- -A bind_address: 로컬 네트워크 인터페이스가 여러 개일 때 특정 통신 인터페이스 선택
- -d: 디버그 메시지 출력
- -h hostname: 브로커 주소. 기본값은 localhost
- -p port_number: 브로커 포트 번호. 기본값은 1883
- -u username, -P passwd: 브로커 인증 ID와 패스워드
- -i client_id: 클라이언트 ID. 기본값은 클라이언트의 프로세스 ID
- -k keepalive_time: 킵얼라이브 만료 시간 설정. 기본값은 60초
- -q qos: 0 ~ 2 사이 서비스 품질 설정. 기본값은 0
- -t topic: 토픽

다음은 mosquitto_pub 전용 옵션입니다.

- -r: 유지 비트를 붙여 브로커에 마지막 게시 메시지 유지
- -l: <Enter>를 누를 때마다 표준 입력에서 읽은 메시지 전송. 빈 줄은 전송하지 않음
- -s: <Ctrl>d를 누를 때까지 표준 입력에서 읽은 전체 내용을 단일 메시지로 전송
- -n: 길이가 0인 메시지 전송
- -f file: 파일 내용 전송. '-s < file' 과 같음
- -m message: 단일 메시지 전송

mosquitto_pub 사용 예는 다음과 같습니다.

- mosquitto_pub -t soda/test "hello mqtt"
    - 로컬 브로커에 연결한 soda/test 토픽 메시지 "hello mqtt" 게시
- mosquitto_pub -q 1 -t soda/sensors/temp 27
    - QoS 1로 soda/sensors/temp 토픽 메시지 27 게시
- mosquitto_pub -r -t soda/sensors/temp 30
    - 브로커에 게시된 메시지 30 보존
- mosquitto_pub -h 192.168.10.2 -t soda/sensors/humi 32
    - 192.168.10.2 브로커에 soda/sensors/humi 토픽 메시지 32 게시
- mosquitto_pub -d -h 192.168.10.2 -t soda/actions -f my_action
    - 디버깅 메시지를 출력하면서 soda/actions 토픽에 my_action 파일 내용 게시

mosquitto_sub 전용 옵션은 다음과 같습니다.

- -c: 클린 세션 비활성화.
    - 연결이 끊어져도 구독이 유지되므로 이후 도착하는 QoS 1, 2 메시지 유지
    - 다시 연결하면 대기 중인 모든 메시지 수신
- -C msg_count: 주어진 수만큼 메시지를 수신하면 연결을 해제한 후 프로그램 종료
- -R: 유지 비트가 설정된 수신 메시지 출력 안 함
- -N: 수신한 메시지를 출력할 때 줄 바꿈 문자를 추가로 출력 안 함
    - -d 옵션을 사용하면 디버깅 메시지로 인해 효과가 없음

- -v: 수신한 메시지의 토픽과 페이로드를 함께 출력. 생략하면 페이로드만 출력
    - 다중 토픽 필터를 설정할 때 사용
- -T filter-out...: 필터와 일치하는 토픽의 수신 메시지 출력 안 함. 옵션 반복 허용
    - 다중 토픽 필터를 설정한 후 특정 토픽은 무시할 때 사용

다음은 mosquitto_sub 사용 예입니다.

- mosquitto_sub q 2 -t soda/sensors/#
    - 로컬 브로커에 연결한 후 QoS 2로 soda/sensors의 모든 토픽 구독
- mosquitto_sub -v -t soda/sensors/#
    - 수신한 메시지를 출력할 때 토픽도 함께 출력
- mosquitto_pub -R -d -t soda/sensors/#
    - 유지 비트가 설정된 메시지는 출력하지 않고 모든 디버깅 메시지 출력
- mosquitto_pub -h 192.168.10.2 -t soda/sensors/# -c
    - 192.168.10.2 브로커에 연결. 연결이 끊어져도 브로커는 세션 유지
- mosquitto_pub -h 192.168.10.2 -t soda/sensors/# -T soda/sensors/temp -T soda/sensors/humi
    - soda/sensors/temp과 soda/sensors/humi 토픽 메시지 출력 제외

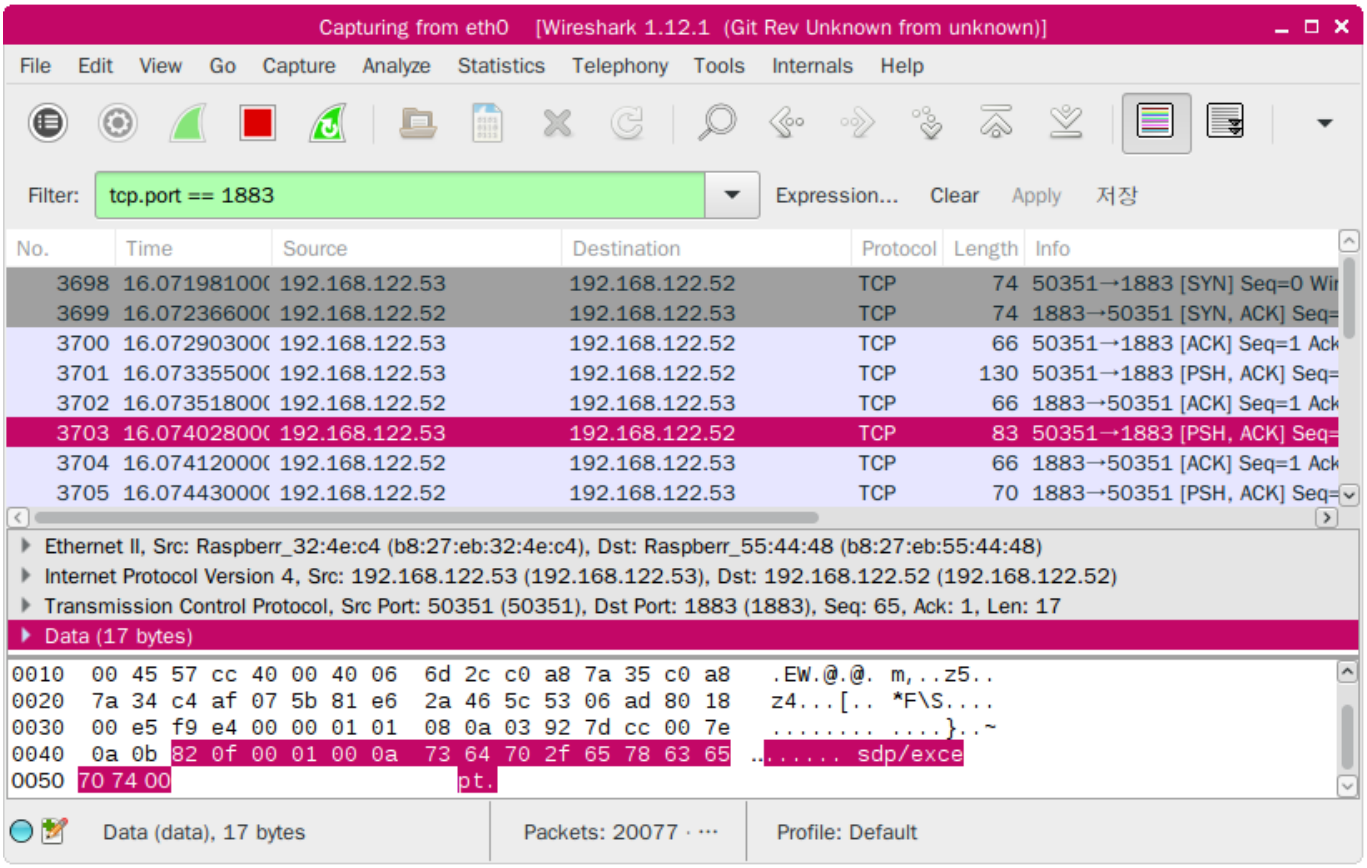
패킷 분석

TCP/IP 프로토콜 분석을 위해 주로 사용하는 Wireshark는 오픈소스로 GUI 환경에서 특정 네트워크 인터페이스를 통해 송수신되는 모든 패킷을 캡처해 사용자에게 보여줍니다. MQTT는 TCP 기반 메시지 버스 서비스이므로 Wireshark로 모든 통신 흐름을 파악할 수 있습니다.

Soda에는 다음 명령으로 Wireshark가 미리 설치되어 있습니다. 따라서 사용자는 '패널 > 보조 프로그램 > 네트워크 분석기'를 선택해 Wireshark를 실행합니다.

```
sudo apt install wireshark
```

Wireshark가 실행되면 인터페이스 목록에서 'eth0'와 같은 인터페이스를 선택한 후 도구 모음에서 'Start a new live capture' 버튼을 눌러 모니터링을 시작합니다. 브로커는 기본적으로 1883 포트 번호를 사용하므로 디스플레이 필터에 tcp.port == 1883을 설정하면 MQTT 패킷만 화면에 표시합니다.



[그림 30 Wireshark로 MQTT 패킷 캡처]