MQTT Tutorial I

by chanmin.park (devcamp@gmail.com)

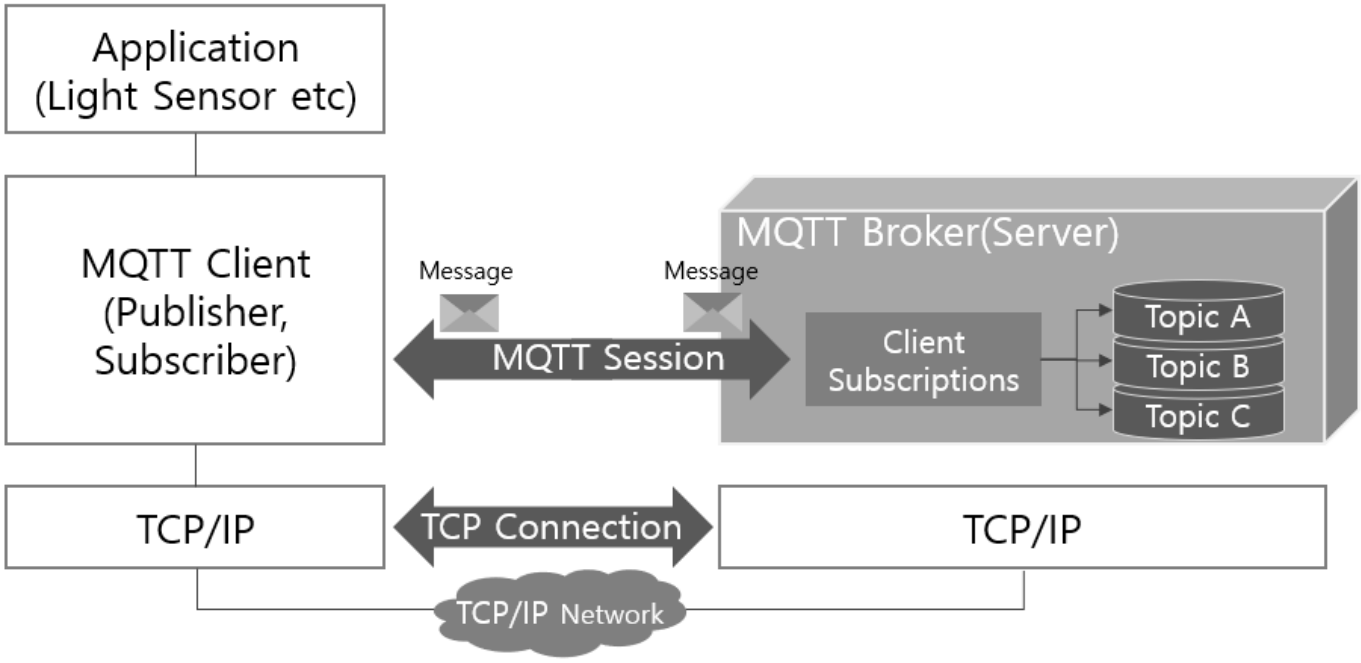
MQTT*Message Queue Telemetry Transport*는 앤디 스텐포드-클락*andy stanford-clark*(IBM)과 알렌 니퍼*arlen nipper*가 인공위성을 통해 오일 파이프라인 원격 검침 시스템을 연결하기 위해 1999년부터 설계한 것으로 2010년 로열티 없는 상태로 출시되어 2014년 OASIS*advancing open standards for the information society* 표준이 되었습니다. 이후 MQTT V3.1을 중심으로 IoT Internet of Thing 장치 사이 낮은 대역폭 통신을 제공하는 가벼운 메시징 프로토콜 중 하나로 자리매김하고 있으며 2018년 1월 최신 버전인 MQTT V5가 승인된 상태입니다.

MQTT 시스템은 중계 서버 역할의 브로커*broker*와 브로커에 토픽을 기반으로 메시지를 보내고 받는 클라이언트로 구성되는데, 클라이언트는 브로커로부터 토픽 메시지를 구독하는 구독자*subscriber*와 토픽에 메시지를 발행하는 발행자*publisher*로 나누어집니다.

[그림 1 MQTT 구성]

MQTT 개요

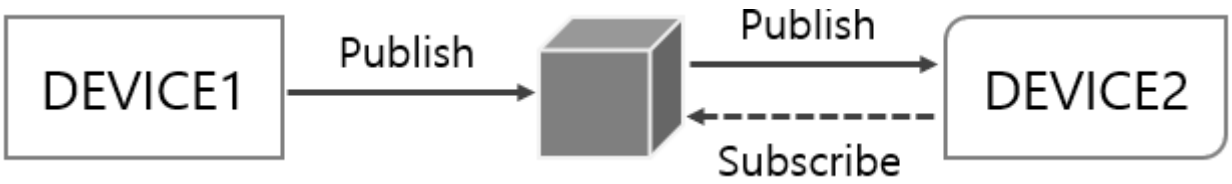
MQTT는 TCP/IP를 기반으로 한 메시지 버스 시스템으로 클라이언트(발행자, 구독자)와 서버(브로커), 세션, 서브스크립션, 토픽으로 구성됩니다. 브로커의 디폴트 포트 번호는 1883입니다.



[그림 2 MQTT 시스템 아키텍처]

MQTT 클라이언트

현재 기기가 발행자면 토픽에 대한 메시지를 발행해 브로커에 게시할 수 있고 구독자면 브로커의 특정 토픽에 가입해 메시지를 구독할 수 있습니다. 하나의 클라이언트는 구독자 또는 발행자이거나 모두일 수 있으며 브로커에 연결할 때 사용자 ID와 비밀번호 또는 인증서를 이용해 인증합니다.



[그림 3 발생과 구독]

클라이언트는 일반적으로 공개된 클라이언트 라이브러리를 이용해 구현하지만 크롬 브라우저의 확장 플러그인 MQTTlens나 리눅스 쉘 또는 윈도우 명령행에서 동작하는 mosquitto-clients처럼 미리 범용으로 구현된 제품들도 있습니다. Soda OS(이하 Soda)에는 mosquitto-client가 다음 명령으로 미리 설치되어 있습니다.

```
sudo apt install mosquitto-clients
```

mosquitto-client를 설치하면 범용 발행자인 mosquitto_pub과 범용 구독자인 mosquitto_sub를 사용할 수 있습니다.

MQTT 브로커

브로커는 일종의 서버 프로그램으로 발행자와 구독자 사이 중계자 역할만 하므로 서버 대신 브로커란 용어를 사용합니다. 브로커가 발행자로부터 메시지를 수신하면 토픽을 기반으로 이를 게시한 후 토픽에 가입한 모든 구독자에게 배포합니다. 일반적으로 게시된 메시지 중 가입자가 없는 메시지나 배포된 메시지는 제거되지만 보존 속성을 가진 메시지나 영구 세션에서 발행한 메시지는 최소한 다음 배포까지 유지됩니다.

브로커는 IBM MQ와 같은 상용 제품부터 EMQ, mosquitto와 같은 오픈소스에 이르기까지 다양한 제품들이 출시되어 있는데, Sdoa에는 mosquitto가 미리 설치되어 시스템이 부팅할 때마다 자동으로 실행됩니다. 또한 초기 설정은 익명을 허용하므로 클라이언트는 사용자 ID나 비밀번호 없이 브로커에 연결할 수 있습니다.

```
sudo apt install mosquitto
```

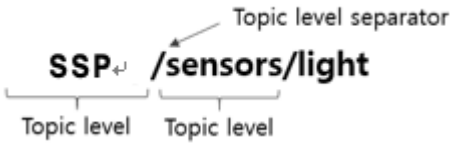
참고로 /etc/mosquitto/conf.d/mosquitto.conf에는 브로커의 보안을 높여 익명을 허용하지 않을 때를 대비한 설정이 들어 있으나 자동 실행되는 mosquittor 브로커에는 간편한 사용을 위해 적용하지 않고 있습니다. 이 파일을 이용해 mosquitto를 실행하면 클라이언트는 브로커에 연결할 때마다 반드시 /etc/mosquitto/passwd에 추가된 사용자 ID와 비밀번호를 사용해야 합니다.

토픽

토픽은 메시지에 대한 발생/구독 패턴의 기준으로 클라이언트 사이 미리 정의한 의미대로 정보를 교환할 수 있게 합니다. 대소 문자를 구분하는 계층 구조의 UTF-8 문자열로 파일 시스템의 경로와 같이 슬래시(/)로 구분되며 발행자와 브로커, 브로커와 구독자 사이 토픽에 따른 메시지 흐름을 구분합니다.

토픽 구조

$SYS 토픽을 제외하고 기본 또는 표준화된 토픽 구조는 없습니다. $SYS 토픽은 브로커에 대한 정보를 공개하기 위해 대부분의 브로커에서 예약되어 있습니다.



[그림 4 토픽]

다음 예에서 클라이언트와 브로커는 첫 번째 레벨 soda를 기준으로 8개의 토픽을 사용합니다. soda의 두 번째 레벨은 pir, cds, motor이고 pir과 cds의 다음 레벨은 각각 value이며 motor의 다음 레벨은 direction과 speed입니다.

- soda/pir/value
- soda/cds/value
- soda/motor/direction
- soda/motor/speed

토픽 생성 및 제거

발행자는 메시지를 발행할 때마다 토픽을 포함하므로 브로커는 토픽이 없으면 만든 후 게시합니다. 또한 구독자도 토픽에 가입할 때 토픽이 없으면 브로커는 만든 후 가입시킵니다. 토픽은 세션이 제거될 때 제거됩니다.

토픽 필터

발행자가 토픽에 메시지를 게시할 때는 항상 개별적인 발행만 허용하므로 두 가지 토픽에 메시지를 게시하려면 두 번 발행해야 합니다. 하지만 구독자는 여러 토픽에 가입하는 대신 토픽 필터를 적용해 한 번만 가입해도 됩니다. 토픽 필터는 단일 레벨 필터일 '+'와 다중 레벨 필터인 '#' 중 하나를 이용해 레벨 또는 이후 모든 레벨을 와일드카드wildcard로 설정합니다.

- +: 단일 레벨 와일드카드로 레벨의 모든 문자 치환
- #: 다중 레벨 와일드카드로 현재 레벨과 이후 모든 레벨의 문자 치환
    - $SYS/#: 브로커의 모든 토픽에 가입

앞의 예에서 발행자가 soda 또는 soda/pir이나 soda/pir/value 토픽 등으로 메시지를 발행하면 구독자는 일치하는 토픽 구독 대신 다음과 같이 구독 필터를 적용할 수 있습니다.

- soda/+/value: 2개의 토픽 구독
    - soda/pir/value
    - soda/cds/value
- soda/#: soda를 포함한 모든 하위 토픽 구독
    - soda
    - soda/pir
    - soda/pir/value
    - soda/cds
    - soda/cds/value
    - soda/motor
    - soda/motor/direction
    - soda/motor/speed

세션

세션은 연결된 클라이언트와 브로커 사이 상호작용을 관리하기 위한 것으로 클라이언트와 브로커는 세션 정보 중 서비스 품질을 나타내는 QoS를 통해 "적어도 한 번" 및 "정확히 한 번" 발행과 "정확히 한 번" 구독을 보장합니다. 또한 구독자가 작성한 구독도 세션 정보에 포함됩니다.

다음은 세션에 저장되는 내용입니다.

- 구독이 없는 세션의 존재 여부

- 모든 구독
- 클라이언트가 확인하지 않은 QoS = 1 또는 QoS = 2 메시지
- 클라이언트가 끊어진 동안 놓친 모든 새로운 QoS = 1 또는 QoS = 2 메시지
- 클라이언트에서 아직 확인되지 않은 모든 수신된 QoS = 2 메시지

클라이언트를 연결할 때 이전 세션 정보를 유지하거나 유지하지 않도록 선택할 수 있는데, 영구 세션(CleanSession = False)은 세션 상태를 유지하고 클린 세션(CleanSession = True)은 유지하지 않습니다.

클라이언트가 클라이언트 ID와 브로커 주소, 클린 세션 여부, 사용자 ID, 비밀번호 또는 인증서를 이용해 브로커에 연결을 요청하면 브로커는 세션 정보가 이전 연결에서 저장되어 있는지 확인합니다. 이전 세션이 존재할 때 클린 세션이면 클라언트와 브로커의 이전 세션 상태는 지워지고 영구 세션이면 이전 세션 상태를 다시 사용합니다. 이전 세션이 없으면 새 세션이 시작됩니다.

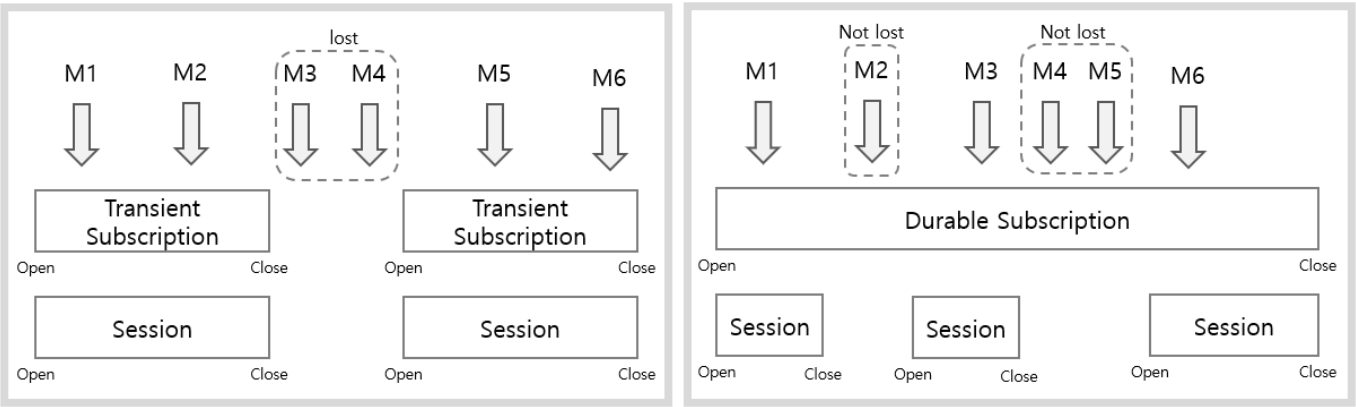
발행

클린 세션으로 클라이언트를 연결하면 보류 중인 모든 발행의 송신이 제거됩니다. 또한 클린 세션에서 QoS = 0으로 발행된 메시지는 영향을 주지 않지만 QoS = 1 및 QoS = 2로 발행한 메시지는 게시되지 않고 잃어버릴 수 있습니다.

구독

세션 상태에서 구독은 논리적으로 클라이언트의 토픽을 연결하며 토픽에 가입한 클라이언트는 게시된 메시지를 수신할 수 있습니다. 클린 세션으로 클라이언트를 연결하면 브로커는 클라이언트에 대한 이전 구독을 모두 제거합니다. 또한 연결이 끊어지면 세션 중에 클라이언트가 만든 모든 새 구독도 함께 제거됩니다. 반면에 영구 세션에서는 클라이언트가 만드는 구독이 연결되기 전에 클라이언트에 있었던 모든 구독에 추가됩니다. 클라이언트가 연결을 끊어도 모든 구독은 활성 상태를 유지합니다.

클린 세션은 기본 세션 모드로 세션 범위 내에서만 게시된 메시지를 받을 수 있으므로 구독은 일시적입니다. 따라서 세션이 닫히면 세션 상태도 지워지므로 이후 게시된 메시지는 잃어버린다. 하지만 영구 세션은 구독이 영구적이므로 클라이언트의 연결을 끊었다가 다시 연결하면 QoS = 1 또는 QoS = 2로 게시된 메시지를 받게 됩니다.



[그림 5 클린 세션과 영구 세션]

어떤 세션이든 한 번 만들어지면 전체 세션 동안 지속되므로 모드를 바꾸려면 클라이언트의 연결을 끊었다가 다시 연결하는 방법밖에 없습니다. 영구 세션을 끊고 클린 세션으로 다시 연결하면 클라이언트에 대한 모든 이전 구독 및 수신되지 않은 모든 발행이 삭제됩니다.

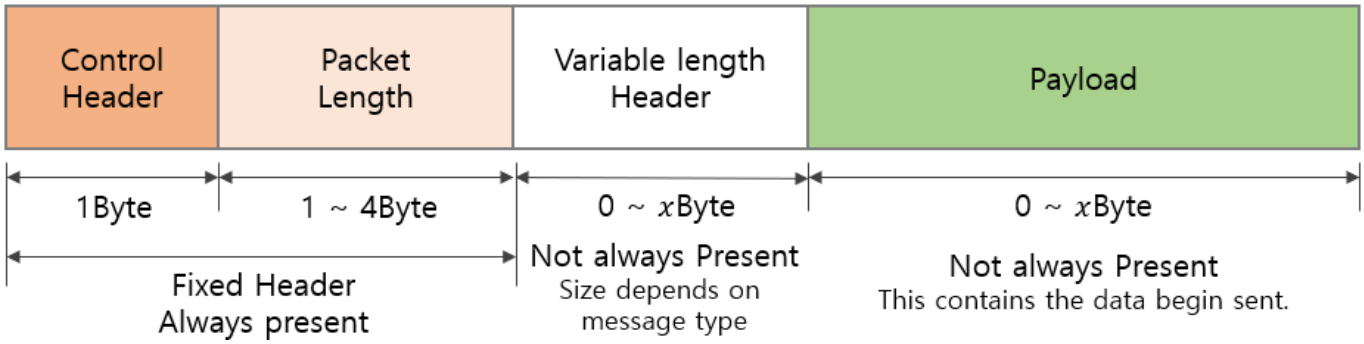
클라이언트 ID

클라이언트 ID는 브로커가 세션 안에서 각각의 클라이언트를 구분하는 유일한 식별자로 숫자와 영문자 대소문자 및 '_' 문자를 조합해 만듭니다. CleanSession = True일 때는 브로커가 동적으로 만들지만 CleanSession = False에서는 클라이언트가 반드시 유효한 문자열로 클라이언트 ID를 부여해야 하는데, 물리적으로 다른 장치에서 동일한 클라이언트 ID를 사용하면 보류 중인 게시 및 활성 구독이 자동으로 새 장치로 전송되므로 장애가 발행한 장치의 이전이 쉽습니다.

브로커가 동적으로 만드는 클라이언트 ID는 고유성이 보장되지만 클라이언트에서 명시적으로 만들 때는 브로커에 이미 등록된 클라이언트 ID인지 알 수 없으므로 128bit 고유 문자열을 생성하는 uuid 툴이나 네트워크 인터페이스의 MAC 주소 등을 사용하는 것과 같이 스스로 특별한 규칙을 적용할 필요가 있습니다.

MQTT 제어 패킷

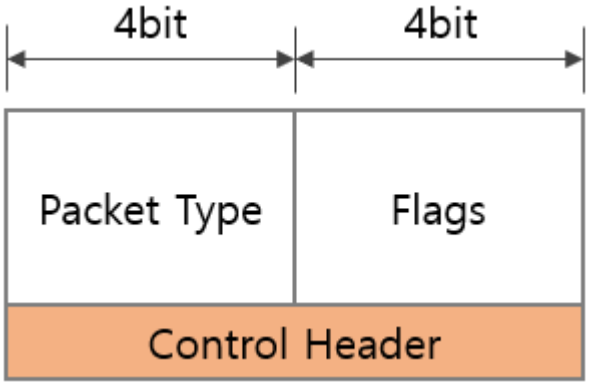
MQTT는 TCP/IP를 사용하므로 MQTT 제어 패킷은 TCP의 사용자 데이터인 페이로드에 대응합니다. MQTT 제어 패킷은 브로커와 클라이언트 사이 데이터 교환 단위로 제어 헤더(1byte)와 패킷 길이(1 ~ 4byte), 가변 길이 헤더(0 ~ xbyte), 페이로드(0 ~ xbyte)로 구성되는데, 제어 헤더와 패킷 길이는 항상 고정되고 가변 길이 헤더와 페이로드는 패킷 유형에 따라 달라집니다. 패킷의 최소 크기는 고정 헤더로만 구성된 2바이트이고, 최대 크기는 256MB입니다.



[그림 6 MQTT 메시지 패킷 구조]

제어 헤더

제어 필드로도 불리는 제어 헤더는 4비트 2개로 나눠 상위 4bit는([7:4]) 명령 또는 메시지를 구분하는 패킷 유형으로 사용하고 나머지 4bit는([3:0]) 패킷 유형에 따른 여분의 플래그로 사용합니다.



[그림 7 제어 필드 구조]

패킷 유형

메시지 유형으로도 불리는 패킷 유형은 클라이언트와 브로커 사이에 주고받는 명령 또는 메시지의 종류를 구분하는 것으로 다음과 같이 정의되어 있습니다.

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server/ Server to Client	Publish message
PUBACK	4	Client to Server/ Server to Client	Publish
PUBREC	5	Client to Server/ Server to Client	Publish received
PUBREL	6	Client to Server/ Server to Client	Publish release
PUBCOMP	7	Client to Server/ Server to Client	Publish complete
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Subscribe acknowledgment
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved

[표 1 MQTT 패킷 유형]

클라이언트가 브로커에 CONNECT 패킷은 전달해 연결을 요청하면 브로커는 CONNACK 패킷으로 응답합니다. PUBLISH 패킷은 클라이언트가 브로커에 메시지를 발행하거나, 브로커가 게시된 메시지를 배포할 때 사용하며 QoS에 따라 응답이 달라집니다.

- QoS = 0
  - 응답 패킷 없음
- QoS = 1
  - PUBACK 패킷 응답
- QoS = 2
  - 첫 번째로 PUBREC 패킷 응답
  - 상대 쪽은 PUBREL 패킷 전달
  - 마지막으로 PUBCOMP 패킷 응답

SUBSCRIBE 패킷은 하나 이상의 구독을 생성하기 위해 클라이언트에서 브로커로 전송되며 각 구독은 자신만의 QoS와 함께 클라이언트의 관심을 하나 이상의 토픽에 등록합니다. 브로커는 메시지가 게시되면 이를 배포하기 위해 클라이언트에 PUBLISH 패킷을 보냅니다. 브로커는 SUBSCRIBE 패킷을 받으면 SUBACK 패킷으로 응답합니다. UNSUBSCRIBE 패킷은 구독을 취소하기 위해 클라이언트에서 브로커로 전달되며, 브로커는 UNSUBACK 패킷으로 응답합니다.

PINGREQ 패킷은 킵얼라이브 프로토콜을 위해 클라이언트에서 브로커로 전송되며 다른 제어 패킷이 브로커로 전송되지 않을 때 클라이언트가 살아있음을 브로커에 알리거나 반대로 서버가 살아있는지 확인하기 위해 사용합니다. PINGREQ 패킷을 받은 브로커는 PINGRESP 패킷으로 응답합니다. DISCONNECT 패킷은 클라이언트의 연결 해제를 브로커에 알리는 최종 제어 패킷이며 브로커는 응답 패킷 없이 클라이언트와의 연결을 끊습니다.

플래그

플래그는 패킷 유형에 대한 추가 정보로 브로커나 클라이언트는 유효하지 않은 플래그가 수신되면 네트워크 연결을 해제합니다. PUBLISH를 제외한 모든 플래그는 다음과 같이 예약되어 있습니다.

Name	flags	bit 3	bit 2	bit 1	bit 0
CONNECT	Reserved	0	0	0	0
CONNACK	Reserved	0	0	0	0
PUBLISH	Used in MQTT v3.1.1	DUP	QoS	QoS	RETAIN
PUBACK	Reserved	0	0	0	0
PUBREC	Reserved	0	0	0	0
PUBREL	Reserved	0	0	1	0
PUBCOMP	Reserved	0	0	0	0
SUBSCRIBE	Reserved	0	0	1	0
SUBACK	Reserved	0	0	0	0
UNSUBSCRIBE	Reserved	0	0	1	0
UNSUBACK	Reserved	0	0	0	0
PINGREQ	Reserved	0	0	0	0
PINGRESP	Reserved	0	0	0	0
DISCONNECT	Reserved	0	0	0	0

[표 2 MQTT 플래그]

PUBLISH 패킷의 플래그는 DUP*duplicate delivery*, QoS(0 ~ 2), RETAIN을 나타내는데, DUP가 0이면 첫 번째 PUBLISH 패킷을 의미하고 1이면 이전 시도의 다시 전송임을 나타냅니다. RETAIN이 1이면 브로커는 이전 메시지를 지우고 현재 메시지를 보존합니다. 즉, 항상 마지막 메시지를 보존합니다. 이렇게 보존된 메시지는 나중에 구독한 클라이언트에도 배포됩니다.

패킷 길이

패킷 길이에는 가변 길이 헤더와 페이로드의 합을 저장하며 이 값이 128보다 작으면 패킷 길이는 1바이트를 사용하고 128바이트에서 16,383까지는 2바이트를 사용합니다. 가장 큰 4바이트는 2,097,152에서 268,435,455바이트 범위의 가변 길이 헤더와 페이로드의 합을 저장합니다. 따라서 실제 제어 패킷 크기는 패킷 길이 필드의 저장 값에 제어 헤더 1바이트와 패킷 길이 필드의 크기인 n바이트 더해야 합니다.

- 1 Digit: 0 (0x00) ~ 127 (0x7F)

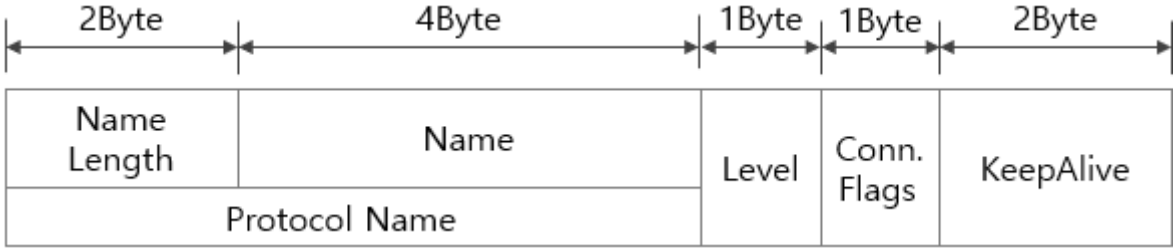
- 2 Digit: 128 (0x80, 0x01) ~ 16,383(0xFF, 0x7F)
- 3 Digit: 16,384(0x80, 0x80, 0x01) ~ 2,097,151(0xFF, 0xFF, 0x7F)
- 4 Digit: 2,097,152(0x80, 0x80, 0x80, 0x01) ~ 268,435,455(0xFF, 0xFF, 0xFF, 0x7F)

가변 길이 헤더와 페이로드

가변 길이 헤더와 페이로드는 패킷 유형에 따라 크기와 의미가 다릅니다.

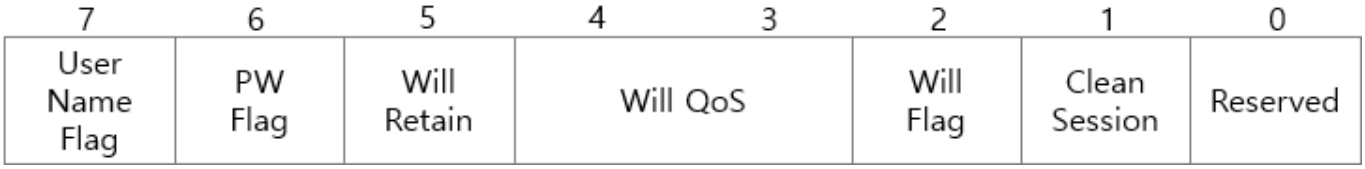
CONNECT

10바이트 크기의 가변 길이 헤더는 프로토콜 이름(6바이트), 프로토콜 수준(1바이트), 연결 플래그(1바이트), 킵얼라이브(2바이트) 순으로 구성됩니다. 페이로드는 연결 플래그에 따라 생략되거나 클라이언트 ID, 윌will 토픽, 윌 메시지, 사용자 ID, 비밀번호순으로 나열되는데, 각 데이터는 길이 접두사(2바이트)를 포함해 데이터를 크기를 구분합니다.



[그림 8 CONNECT 패킷의 가변 길이 헤더]

프로토콜 이름은 처음 2바이트가 이름에 대한 길이고 나머지는 이름입니다. 일반적으로 길이는 4바이트를 의미하는 0x00, 0x04, 이름은 0x0d('M'), 0x51('Q'), 0x54('T), 0x54('T')을 사용합니다. 프로토콜 수준은 v3.1.1 기준으로 0x04를 사용하고 연결 플래그는 비트 단위로 의미가 다릅니다.



[그림 9 CONNECT 패킷의 가변 길이 헤더 중 연결 플래그]

연결 플래그의 CS([1])는 세션의 수명과 관련된 클린 세션의 사용 여부로, CS = 1(클린 세션)이면 항상 새로운 세션을 만들어 상태를 유지하다가 연결이 끊어지면 세션 정보도 함께 제거합니다. CS = 0(영구 세션)이면 브로커는 클라이언트 ID로 구분되는 현재 세션 목록을 CONNECT 패킷의 페이로드에 포함된 클라이언트 ID와 비교해 일치하는 세션이 있으면 이를 이용해 클라이언트와의 통신을 재개하고 없으면 새로운 세션을 만듭니다. 또한 클라이언트와 브로커는 연결이 끊어질 때 세션을 저장하므로 CS = 0으로 다시 연결한 클라이언트는 종료한 동안 브로커에 게시된 QoS 1 또는 QoS 2 메시지를 수신할 수 있습니다.

다음은 클라이언트의 세션 상태입니다.

- 브로커로 전송되었지만 완전히 확인되지 않은 QoS 1, QoS 2 메시지
- 브로커부터 수신되었지만 완전히 확인되지 않은 QoS 2 메시지

다음은 브로커의 세션 상태입니다.

- 구독이 없는 세션의 존재 여부
- 클라이언트의 모든 구독
- 클라이언트에게 전송되었지만 완전히 확인되지 않은 QoS 1, QoS 2 메시지

- 클라이언트에 전송이 보류 중인 QoS 1, QoS 2 메시지
- 클라이언트로부터 수신되었지만 완전히 확인되지 않은 QoS 2 메시지
- 선택적으로, 클라이언트에 전송이 보류 중인 QoS 0 메시지

연결 플래그의 WF([2])가 1이면 페이로드에는 클라이언트의 연결이 비정상적으로 끊어질 때 브로커가 다른 클라이언트에 송신할 윌 토픽과 윌 메시지를 포함합니다. 이때 WR([5])이 1이면 윌 메시지를 지속적으로 브로커에 보존하고, WQ([4:3])는 윌 메시지의 QoS를 나타냅니다. 윌 메시지를 등록한 클라이언트가 DISCONNECT 패킷을 보내면 브로커는 저장된 세션 상태에서 윌 메시지를 제거합니다.

연결 플래그의 최상위 비트인 UF([7])와 PF([6])가 1이면 페이로드에는 사용자 ID와 비밀번호가 포함되어 있어 이를 이용해 브로커와 연결합니다.

클라이언트와 브로커의 연결 유지를 위해 전송되는 제어 패킷 사이 간격이 킵얼라이브 값을 초과하지 않도록 하는 것은 클라이언트의 책임이므로 킬얼라이브 시간이 지날때까지 다른 제어 패킷을 전송하지 않으면 클라이언트는 PINREQ 패킷을 브로커에 전송해야 합니다. PINGREQ 패킷을 보낸 후 일정 시간 내에 브로커로부터 PINGRESP 패킷을 수신할 수 없으면 네트워크 연결이 끊긴 것으로 간주합니다. 60초 킵얼라이브는 0x00, 0x3C 이지만, 0을 사용하면 연결 유지 메커니즘을 사용하지 않습니다.

페이로드의 첫 번째 필드에 위치하는 클라이언트 ID는 브로커와 클라이언트 사이 연결된 세션 상태를 식별하기 위한 것으로 UTF-8로 인코딩된 숫자와 영문자 대소문자만 허용합니다. 클라이언트 ID를 생략할 때는 반드시 CS = 1이어야 하며, 브로커는 자신이 동적으로 생성한 클라이언트 ID를 클라이언트가 전달한 것처럼 사용합니다.

CONNACK

2바이트크기의 가변 길이 헤더는 연결 확인 플래그(1바이트)와 반환 코드(1바이트로)로 구성됩니다. 페이로드는 항상 사용하지 않습니다.

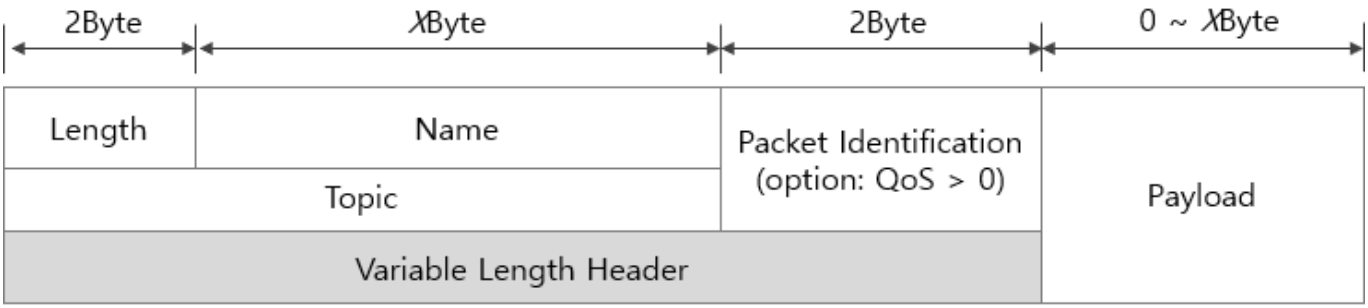
[그림 10 CONNACK 패킷의 가변 길이 헤더]

연결 확인 플래그의 [7:1]은 예약되어 있어 항상 0이고 세션의 존재session present를 나타내는 SP([0])만 사용합니다. 클린 세션으로 연결을 요청했다면 세션 존재 비트는 0이고, 클린 세션이 아니라면 클라이언트 ID와 일치하는 기존 세션 상태가 저장되어 있을 때 1, 아니면 0입니다. 반환 코드는 브로커가 연결을 허용하면 0, 아니면 원인을 나타내는 1 ~ 5사이 값을 갖습니다.

- 0x01: 프로토콜 레벨 불일치
- 0x02: 식별자가 거부됨 → CS = 0에서 클라이언트 ID를 제공하지 않음
- 0x03: 네트워크에 연결되어 있지만 브로커를 사용할 수 없음
- 0x04: 사용자 ID 또는 비밀번호 불일치
- 0x05: 클라이언트의 연결 권한이 없음

PUBLISH

가변 길이 헤더는 토픽(가변 길이)과 패킷 ID(2바이트)로 구성됩니다. 페이로드는 생략될 수 있습니다.



[그림 11 PUBLISH 패킷의 가변 길이 헤더와 페이로드]

토픽의 크기는 가변이므로 처음 2바이트에 길이, 나머지는 길이만큼 토픽을 저장합니다. 메시지 ID로도 불리는 패킷 ID는 비동기 환경에서 QoS > 0인 단위 작업을 구분하기 위해 1부터 시작하는 일련번호입니다. PUBLISH로부터 시작하는 PUBACK, PUBREC, PUBREL, PUBCOMP 패킷은 모두 하나의 단위 작업으로 이들은 PUBLISH의 패킷 ID와 같은 패킷 ID를 사용합니다. SUBSCRIBE와 SUBACK, UNSUBSCRIBE와 UNSUBACK도 하나의 단위 작업이므로 PUBLISH 단위 작업의 패킷 ID가 1이면, 이후 SUBSCRIBE 단위 작업은 2, 다시 PUBLISH 단위 작업은 3이 되는 식입니다.

QoS 0일때 이 필드는 생략되므로 페이로드는 토픽 바로 다음에 위치합니다. PUBLISH 패킷은 QoS에 따라 응답 패킷이 다른데, QoS 0은 응답 패킷이 없고 QoS 1은 PUBACK, QoS 2는 PUBREC 패킷을 사용합니다.

PUBACK

가변 길이 헤더는 패킷 ID(2바이트)로만 구성됩니다. 페이로드는 사용하지 않습니다. PUBLISH 패킷을 보낸 쪽은 동일한 식별자의 PUBACK 패킷을 받을 때 수신이 확인된 것으로 간주합니다. 받은 쪽은 수신되었음을 알리기 위해 PUBLISH 패킷의 식별자를 포함하는 PUBACK 패킷을 응답으로 보냅니다. 이후부터 받은 쪽은 DUP 설정과 관계없이 패킷 ID가 같은 PUBLISH 패킷을 항상 새로 게시합니다.

PUBREC

가변 길이 헤더는 패킷 ID(2바이트)로만 구성됩니다. 페이로드는 사용하지 않습니다. PUBLISH 패킷을 보낸 쪽은 PUBRECpublic received 패킷을 받으면 반드시 PUBRELpublic release 패킷을 보내야 하고 최종적으로 PUBCOMPpublic complete 패킷을 받아야 수신이 확인된 것으로 간주합니다.

PUBLISH 패킷을 받은 쪽은 PUBLISH 패킷의 식별자를 포함하는 PUBREC 패킷을 응답으로 보낸다음 PUBREL 패킷을 받을 때까지 PUBREC 패킷을 보냄으로써 동일한 패킷 ID를 가진 PUBLISH 패킷이 다시 보내지지 않도록 막습니다.

PUBREL

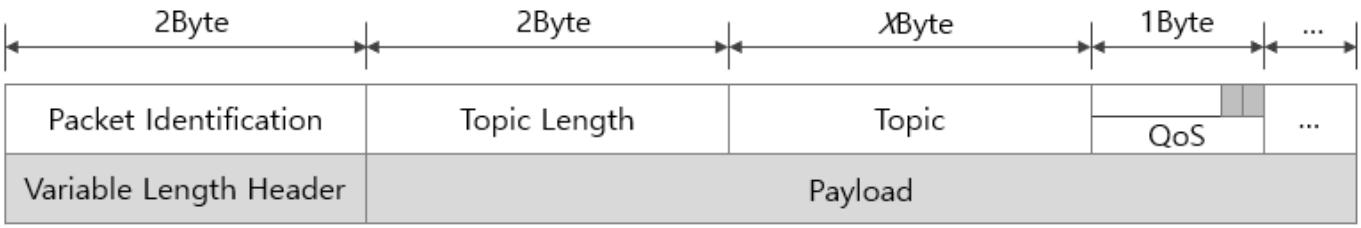
가변 길이 헤더는 패킷 ID(2바이트)로만 구성됩니다. 페이로드는 사용하지 않습니다.

PUBCOMP

가변 길이 헤더는 패킷 ID(2바이트)로만 구성됩니다. 페이로드는 사용하지 않습니다.

SUBSCRIBE

가변 길이 헤더는 패킷 ID(2바이트)로만 구성되며 페이로드에는 UTF-8로 인코딩된 구독할 토픽 또는 토픽 필터의 길이(2바이트)와 내용(N바이트), QoS(1바이트 중 하위 2bit)으로 구성된 목록이 포함됩니다.

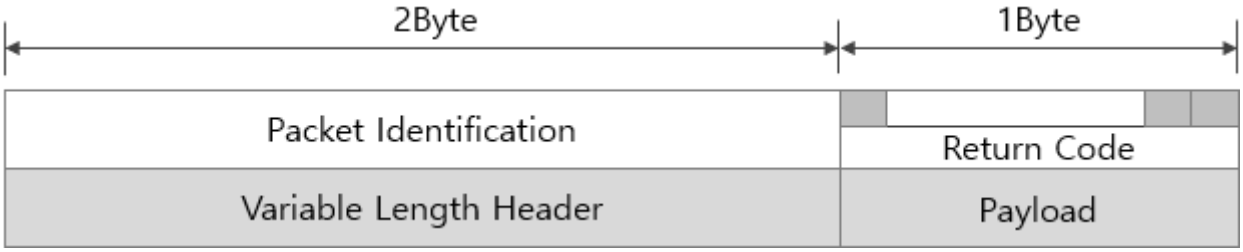


[그림 12 SUBSCRIBE 패킷의 가변 길이 헤더와 페이로드]

브로커는 이미 등록된 토픽 또는 토픽 필터를 다시 수신하면 기존 구독을 새로운 구독으로 완전히 대체하며 보존된 메시지도 다시 보냅니다.

SUBACK

가변 길이 헤더는 패킷 ID(2바이트)로만 구성되며 페이로드에는 반환 코드(1바이트)들이 포함됩니다.



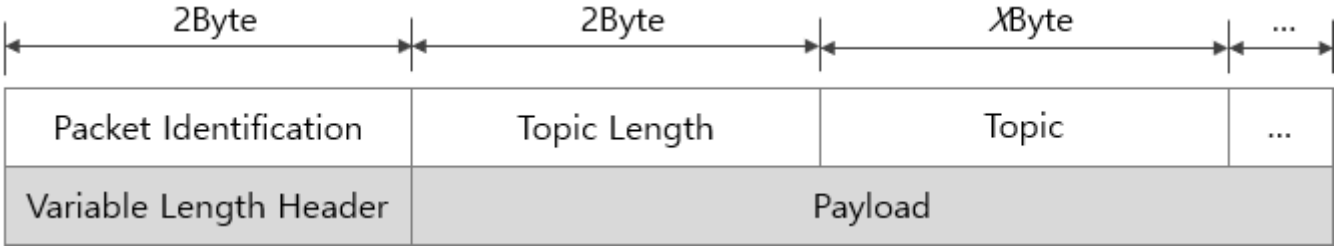
[그림 13 SUBACK 패킷의 가변 길이 헤더와 페이로드]

SUBACK 패킷은 SUBSCRIBE 패킷과 동일한 패킷 ID를 가지며 반환 코드는 실제 브로커가 부여한 QoS와 오류 코드 중 하나로 SUBSCRIBE 패킷의 토픽 순서와 일치합니다.

- 0x00: 성공. QoS 0
- 0x01: 성공. QoS 1
- 0x02: 성공. QoS 2
- 0x80: 실패

UNSUBSCRIBE

가변 길이 헤더는 패킷 ID만 포함합니다. 페이로드는 취소할 토픽 또는 토픽 필터의 길이(2바이트)와 내용(N 바이트)입니다.



[그림 14 UNSUBSCRIBE 패킷의 가변 길이 헤더와 페이로드]

브로커는 UNSUBSCRIBE 패킷에 포함된 페이로드를 자신이 보유하고 있는 토픽 필터 집합과 비교해 일치하면 서브스크립션을 삭제하고 그렇지않으면 무시합니다. 이때 송신하기 시작한 QoS 1 또는 QoS 2 메시지가 있다면 이는 정상적으로 배달됩니다.

UNSUBACK

가변 길이 헤더에는 패킷 ID만 포함하고 페이로드는 사용하지 않습니다. 이때 패킷 ID는 UNSUBSCRIBE 패킷과 같습니다.

PINGREQ

가변 길이 헤더와 페이로드를 사용하지 않습니다.

PINGRESP

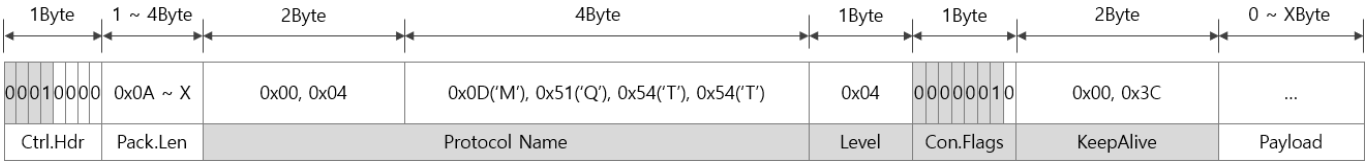
가변 길이 헤더와 페이로드는 사용하지 않습니다.

DISCONNECT

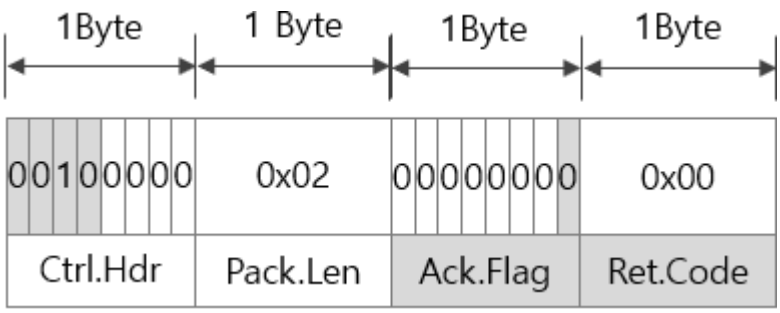
가변 길이 헤더와 페이로드는 사용하지 않습니다. 브로커는 이 패킷을 받으면 클라이언트의 연결을 끊지만 이 패킷 없이 연결 끊김을 감지하면 등록된 윌 메시지가 있을 경우 이를 PUBLISH로 전송합니다.

MQTT 패킷 정리

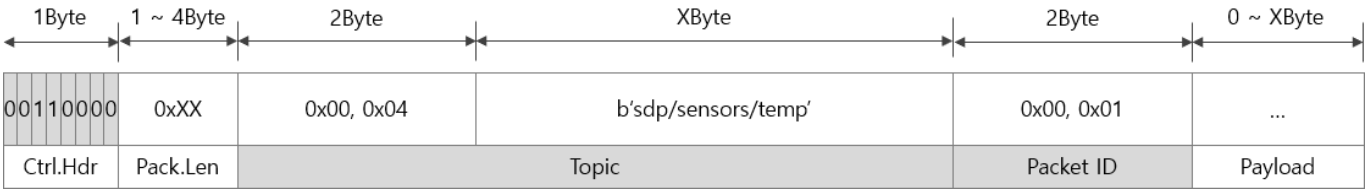
지금까지 설명한 MQTT 패킷을 정리하는 다음과 같습니다. 이러한 패킷 구조의 이해는 Wireshark와 같은 네트워크 모니터링 툴로 실제 통신 중인 MQTT 패킷을 캡처해 해석할 때 필요합니다.



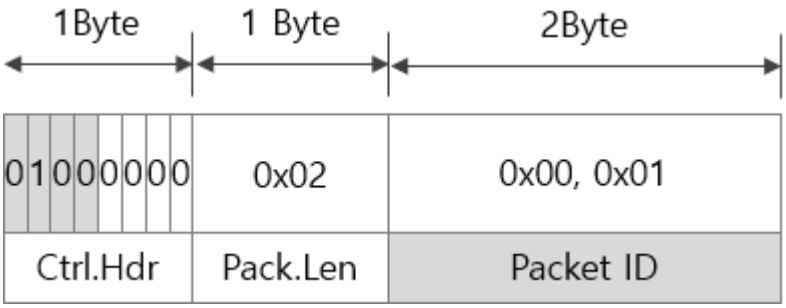
[그림 15 CONNECT 패킷]



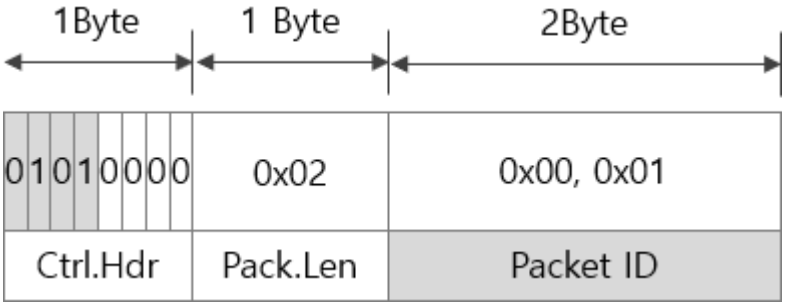
[그림 16 CONNACK 패킷]



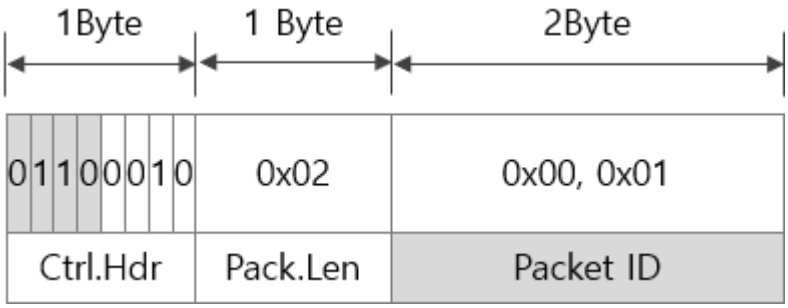
[그림 17 PUBLISH 패킷]



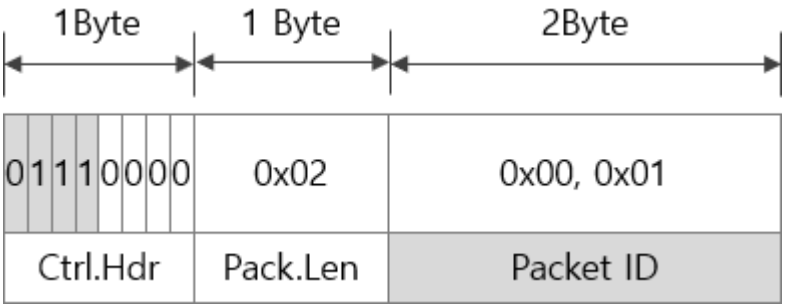
[그림 18 PUBACK 패킷]



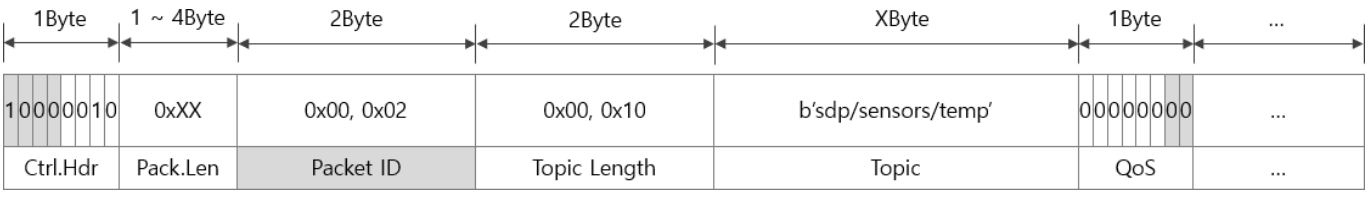
[그림 19 PUBREC 패킷]



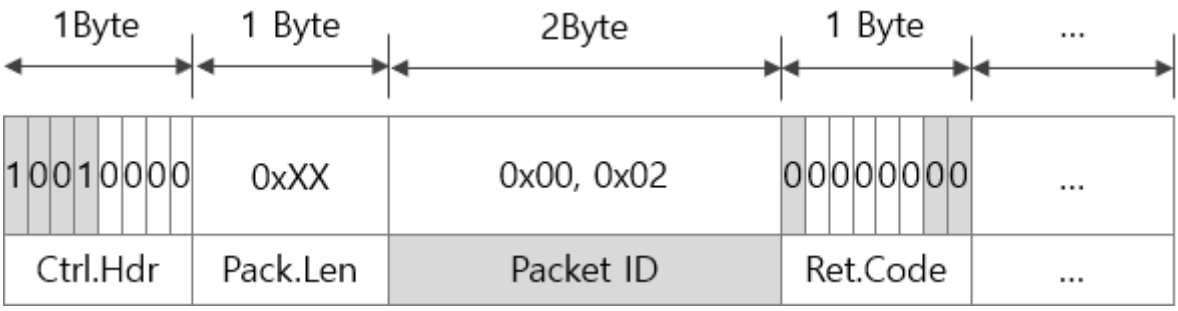
[그림 20 PUBREL 패킷]



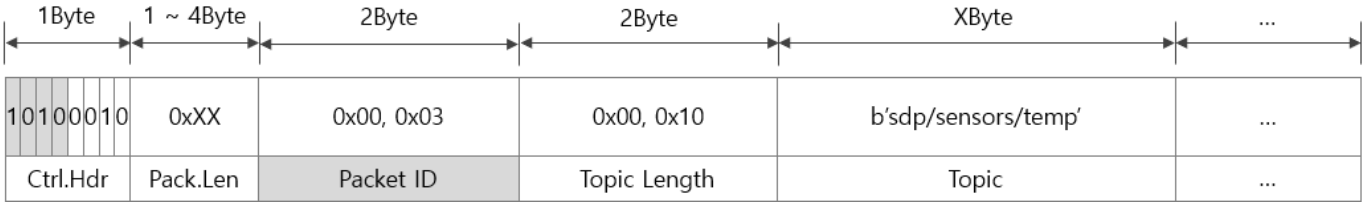
[그림 21 PUBCOMP 패킷]



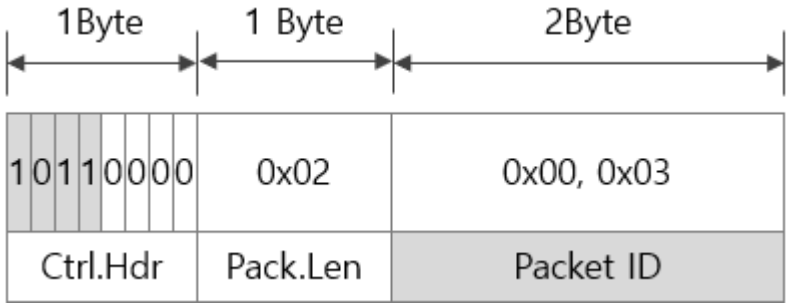
[그림 22 SUBSCRIBE 패킷]



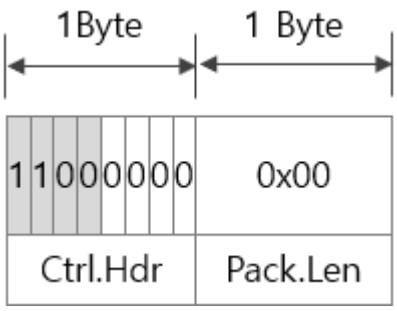
[그림 23 SUBACK 패킷]



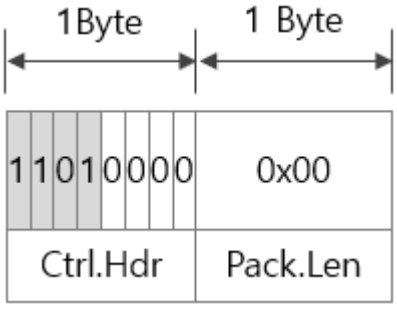
[그림 24 UNSUBSCRIBE 패킷]



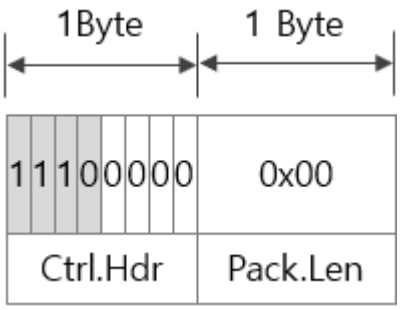
[그림 25 UNSUBACK 패킷]



[그림 26 PINGREQ 패킷]



[그림 27 PINGRESP 패킷]



[그림 28 DISCONNECT 패킷]

MQTT 클라이언트 라이브러리

대부분의 브로커는 MQTT 클라이언트 라이브러리를 함께 제공하므로 MQTT 클라이언트 라이브러리의 종류는 브로커만큼 다양합니다. 그중 IBM으로부터 시작된 Paho API는 Java부터 C, 파이썬에 이르기까지 다양한 언어를 지원하는 오픈소스 MQTT 클라이언트 라이브러리로, 동기/블로킹 및 비동기 API를 지원합니다. 동기/블로킹 기능을 사용하면 MQTT 로직을 쉽고 간결하게 구현할 수 있고, 비동기 기능은 고성능 MQTT 클라이언트를 작성할 수 있도록 높은 처리량을 제공합니다.

Paho API는 콜백 기반이며 사용자 정의 비즈니스 로직을 다른 이벤트(예: 메시지 수신시 또는 브로커 연결이 끊어진 경우)에 연결합니다. 또한 TLStransport layer security 기반 보안 통신을 비롯해 표준 MQTT v3.1 기능을 모두 지원합니다. Soda에는 다음 명령으로 파이썬용 Paho 라이브러리가 미리 설치되어 있습니다.

```
sudo pip3 install paho-mqtt
```

파이션용 Paho API는 MQTT 클라이언트 구현을 지원하는 client 모듈 외에 구독자와 발행자를 좀 더 쉽게 구현할 수 있도록 publish 모듈과 subscribe 모듈을 함께 제공합니다. 일반적으로 사용자는 다음과 같은 방법으로 client 모듈을 로드합니다.

```
01: import paho.mqtt.client as mqtt
```

client 모듈

client 모듈의 Client 클래스는 MQTT 클라이언트의 모든 기능을 지원하는데, 클라이언트 객체가 만들어지면 `connect*()` 중 하나를 사용해 브로커에 연결한 다음 `loop*()` 중 하나로 브로커와의 연결을 유지합니다. `subscribe()`는 토픽을 구독하고 `publish()`는 토픽에 메시지를 발행하며 `disconnect()`는 브로커와의 연결을 끊습니다.

클라이언트 객체 생성 및 다시 초기화

처음 `Client()`를 호출하면 세션 모드가 설정된 MQTT 클라이언트 객체가 만들어집니다. 첫 번째 인자인 client_id를 생략하면 브로커에서 클라이언트 ID를 만들지만, 영구 세션을 만들기 위해 두 번째 인자인 clean_session에 False를 전달할 때는 반드시 유효한 문자열을 설정해야 합니다. 필요에 따라 `reinitialize()`를 호출해 클라이언트 객체를 다시 초기화할 수 있습니다.

- Client(client_id="", clean_session=True, userdata=None, protocol=MQTTv311, transport="tcp")
  - 세션 모드가 설정된 클라이언트 객체 생성
  - client_id: 브로커에 연결할 때 사용할 클라이언트 ID. 기본값은 브로커 자동 생성
  - clean_session = False이면 반드시 설정해야 함

    - 유일한 UTF-8 문자열로 만들어야 함
    - clean_session: 세션 모드 지정
    - True: 클린 세션
    - False: 영구 세션. 반드시 client_id 설정 필요
    - userdata: 콜백 함수가 호출될 때 함께 전달할 사용자 데이터
    - protocol: 사용할 프로토콜 버전. MQTTv31 또는 MQTTv311(기본값) 사용
    - transport: 전송 계층 선택
    - "websockets": 웹 소켓
    - "tcp": 기본값으로 TCP 소켓
- reinitialise(client_id="", clean_session=True, userdata=None)
    - 클라이언트 객체 다시 초기화
    - 브로커에 연결된 상태라면 리셋을 전달해 즉시 연결을 끊음

다음은 클라이언트 객체를 만드는 예입니다. 인자를 사용하지 않으면 클린 세션을 사용하며 브로커에서 자동으로 클라이언트 ID를 생성합니다.

```
01: import paho.mqtt.client as mqtt
02:
03: client = mqtt.Client()
04: ...
05: client.reinitialise()
```

옵션 기능

옵션 기능은 클라이언트 객체의 동작을 수정하며 대부분은 `connect*()`로 브로커에 연결하기 전에 적용해야 합니다. `username_pw_set()`을 이용해 연결할 브로커의 사용자 ID와 비밀번호를 설정하는 것은 브로커가 익명을 허용하지 않을 때만 필요합니다.

- max_infcds_messages_set(infcds): 최대 메시지 수 설정
    - infcds: 기본값은 20
        - 늘리면 처리량은 증가하지만 더 많은 메모리 소비
- max_queued_messages_set(queue_size): 보내는 메시지를 보관할 최대 수 설정
    - queue_size: 기본값은 0으로 무제한
        - 가득 차면 이후 모든 보내는 메시지는 버려짐
- message_retry_set(retry): 브로커의 응답이 없을 때 다시 시도할 시간 설정
    - retry: 기본값인 5초 권장
- ws_set_options(path="/mqtt", headers=None): 웹 소켓 옵션 설정
    - 클라이언트 객체를 생성할 때 transport가 "websockets"일 때만 유효
    - path: 브로커에서 사용할 MQTT 경로
    - headers: 표준 웹 소켓 헤더에 추가할 헤더 목록
- tls_set(ca_certs=None, certfile=None, keyfile=None, cert_reqs=ssl.CERT_REQUIRED, tls_version=ssl.PROTOCOL_TLS, ciphers=None)
    - 네트워크 비밀번호화 및 인증 옵션 설정. SSL/TLS 지원
    - ca_certs: 신뢰할 수 있는 인증 기관의 인증서 파일에 대한 문자열 경로
    - certfile, keyfile: PEM으로 인코딩된 클라이언트 인증서와 키 파일 경로
        - TLS 기반 인증을 위해 사용하며 파일이 비밀번호화되어 있다면 명령줄에서 비밀번호 요구

    - cert_reqs: 클라이언트가 브로커에 요구하는 인증서 요구 사항
        - 기본값인 ssl.CERT_REQUIRED은 브로커가 인증서(certfile, keyfile) 제공
        - ssl pydoc 참조
    - tls_version: 사용할 SSL/TLS 프로토콜 버전. 기본적으로 가장 높은 TLS 버전 감지
    - ciphers: 연결에 대해 허용할 문자열 타입의 비밀번호 설정
        - ssl pydoc 참조
- tls_set_context(context=None): SSL/TLS 지원을 위한 인증 컨텍스트 설정
    - context: ssl.create_default_context()에 의해 생성된 ssl.SSLContext 객체
- enable_logger(logger=None): 파이썬 로깅 패키지를 이용해 로깅 활성화
    - logger: 로깅할 logging.Logger 객체. 기본값은 자동 생성. 로깅 수준은 다음과 같음
        - MQTT_LOG_ERROR: logging.ERROR
        - MQTT_LOG_WARNING: logging.WARNING
        - MQTT_LOG_NOTICE: 항목 없음. logging.INFO로 대체
        - MQTT_LOG_INFO: logging.INFO
        - MQTT_LOG_DEBUG: logging.DEBUG
- disable_logger(): 로깅 비활성
- username_pw_set(username, password=None): 브로커 인증 설정
    - username, password: 사용자 이름과 비밀번호
- user_data_set(userdata): 콜백 함수가 호출될 때 함께 전달할 사용자 데이터 설정
- will_set(topic, payload=None, qos=0, retain=False): 클라이언트가 강제 종료될 때 전달할 메시지
    - topic: 토픽 문자열
    - payload: 메시지
        - int 또는 float 타입 인자는 내부에서 바이트 배열로 변환
        - 구조체 타입은 미리 struct.pack()으로 변환해 적용
    - qos: 0 ~ 2 사이 서비스 품질 수준. 기본값은 0
        - 0: 최대 1회 전송
            - 구독자의 메시지 수신을 보장하지 않음
        - 1: 최소 1회 전송
            - 구독자의 메시지 수신이 불확실하면 재전송
            - 메시지 중복이 발생할 수 있음
        - 2: 정확히 1회 전송
            - 구독자 정확히 메시지를 한 번 수신할 수 있도록 보장
            - retain: True이면 마지막으로 전달한 메시지 보관
- reconnect_delay_set(min_delay=1, max_delay=120): 다시 연결 시간 설정
    - 네트워크 장애 등으로 연결이 끊어질 때 이 설정을 참조해 다시 연결 시도
    - min_delay: 최소 지연 시간. 기본값은 1초
    - max_delay: 최대 지연 시간. 기본값은 60초

연결, 해제 및 다시 연결

브로커에 메시지를 게시하거나 배포된 메시지를 수신하려면 `connect*()` 중 하나를 사용해 브로커에 연결해야 합니다. `disconnect()`는 연결을 명시적으로 해제하고 `reconnect()`는 연결이 해제된 상태에서 이전 정보를 이용해 다시 연결합니다. connect()는 블로킹 모드로 동작하므로 유효하지 않은 브로커 주소를 인자로 전달하면 일정 시간 프로그램이 블로킹 되지만 `connect_async()`는 비동기 모드로 동작하므로 블로킹 되지 않습니다. `connect*()`와 `disconnect()`는 동작이 완료되면 `on_connect()`와 `on_disconnect()` 콜백을 호출해 응용프로그램에 이 사실을 알립니다.

- connect(host, port=1883, keepalive=60, bind_address=""): 동기 방식으로 브로커에 연결
  - host: 문자열 타입의 연결할 브로커 주소
  - port: 연결할 브로커의 포트 번호. 기본값은 1883
  - keepalive: 킵얼라이브 만료시간. 기본값은 60초
  - bind_address: 로컬 인터페이스가 여러 개일 때 클라이언트를 바인드하는 인터페이스 주소
  - 연결이 완료된 후 on_connect() 콜백이 존재하면 호출
- connect_async(host, port=1883, keepalive=60, bind_address=""): 비동기 연결
  - loop_start()와 함께 사용. loop_start()가 호출될 때까지 연결이 완료되지 않음
  - 연결이 완료된 후 on_connect() 콜백이 존재하면 호출
- connect_srv(domain, keepalive=60, bind_address=""): DNS 질의와 함께 연결
  - domain: 브로커의 도메인 이름
  - 연결이 완료된 후 on_connect() 콜백이 존재하면 호출
- reconnect(): 이전 정보로 다시 연결
  - 이 전 connect*() 호출이 있어야 함
- disconnect(): 연결 해제
  - will 메시지를 보내지 않음. 대기 중인 메시지는 버려짐
  - 연결이 해제된 후 on_disconnect() 콜백이 존재하면 호출

다음은 클라이언트를 브로커에 연결한 후 해제하는 예입니다.

```
01: client = mqtt.Client()
02:
03: client.connect("192.168.10.2")
04: ...
05: client.disconnect()
```

네트워크 루프

브로커에 연결 요청이 완료되거나 발행한 메시지가 수신될 때마다 클라이언트 객체는 콜백을 통해 이를 응용프로그램에 알립니다. 심지어 발행한 메시지가 토픽에 게시되어도 콜백이 호출되는데, 이는 사용자를 대신해 네트워크 상태를 감시하는 네트워크 루프가 실행 중이기 때문입니다.

`loop_start()`는 스레드를 만들어 백그라운드에서 네트워크 루프를 실행하므로 즉시 반환되지만 `loop_forever()`는 내부 while 루프를 실행하므로 `disconnect()`를 호출할 때까지 반환되지 않습니다.

- loop(timeout=1.0, max_packets=1): 메시지 송/수신을 위해 select() 기반 네트워크 루프 실행
  - 한 번만 실행되므로 지속적으로 루프를 실행하려면 for나 while 필요.
  - timeout: 킵얼라이브 미만의 최대 프로그램 블로킹 시간. 기본값은 1초
    - 초과하면 브로커에 의해 정기적으로 연결이 끊어짐
  - max_packets: 사용하지 않으므로 설정하지 말 것
- loop_start(): 작업 스레드를 만들어 지속적으로 loop() 실행
- loop_stop(force=False): loop_start()로 실행한 스레드 중지
  - force: 현재 무시됨
- loop_forever(timeout=1.0, max_packets=1, retry_first_connection=False)
  - select() 기반 내부 while 루프에서 지속적으로 loop() 실행
  - disconnect()가 호출될 때까지 반환되지 않음
  - 자동으로 다시 연결 처리

        - reconnect_delay_set()으로 다시 연결 시간 설정 허용
    - retry_first_connection: 첫 번째 연결 실패 때 동작 설정
        - True: connect_async()에서 첫 번째 연결이 실패하면 다시 연결 시도

다음과 같이 loop_forever()로 네트워크 루프를 실행하면 loop_forever() 이후 명령은 disconnect()를 호출할 때까지 실행되지 않습니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     print(flags, rc)
03:     client.disconnect()
04:
05: client = mqtt.Client()
06: client.on_connect = _on_connect
07: client.connect("192.168.10.2")
08:
09: client.loop_forever()
10: print("good-bye")
```

메시지 발행

클라이언트가 publish()로 메시지를 발행하면 브로커는 이를 토픽에 게시합니다. publish()의 반환 객체인 MQTTMessageinfo를 이용해 is_published()를 호출하면 메시지 전송 상태를 알려주고 wait_for_publish()는 메시지가 전송될 때까지 응용프로그램을 대기 상태로 만듭니다.

- publish(topic, payload=None, qos=0, retain=False): 브로커에 메시지 발행
    - payload: 토픽에 게시할 메시지. 최대 268,435,455바이트
        - O 또는 None이면 길이가 0인 메시지 사용
        - 문자열 또는 int, float는 내부에서 바이트 배열로 변환
    - retain: True이면 브로커에 마지막 메시지 보존. 기본값인 False는 보존하지 않음
        - MQTTMessageInfo 객체 반환
            - rc: 결과 코드
                - 0 (MQTT_ERR_SUCCESS): 전송 성공. → 브로커에 게시된 것을 확인하지 않음
                - 4 (MQTT_ERR_NO_CONN): 전송 실패
                - 15 (MQTT_ERR_QUEUE_SIZE): 전송 큐가 가득 차 메시지가 버려짐
            - mid: 메시지 ID
            - is_published(): 메시지가 전송되었으면 True 반환. 아니면 False
            - wait_for_publish(): 메시지가 전송될 때까지 대기
        - 발행한 후 on_publish() 콜백이 존재하면 호출

다음은 측정한 센서값들을 토픽에 게시하는 예입니다. 무효한 토픽 문자열이나 QoS, 메시지 길이 초과 등은 ValueError 예외를 발생시킵니다. 실제 브로커에 게시되는 메시지의 타입은 바이트 배열입니다.

```
01: th = TempHumi(4)
02: client = mqtt.Client()
03:
04: client.connect("192.168.10.2")
05:
```

```
06: while True:
07:     data = th.read()
08:     temp = round(data.temp)
09:     humi = round(data.humi)
10:     try:
11:         mi = client.publish("soda/sensors/temp", temp)
12:         mi.wait_for_publish()
13:         mi = client.publish("soda/sensors/humi", humi)
14:         mi.wait_for_publish()
15:     except ValueError:
16:         Break
```

메시지 구독

브로커가 배포하는 메시지를 수신하려면 토픽을 구독해야 하는데, subscribe()는 구독할 토픽에 가입하고 unsubscribe()는 취소합니다. 브로커는 토픽에 메시지가 게시될 때마다 구독에 가입한 모든 클라이언트에 메시지를 배포하는데, 이를 수신하려면 반드시 on_message() 콜백이 필요합니다.

- subscribe(topic, qos=0): 하나 이상의 토픽 구독
  - topic과 qos를 튜플로 묶어 전달할 수 있음
    - subscribe(("soda/sensor/temp", 0))
  - 두 개 이상의 구독은 튜플을 요소로 갖는 리스트 전달
    - subscribe([("soda/sensor/temp", 0), ("soda/sensor/humi", 2)])
  - rc, mid를 튜플로 묶어 반환. 토픽이나 QoS가 유효하지 않으면 ValueError 예외 발생
  - 구독에 가입한 후 on_subscribe() 콜백이 존재하면 호출
- unsubscribe(topic): 하나 이상의 토픽 구독 취소
  - 구독 취소가 완료된 후 on_unsubscribe() 콜백이 존재하면 호출

여러 개의 토픽에 가입할 때는 subscribe()를 여러 번 호출해야 하는데, 이를 튜플 요소의 리스트로 만들면 subscribe()를 한 번만 호출할 수 있습니다.

```
01: client = mqtt.Client()
02:
03: client.connect("192.168.10.2")
04: client.subscribe([("soda/sensors/temp", 0), ("soda/sensors/humi", 0)])
```

콜백

네트워크 루프는 사용자를 대신해 네트워크 상태를 감시하다가 이벤트가 발행하면 이를 콜백으로 응용프로그램에 알립니다. 따라서 호출되는 콜백은 이벤트의 종류에 따라 다릅니다. on_connect()는 CONNACT 응답 패킷이 수신되면 호출되고 on_disconnect()는 DISCONNECT 패킷을 송신한 후 호출됩니다. on_message()는 PUBLISH 패킷이 수신될 때마다 호출되고 on_publish()는 PUBLISH 패킷을 송신한 후 호출됩니다. on_subscribe()와 on_unsubscribe()는 각각 SUBACK와 UNSUBACK 패킷이 수신되면 호출됩니다. 마지막으로 on_log()는 enable_logger()를 호출한 후 내부 이벤트가 발생할 때마다 호출됩니다.

- on_connect(client, userdata, flags, rc): CONNACK 패킷이 수신되면 호출
  - connect()로 송신한 CONNECT 패킷의 응답

        - client: Client()로 생성한 클라이언트 객체
        - userdata: Client() 또는 user_data_set()으로 설정한 사용자 데이터
        - flags: 브로커가 보낸 딕셔너리 타입 응답 플래그
            - flags["session present"]: 영구 세션일 때 현재 세션 상태 반환
                - 0: 새로운 세션. 1: 기존 세션
        - rc: 결과 코드
            - 0: 성공
            - 1: 프로토콜 버전 문제로 실패
            - 2: 잘못된 클라이언트 ID로 실패
            - 3: 브로커를 사용할 수 없으므로 실패
            - 4: 잘못된 사용자 ID 또는 비밀번호로 실패
            - 5: 권한 없음므로 실패
- on_disconnect(client, userdata, rc): disconnect()로 DISCONNECT 패킷을 송신한 후 호출
    - 브로커와의 연결이 끊어진 상태이므로 client는 다시 연결하는 용도로만 사용
    - 통신 장애 등으로 연결이 끊어져도 호출됨
- on_message(client, userdata, message): PUBLISH 패킷이 수신될 때 호출
    - subscribe()로 가입한 토픽의 메시지 수신
    - message: 수신한 메시지를 MQTTMessage 객체로 전달
        - topic: 토픽
        - payload: 메시지
        - qos: 메시지의 QoS
        - retain: 지속 데이터 여부
- on_publish(client, userdata, mid): publish()로 PUBLISH 패킷을 송신한 후 호출
    - 유사한 MQTTMessageInfo.wait_for_publish()는 동기 모드
    - mid: 메시지 ID 또는 패킷 ID
- on_subscribe(client, userdata, mid, granted_qos): SUBACK 패킷을 수신하면 호출
    - subscribe()로 송신한 SUBSTRIBE 패킷의 응답
    - granted_qos: 다중 구독 요청에 대한 QoS 목록
- on_unsubscribe(client, userdata, mid): UNSUBACK 패킷을 수신하면 호출
    - unsubscribe()로 송신한 UNSUBSCRIBE 패킷의 응답
- on_log(client, userdata, level, buf): 네트워크 이벤트가 발생할 때마다 호출
    - enable_logger()와 disable_logger()로 활성화, 비활성화
    - level: MQTT_LOG_INFO, _NOTICE, _WARNING, _ERR, _DEBUG 중 하나
    - buf: 로그 메시지

클라이언트 객체의 초기 콜백 속성들은 빈 상태이므로 사용자 함수를 대입해 사용합니다. 다음은 메시지를 구독하는 예로 on_connect() 콜백이 호출되면 rc를 확인해 브로커 연결이 성공이면 토픽에 가입한 후 on_message() 콜백으로 수신된 메시지를 출력합니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     if rc == 0:
03:         client.subscribe([("soda/sensors/temp", 0), ("soda/sensors/humi",
0)])
04:
05: def _on_message(client, userdata, message):
06:     print(message.topic, message.payload, message.qos, message.retain)
07:
```

```
08: mqttc = mqtt.Client()
09:
10: mqttc.on_connect = _on_connect
11: mqttc.on_message = _on_message
12:
13: mqttc.connect("192.168.10.2")
14:
15: mqttc.loop_forever()
```

publish()로 발행한 메시지가 브로커에 송신되면 on_publish() 콜백이 호출되는데, 이곳에서 다시 publish()를 호출하면 on_publish() 콜백과 publish() 호출이 지속됩니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     if rc == 0:
03:         client.publish("soda/sensors/cds", cds.readAverage)
04:
05: def _on_publish(client, userdata, mid):
06:     client.publish("soda/sensors/cds", cds.readAverage)
07:
08: cds = Cds(27)
09: mqttc = mqtt.Client()
10:
11: mqttc.on_connect = _on_connect
12: mqttc.on_publish = _on_publish
13:
14: mqttc.connect("192.168.10.2")
15:
16: mqttc.loop_forever()
```

message_callback_add()는 수신한 메시지 중 일부를 별도의 사용자 콜백에서 처리하도록 수신 필터와 수신 콜백을 등록하고 message_callback_remove()는 이를 제거합니다. 따라서 subscribe()로 토픽 필터를 지정한 후 message_callback_add()으로 수신 필터를 지정하면 수신 필터와 일치하는 메시지는 수신 콜백으로 전달되고 나머지만 on_message()로 전달됩니다.

- message_callback_add(sub, callback): 수신 콜백 추가
  - sub: 수신 필터 → 토픽 또는 토픽 필터 형식
  - callback: 수신 콜백
    - 수신 필터와 일치하지 않는 메시지는 on_message()에 전달
- message_callback_remove(sub): 수신 콜백 제거

다음은 토픽 필터에 가입한 후 수신 필터를 설정해 토픽별로 콜백을 분리한 예입니다. _on_message_temp()는 "soda/sensros/temp" 토픽이 수신될 때 호출되고, _on_message_humi()는 "soda/sensors/humi" 토픽이 수신될 때 호출됩니다. 그 밖의 메시지는 _on_message()에 전달됩니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     if rc == 0:
03:         client.subscribe("soda/sensors/+")
04:         client.message_callback_add("soda/sensors/temp", _on_message_temp)
```

```
05:         client.message_callback_add("soda/sensors/humi", _on_message_humi)
06:
07: def _on_message(client, userdata, message):
08:     print("OTHER:", message.topic, message.payload)
09:
10: def _on_message_temp(client, userdata, message):
11:     print("TEMP: ", int(message.payload))
12:
13: def _on_message_humi(client, userdata, message):
14:     print("HUMI: ", int(message.payload))
15:
16: client = mqtt.Client()
17:
18: mqttc.on_connect = _on_connect
19: mqttc.on_message = _on_message
20:
21: mqttc.connect("192.168.10.2")
22:
23: mqttc.loop_forever()
```

하나의 응용프로그램에 발행자와 구독자를 함께 구현할 때 on_publish() 콜백에서 publish()로 송신하는 메시지는 반드시 QoS > 0이어야 on_message() 콜백이 정상적으로 호출됩니다. 다음 예처럼 자신이 발행한 메시지를 자신이 구독할 수도 있습니다.

```
01: def _on_connect(client, userdata, flags, rc):
02:     client.subscribe("soda/sensors/cds")
03:     client.publish("soda/sensors/cds", cds.readAverage(), 1)
04:
05: def _on_publish(client, userdata, mid):
06:     client.publish("soda/sensors/cds", cds.readAverage(), 1)
07:
08: def _on_message(client, userdata, message):
09:     print(int(message.payload))
10:
11: cds = Cds(27)
12: client = mqtt.Client()
13:
14: client.on_connect = _on_connect
15: client.on_publish = _on_publish
16: client.on_message = _on_message
17:
18: client.connect("192.168.122.53")
19:
20: client.loop_forever()
```

publish 모듈

publish 모듈의 single()과 multiple()은 간단하게 단일 또는 다중 메시지 발행을 지원합니다.

- single(topic, payload=None, qos=0, retain=False, hostname="localhost", port=1883, client_id="", keepalive=60, will=None, auth=None, tls=None, protocol=mqtt.MQTTv311, transport="tcp")
    - will: 딕셔너리 타입의 강제 종 시 배포할 메시지. 기본값은 사용 안 함
        - will = { 'topic': "", "payload": "<payload">, 'qos': , 'retain': }
    - auth: 딕셔너리 타입의 인증 매개변수. 기본값은 사용 안 함
        - auth = { 'username': "", "password": ""}
            - username은 필수. password는 옵션
    - tls: 딕셔너리 타입의 TLS 인증 매개변수. 기본값은 사용 안 함
        - dict = { 'ca_certs': "<ca_certs>", "certfile": "", "keyfile": "", "tls_version": "<tls_version>", "ciphers": "<ciphers ">}
            - ca_certs는 필수. 나머지는 옵션
- multiple(msgs, hostname="localhost", port=1883, client_id="", keepalive=60, will=None, auth=None, tls=None, protocol=mqtt.MQTTv311, transport="tcp")
    - msgs: 딕셔너리 또는 튜플 타입의 발행할 메시지 목록.
        - 반드시 토픽 포함. 나머지는 기본값 사용 허용
        - msgs = {"topic":"", "payload":"", "qos":, "retain":}
        - msgs = ("", "", qos, retain)

다음은 `multiple()`을 사용해 한 번에 "soda/msg1"과 "soda/msg2" 토픽에 메시지를 발행하는 예입니다.

```
01: import paho.mqtt.publish as publish
02:
03: msgs = [
04:     {'topic':"soda/msg1", 'payload':"Soda OS"},
05:     ("soda/msg2", "Welcome to IoT World!", 0, False)
06: ]
07:
08: publish.multiple(msgs)
```

subscribe 모듈

subscribe 모듈은 `simple()`과 `callback()`으로 메시지의 간단한 구독 및 처리를 지원합니다. `simple()`은 토픽에 대한 MQTTMessage 객체가 반환되고 `callback()`은 인자로 전달한 사용자 함수가 호출되면서 인자로 MQTTMessage 객체가 전달됩니다.

- simple(topics, qos=0, msg_count=1, retained=False, hostname="localhost", port=1883, client_id="", keepalive=60, will=None, auth=None, tls=None, protocol=mqtt.MQTTv311)
    - topic: 구독할 토픽 문자열로 필터 적용 허용
    - msg_count: 브로커에서 찾을 메시지 수. 기본값은 1
    - msg_count == 1: MQTTMessage 객체 반환
    - msg_count > 1: MQTTMessage 객체 목록 반환
- callback(callback, topics, qos=0, userdata=None, hostname="localhost", port=1883, client_id="", keepalive=60, will=None, auth=None, tls=None, protocol=mqtt.MQTTv311)
    - callback: 메시지가 수신되면 호출할 사용자 함수
        - def on_message(client, userdata, message)
    - userdata: callback이 호출될 때 함께 전달할 사용자 제공 인자

MQTTMessage 객체의 topic과 payload 속성에는 수신된 토픽과 바이트 배열 타입의 메시지가 저장되어 있으며 토픽 필터를 적용했다면 topic을 통해 수신된 토픽을 확인할 수 있습니다. `callback()`은 내부 무한 루프에서 `select()`를 사용해 메시지를 수신하므로 프로그램은 무한 대기 상태가 됩니다.

```
01: import paho.mqtt.subscribe as subscribe
02:
03: def on_message_print(client, userdata, message):
04:     print("%s %s" % (message.topic, message.payload.decode()))
05:
06: try:
07:     subscribe.callback(on_message_print, "soda/+", hostname="192.168.10.2")
08: except KeyboardInterrupt:
09:     pass
```