MQTT:
Message Queuing Telemetry Transport

Dr. Manas Khatua
Associate Professor
Dept. of CSE, IIT Guwahati
E-mail: manaskhatua@iitg.ac.in

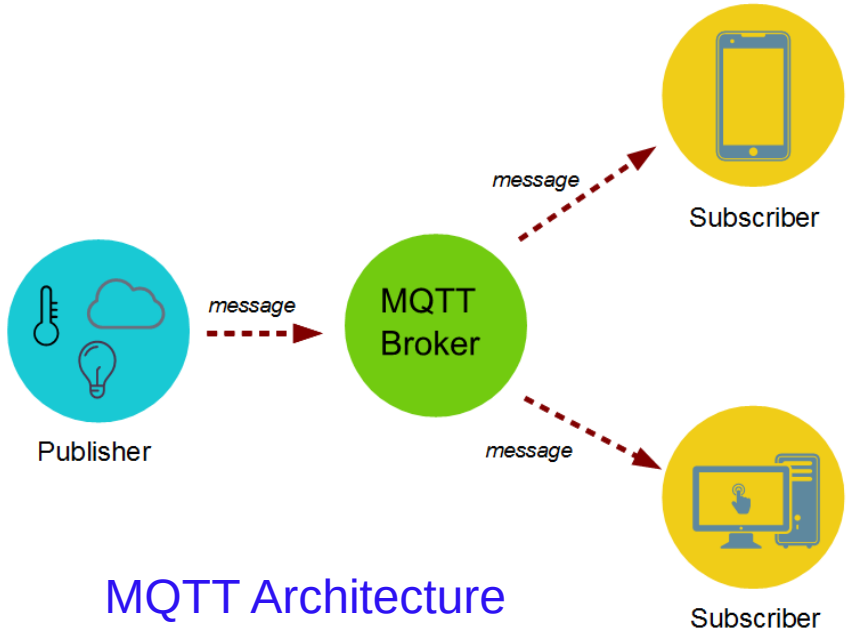
What is MQTT?

MQTT : Message Queueing Telemetry Transport protocol.

- **Reliable, Lightweight, Cost-effective protocol** that transports messages between devices.
- Suited for the **transport of telemetry data** (sensor and actuator data)

- Invented by *Andy Stanford Clark* of IBM and *Arlen Nipper* of Arcom (now Eurotech) in 1999

- Used by real-life IoT frameworks:
  - Amazon Web Services (AWS),
  - IBM WebSphere MQ,
  - Microsoft Azure IoT,
  - Facebook Messenger,
  - etc.



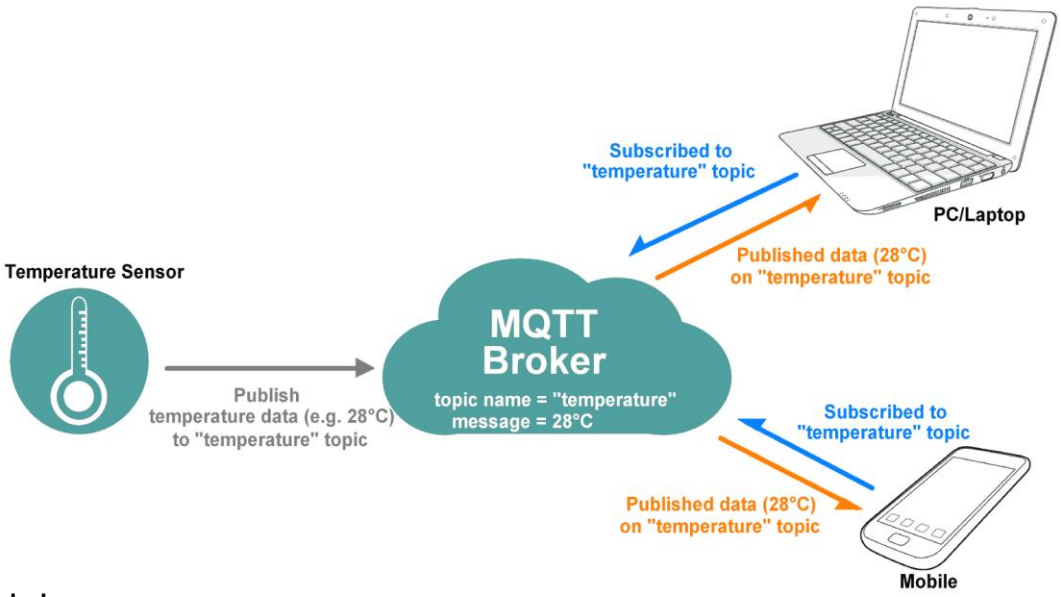
MQTT Architecture

Example:

- **Light sensor** continuously **publish** sensor data to the **broker**.
- **Building control application** **subscribes** to light sensor data and so receives it from the broker. Then it decides to activate **Camera**.
- **The application** sends an activation message to the **camera** node (i.e. actuator) through the **broker**.

MQTT Characteristics

- In 2013-14, MQTT was adopted and **published as an official standard** by OASIS
  - OASIS: Organization for the Advancement of Structured Information Standard
- MQTT was mainly designed for M2M communication

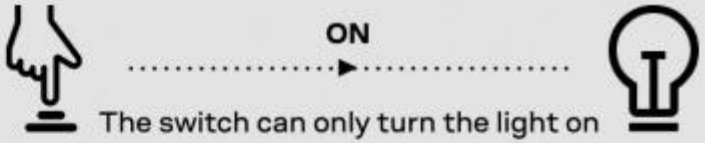


- **Publish / Subscribe** (PubSub) model
  - **Decoupling** of data producer (publisher) and data consumer (subscriber) through topics (message queues)
- **Asynchronous communication model** with messages (events)
  - Publisher and Subscriber need not be online at the same time
- Low overhead **(2 bytes header)**
  - Suitable for low network bandwidth applications
- Runs on connection-oriented transport (**TCP**), but it will be used in conjunction with **6LoWPAN HC** (TCP header compression).

Advantages of MQTT

Simplified communication

Communication is a complex problem. MQTT reduces complexity, allowing a single connection to a message topic. Data is logically structured and can be processed flexibly.



Eliminate polling

MQTT allows instantaneous, push-based delivery, eliminating the need for message consumers to periodically check or "poll" for new information. This dramatically reduces network traffic.

Dynamic targeting

MQTT makes discovery of services easier and less error prone. Instead of maintaining a roster of peers that an application can send messages to, a publisher will simply post messages to a topic.

Decouple and scale

MQTT also makes solutions more flexible and enables scale. It allows changes in communication patterns, adding or changing functionality without sending ripple effects across the system.

Source: https://www.u-blox.com/en/blogs/insights/mqtt-beginners-guide

Publish & Subscribe Messaging

- A producer sends (publishes) a message (publication) on a topic (subject)
- A consumer subscribes (makes a subscription) for messages on a topic (subject)
- A message server / broker matches publications to subscriptions

Who will get the message ?

- If no matches, the message is discarded
- If one / more matches, the message is delivered to each matching subscriber/consumer

Topic

- A topic forms the namespace in hierarchical with each "sub topic" separated by **/** (forward slash)
- An example topic space :
  - A house publishes information about itself on**:**
    <country>/<region>/<town>/<postalcode>/<house no>/energyConsumption
    <country>/<region>/<town>/<postalcode>/<house no>/solarEnergy
  - It subscribes for control commands:
    <country>/<region>/<town>/<postalcode>/<house no>/thermostat/setTemp

Cont…

Wildcards

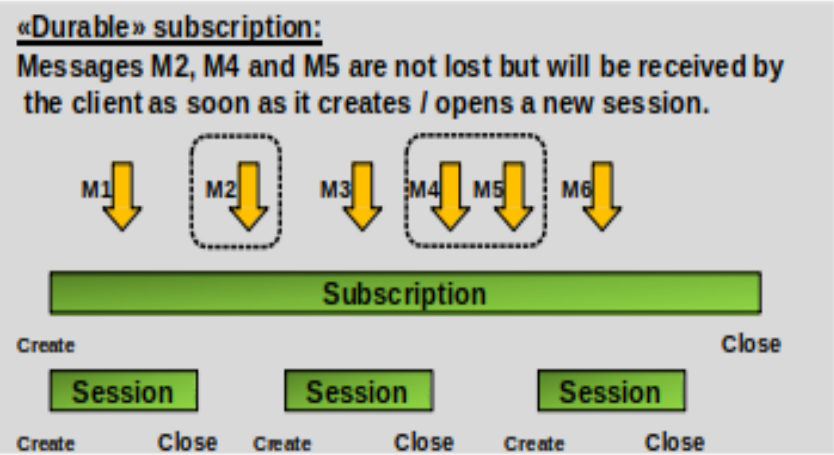
- A **subscriber** can subscribe to an absolute topic OR can use wildcards:
  - Single-level wildcards **"+"** can appear anywhere in the topic string
    For example:
    Let the Topic: *<country>/<region>/<town>/<postalcode>/<house no>/energyConsumption*
    - Energy consumption **for 1 house** in Hursley
      - *UK/Hants/Hursley/SO212JN/+/energyConsumption*
    - Energy consumption **for all houses** in Hursley
      - *UK/Hants/Hursley/+/+/energyConsumption*
  - Multi-level wildcards **"#"** must appear at the end of the string
    For example:
    - Details of energy consumption, solar and alarm **for all houses in SO212JN**
      - *UK/Hants/Hursley/SO212JN/#*

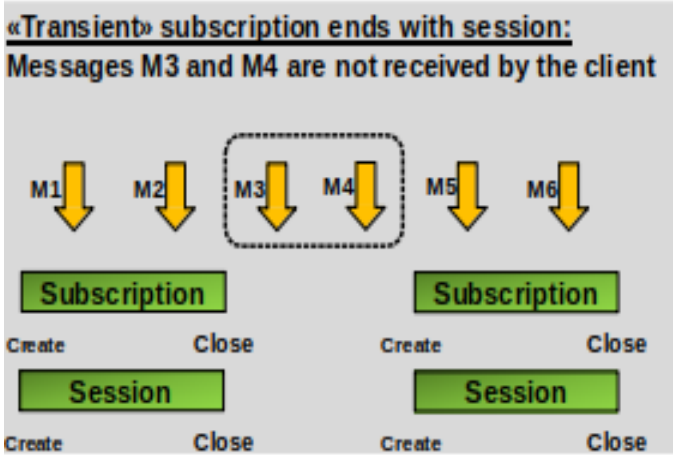
NOTE :

- Wildcards must be **next to a separator**
- Wildcards **cannot** be used when **publishing**

Cont…

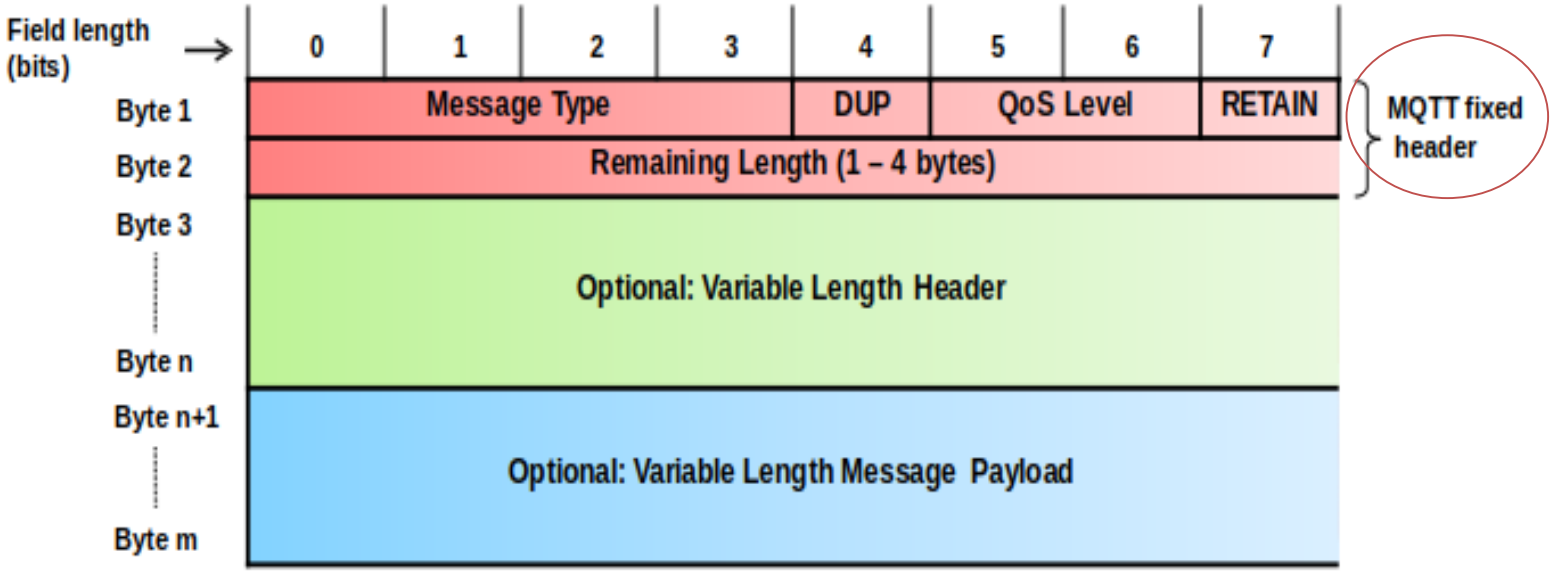
- A subscription can be **durable** or **non-durable**
- **Durable:**
  - Once a subscription is in place,
    - A broker will forward matching messages to the subscriber immediately if the subscriber is connected.
    - If the subscriber is not connected, messages are stored on the server/broker until the next time the subscriber connects.





- **Non-durable** / **Transient** (i.e. subscription ends with client session):
  - The subscription lifetime is the same as the time the subscriber is connected to the server / broker

MQTT Message Format



- Message Type: identifies the kind of MQTT packet within a message
- DUP: Duplicate flag – indicates whether the packet has been sent previously or not

- QoS Level: it allows to select different QoS level
- Retain: this flag notifies the server to hold onto the last received PUBLISH message data
- Remaining Length: specifies the size of optional fields

- MQTT is lightweight
  - because each packets consists of a 2-byte fixed header with optional variable header and payload fields

Cont…

Message header field	Description / Values	
Message Type (4 bits) 14 message types, 2 are reserved	0: Reserved	8: SUBSCRIBE
	1: CONNECT	9: SUBACK
	2: CONNACK	10: UNSUBSCRIBE
	3: PUBLISH	11: UNSUBACK
	4: PUBACK (Publish ACK)	12: PINGREQ
	5: PUBREC (Publish Received)	13: PINGRESP
	6: PUBREL (Publish Release)	14: DISCONNECT
	7: PUBCOMP (Publish Complete)	15: Reserved
DUP (1 bit)	Duplicate message flag. Indicates to the receiver that this message may have already been received. 1: Client or server (broker) re-delivers a PUBLISH, PUBREL, SUBSCRIBE or UNSUBSCRIBE message (duplicate message).	
QoS Level (2 bits)	Indicates the level of delivery assurance of a PUBLISH message. 0: At-most-once delivery, no guarantees, «Fire and Forget». 1: At-least-once delivery, acknowledged delivery. 2: Exactly-once delivery.	
RETAIN (1 bit)	1: Instructs the server to retain the last received PUBLISH message and deliver it as a first message to new subscriptions.	
Remaining Length (1-4 bytes)	Indicates the number of remaining bytes in the message, i.e. the length of the (optional) variable length header and (optional) payload.	

Message Types

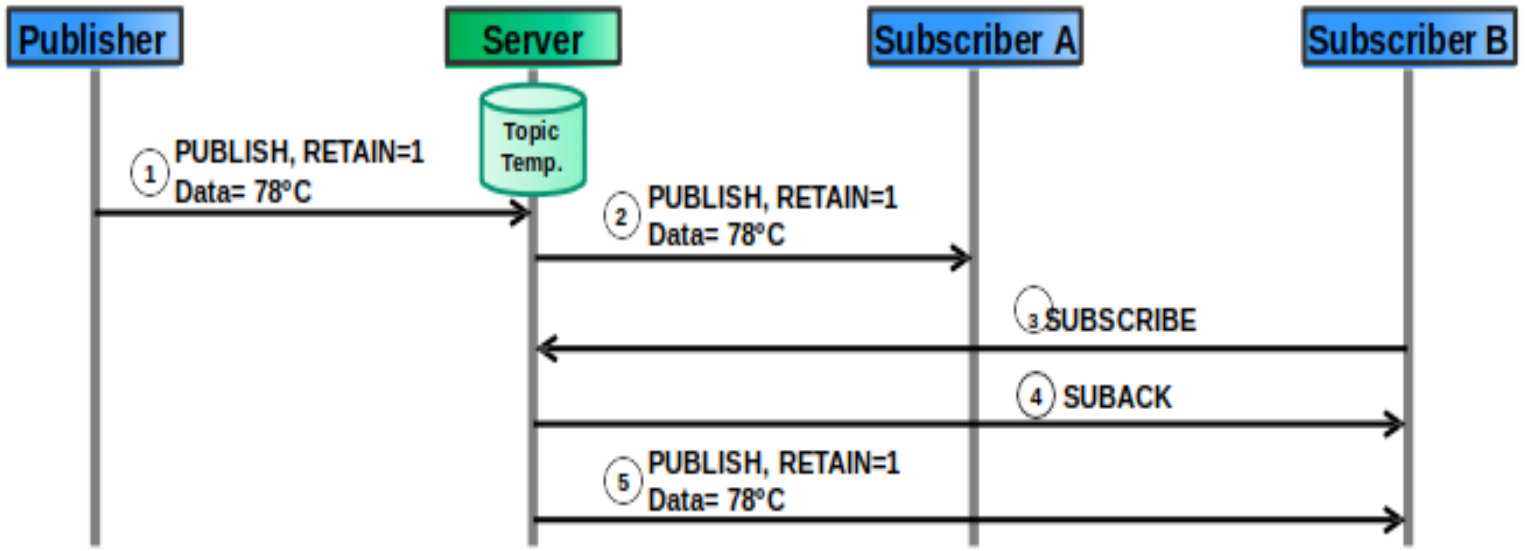
Name	Value	Direction	Description	Flags
Reserved	0	Forbidden		
CONNECT	1	C $\rightarrow$ S	Client connection request	0000
CONNACK	2	C $\leftarrow$ S	Server response to CONNECT	0000
PUBLISH	3	C $\leftrightarrow$ S	Message publishing	Varying
PUBACK	4	C $\leftrightarrow$ S	PUBLISH acknowledgement	0000
PUBREC	5	C $\leftrightarrow$ S	PUBLISH received	0000
PUBREL	6	C $\leftrightarrow$ S	PUBLISH release	0010
PUBCOMP	7	C $\leftrightarrow$ S	PUBLISH complete	0000
SUBSCRIBE	8	C $\rightarrow$ S	Client subscription to topic	0010
SUBACK	9	C $\leftarrow$ S	Server response to SUBSCRIBE	0000
UNSUBSCRIBE	10	C $\rightarrow$ S	Client unsubscription to a topic	0010
UNSUBACK	11	C $\leftarrow$ S	Server response to UNSUBSCRIBE	0000
PINGREQ	12	C $\rightarrow$ S	Keepalive PING request	0000
PINGRESP	13	C $\leftarrow$ S	Keepalive PING response	0000
DISCONNECT	14	C $\rightarrow$ S	Client is disconnecting	0000
Reserved	15	Forbidden		

RETAIN

RETAIN=1 in a PUBLISH message instructs the server to keep the message for this topic.
When a new client subscribes to the topic, the server sends the retained message quickly.

- **Typical application scenarios:**
  - Clients publish only **changes** in data, so subscribers receive the last known good value.

- **Example:**
  - Subscribers receive last known temperature value from the temperature data topic.
  - RETAIN=1 indicates to subscriber B that the message may have been published some time ago.

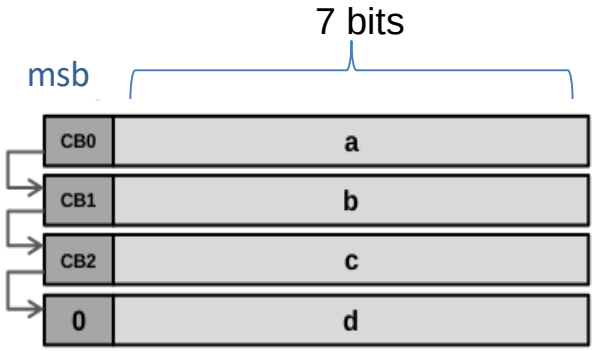


Remaining Length (RL)

- The *remaining length* field encodes the sum of the lengths of:
  1. (Optional) variable length header
  2. (Optional) variable length payload

- To save bits, *RL* is a variable length field with 1 to 4 bytes.
  - The most significant bit (msb) of a length field byte has the meaning continuation bit (CB).
  - If more bytes follow, it is set to 1.

RL is encoded as:: $\mathbf{a * 128^0 + b * 128^1 + c * 128^2 + d * 128^3}$
and placed into the RL field bytes as follows:



Byte 0 = LSB ($a * 128^0$, CB0=1 if b > 0)

Byte 1 ($b * 128^1$, CB1=1 if c > 0)

Byte 2 ($c * 128^2$, CB2=1 if d > 0)

Byte 3 = MSB ($d * 128^3$)

0 (0x00)	127 (0x7F)
128 (0x80, 0x01)	16 383 (0xFF, 0x7F)
16 384 (0x80, 0x80, 0x01)	2 097 151 (0xFF, 0xFF, 0x7F)
2 097 152 (0x80, 0x80, 0x80, 0x01)	268 435 455 (0xFF, 0xFF, 0xFF, 0x7F)

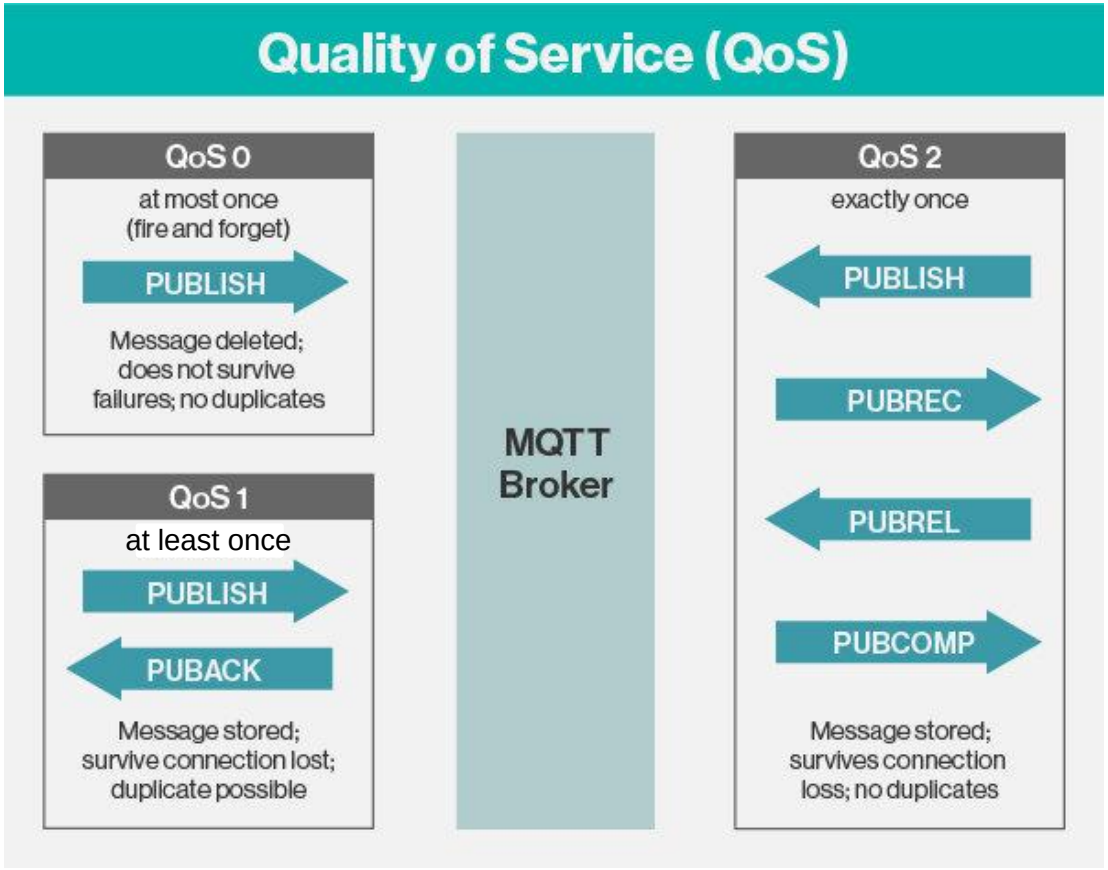
Example 1: **RL = 364 = $108*128^0+2*128^1$ → a=108, CB0=1, b=2, CB1=0, c=0, d=0, CB2=0**

Example 2: **RL = 25'897 = $41*128^0 + 74*128^1 + 1*128^2$ → a=41, CB0=1, b=74, CB1=1, c=1, CB2=0, d=0**

MQTT QoS

- Even though TCP/IP provides guaranteed data delivery, **data loss** can still occur if a TCP connection breaks down and messages in transit are lost.

- Therefore, MQTT adds **3 quality of service (QoS)** levels on top of TCP



PUBREC (REC: received) Packet is the response to a PUBLISH Packet

PUBREL (REL: release) Packet is the response to a PUBREC Packet

PUBCOMP (COMP: complete) Packet is the response to a PUBREL Packet.

Cont…

QoS level 0:

- At-most-once delivery («best effort»).
- Messages are delivered according to the delivery guarantees of the underlying network (TCP/IP).
  - *Example application*: Temperature sensor data which is regularly published. Loss of an individual value is not critical since applications (i.e. consumers of the data) will anyway integrate the values over time

QoS level 1:

- At-least-once delivery.
- Messages are guaranteed to arrive, but there may be duplicates.
  - *Example application*: A door sensor senses the door state. It is important that door state changes (closed->open, open->closed) are published losslessly to subscribers (e.g. alarming function). Applications simply discard duplicate messages by comparing the message ID field.

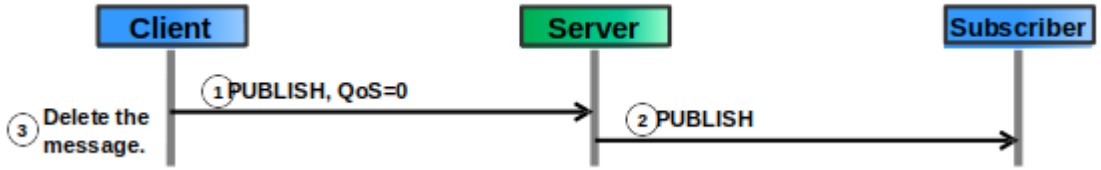
QoS level 2:

- Exactly-once delivery.
- This is the highest level that also incurs most overhead in terms of control messages and the need for locally storing the messages.
- Exactly-once is a combination of at-least-once and at-most-once delivery guarantee.
  - *Example application*: Applications where duplicate events could lead to incorrect actions, e.g. sounding an alarm as a reaction to an event received by a message. So, it avoids duplicate.

PUBLISH msg flow

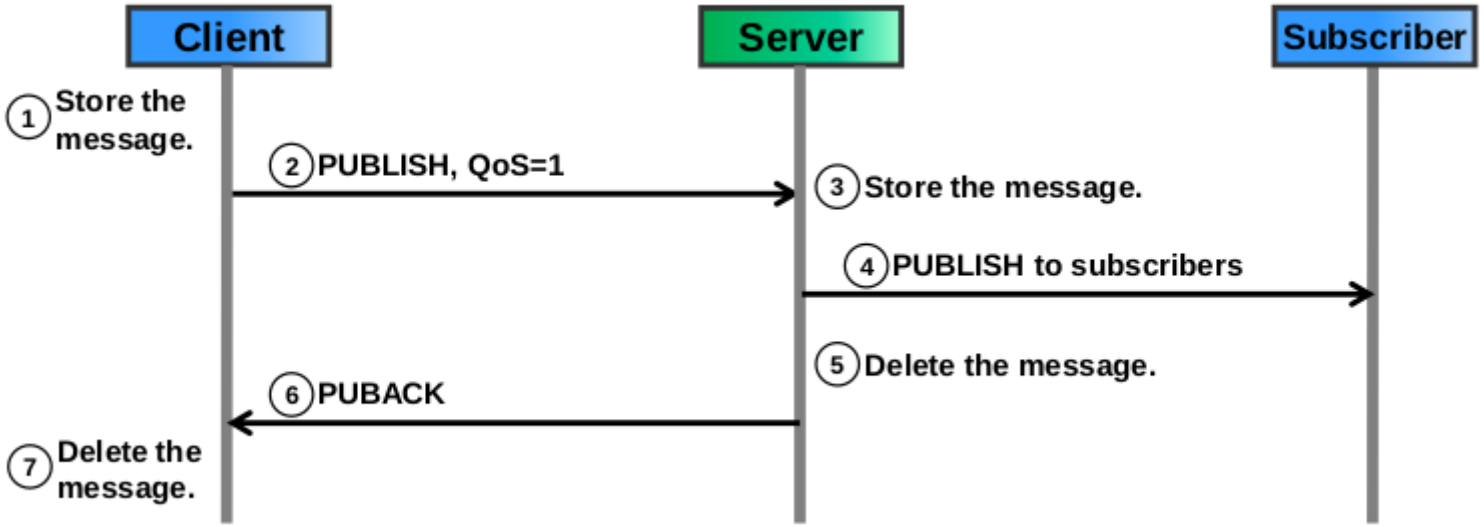
QoS level 0:

- With QoS level 0, a message is delivered with **at-most-once delivery semantics** («fire-and-forget»).



QoS level 1:

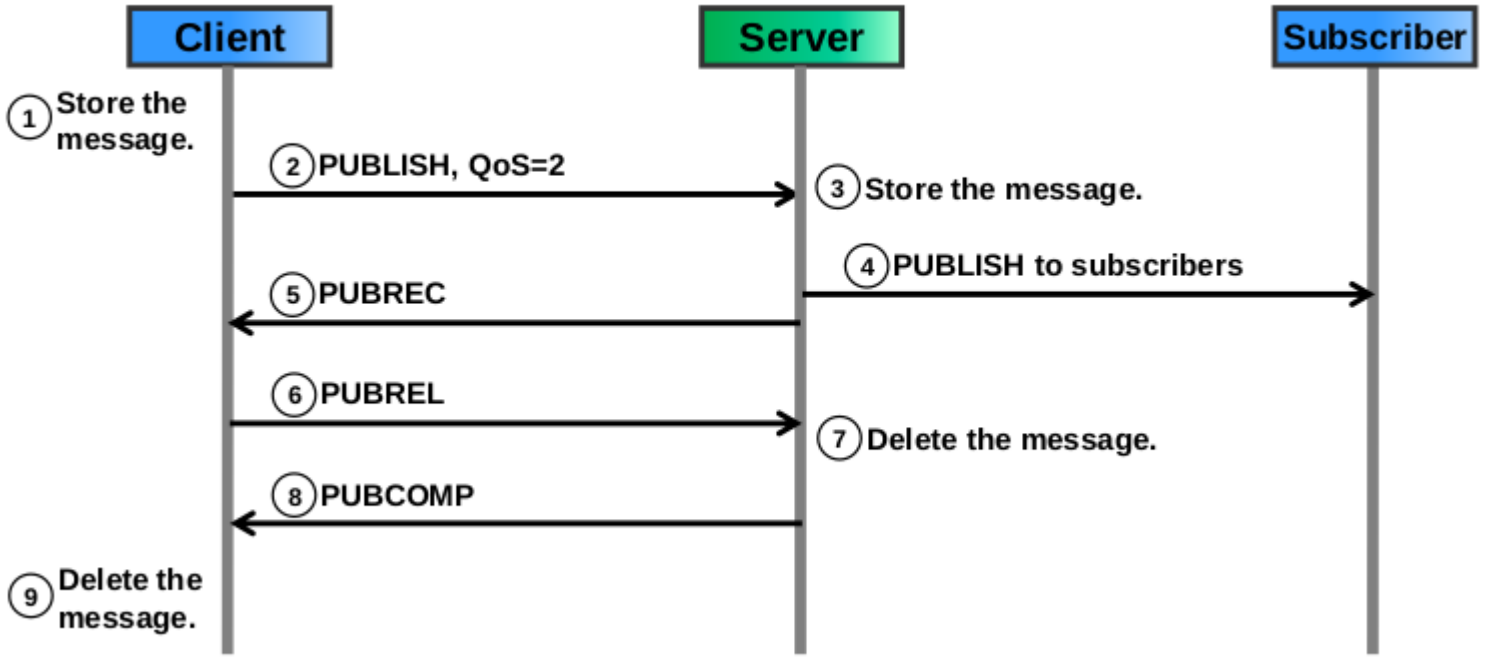
- QoS level 1 affords **at-least-once delivery semantics.** If the client does not receive the **PUBACK** in time, it **re-sends** the message.



Cont…

QoS level 2:

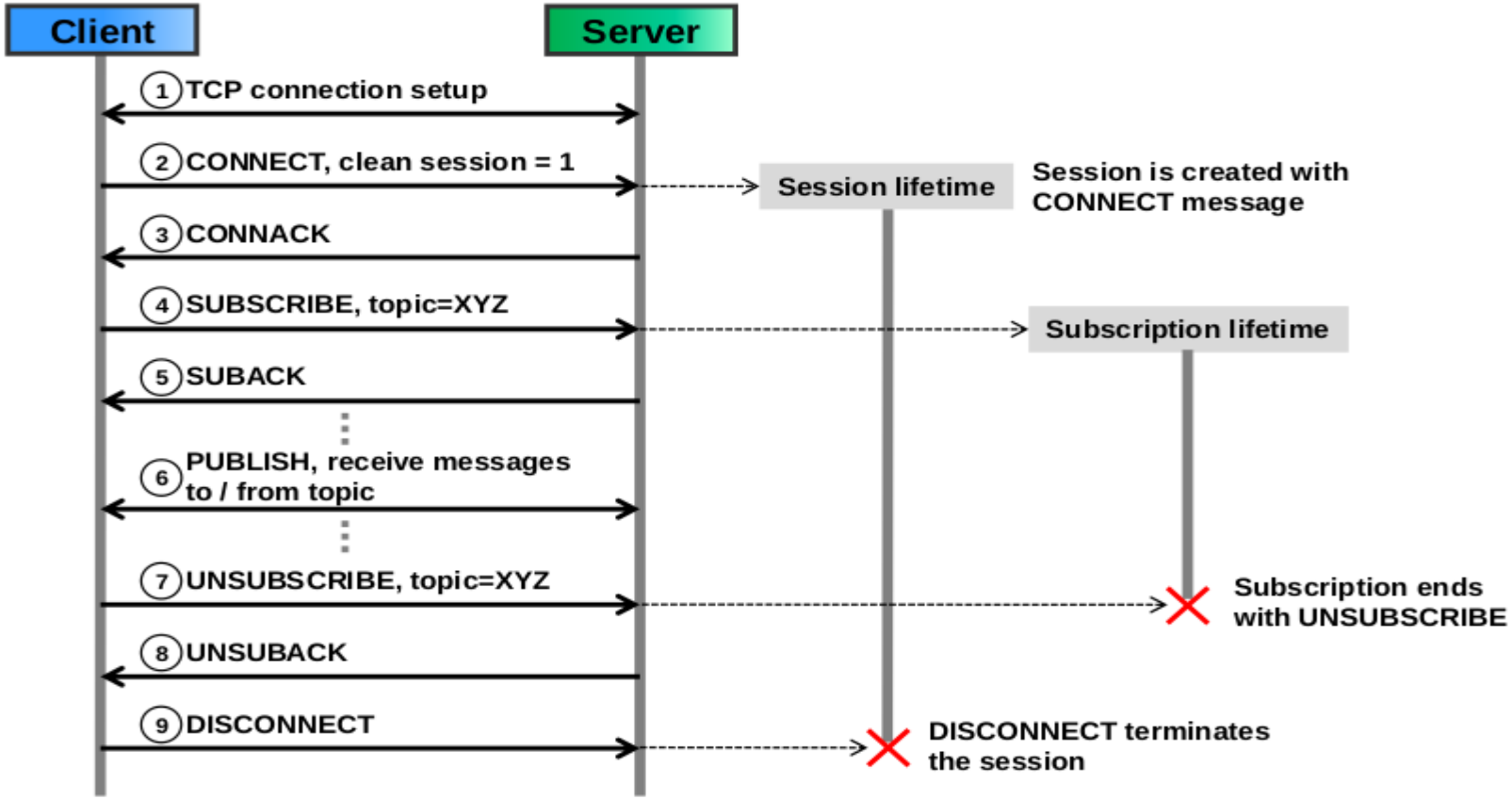
- QoS level 2 affords the highest quality delivery semantics **exactly-once,** but comes with the cost of **additional control messages.**



Example: CONNECT & SUBSCRIBE msg flow

Case 1:

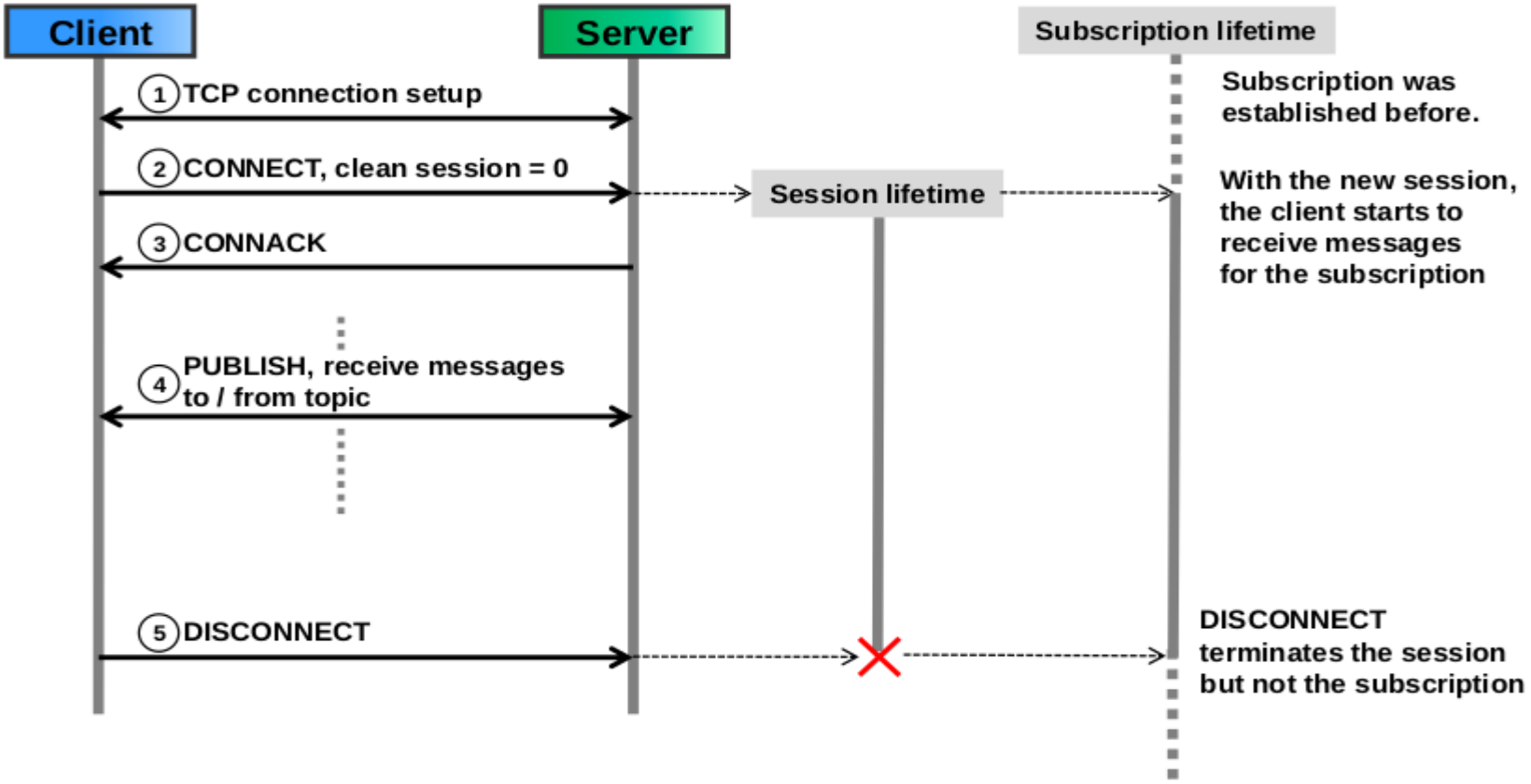
- Session/connection and subscription setup with clean session flag = 1 (non-durable subscription)



Cont…

Case 2:

- Session/connection and subscription setup with clean session flag = 0 (durable subscription)



What is MQTT-SN?

MQTT-SN (MQTT for sensor networks) is a variant of MQTT that has been optimized for use in low power environments such as sensor networks, as the name suggests.

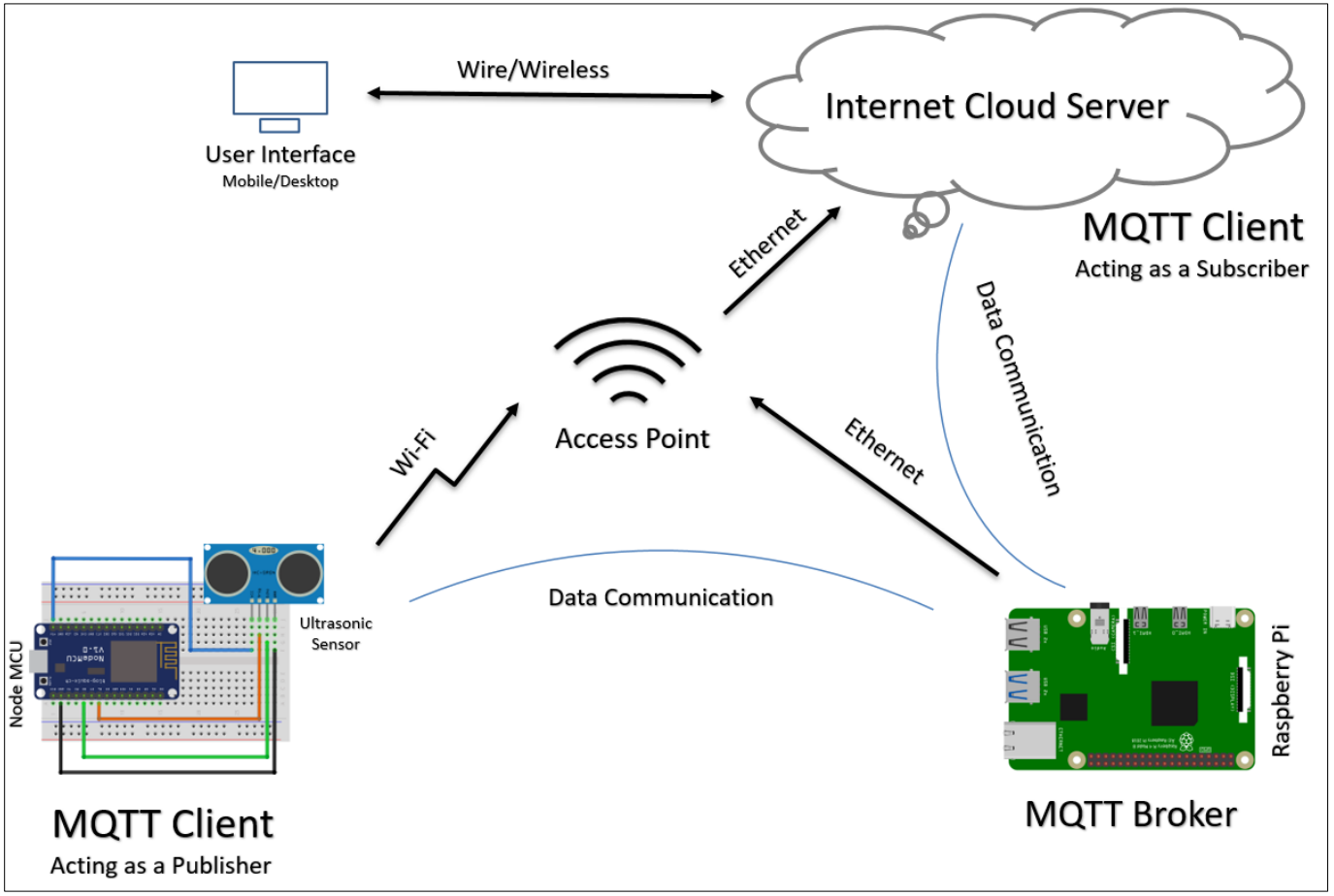
SN adds extra functionality for use cases where lower power is required.

- QoS mode -1: allows for fire-and-forget messaging
- Topic aliases: allows for simplified publishing and reduced data overheads
- Sleep mode (disconnected sessions): allows messages to be queued on the broker while the remote Thing or device is powered off

Demo using MQTT Protocol

Send Sensor values to Cloud using MQTT Protocol:

- The Ultrasonic Sensor reads the distance between itself & the object
- Sends the measured distance to the cloud server using MQTT protocol.
- Here, the cloud is ThingSpeak.com



Requirements

Hardware

- •ESP8266 NodeMCU
- •Breadboard
- •Jumper Wires
- •HC-SR04 Ultrasonic Sensor
- •Raspberry Pi

Software

- •Arduino IDE
- •Fritzing

Cloud Platform

- •ThingSpeak Account

ThingSpeak is an IoT analytics PaaS that allows you to aggregate, visualize, and analyze live data streams

ARDUINO

ESP8266 NodeMCU: Libraries

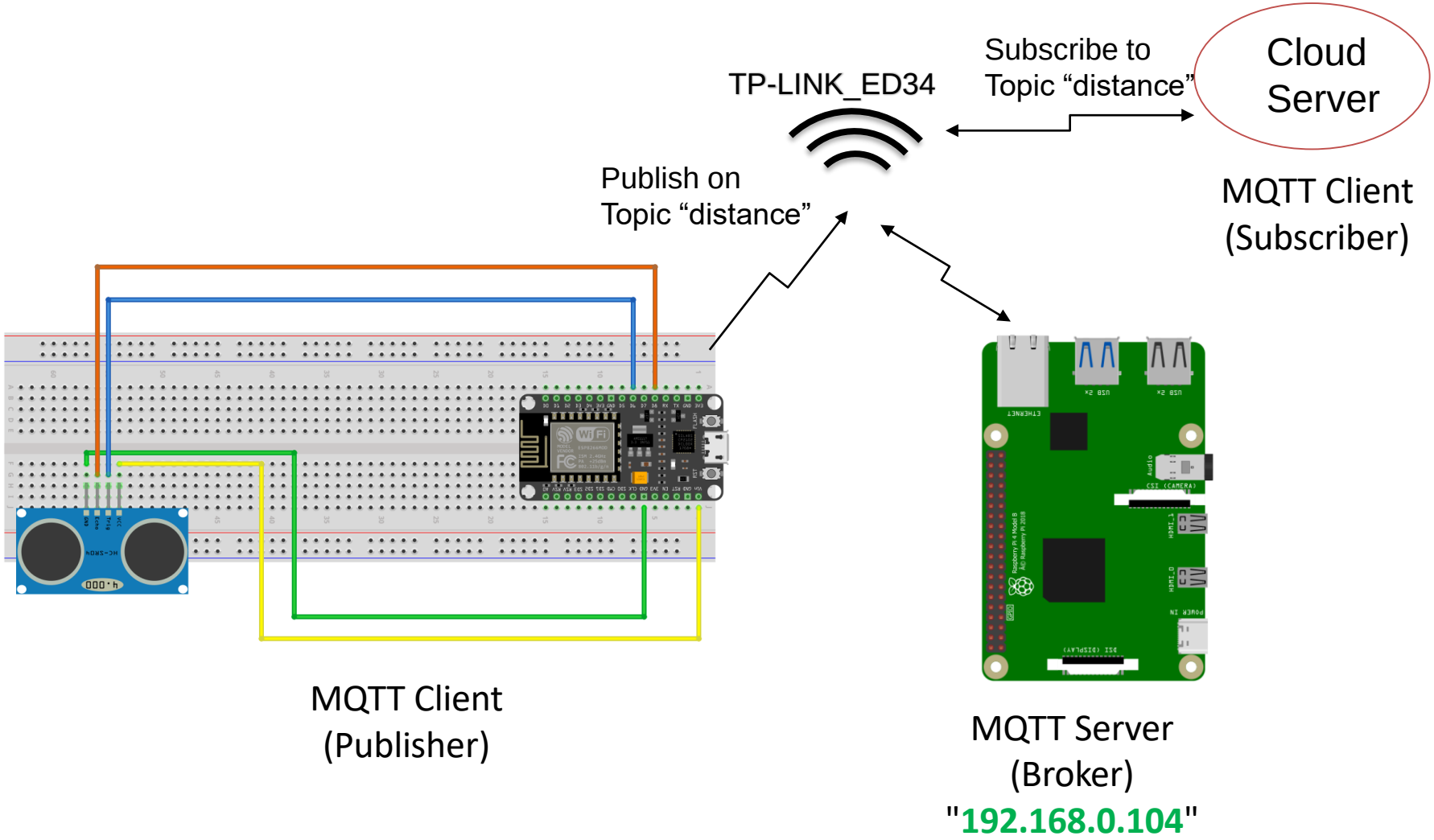
The ESP8266 NodeMCU requires certain libraries for optimal use of the board.

- PubSubClient – It is required for the MQTT Messaging.
- ThingSpeak – It is a communication library for supporting to send data to the cloud.
- ESP8266WiFi – It is required to use the Wi-Fi capabilities of the NodeMCU.

Step to install above libraries for the programming part:

- Open Arduino IDE
- Move on to Tools
- Manage Libraries…
- Search and install the above mentioned libraries

Setup Connections



Programming ESP8266

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>

#define TRIGGER D6
#define ECHO D8
long duration, cm;

const char* ssid = "TP-LINK_ED34";
const char* password = "48193580";

//Raspberry Pi IP Address
const char* mqtt_server = "192.168.0.104";

WiFiClient   espClient;

PubSubClient   client(espClient);
```

```
void setup_wifi () {
   delay(10);
   Serial.println();
   Serial.print("Connecting to ");
   Serial.println(ssid);

   WiFi.begin(ssid, password);
   while (WiFi.status() != WL_CONNECTED) {
       delay(500);
       Serial.print(".");
   }
   Serial.println("");
   Serial.print("WiFi connected - ESP IP address: ");

   Serial.println(WiFi.localIP());
}
```

Cont…

```
void callback (String topic, byte* message,
                    unsigned int length)
{
   Serial.print("Message arrived on topic: ");
   Serial.print(topic);
   Serial.print("Message: ");

   String messageTemp;
   for (int i = 0; i < length; i++) {
         Serial.print((char)message[i]);
         messageTemp += (char)message[i];
   }
   Serial.println();
}
```

A **callback** is a function that is passed as an argument to another function.

```
void reconnect ()
{
  // Loop until we're reconnected
 while (! client.connected ()) {
     Serial.print("Attempting MQTT connection...");

     if (client.connect("ESP8266Client"))
     {
         Serial.println("connected");
         client.subscribe("esp8266/4");
     }
     else {
         Serial.print("failed, rc=");
         Serial.print(client.state());
         Serial.println(" try again in 5 seconds");
         delay(15000);
     }
  }
}
```

Cont…

```
void setup () {
   Serial.begin(115200);
   pinMode(TRIGGER,OUTPUT);
   pinMode(ECHO,INPUT);

   setup_wifi ();

   client.setServer (mqtt_server, 1883);

   client.setCallback (callback);
}



void loop () {
  if (! client.connected ()) {
      reconnect();
  }
  if(! client.loop() )
     client.connect("ESP8266Client");
     digitalWrite(TRIGGER, LOW);
     delayMicroseconds(2);
     digitalWrite(TRIGGER, HIGH);
     delayMicroseconds(10);
     digitalWrite(TRIGGER, LOW);

     duration = pulseIn(ECHO, HIGH);

     cm = duration / 29 / 2;
     static char distanceincm[7];

     dtostrf(cm, 6, 2, distanceincm);

     client.publish("/esp8266/distance", distanceincm);

     Serial.print("Distance: ");
     Serial.print(cm);
     Serial.print("cm");
     Serial.println();
     delay(5000);
}
```

Programming R. Pi

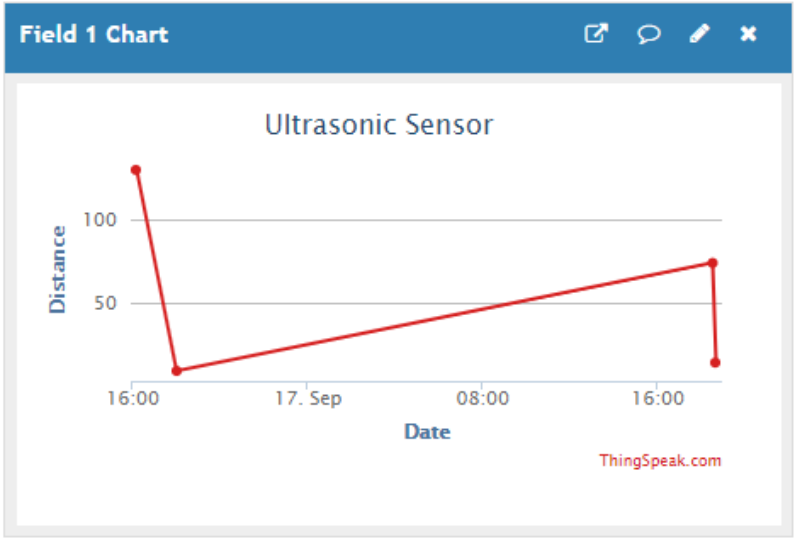
```
import paho.mqtt.client as mqtt
import urllib3                  # powerful HTTP client for Python

myAPI = 'SMLI56456456RHUB'
baseURL = 'https://api.thingspeak.com/update?api_key=%s'
                                                    % myAPI
val=''
http = urllib3.PoolManager()

def on_connect (client, userdata, flags, rc):
      print("Connected with result code "+ str(rc))
      client.subscribe("/esp8266/distance")

def on_message (client, userdata, message):
     print("Received message '" + str(message.payload) +
                                  "' on topic '" + message.topic)
     distance = 0
     if message.topic == "/esp8266/distance":
          print("Distance updated")
          distance = str(message.payload, 'UTF-8')
          distance = distance.strip()
          print(distance)
          global val
          val = distance
```

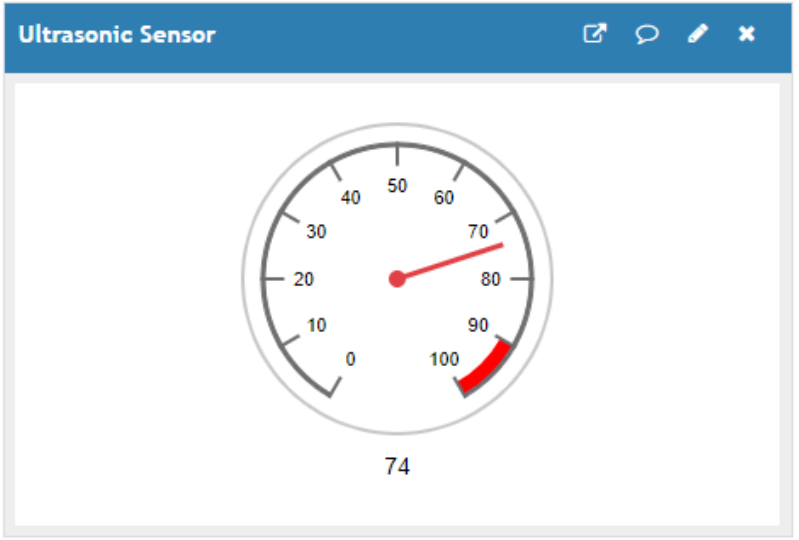
```
  if val != '' :
     conn = http.request('GET', baseURL +'&field1=%s'%(val))
     print(conn.status)
     conn.read()
     conn.close()
     val = ''

def main():
   mqtt_client = mqtt.Client()
   mqtt_client.on_connect = on_connect
   mqtt_client.on_message = on_message

   mqtt_client.connect ('localhost', 1883, 60)
   # Connect to the MQTT server and process messages in a
                                          background thread.
   mqtt_client.loop_start()

if __name__ == '__main__':
   print('MQTT to InfluxDB bridge')
   main()
```

ThingSpeak Cloud Dashboard

Channel Stats

Created: 8 days ago
Last entry: less than a minute ago
Entries: 3003





Thanks!