Web Protocols

HTTP & Web Sockets

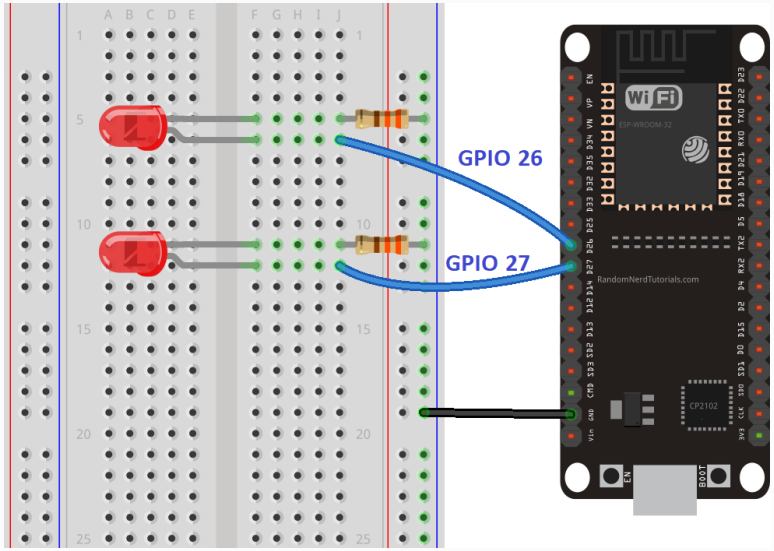
Recall REST & HTTP

REST

HTTP

WiFi ESP 32 - HTTP Server

```
// Rui Santos
// Complete Project Details http://randomnerdtutorials.com
#include <WiFi.h>
// Replace with your network credentials
const char* ssid     = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

WiFiServer server(80);

const int led1 =  26;      // the number of the LED pin
const int led2 =  27;      // the number of the LED pin

// Client variables
char linebuf[80];
int charcount=0;

void setup() {
  // initialize the LEDs pins as an output:
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);

  //Initialize serial and wait for port to open:
  Serial.begin(115200);
  while(!Serial) {// wait for serial port to connect. Needed for native USB port only
  }

  // We start by connecting to a WiFi network
  Serial.println();  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  // attempt to connect to Wifi network:
  while(WiFi.status() != WL_CONNECTED) {
    // Connect to WPA/WPA2 network. Change this line if using open or WEP network:
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  server.begin();
```



https://randomnerdtutorials.com/esp32-access-point-ap-web-server/

https://randomnerdtutorials.com/esp32-web-server-arduino-ide/

WiFi ESP 32 - HTTP Server

```
void loop() {
  // listen for incoming clients
  WiFiClient client = server.available();
  if (client) {
    Serial.println("New client");
    memset(linebuf,0,sizeof(linebuf));
    charcount=0;
    // an http request ends with a blank line
    boolean currentLineIsBlank = true;
    while (client.connected()) {
      if (client.available()) {
        char c = client.read();
        Serial.write(c);
        //read char by char HTTP request
        linebuf[charcount]=c;
        if (charcount<sizeof(linebuf)-1) charcount++;
        // if you've gotten to the end of the line (received a newline
        // character) and the line is blank, the http request has ended,
        // so you can send a reply
        if (c == '\n' && currentLineIsBlank) {
          // send a standard http response header
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println("Connection: close");  // the connection will be closed after completion of the response
          client.println();
          client.println("<!DOCTYPE HTML><html><head>");
          client.println("<meta name=\"viewport\" content=\"width=device-width, initial-scale=1\"></head>");
          client.println("<h1>ESP32 - Web Server</h1>");
          client.println("<p>LED #1 <a href=\"on1\"><button>ON</button></a> <a
href=\"off1\"><button>OFF</button></a></p>");
          client.println("<p>LED #2 <a href=\"on2\"><button>ON</button></a> <a
href=\"off2\"><button>OFF</button></a></p>");
          client.println("</html>");
          break;
        }
```

```
if (c == '\n') {
          // you're starting a new line
          currentLineIsBlank = true;
          if (strstr(linebuf,"GET /on1") > 0){
            Serial.println("LED 1 ON");
            digitalWrite(led1, HIGH);
          }
          else if (strstr(linebuf,"GET /off1") > 0){
            Serial.println("LED 1 OFF");
            digitalWrite(led1, LOW);
          }
          else if (strstr(linebuf,"GET /on2") > 0){
            Serial.println("LED 2 ON");
            digitalWrite(led2, HIGH);
          }
          else if (strstr(linebuf,"GET /off2") > 0){
            Serial.println("LED 2 OFF");
            digitalWrite(led2, LOW);
          }
          // you're starting a new line
          currentLineIsBlank = true;
          memset(linebuf,0,sizeof(linebuf));
          charcount=0;
        } else if (c != '\r') {
          // you've gotten a character on the current
line
          currentLineIsBlank = false;
        }
      }
    }
    // give the web browser time to receive the data
    delay(1);

    // close the connection:
    client.stop();
    Serial.println("client disconnected");
  }
}
```

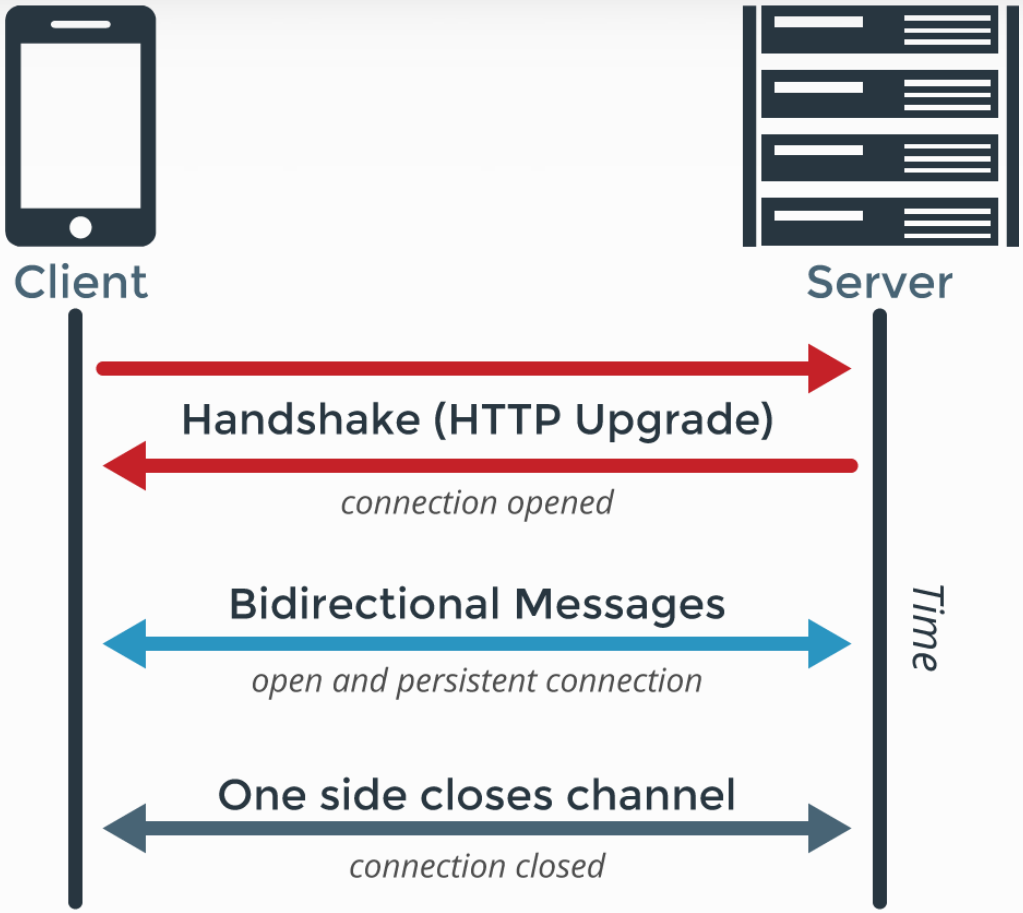
Web Socket

What is WebSocket

- WebSockets provide persistent connection between client and server that both parties can use to start sending data at any time.
- Client establishes WebSocket connection through a process known as the WebSocket handshake.
- Handshake process starts with the client sending a regular HTTP request to the server. An Upgrade header is included in this request informing the server that the client wishes to establish connection.

https://www.instructables.com/howto/ESP32+WEBSOCKET/

WebSockets Communication flow



Install Lib from here: https://github.com/Links2004/arduinoWebSockets

WebSockets Communication flow

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Wed, 16 Oct 2013 10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

When the handshake is complete, the initial HTTP connection is replaced by a WebSocket connection that uses the same underlying TCP/IP connection. At this point **Both parties** can send data.

How WebSockets Work

- Data is transferred through a WebSocket as ***messages.***
- Each message consists of one or more *frames* containing the data you are sending (aka payload).
- Each frame is prefixed with 4-12 bytes of data about the payload.

http://codepen.io/matt-west/pen/tHlBb

ESP 32 - WebSockets

```
/*
 * WebSocketServer.ino
 *
 *  Created on: 22.05.2015
 *  Follow https://techtutorialsx.com/2018/08/14/esp32-async-http-web-server-websockets-introduction/
 */

#include <Arduino.h>

#include <WiFi.h>
#include <WiFiMulti.h>
#include <WiFiClientSecure.h>

#include <WebSocketsServer.h>

WiFiMulti WiFiMulti;
WebSocketsServer webSocket = WebSocketsServer(81);

#define USE_SERIAL Serial

void hexdump(const void *mem, uint32_t len, uint8_t cols = 16) {
                        const uint8_t* src = (const uint8_t*) mem;
                        USE_SERIAL.printf("\n[HEXDUMP] Address: 0x%08X len: 0x%X (%d)", (ptrdiff_t)src, len, len);
                        for(uint32_t i = 0; i < len; i++) {
                                        if(i % cols == 0) {
                                                        USE_SERIAL.printf("\n[0x%08X] 0x%08X: ", (ptrdiff_t)src, i);
                                        }
                                        USE_SERIAL.printf("%02X ", *src);
                                        src++;
                        }
                        USE_SERIAL.printf("\n");
}
```

ESP 32 - WebSockets

```
void webSocketEvent(uint8_t num, WStype_t type, uint8_t * payload, size_t length) {

    switch(type) {
        case WStype_DISCONNECTED:
            USE_SERIAL.printf("[%u] Disconnected!\n", num);
            break;
        case WStype_CONNECTED:
            {
                IPAddress ip = webSocket.remoteIP(num);
                USE_SERIAL.printf("[%u] Connected from %d.%d.%d.%d url: %s\n", num, ip[0], ip[1],
ip[2], ip[3], payload);

                                                                                    // send
message to client

                        webSocket.sendTXT(num, "Better Connected");
            }
            break;
        case WStype_TEXT:
            USE_SERIAL.printf("[%u] get Text: %s\n", num, payload);

            // send message to client
            // webSocket.sendTXT(num, "message here");

            // send data to all connected clients
            // webSocket.broadcastTXT("message here");
            break;
        case WStype_BIN:
            USE_SERIAL.printf("[%u] get binary length: %u\n", num, length);
            hexdump(payload, length);

            // send message to client
            // webSocket.sendBIN(num, payload, length);
            break;
                                        case WStype_ERROR:

                                        case WStype_FRAGMENT_TEXT_START:
                                        case WStype_FRAGMENT_BIN_START:
                                        case WStype_FRAGMENT:
                                        case WStype_FRAGMENT_FIN:
                                                    break;
    }}
```

ESP 32 - WebSockets

```
void setup() {
    // USE_SERIAL.begin(921600);
    USE_SERIAL.begin(115200);

    //Serial.setDebugOutput(true);
    USE_SERIAL.setDebugOutput(true);

    USE_SERIAL.println();
    USE_SERIAL.println();
    USE_SERIAL.println();

    for(uint8_t t = 4; t > 0; t--) {
        USE_SERIAL.printf("[SETUP] BOOT WAIT %d...\n", t);
        USE_SERIAL.flush();
        delay(1000);
    }

    WiFiMulti.addAP("LivingStone Extra", "alatb@123454321");

    while(WiFiMulti.run() != WL_CONNECTED) {
      Serial.println("...");
        delay(100);
    }

    Serial.println("");
    Serial.println("WiFi connected.");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());

    webSocket.begin();
    webSocket.onEvent(webSocketEvent);
}

void loop() {
    webSocket.loop();
}
```

ESP 32 - WebSockets

```
<!DOCTYPE HTML>
<html>
<head>
<script type="text/javascript">
function WebSocketTest() {

if ("WebSocket" in window) {
alert("WebSocket is supported by your Browser!");

// Let us open a web socket
var ws = new WebSocket("ws://192.116.222.205:81");

ws.onopen = function() {

// Web Socket is connected, send data using send()
ws.send("Bigger");
alert("Message is sent...");
};

ws.onmessage = function (evt) {
var received_msg = evt.data;
alert("Message is received..."+received_msg);
};

ws.onclose = function() {

// websocket is closed.
alert("Connection is closed...");
};
} else {

// The browser doesn't support WebSocket
alert("WebSocket NOT supported by your Browser!");
}
}
</script>

</head>
<body>
<div id = "sse">
<a href = "javascript:WebSocketTest()">Run WebSocket</a>
</div>
</body>
</html>
```

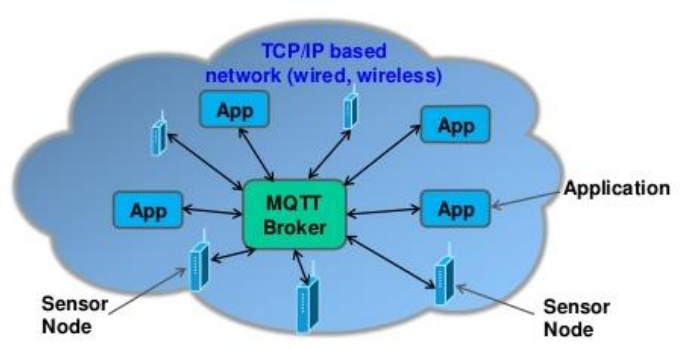
Web Protocols

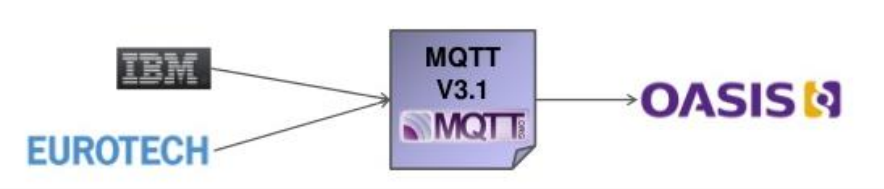
MQTT

Messaging & Protocols

MQTT

- הומצא בשנת 1999 בבדי ליצור קשר לוויני עם קווי אספקת נפט ולאוטומצית מכשירים ביתיים.
- הפרוטוקול שוחרר ללא תשלום 2010
- מיועד לאפליקציות M2M, WSN, ו- IoT ועומד להפוך לתקן ע"י OASIS.

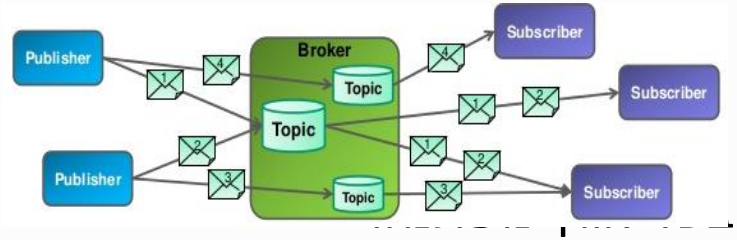




Messaging & Protocols

MQTT

- מבוסס תבנית `Publish/Subscribe`
- תקשורת אסינכרונית , 1:∞
- סט פקודות מצומצם: `Connect, Publish, Subscribe/Unsubscribe, Disconnect`
- פורמט העברת נתונים מינימליסטי
  - המסר מועבר כמערך בייטים פשוט.
  - ללא `Application Headers`.
  - הפרוטוקול דוחס ע"י מניפולציית ביטים שדות.
  - גודל חבילה מינימלית : 2 בייטים.
  - תמיכה במצבי נתק לרבות "צוואה" לפרסום המסר.
- עמידות באמצעות אסטרטגית `Stateful Roll-Forward`



MQTT

- מבני Topics היררכים ורישום מאפשר פילטרים (WildCards).
- 3 רמות שירות :
  - 0-שלח פעם אחד לכל היותר
  - 1-שלח לפחות פעם אחת
  - 2-שלח רק פעם אחת
- מסרים עוברים שמירה (Persistent)

Messaging & Protocols

- השוואה
  - **MQTT** אדיש לתוכן המועבר (תמיד מערך בייטים), HTTP לא.
  - **MQTT** – סט פקודות מצומצם יחסית ל-HTTP.
  - **MQTT** – מסר מינימלי באורך 2 בייטים, ל-HTTP תקורה ניכרת.
  - **MQTT** – מסוגל לשדר במגוון מתארים (1:1, 1:0, 1:*) , HTTP רק 1:1 (בסיוע מתווכים ניתן, POSTS רבים).

Messaging & Protocols

MQTT

- Messaging Protocol
- Simple
- On top of TCP
- Publish / Subscribe Architecture
- Binary protocol
- Minimal Overhead
- Designed for unreliable networks
- Data agnostic

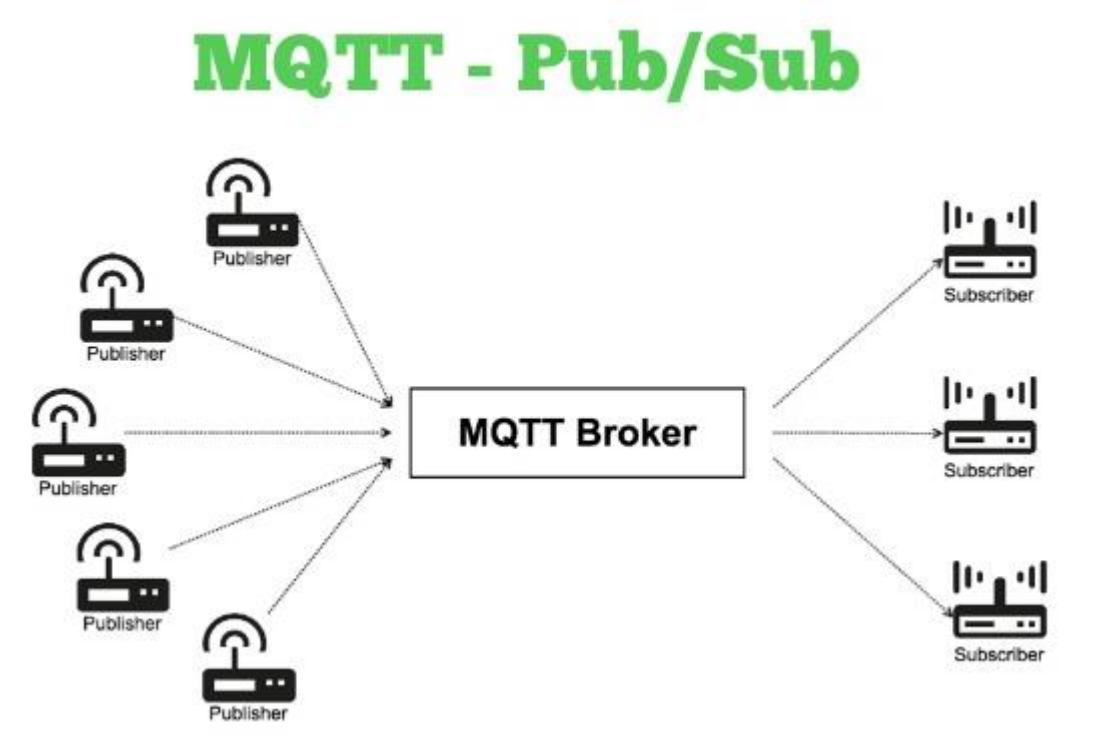
Use cases

- Push instead of Poll
- Bandwidth is at a premium
- Enterprise applications should interact with mobile applications
- Reliable delivery of messages over unreliable networks
- Constrained devices
- Low latency

Features

- Topic Wildcards
- 3 Quality of Service Levels
- Retained Messages
- Last Will and Testament
- Persistent Sessions
- Heartbeats

Messaging & Protocols



Messaging & Protocols

MQTT - Security

Protocol

- Username / Password
- Payload Encryption

Transport

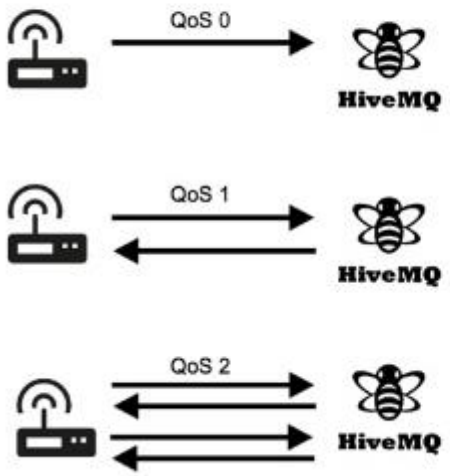
- TLS
- Client certificate authentication

Broker

- Publish / Subscribe Permissions
- Integration to other systems (databases, APIs, ...)

Messaging & Protocols

Quality of Service Levels



QoS 0

At most once delivery

The message is delivered once or never.

QoS 1

At least once delivery

The message is delivered once or more.

QoS 2

Exactly once delivery

The message is delivered exactly once.

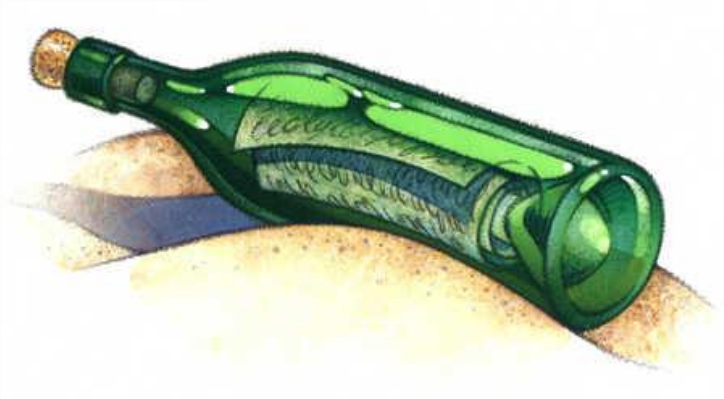
Messaging & Protocols

Last Will and Testament

- Clients can specify a LWT
- Broker publishes the LWT message on behalf of the client on “death”
- Useful for reporting problems
- Real push on device “death”
- Mostly used for reporting the connection status of a device

Messaging & Protocols

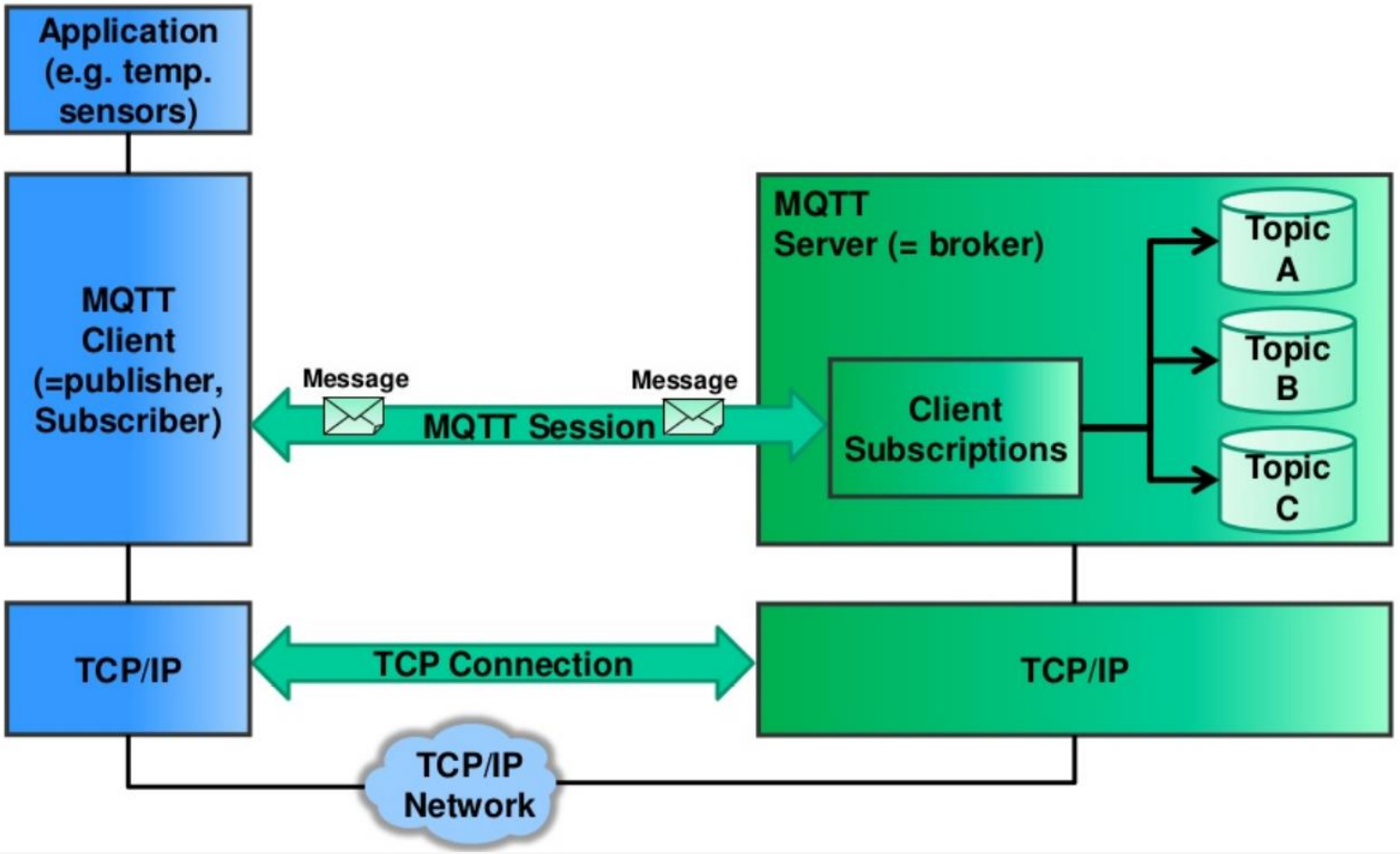
Retained Messages



- Last known good value
- Last message is stored for a topic
- Publishing clients decide if the message should be retained
- Clients automatically receive the retained message after subscribing

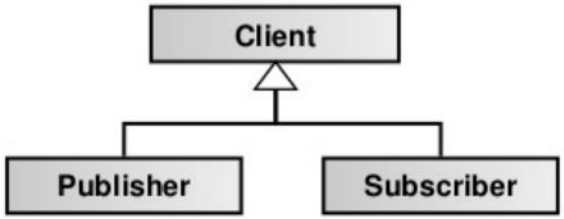
MQTT MODEL

The core elements of MQTT are clients, servers (=brokers), sessions, subscriptions and topics.



MQTT MODEL

MQTT client (=publisher, subscriber):
Clients subscribe to topics to publish and receive messages.
Thus subscriber and publisher are special roles of a client.



MQTT server (=broker):
Servers run topics, i.e. receive subscriptions from clients on topics, receive messages from clients and forward these, based on client's subscriptions, to interested clients.

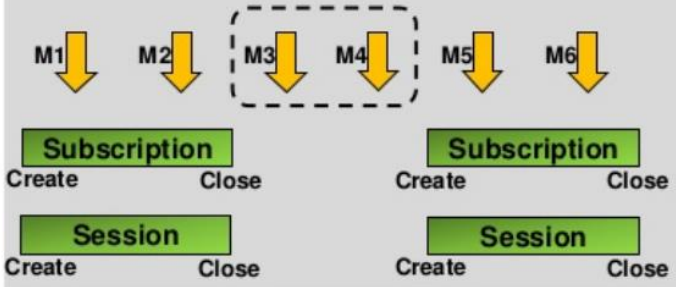
Topic:
Technically, topics are message queues. Topics support the publish/subscribe pattern for clients.
Logically, topics allow clients to exchange information with defined semantics.
Example topic: Temperature sensor data of a building.

MQTT MODEL

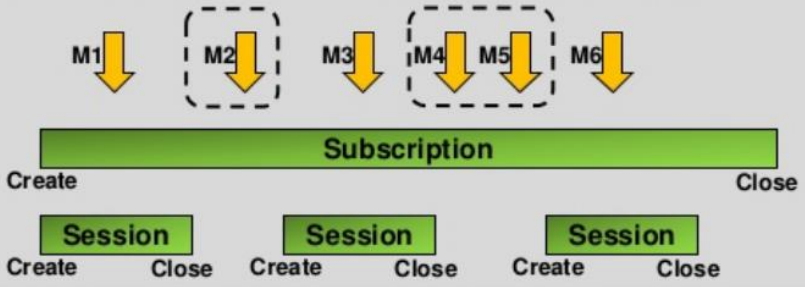
Session:
A session identifies a (possibly temporary) attachment of a client to a server. All communication between client and server takes place as part of a session.

Subscription:
Unlike sessions, a subscription logically attaches a client to a topic. When subscribed to a topic, a client can exchange messages with a topic.
Subscriptions can be «transient» or «durable», depending on the clean session flag in the CONNECT message:

«Transient» subscription ends with session:
Messages M3 and M4 are not received by the client



«Durable» subscription:
Messages M2, M4 and M5 are not lost but will be received by the client as soon as it creates / opens a new session.



Message:
Messages are the units of data exchange between topic clients.
MQTT is agnostic to the internal structure of messages.

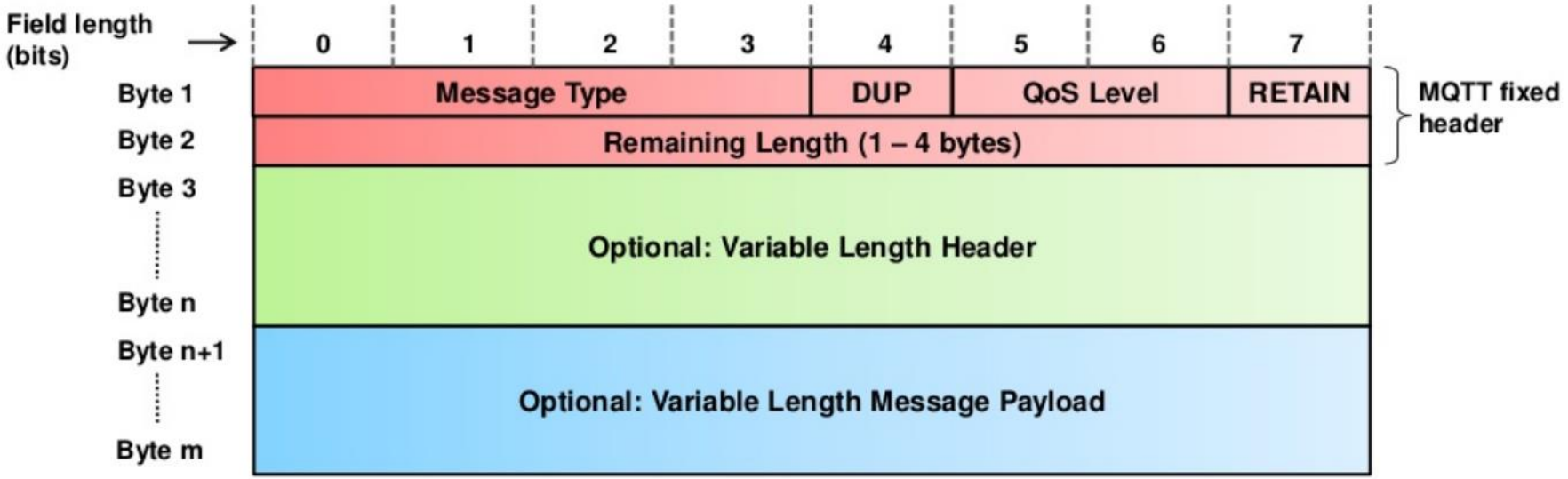
MQTT MODEL

Message format:

MQTT messages contain a mandatory fixed-length header (2 bytes) and an optional message-specific variable length header and message payload.

Optional fields usually complicate protocol processing.
However, MQTT is optimized for bandwidth constrained and unreliable networks (typically wireless networks), so optional fields are used to reduce data transmissions as much as possible.

MQTT uses network byte and bit ordering.



MQTT MODEL

Overview of fixed header fields:

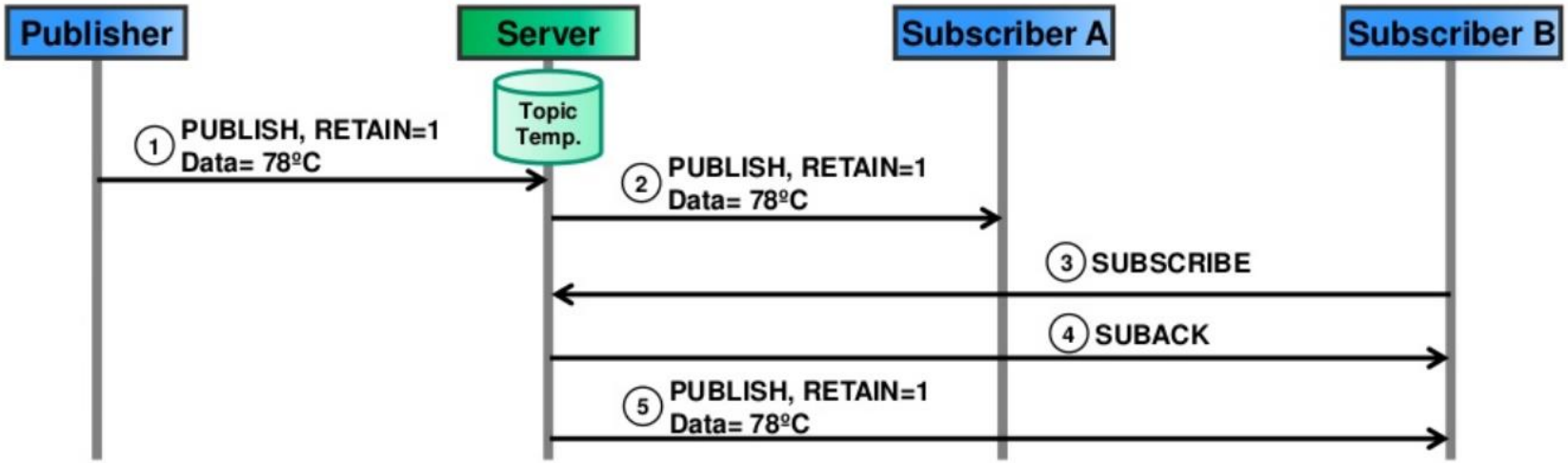
Message fixed header field	Description / Values	
Message Type	0: Reserved	8: SUBSCRIBE
	1: CONNECT	9: SUBACK
	2: CONNACK	10: UNSUBSCRIBE
	3: PUBLISH	11: UNSUBACK
	4: PUBACK	12: PINGREQ
	5: PUBREC	13: PINGRESP
	6: PUBREL	14: DISCONNECT
	7: PUBCOMP	15: Reserved
DUP	Duplicate message flag. Indicates to 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	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. Further details see MQTT QoS.	
RETAIN	1: Instructs the server to retain the last received PUBLISH message and deliver it as a first message to new subscriptions. Further details see RETAIN (keep last message).	
Remaining Length	Indicates the number of remaining bytes in the message, i.e. the length of the (optional) variable length header and (optional) payload. Further details see Remaining length (RL).	

MQTT MODEL

RETAIN (keep last message):
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.

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.



MQTT MODEL

Remaining length (RL):

The remaining length field encodes the sum of the lengths of:

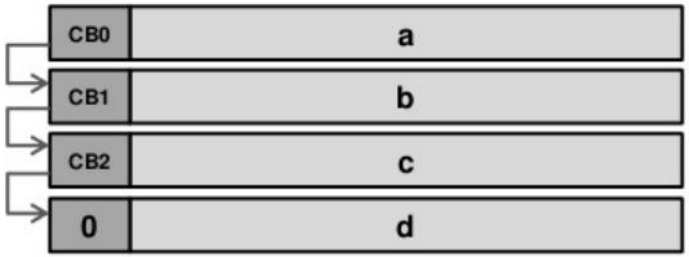
a. (Optional) variable length header

b. (Optional) payload

To save bits, remaining length is a variable length field with 1…4 bytes.

The most significant bit of a length field byte has the meaning «continuation bit» (CB). If more bytes follow, it is set to 1.

Remaining length is encoded as $a * 128^0 + b * 128^1 + c * 128^2 + d * 128^3$ 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 ($d * 128^3$)

Key:
LSB: Least Significant Byte
MSB: Most Significant Byte

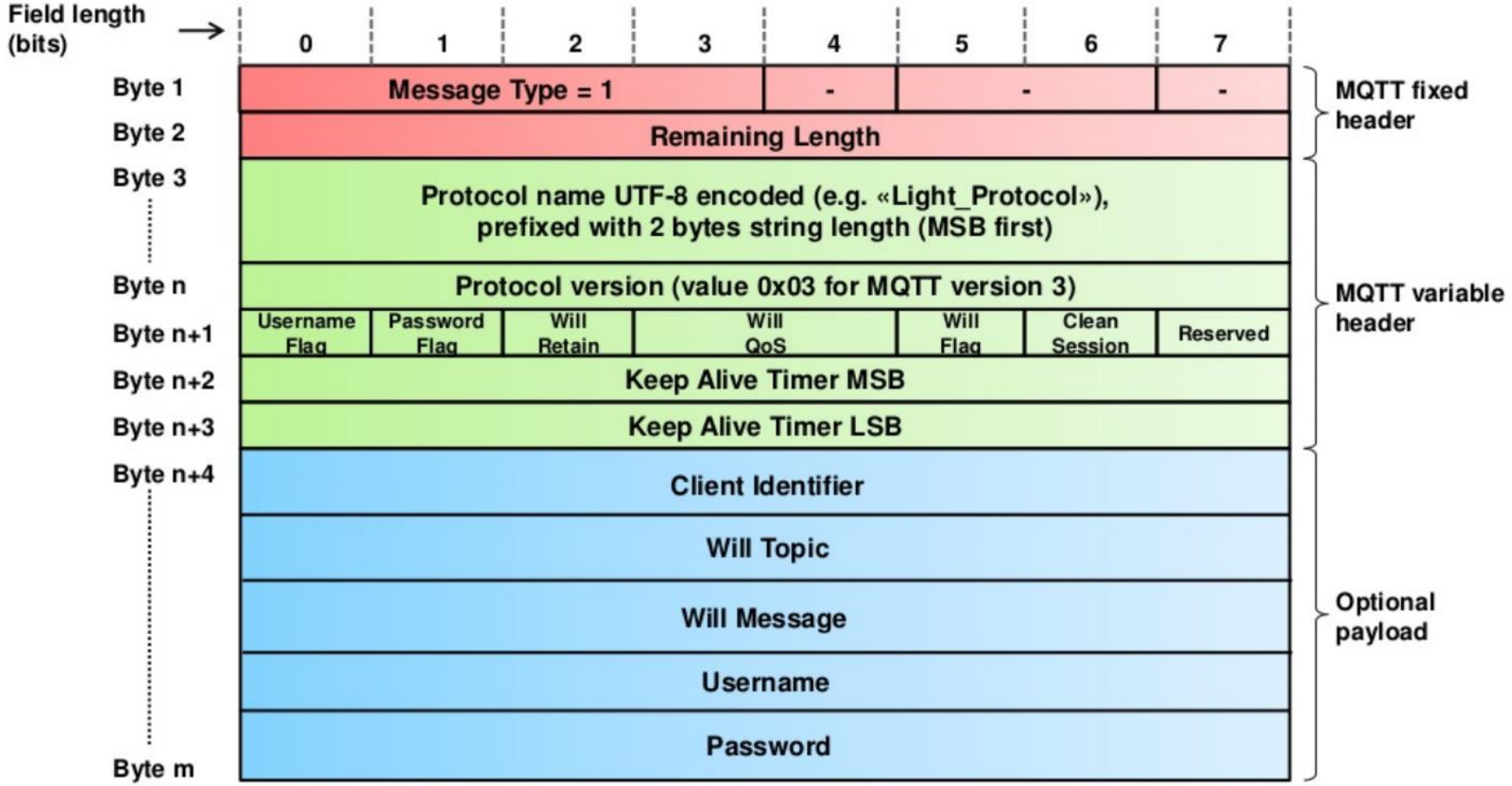
Example 1: **RL = 364 = $108*128^0+2*128^1$ → a=108, CB0=1, b=2, CB1=0, c=0, d=0, CB2=0**

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 MODEL

CONNECT message format:

The CONNECT message contains many session-related information as optional header fields.



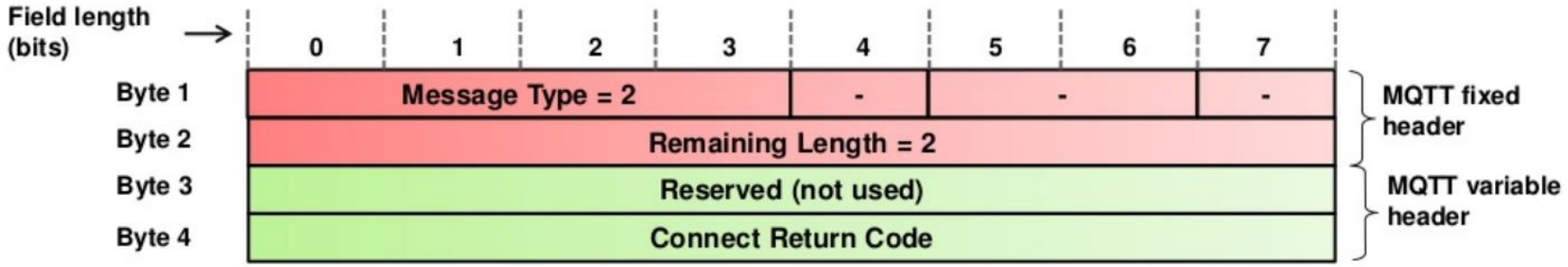
MQTT MODEL

Overview CONNECT message fields:

CONNECT message field	Description / Values
Protocol Name	UTF-8 encoded protocol name string. Example: «Light_Protocol»
Protocol Version	Value 3 for MQTT V3.
Username Flag	If set to 1 indicates that payload contains a username.
Password Flag	If set to 1 indicates that payload contains a password. If username flag is set, password flag and password must be set as well.
Will Retain	If set to 1 indicates to server that it should retain a Will message for the client which is published in case the client disconnects unexpectedly.
Will QoS	Specifies the QoS level for a Will message.
Will Flag	Indicates that the message contains a Will message in the payload along with Will retain and Will QoS flags. More details see MQTT will message.
Clean Session	If set to 1, the server discards any previous information about the (re)-connecting client (clean new session). If set to 0, the server keeps the subscriptions of a disconnecting client including storing QoS level 1 and 2 messages for this client. When the client reconnects, the server publishes the stored messages to the client.
Keep Alive Timer	Used by the server to detect broken connections to the client. More details see Keepalive timer.
Client Identifier	The client identifier (between 1 and 23 characters)uniquely identifies the client to the server. The client identifier must be unique across all clients connecting to a server.
Will Topic	Will topic to which a will message is published if the will flag is set.
Will Message	Will message to be puslished if will flag is set.
Username and Password	Username and password if the corresponding flags are set.

MQTT MODEL

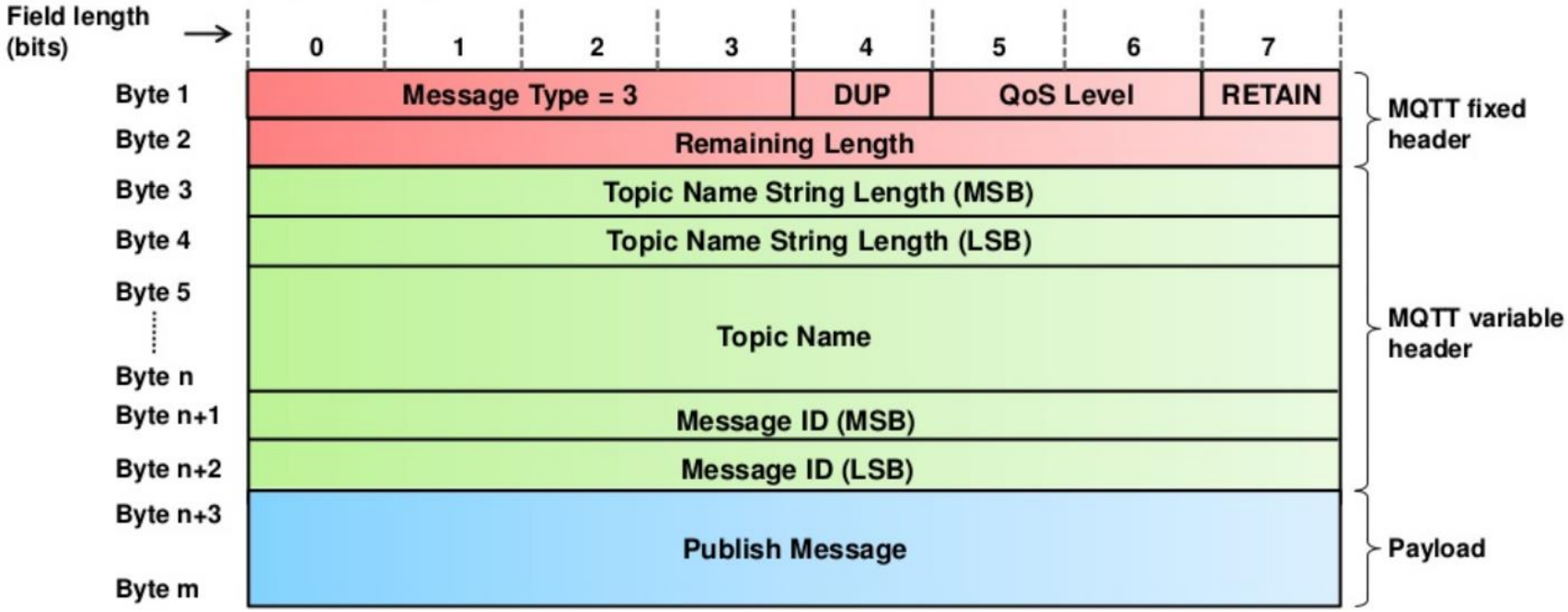
CONNACK message format:



CONNACK message field	Description / Values
Reserved	Reserved field for future use.
Connect Return Code	0: Connection Accepted 1: Connection Refused, reason = unacceptable protocol version 2: Connection Refused, reason = identifier rejected 3: Connection Refused, reason = server unavailable 4: Connection Refused, reason = bad user name or password 5: Connection Refused, reason = not authorized 6-255: Reserved for future use

MQTT MODEL

PUBLISH message format:



PUBLISH message field	Description / Values
Topic Name with Topic Name String Length	Name of topic to which the message is published. The first 2 bytes of the topic name field indicate the topic name string length.
Message ID	A message ID is present if QoS is 1 (At-least-once delivery, acknowledged delivery) or 2 (Exactly-once delivery).
Publish Message	Message as an array of bytes. The structure of the publish message is application-specific.

MQTT MODEL

PUBACK message format:

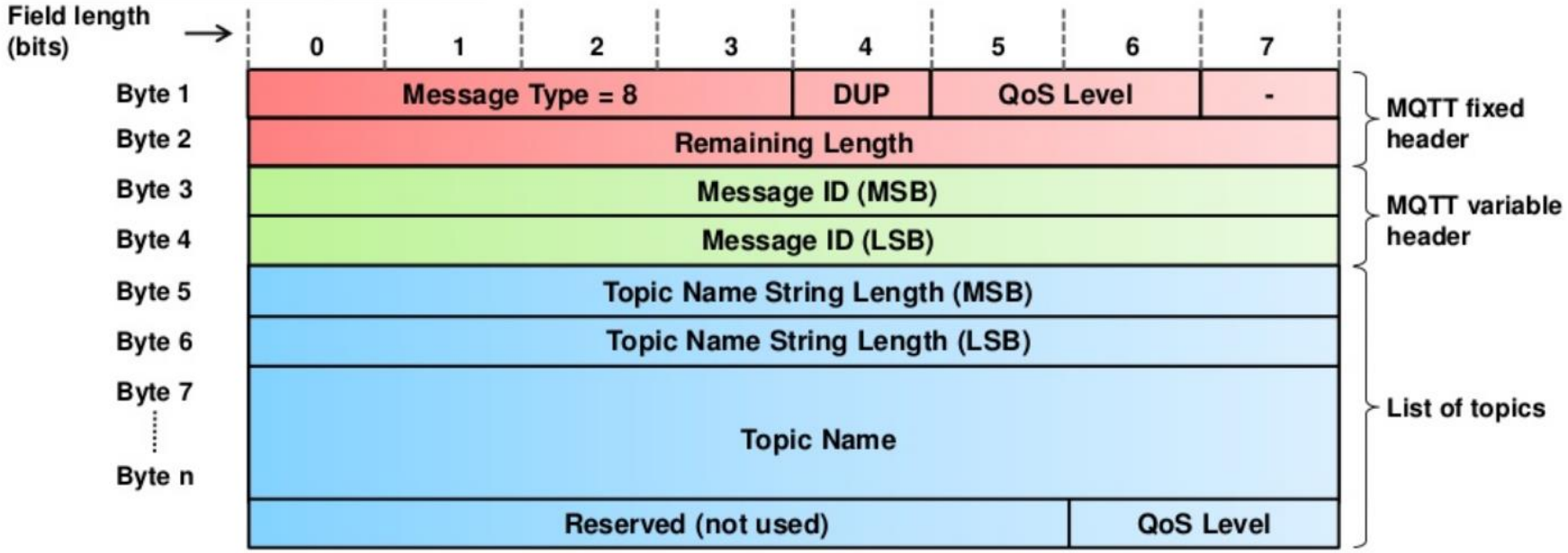
PUBACK message field	Description / Values
Message ID	The message ID of the PUBLISH message to be acknowledged.

PUBREC message format:

PUBREC message field	Description / Values
Message ID	The message ID of the PUBLISH message to be acknowledged.

MQTT MODEL

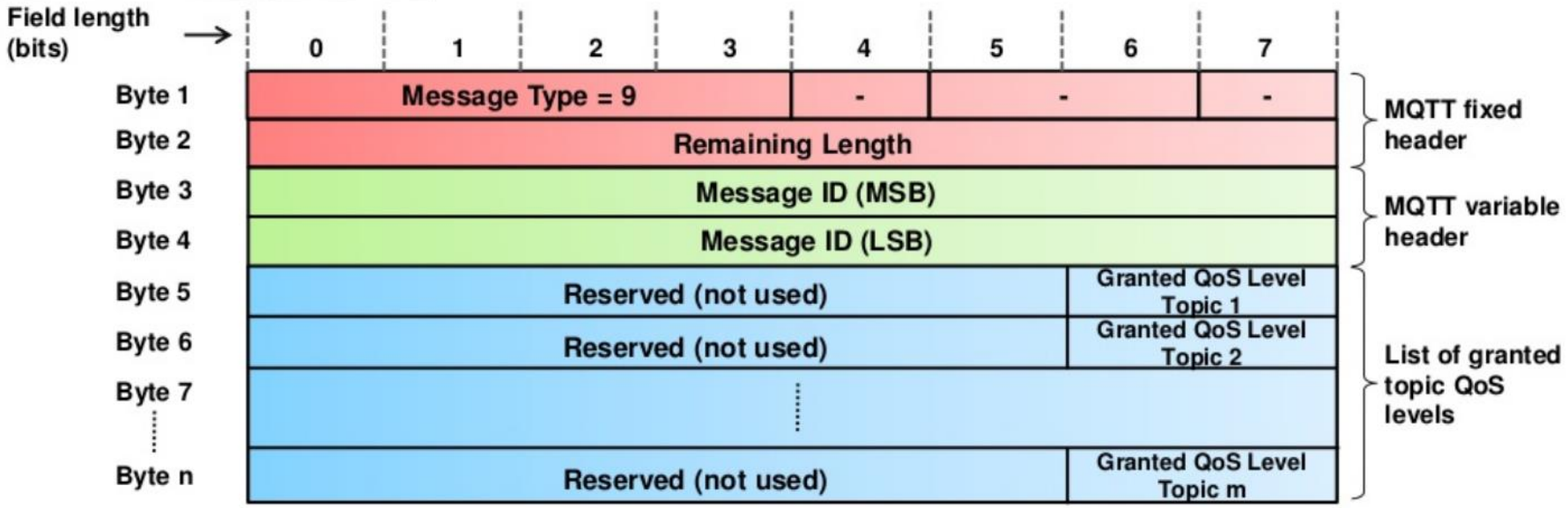
SUBSCRIBE message format:



SUBSCRIBE message field	Description / Values
Message ID	The message ID field is used for acknowledgment of the SUBSCRIBE message since these have a QoS level of 1.
Topic Name with Topic Name String Length	Name of topic to which the client subscribes. The first 2 bytes of the topic name field indicate the topic name string length. Topic name strings can contain wildcard characters as explained under Topic wildcards. Multiple topic names along with their requested QoS level may appear in a SUBSCRIBE message.
QoS Level	QoS level at which the clients wants to receive messages from the given topic.

MQTT MODEL

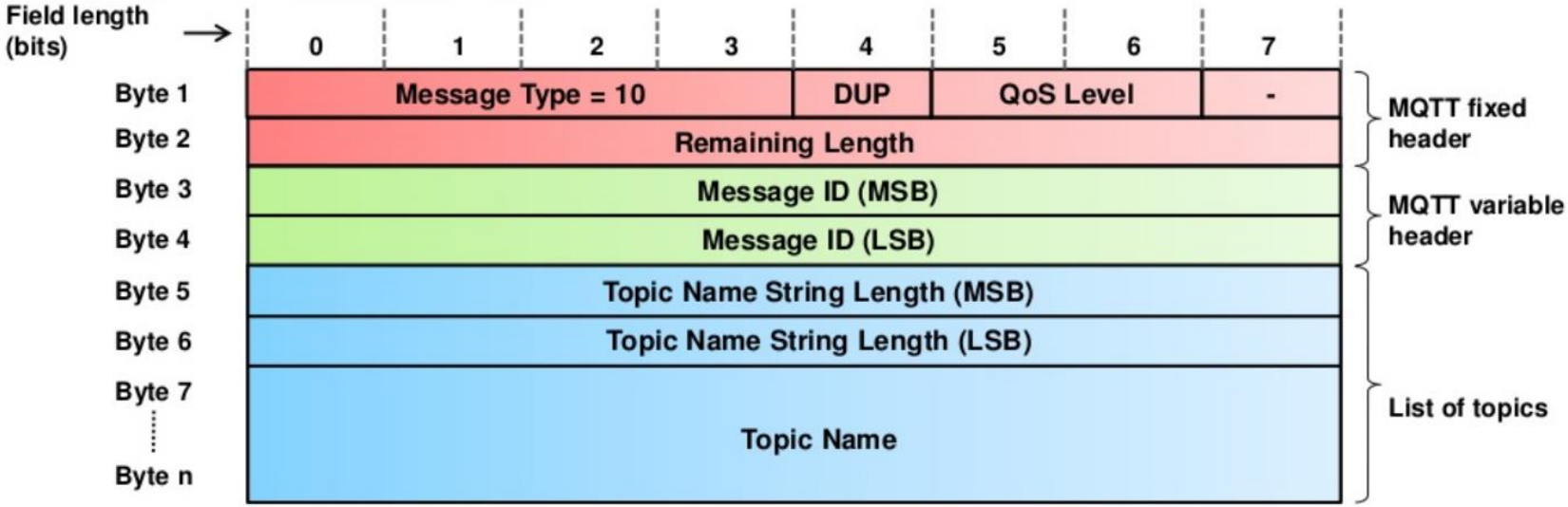
SUBACK message format:



SUBACK message field	Description / Values
Message ID	Message ID of the SUBSCRIBE message to be acknowledged.
Granted QoS Level for Topic	List of granted QoS levels for the topics list from the SUBSCRIBE message.

MQTT MODEL

UNSUBSCRIBE message format:



UNSUBSCRIBE message field	Description / Values
Message ID	The message ID field is used for acknowledgment of the UNSUBSCRIBE message (UNSUBSCRIBE messages have a QoS level of 1).
Topic Name with Topic Name String Length	Name of topic from which the client wants to unsubscribe. The first 2 bytes of the topic name field indicate the topic name string length. Topic name strings can contain wildcard characters as explained under Topic wildcards. Multiple topic names may appear in an UNSUBSCRIBE message.

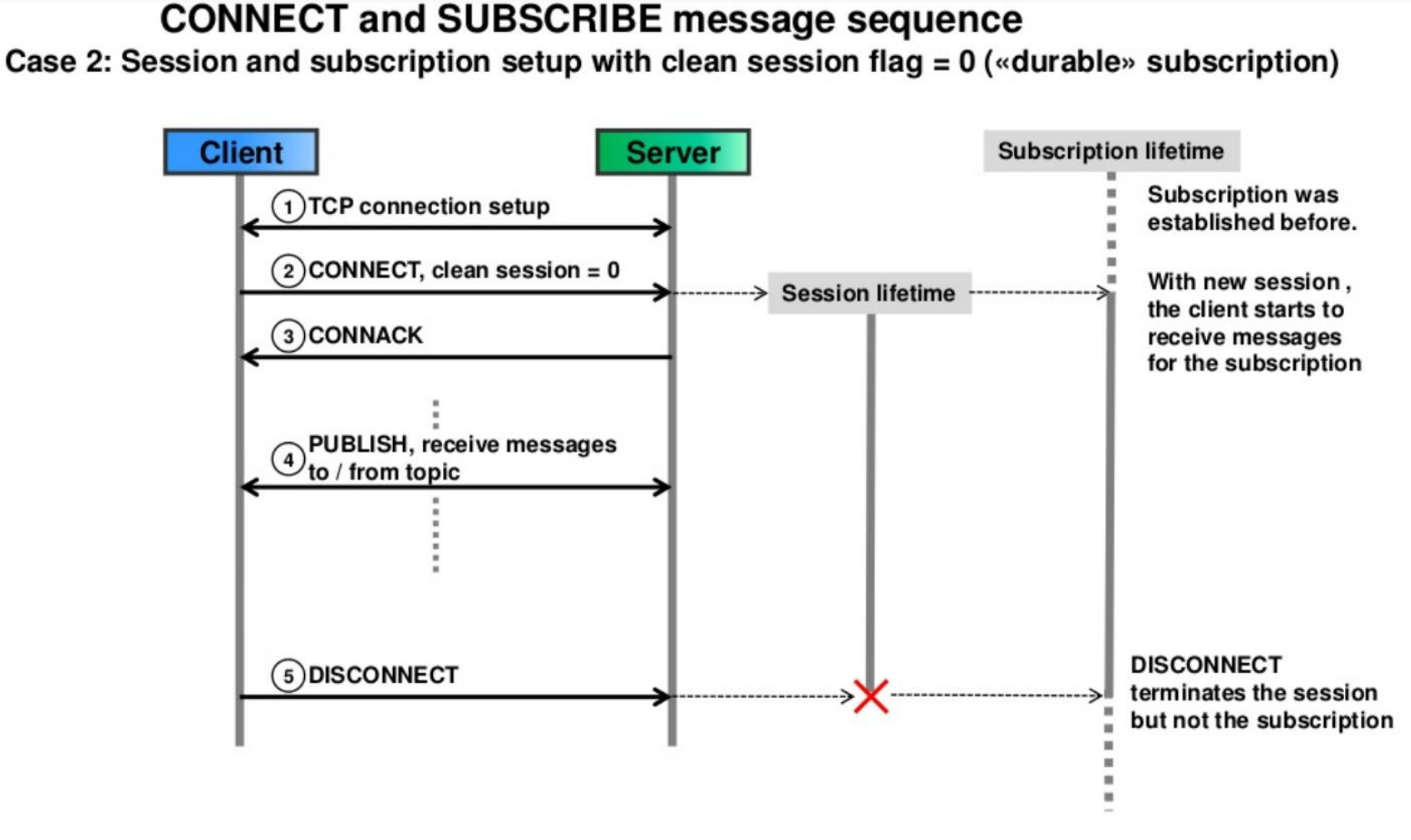
MQTT MODEL

UNSUBACK message format:

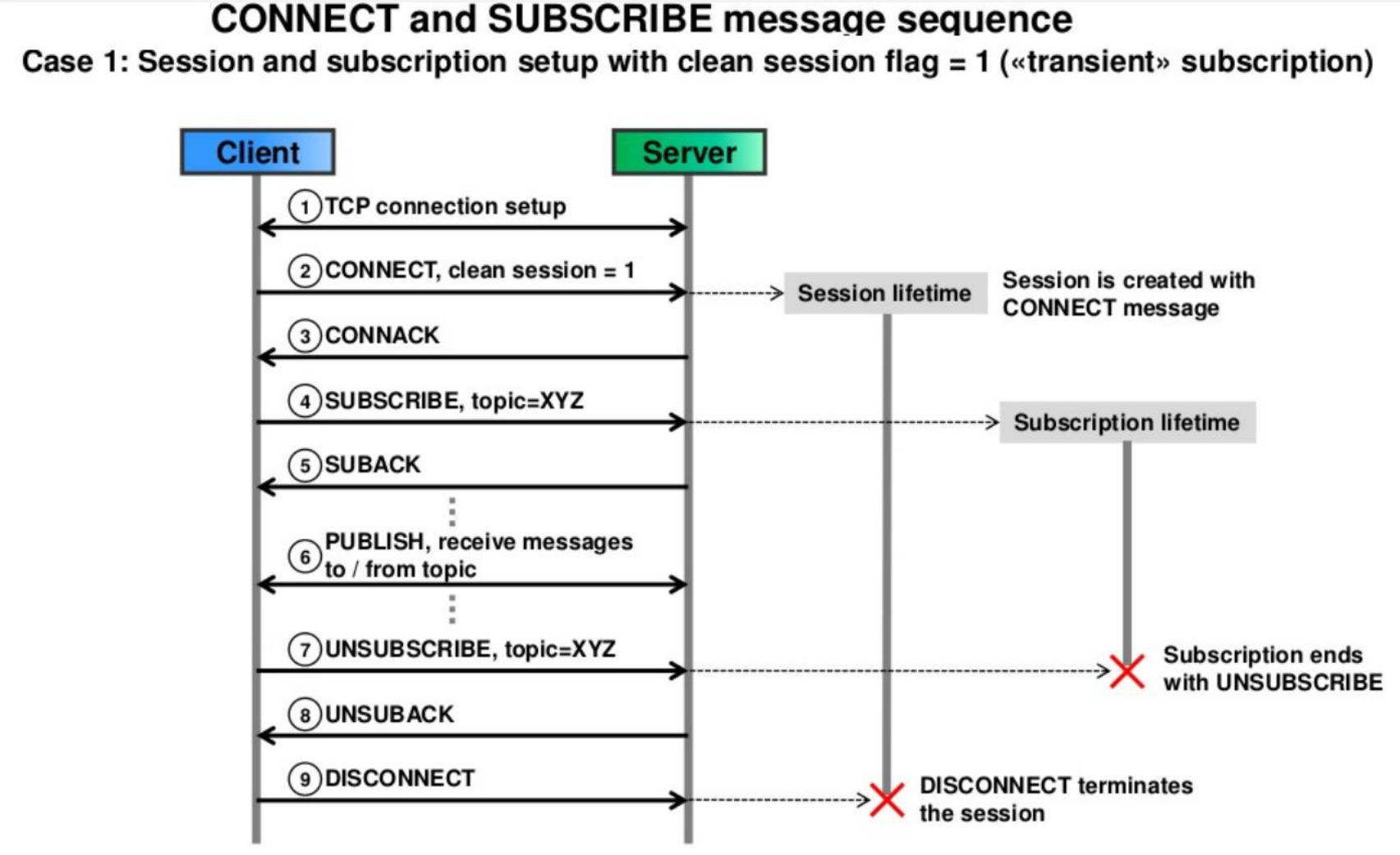
UNSUBACK message field	Description / Values
Message ID	The message ID of the UNSUBSCRIBE message to be acknowledged.

DISCONNECT, PINGREQ, PINGRESP message formats:

MQTT MODEL


CONNECT and SUBSCRIBE message sequence
Case 2: Session and subscription setup with clean session flag = 0 («durable» subscription)
Client
Server
Subscription lifetime
1 TCP connection setup
2 CONNECT, clean session = 0
Session lifetime
Subscription was established before.
With new session , the client starts to receive messages for the subscription
3 CONNACK
4 PUBLISH, receive messages to / from topic
5 DISCONNECT
DISCONNECT terminates the session but not the subscription

MQTT MODEL

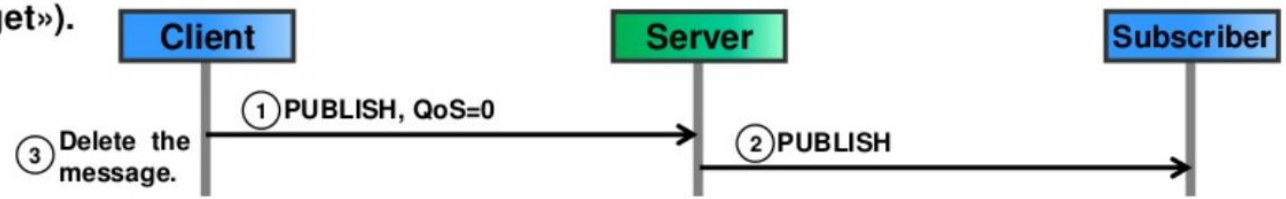

CONNECT and SUBSCRIBE message sequence
Case 1: Session and subscription setup with clean session flag = 1 («transient» subscription)
Client
Server
1 TCP connection setup
2 CONNECT, clean session = 1
Session lifetime
Session is created with CONNECT message
3 CONNACK
4 SUBSCRIBE, topic=XYZ
Subscription lifetime
5 SUBACK
6 PUBLISH, receive messages to / from topic
7 UNSUBSCRIBE, topic=XYZ
Subscription ends with UNSUBSCRIBE
8 UNSUBACK
9 DISCONNECT
DISCONNECT terminates the session

MQTT MODEL

PUBLISH message flows

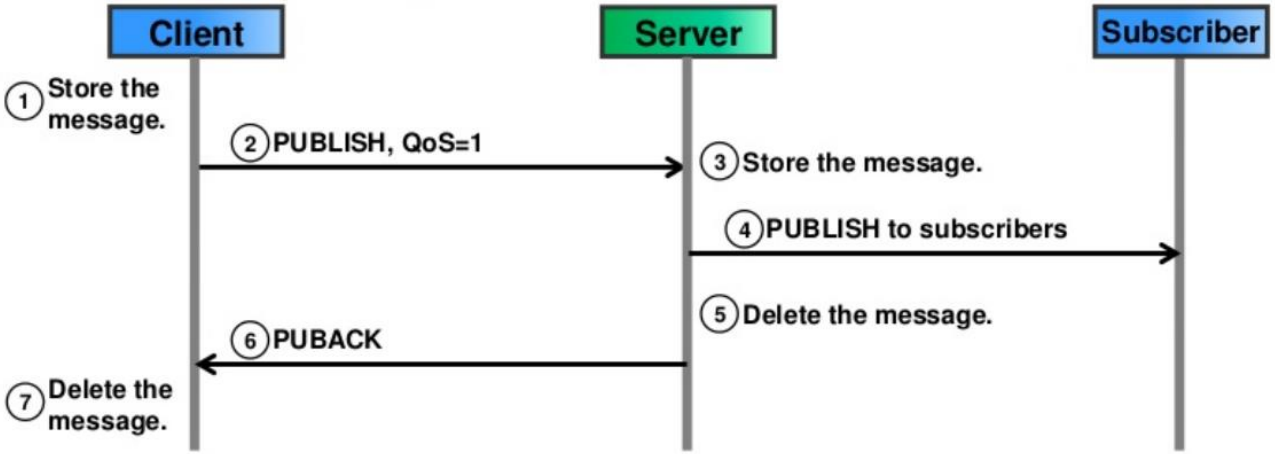
QoS level 0:

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.

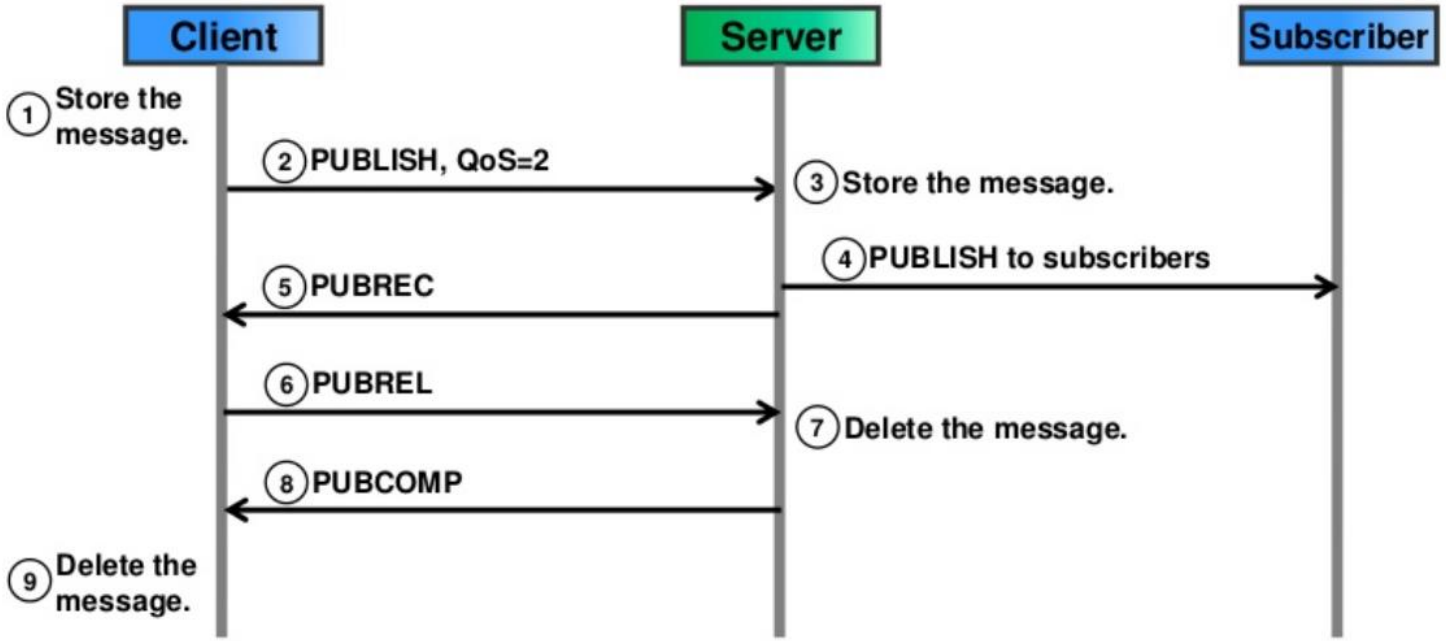


MQTT MODEL

PUBLISH message flows

QoS level 2:

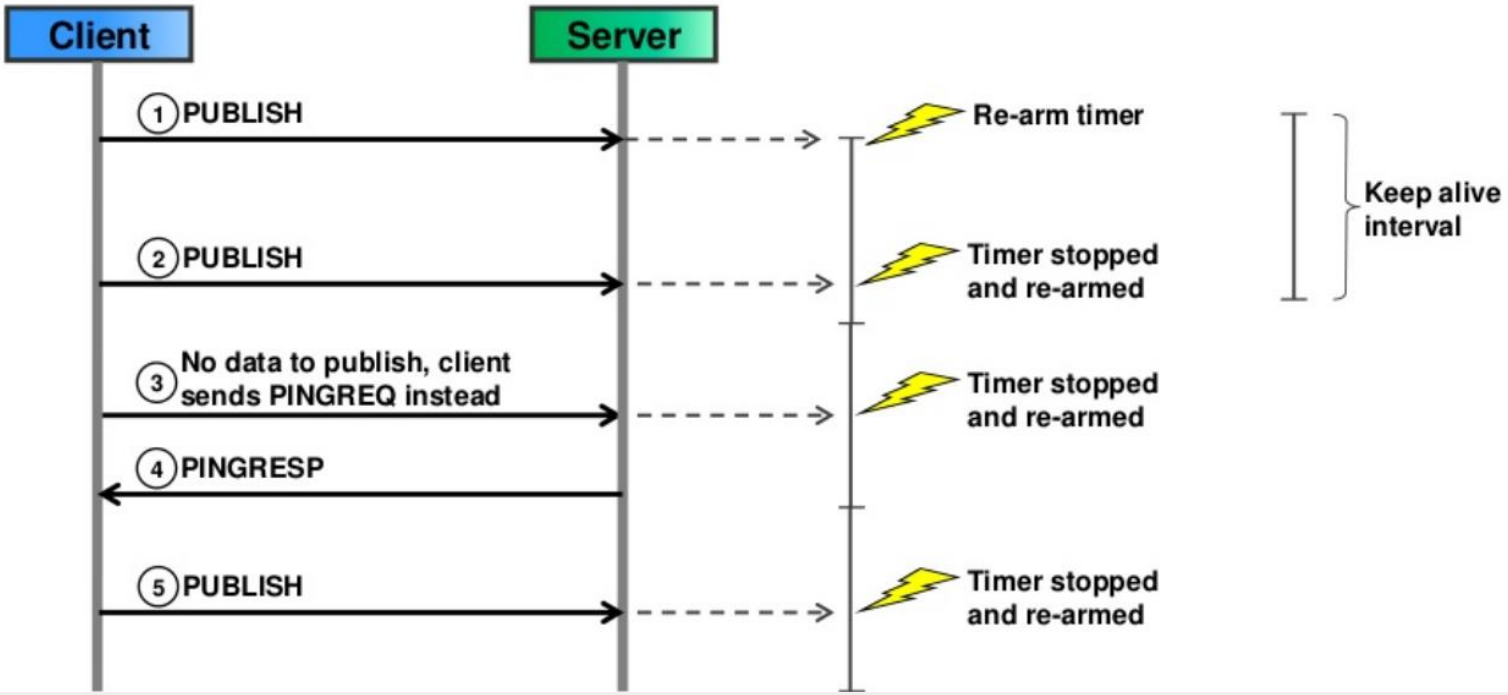
QoS level 2 affords the highest quality delivery semantics exactly-once, but comes with the cost of additional control messages.



MQTT MODEL

Keep alive timer, breath of live with PINGREQ

The keep alive timer defines the maximum allowable time interval between client messages.
The timer is used by the server to check client's connection status.
After 1.5 * keepalive-time is elapsed, the server disconnects the client (client is granted a grace period of an additional 0.5 keepalive-time).
In the absence of data to be sent, the client sends a PINGREQ message instead.
Typical value for keepalive timer are a couple of minutes.



MQTT will message

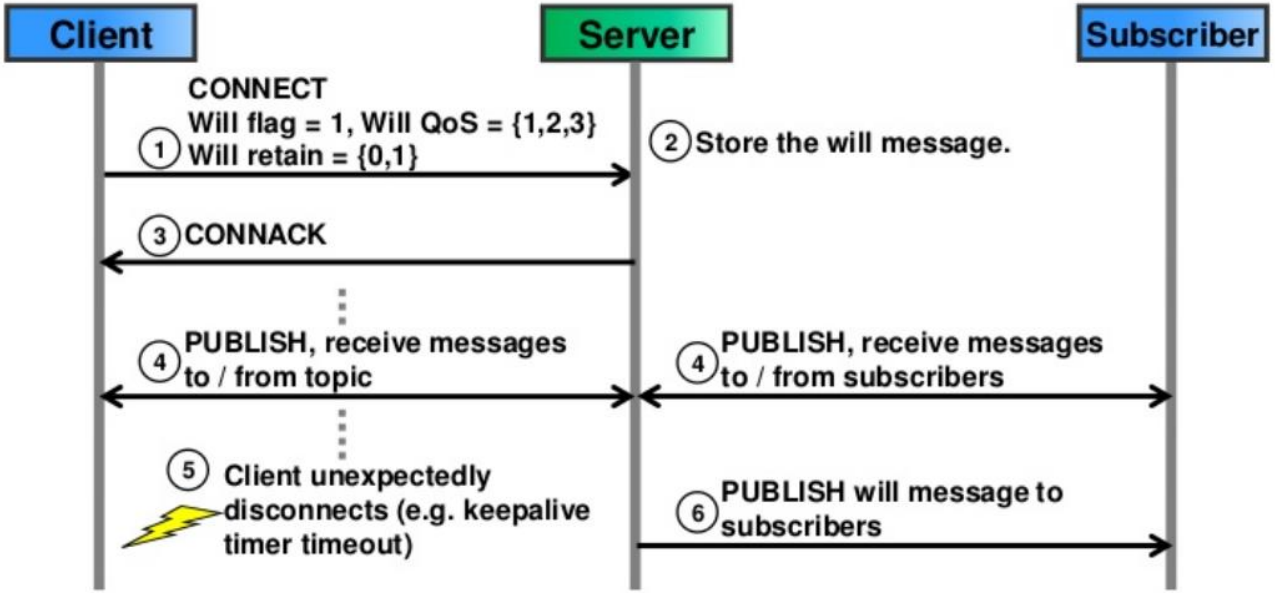
Problem:

In case of an unexpected client disconnect, depending applications (subscribers) do not receive any notification of the client's demise.

MQTT solution:

Client can specify a will message along with a will QoS and will retain flag in the CONNECT message payload.

If the client unexpectedly disconnects, the server sends the will message on behalf of the client to all subscribers («last will»).



MQTT MODEL

Topic wildcards

Problem:

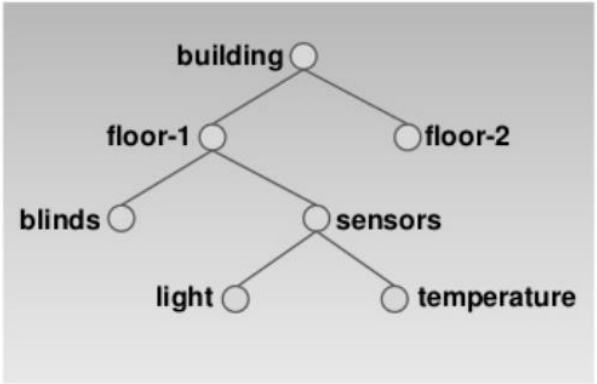
Subscribers are often interested in a great number of topics.
Individually subscribing to each named topic is time- and resource-consuming.

MQTT solution:

Topics can be hierarchically organized through wildcards with path-type topic strings and the wildcard characters '+' and '#'.
Subscribers can subscribe for an entire sub-tree of topics thus receiving messages published to any of the sub-tree's nodes.

Topic string special character	Description
/	Topic level separator. Example: *building / floor-1 / sensors / temperature*
+	Single level wildcard. Matches one topic level. Examples: *building / floor-1 / +* (matches *building / floor-1 / blinds* and *building / floor-1 / sensors*) *building / + / sensors* (matches *building / floor-1 / sensors* and *building / floor-2 / sensors*)
#	Multi level wildcard. Matches multiple topic levels. Examples: *building / floor-1 / #* (matches all nodes under *building / floor-1*) *building / # / sensors* (invalid, '#' must be last character in topic string)

Example topic tree:



MQTT MODEL

MQTT provides the typical delivery quality of service (QoS) levels of message oriented middleware.
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 levels on top of TCP.

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 (consumers of the data) will anyway integrate the values over time and loss of individual samples is not relevant.

MQTT MODEL

QoS level 1:

At-lest-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 evaluating the message ID field.

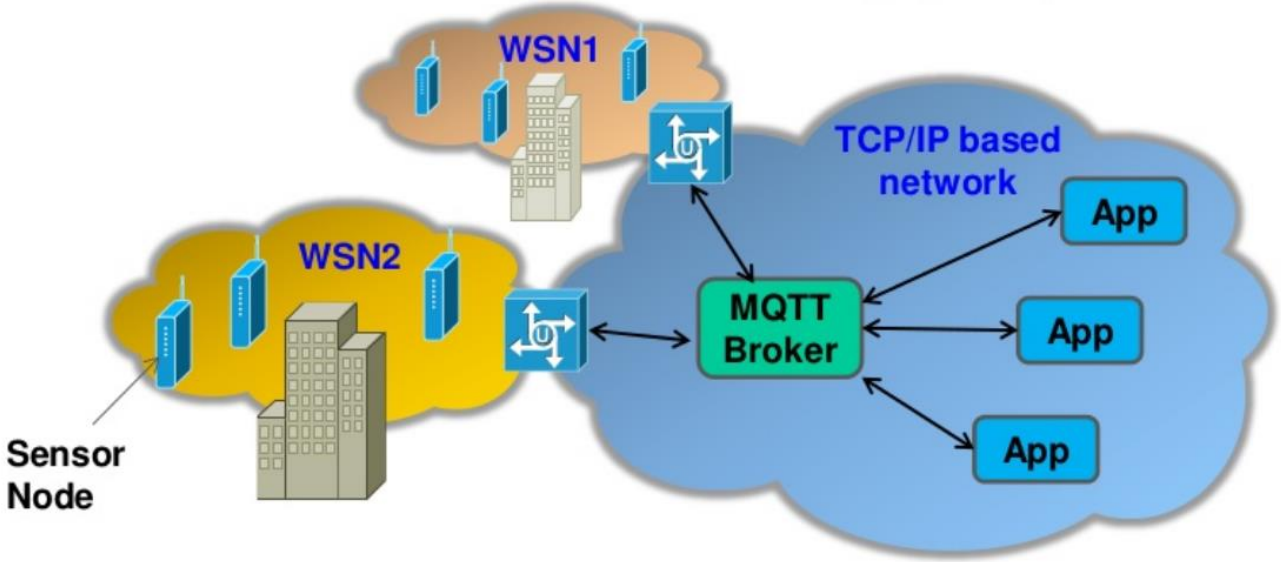
QoS level 2:

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.

MQTT MODEL

MQTT-S

WSNs (Wireless Sensor Networks) usually do not have TCP/IP as transport layer. They have their own protocol stack such as ZigBee on top of IEEE 802.15.4 MAC layer. Thus, MQTT which is based on TCP/IP cannot be directly run on WSNs.
WSNs are connected to traditional TCP/IP networks through gateway devices.



MQTT-S is an extension of MQTT for WSNs.
MQTT-S is aimed at constrained low-end devices, usually running on a batttery, such as ZigBee devices.

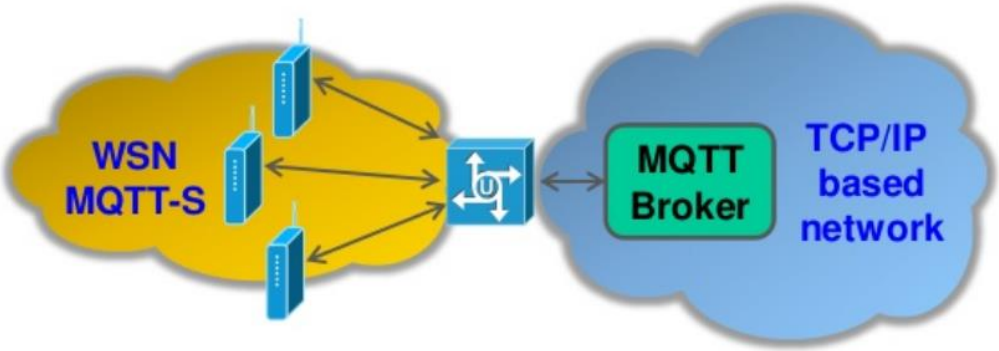
MQTT MODEL

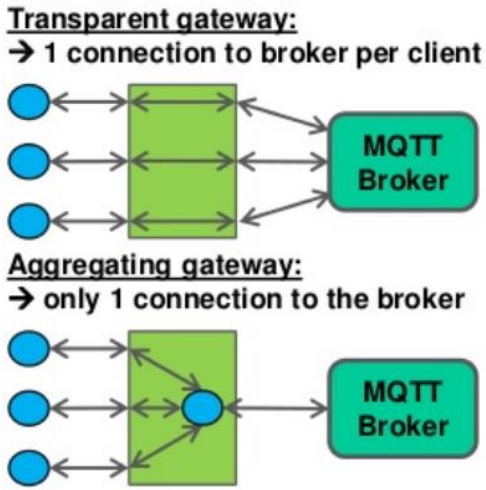
MQTT-S

MQTT-S is a largely based on MQTT, but implements some important optimizations for wireless networks:

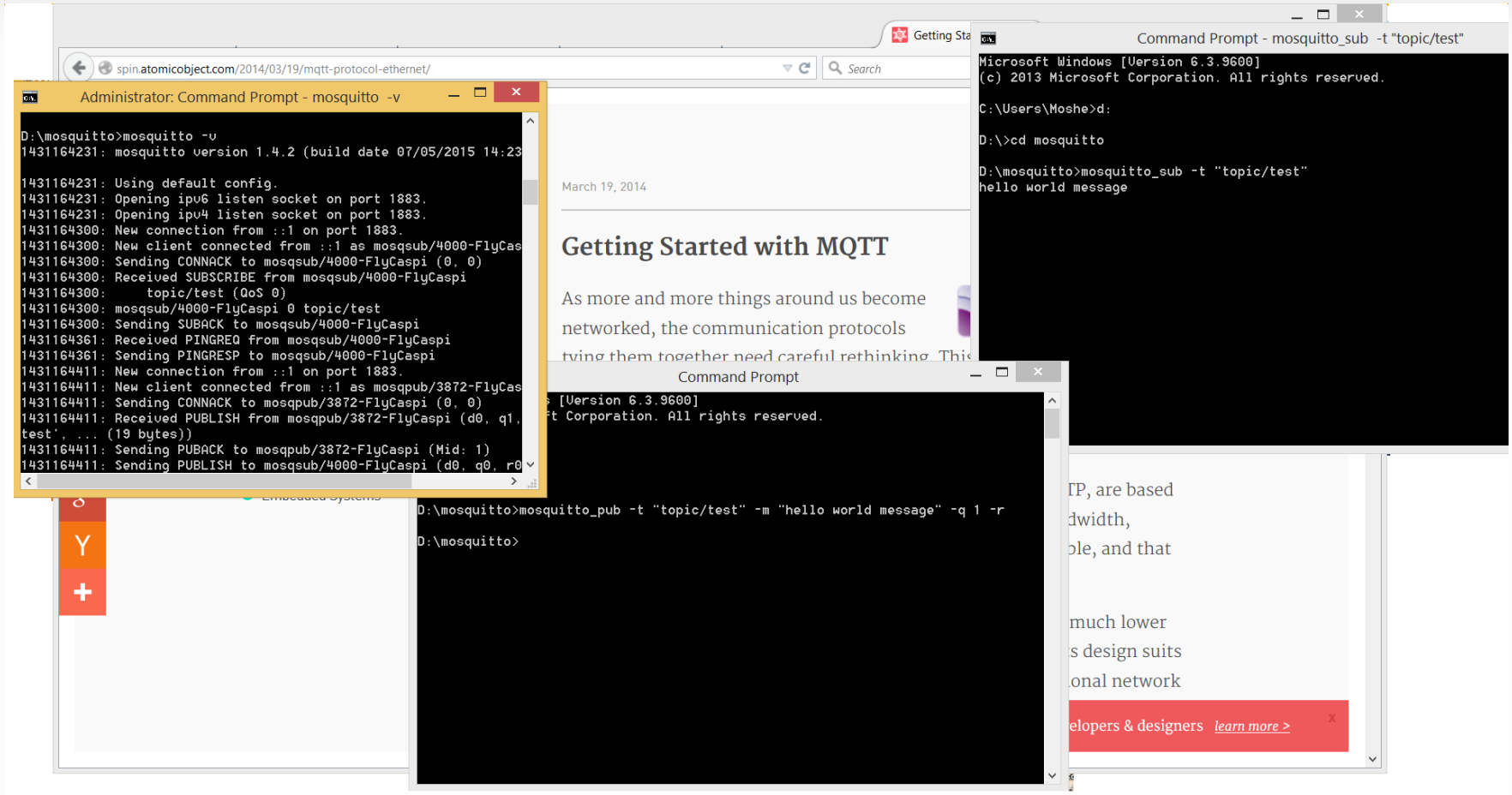
- **Topic string replaced by a topic ID (fewer bytes necessary)**
- **Predefined topic IDs that do not require a registration**
- **Discovery procedure for clients to find brokers (no need to statically configure broker addresses)**
- **Persistent will message (in addition to persistent subscriptions)**
- **Off-line keepalive supporting sleeping clients (will receive buffered messages from the server once they wake up)**

MQTT-S gateways (transparent or aggregating) connect MQTT-S domains (WSNs) with MQTT domains (traditional TCP/IP based networks).





MQTT Demo



http://spin.atomicobject.com/2014/03/19/mqtt-protocol-ethernet/

WiFi ESP 32 - MQTT

```
/*********
  Rui Santos
  Complete project details at https://randomnerdtutorials.com
*********/

#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace the next variables with your SSID/Password combination
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

// Add your MQTT Broker IP address, example:
//const char* mqtt_server = "192.168.1.144";
const char* mqtt_server = "YOUR_MQTT_BROKER_IP_ADDRESS";

WiFiClient espClient;
PubSubClient client(espClient);
long lastMsg = 0;
char msg[50];
int value = 0;

//uncomment the following lines if you're using SPI
/*#include <SPI.h>
#define BME_SCK 18
#define BME_MISO 19
#define BME_MOSI 23
#define BME_CS 5*/

Adafruit_BME280 bme; // I2C
//Adafruit_BME280 bme(BME_CS); // hardware SPI
//Adafruit_BME280 bme(BME_CS, BME_MOSI, BME_MISO, BME_SCK); // software SPI
float temperature = 0;
float humidity = 0;

// LED Pin
const int ledPin = 4;
```

```
void setup() {
  Serial.begin(115200);
  // default settings
  // (you can also pass in a Wire library object like &Wire2)
  //status = bme.begin();
  if (!bme.begin(0x76)) {
    Serial.println("Could not find a valid BME280 sensor, check wiring!");
    while (1);
  }
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);

  pinMode(ledPin, OUTPUT);
}

void setup_wifi() {
  delay(10);
  // We start by connecting to a WiFi network
  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.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

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

https://techtutorialsx.com/2017/04/24/esp32-publishing-messages-to-mqtt-topi

WiFi ESP 32 - MQTT

https://techtutorialsx.com/2017/04/24/esp32-publishing-messages-to-mqtt-topic/

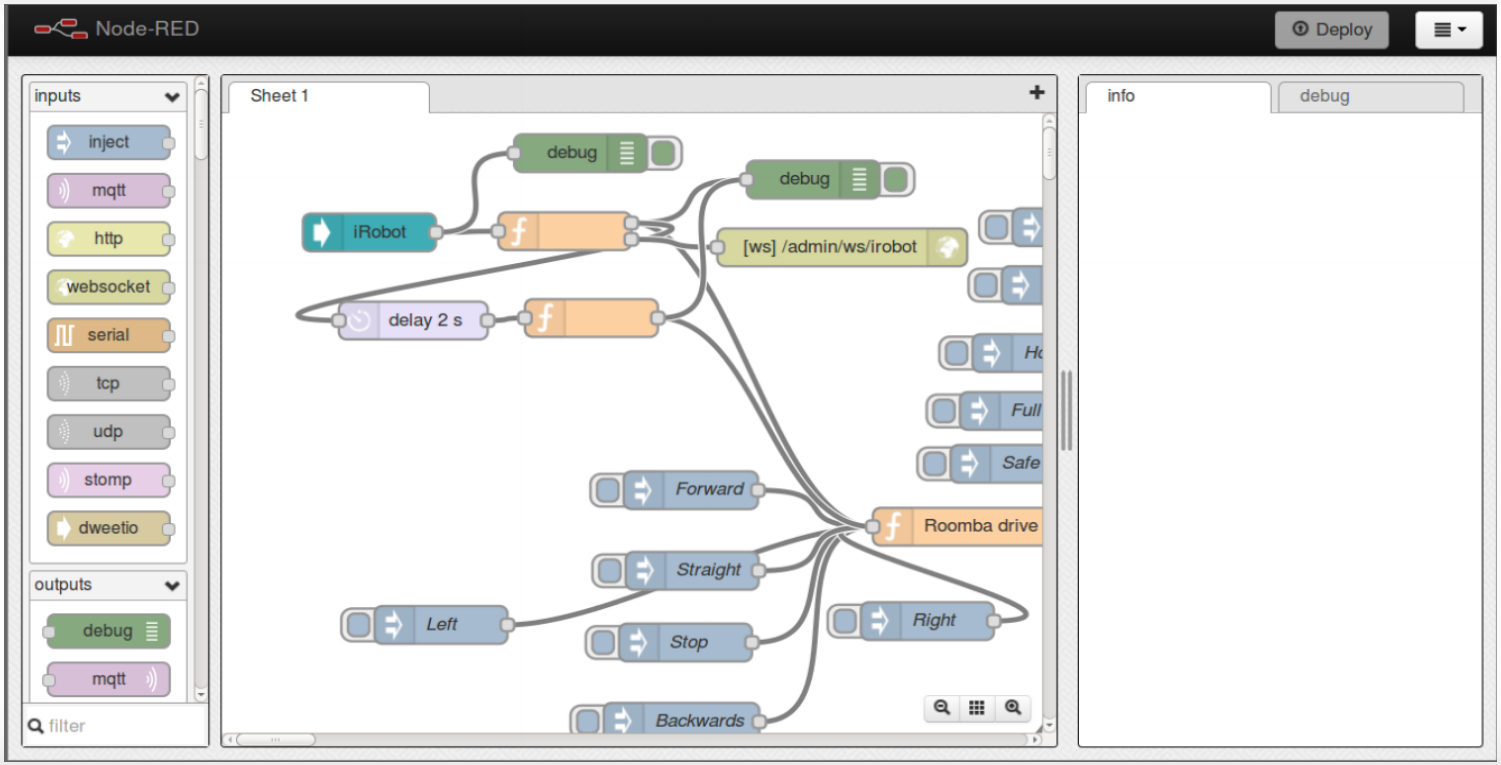
https://techtutorialsx.com/2017/04/24/esp32-subscribing-to-mqtt-topic/

Visual Tool for IoT Wiring

Node Red

The Need for Node-RED

We need tools that make it easier for developers at all levels to bring together the different streams of events, both physical and digital, that make up the Internet of Things.



What is Node-RED

Node-RED is

an application composition tool.

a lightweight proof of concept runtime.

easy to use for simple tasks.

simple to extend to add new capabilities and types of integration.

capable of creating the back-end glue between social applications.

a great way to try ...

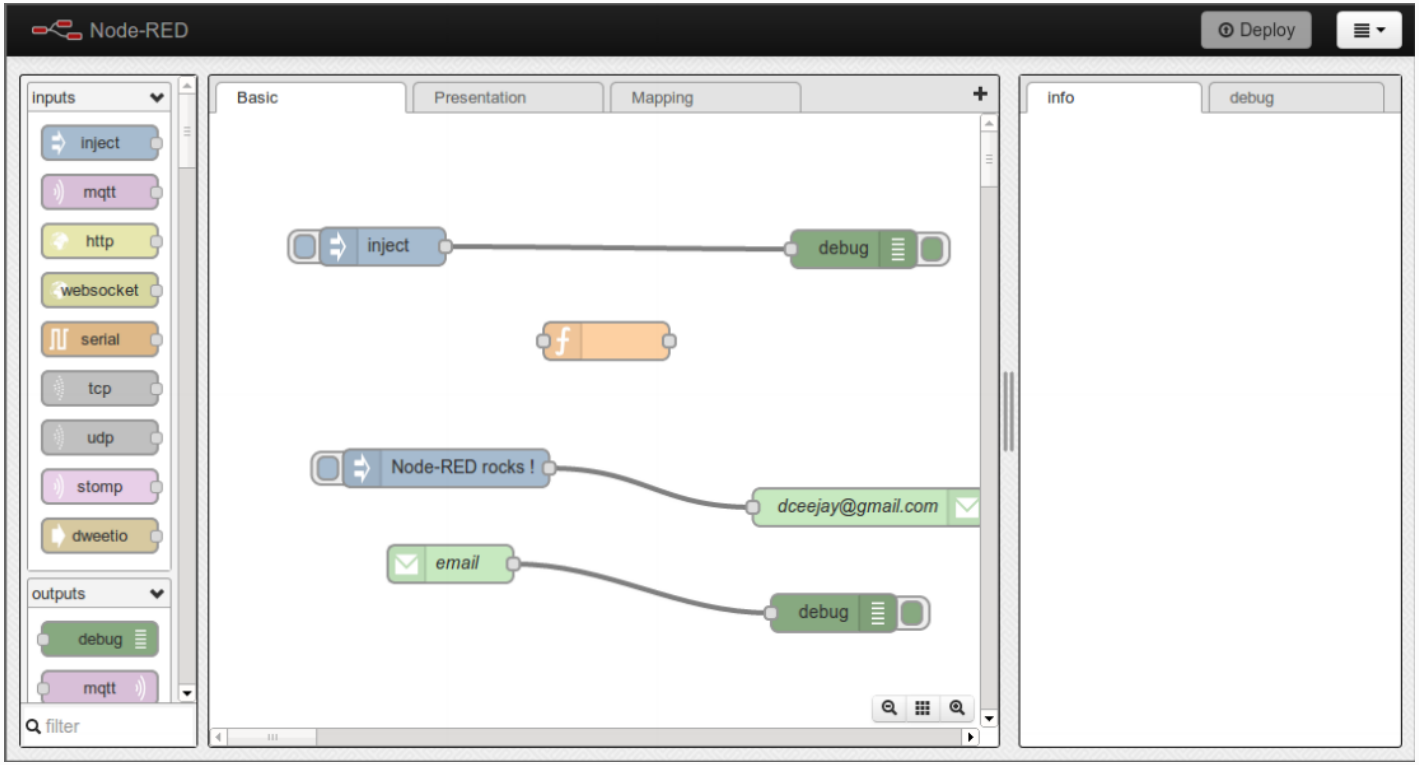
"can I just get this data from here to here ?"

"...and change it slightly on the way"

http://noderedguide.com/

Core Nodes

- Inject
- Debug
- Function
- Change
- Switch
- Template
- TCP/UDP
- Logic
- HTTP
- MQTT
- Forms
- Dashboard



Core Nodes

The Inject node can be used to manual trigger a flow by clicking the node's button within the editor. It can also be used to automatically trigger flows at regular intervals.

The message sent by the Inject node can have its `payload` and `topic` properties set.

The `payload` can be set to a variety of different types:

- a flow or global context property value
- a String, number, boolean, Buffer or Object
- a Timestamp in milliseconds since January 1st, 1970

Core Nodes

The Debug node can be used to display messages in the Debug sidebar within the editor.

The sidebar provides a structured view of the messages it is sent, making it easier to explore the message.

Alongside each message, the debug sidebar includes information about the time the message was received and which Debug node sent it. Clicking on the source node id will reveal that node within the workspace.

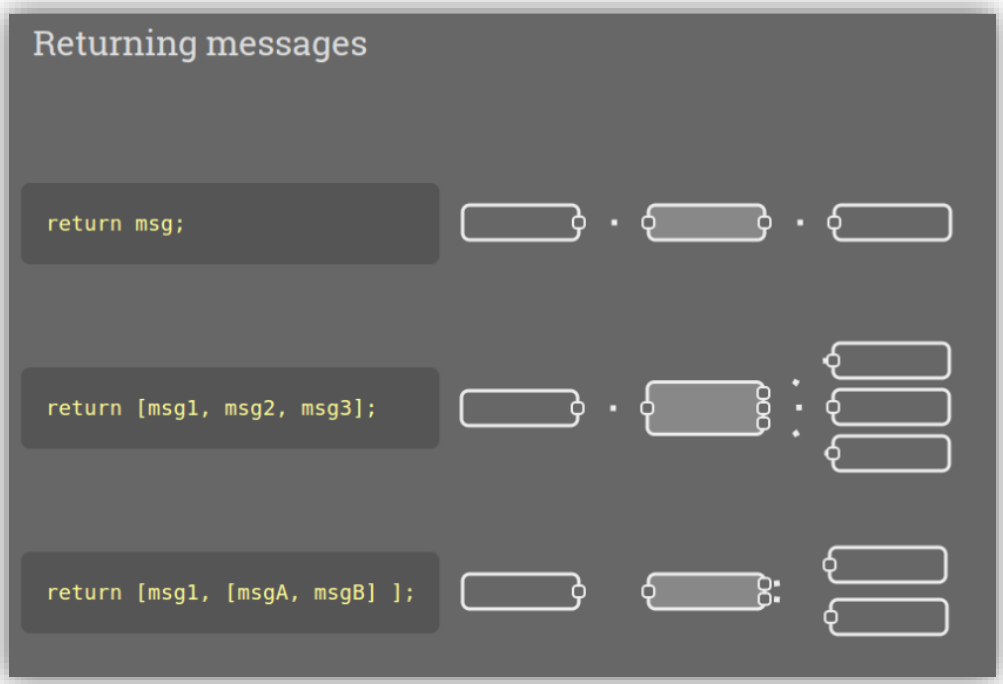
The button on the node can be used to enable or disable its output. It is recommended to disable or remove any Debug nodes that are not being used.

The node can also be configured to send all messages to the runtime log, or to send short (32 characters) to the status text under the debug node.

Core Nodes

The Function node allows JavaScript code to be run against the messages that are passed through it.

A complete guide for using the Function node is available here.



Core Nodes

The Change node can be used to modify a message's properties and set context properties without having to resort to a Function node.

Each node can be configured with multiple operations that are applied in order. The available operations are:

- **Set** - set a property. The value can be a variety of different types, or can be taken from an existing message or context property.
- **Change** - search and replace parts of a message property.
- **Move** - move or rename a property.
- **Delete** - delete a property.

Core Nodes

The Switch node allows messages to be routed to different branches of a flow by evaluating a set of rules against each message.

The node is configured with the property to test - which can be either a message property or a context property.

There are four types of rule:

- **Value** rules are evaluated against the configured property
- **Sequence** rules can be used on message sequences, such as those generated by the Split node
- A JSONata **Expression** can be provided that will be evaluated against the whole message and will match if the expression returns a `true` value.
- An **Otherwise** rule can be used to match if none of the preceding rules have matched.

The node will route a message to all outputs corresponding to matching rules. But it can also be configured to stop evaluating rules when it finds one that matches.

Core Nodes

The Template node can be used to generate text using a message's properties to fill out a template.
It uses the Mustache templating language to generate the result.
For example, a template of:

```
This is the payload: {{payload}} !
```

Will replace `{{payload}}` with the value of the message's `payload` property.

By default, Mustache will replace certain characters with their HTML escape codes. To stop that happening, you can use triple braces: `{{{payload}}}`.

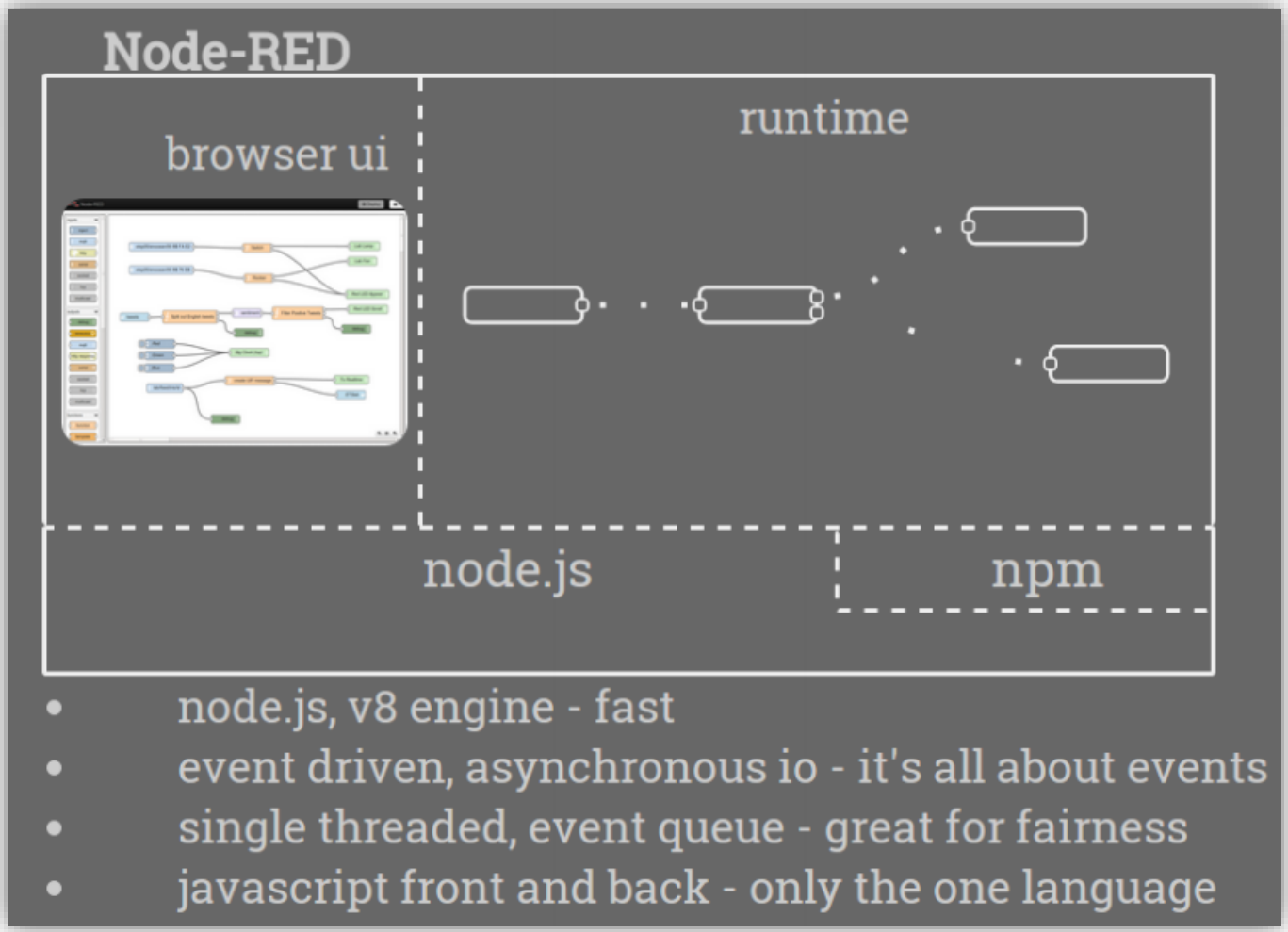
Mustache supports simple loops on lists. For example, if `msg.payload` contains an array of names, such as: `["Nick", "Dave", "Claire"]`, the following template will create an HTML list of the names:

```
<ul>
{{#payload}}
  <li>{{.}}</li>
{{/payload}}
</ul>
```

```
<ul>
  <li>Nick</li>
  <li>Dave</li>
  <li>Claire</li>
</ul>
```

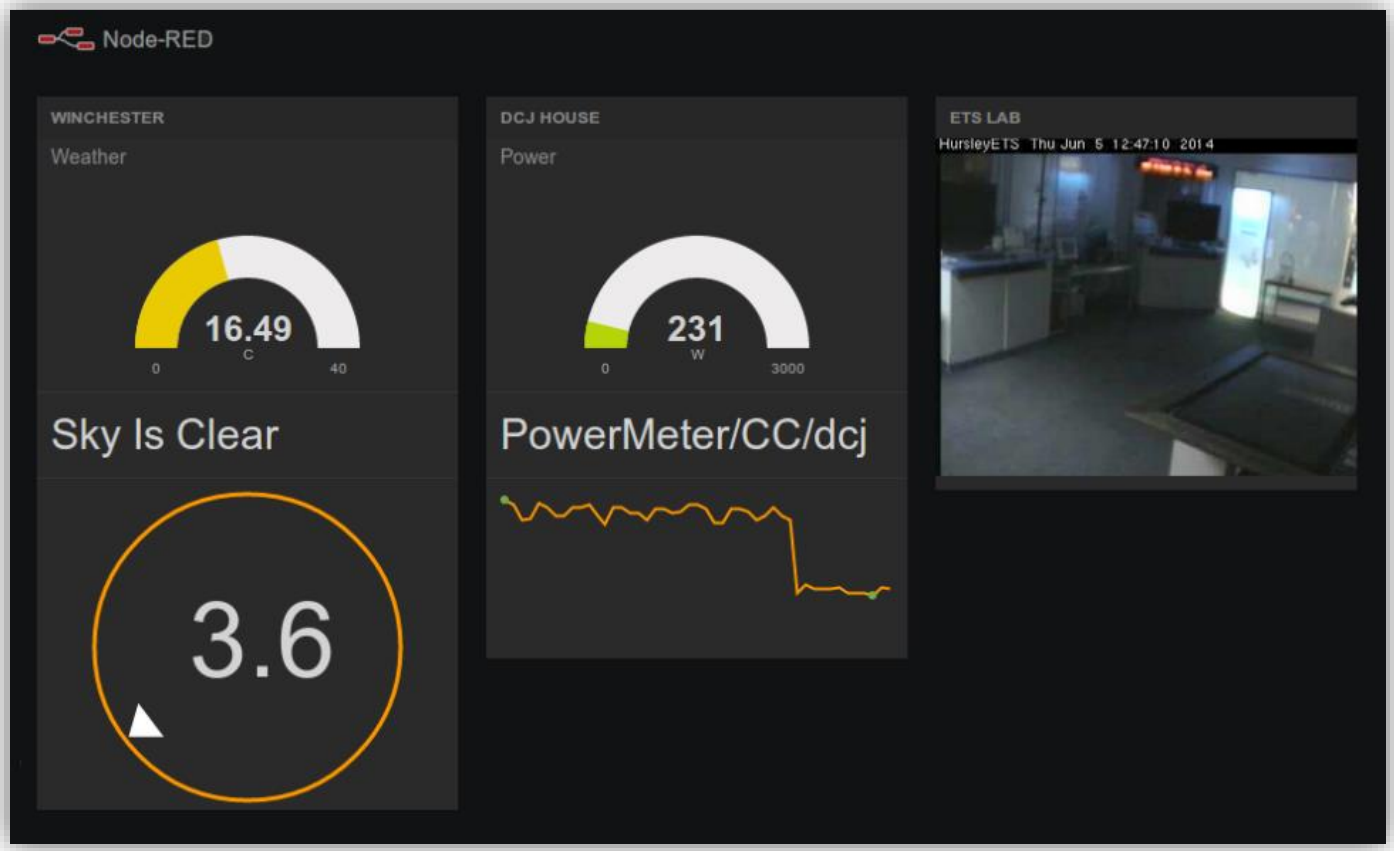
The node will set the configured message or context property with the result of the template. If the template generates valid JSON or YAML content, it can be configured to parse the result to the corresponding JavaScript Object.

Environment modules



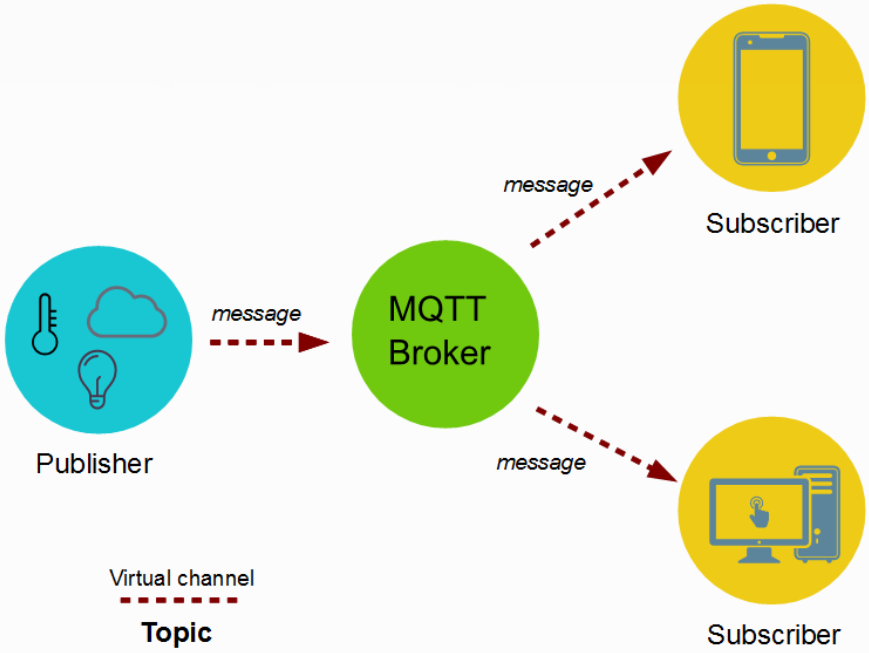
Demo

Node-RED Docs



https://cookbook.nodered.org/http/

MQTT with Node-RED



https://www.npmjs.com/package/node-red-contrib-mqtt-broker

Integrating Cloud & IoT

Edge Computing

INTEGRATIVE
Web Systems

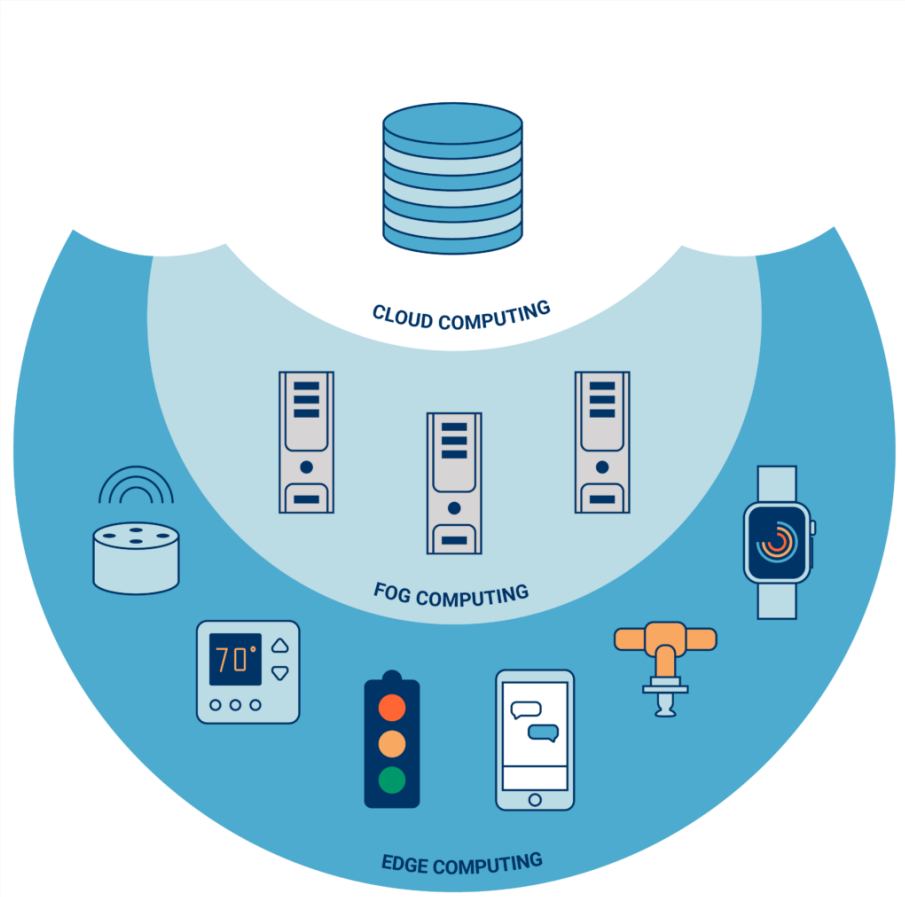
What Is IoT?

What is Edge Computing ?

- **Moving Services from cloud to near-site platforms**
  - Decrease latency - run machine learning algorithms near to IoT devices
  - Elevate robustness – less dependence on connectivity or limited bandwidth
  - Preserve privacy – keep sensitive data local or preprocess on site before sending
  - Sometimes referred to as Fog Computing

What is Edge Computing ?

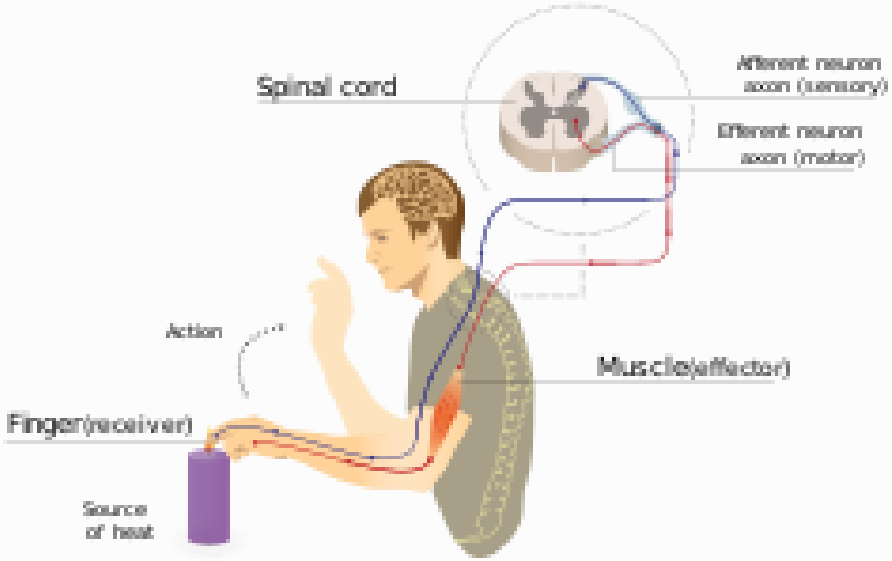
- **Modules include:**
  - MCU, Sensors & Actuators
  - Edge Devices
    - OS
    - Constrained power supply (Battery)
    - Run Logic on sensor Data
    - RT Processing
    - Basic Analytics
    - Buffering/Caching
    - Send commands to actuators
    - Talk to cloud (directly or via Gateway)
  - Edge Gateway
    - OS
    - More storage/Memory/CPU
    - Unconstrained power supply
    - Mediate Edge Devices & cloud
    - Site management services



What is Edge Computing ?

- **Edge Devices & Gateways forward data to cloud services**
  - Raw or pre-processed, Subsets or All

- **Cloud Activities**
  - Storage Services (Big Data)
  - Analytics Services
  - Supply ML Trained Models
  - Commands
  - Request Data/Queries
  - Configure/Manage Devices



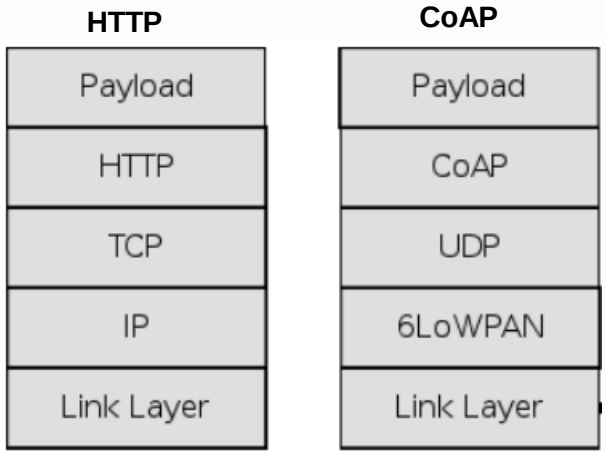
- **The Reflex Arc-Cortex analogy**

Additional Protocols

CoAP & LWM2M

CoAP

- Constrained Application Protocol (CoAP) is a RESTful application layer protocol specifically designed for constrained devices and constrained networks.
- CoAP uses UDP or SMS (Short Message Service) bindings as transport protocol to avoid the overhead created by connection oriented protocols such as TCP.
- CoAP utilizes a request/response based architecture between a CoAP client and a CoAP server.
- CoAP introduces an asynchronous publish/subscribe mechanism to enable serverinitiated communication, which is called “Observe/Notify”



Protocol Stacks Comparison

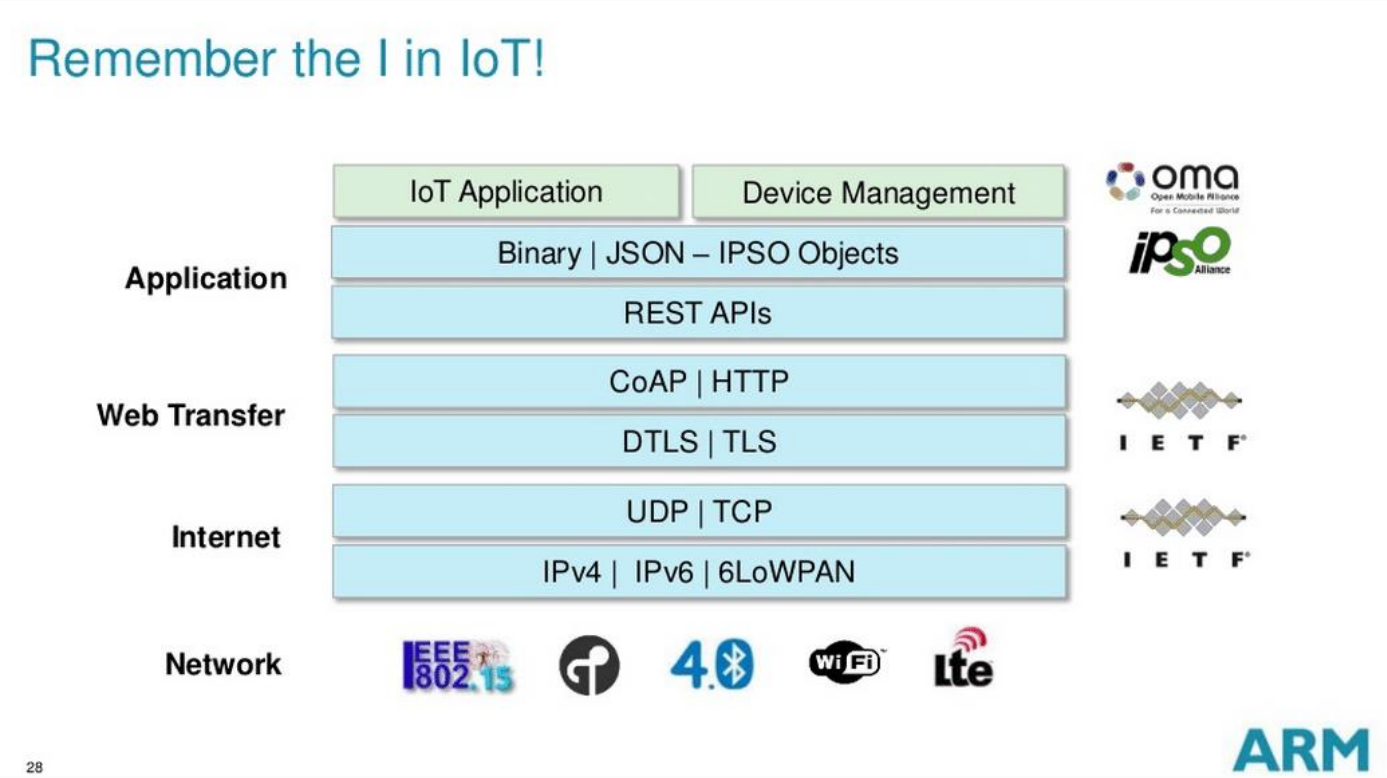
LWM2M

- Lightweight Machine 2 Machine protocol (LWM2M) is a light and compact device management protocol used for managing IoT devices and their resources.
- LWM2M is compatible with any constrained device, which runs CoAP as the transport protocol.
- LWM2M supports all basic device management functionalities, which include but not limited to access control, firmware update, connectivity, location, device meta data etc.

Device Management

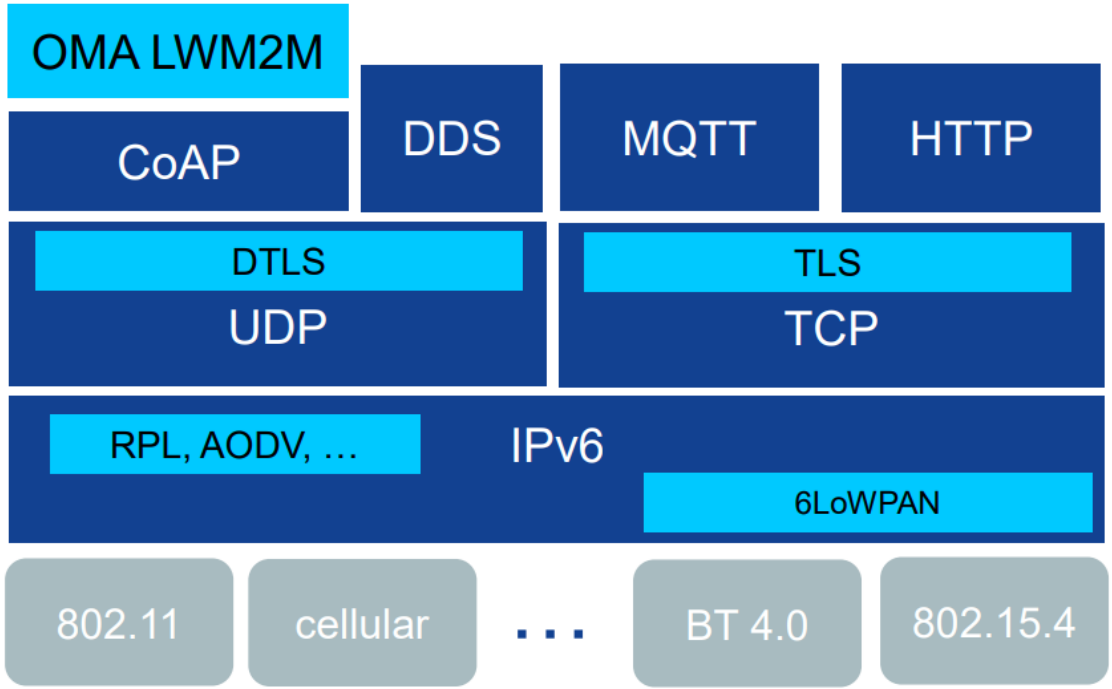
https://www.postscapes.com/internet-of-things-protocols/#graphics

LWM2M



Protocol Stack

IoT protocol stack



Device-to-Server:

LWM2M, MQTT, CoAP, HTTP

Device-to-People:

XMPP

Device-to-Device:

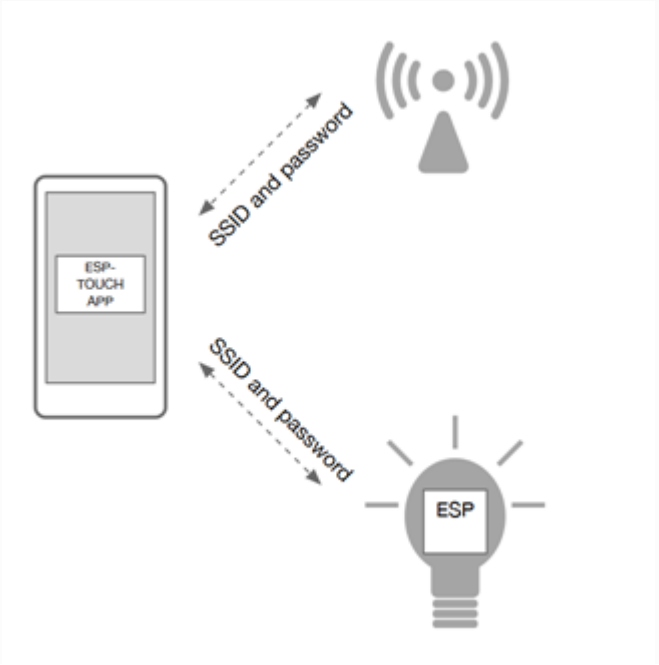
DDS

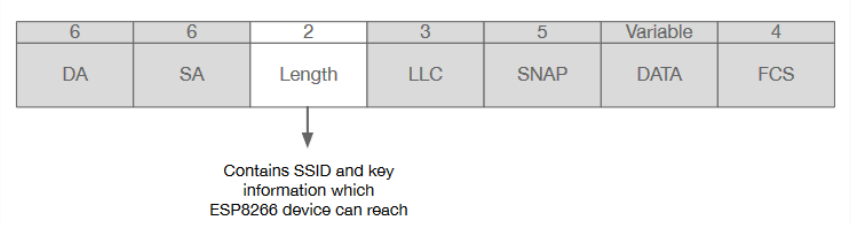
Server-to-Server:

AMQP

ESP 32 - Provisioning

```
#include "WiFi.h"
void setup() {
  Serial.begin(115200);
  /* Set ESP32 to WiFi Station mode */
  WiFi.mode(WIFI_AP_STA);
  /* start SmartConfig */
  WiFi.beginSmartConfig();

  /* Wait for SmartConfig packet from mobile */
  Serial.println("Waiting for SmartConfig.");
  while (!WiFi.smartConfigDone()) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("SmartConfig done.");

  /* Wait for WiFi to connect to AP */
  Serial.println("Waiting for WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("WiFi Connected.");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());
}
void loop() {
}
```





http://www.iotsharing.com/2017/05/how-to-use-smartconfig-on-esp32.html

Architecture

https://aaltodoc.aalto.fi/bitstream/handle/123456789/14829/master_Ocak_Mert_2014.pdf?sequence