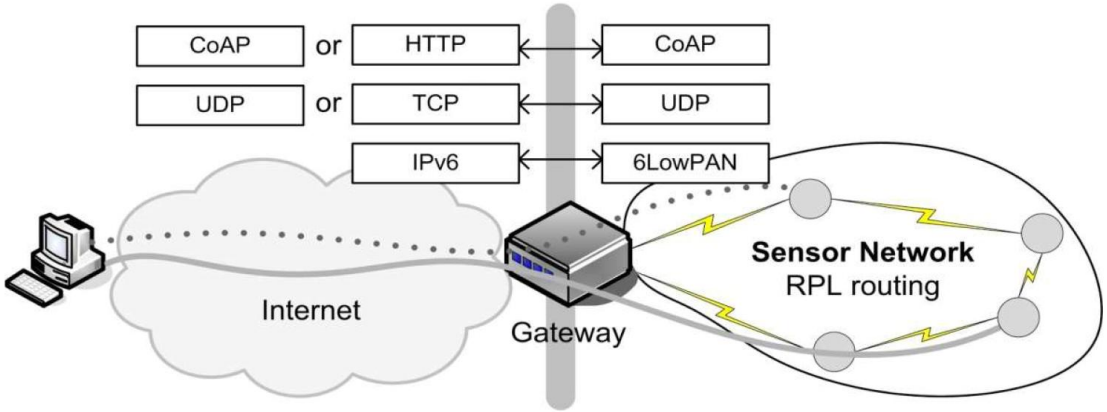
Introduction to Embedded Systems

Unit 2.1: Introduction to Messaging Protocols for IoT

Alexander Yemane
BCIT
INCS 3610
Fall 2025

Use of internet infrastructure in networked embedded systems

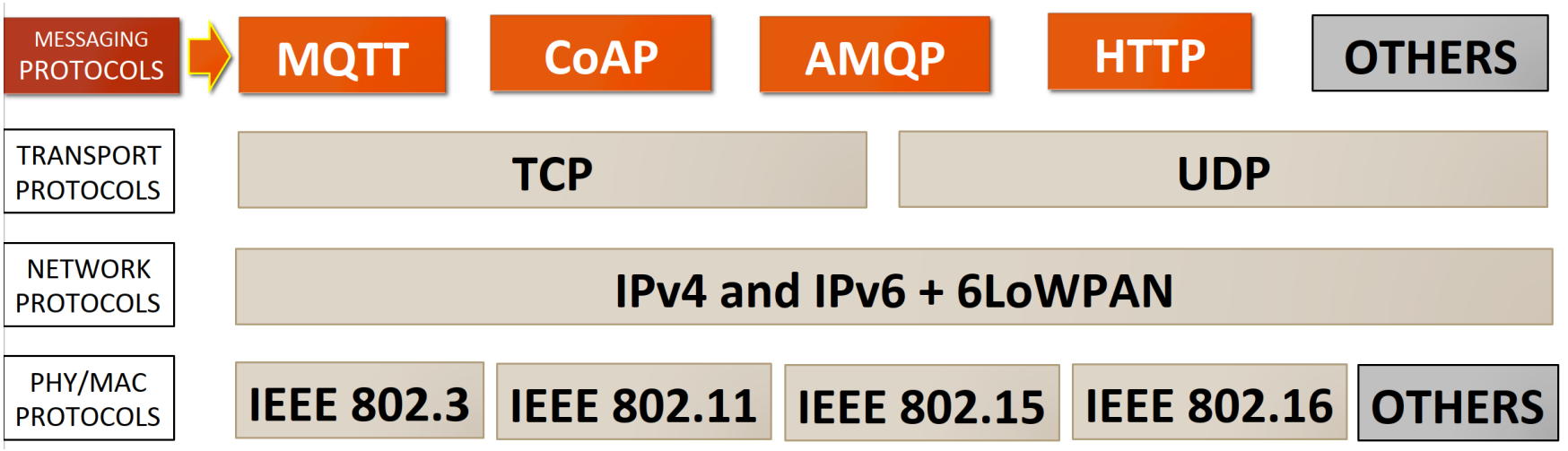
- The general idea behind the (commonly accepted) vision of IoT consists in the extension of Internet protocols to Wireless Sensors Networks (WSNs), composed of sensors, and actuators, and other smart devices



Use of internet infrastructure in networked embedded systems

- Key advantages of TCP/IP networking stack:
  - Open and standard-based
  - Versatile
  - Ubiquitous
  - Scalable
  - Manageable
  - Highly secure
  - Stable and resilient
- The future is web-based:
  - Cloud computing
  - Big data
  - Remote intelligence

IoT protocol stack



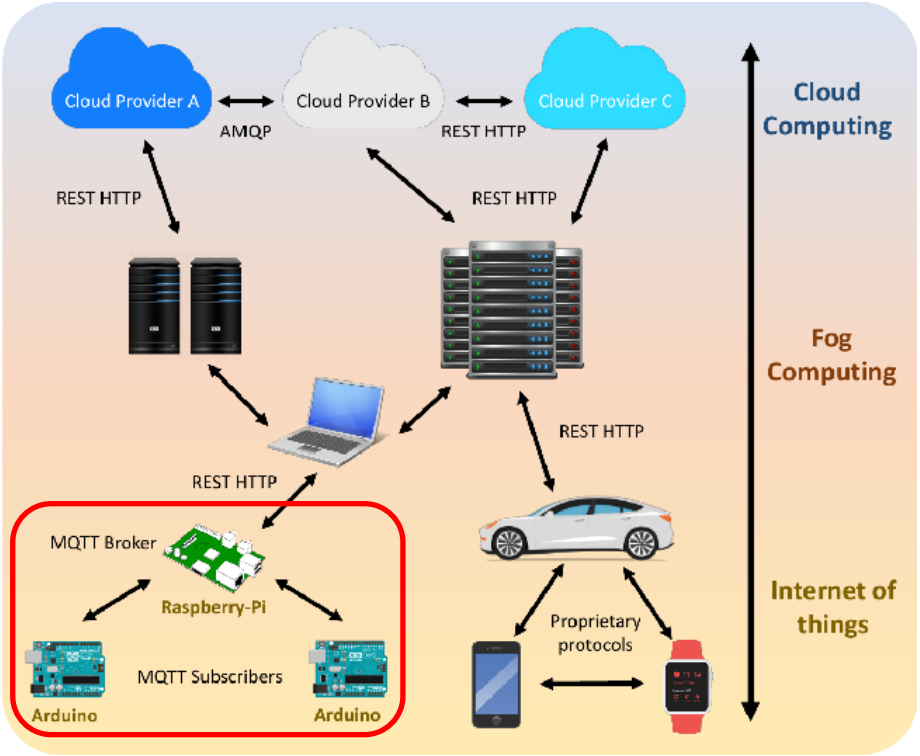
IoT messaging protocols

- **Exist in the session/application layer**
- Provide the abstraction of “message” (elementary unit of data communication among IoT nodes)
- Provide primitives for data communication/message exchange to the upper layer IoT applications (e.g. (dis)connecting, sending, receiving)
- Implement specific networking paradigms (e.g. publish-subscribe or request-response).
- Provide additional reliability or security mechanisms.
- Sometimes adapt pre-existing (not natively M2M) solutions

IoT messaging protocols

- Several protocols thriving to become a de facto standard
- It could be difficult to pick up the best solution, since it depends on requirements of a given application domain
- Some request/response protocols
  - REST architectural principles (REpresentational State Transfer), 2000
  - CoAP (Constrained Application Protocol), 2014 [also pub/sub]
- Some publish/subscribe protocols:
  - MQTT (Message Queue Telemetry Transport), 1999
  - AMQP (Advanced Message Queuing Protocol), 2003
  - DDS (Data Distribution Service), v1.0 2003; v1.2 2007…

IoT messaging protocols in different architectural layers



The MQTT protocol

- MQTT is a lightweight messaging protocol designed for machine-to-machine (M2M) telemetry on top of the TCP/IP protocol
  - Telemetry = Tele-Metering = Remote measurements
- Ideal for use in constrained nodes and networks
  - on embedded devices with limited processor or memory resources
  - where the network is expensive, has low bandwidth, or is unreliable
- **MQTT-SN** for wireless sensor networks, aimed at embedded devices on non-TCP/IP networks

The MQTT protocol

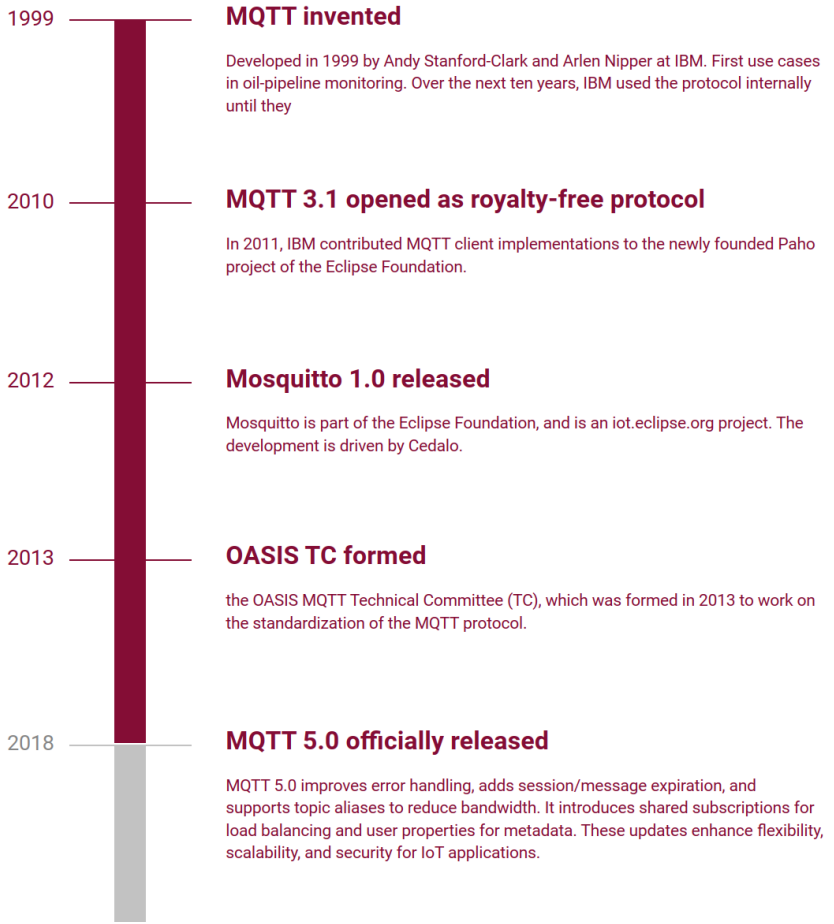
- Pop quiz – “MQTT” stands for:
    1. **M**essage **Q**ueuing **T**elemetry **T**ransport
    2. **MQ** Telemetry Transport
    3. MQTT

What does “MQTT” stand for?

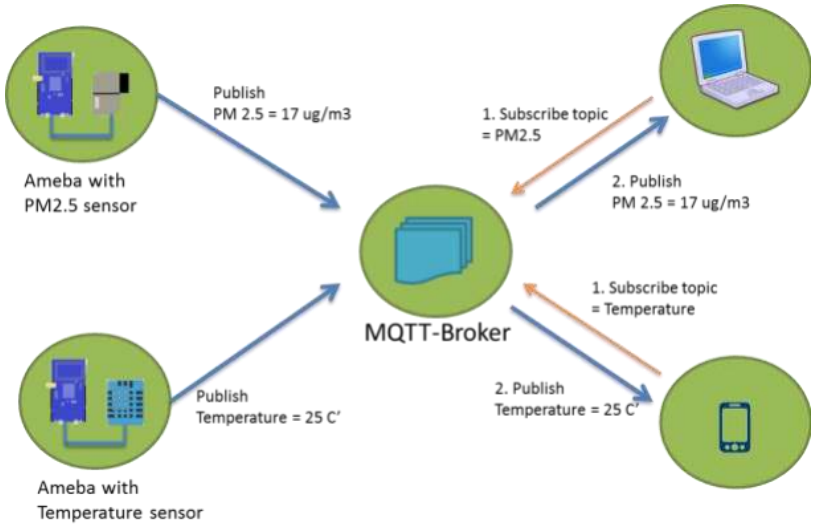
- While it formerly stood for MQ Telemetry Transport, where MQ referred to the MQ Series, a product IBM developed to support MQ telemetry transport, MQTT is no longer an acronym.
- MQTT is now simply the name of the protocol.
- Although many sources label MQTT as a Message Queue Telemetry Transport protocol, this is not entirely accurate. While it is possible to queue messages in certain cases, MQTT is not a traditional message queuing solution

History and evolution of MQTT

- Originally designed for SCADA systems
- Becoming an open standard helped to increase the visibility and adoption of MQTT among the developer community.
- Popular (and open source) implementations are **Paho** (client in Python) and **Mosquitto** (broker and client in C).
- Mosquitto supports MQTT protocol versions 5.0, 3.1.1, and 3.1



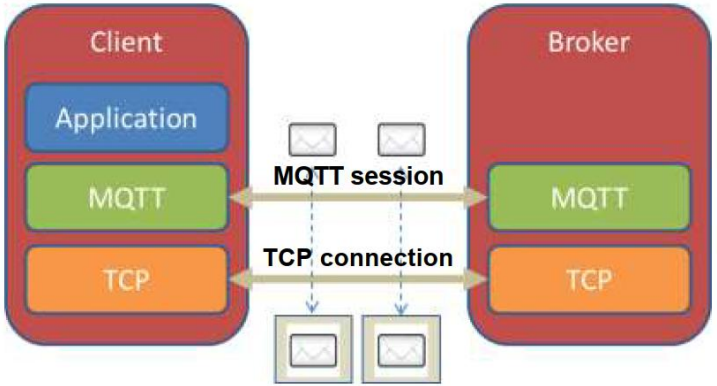
MQTT typical use case



1. Subscribe to one or more topics
2. Publish to a topic
3. Receive messages related to subscribed topics

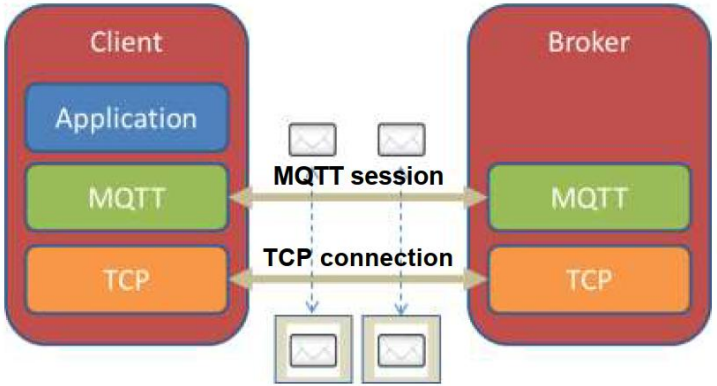
MQTT features

- **Publish/subscribe** message pattern to provide
  - one-to-many message distribution
  - decoupling of applications
- **Push-based**: clients autonomously decide to send messages to servers
- **Small transport overhead** (minimal header length just 2 bytes) and protocol exchanges minimized to reduce network traffic and power consumption
- **Easy to use** with few commands: connect, subscribe, publish, disconnect
- **Retained messages**: an MQTT broker can retain a message that can be sent to newly subscribing clients



MQTT features

- **Three message delivery semantics** with increasing reliability and cost:
  - Quality of Service (QoS): at least once, at most once, exactly once
- **Durable connection**: client subscriptions remain in effect even in case of disconnection
  - subsequent messages with high QoS are stored for delivery after connection reestablishment
- **Wills**: a client can setup a will, a message to be published in case of unexpected disconnection

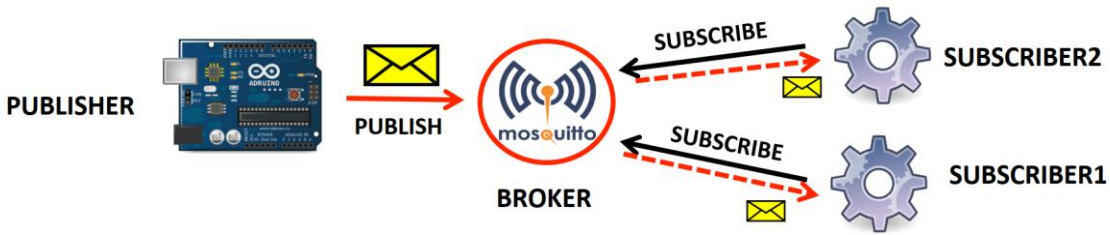


Projects that implement MQTT

- **Amazon Web Services** announced Amazon IoT based on MQTT in 2015
- **Microsoft Azure IoT Hub** uses MQTT as its main protocol for telemetry messages
- **Node-RED** supports MQTT nodes as of version 0.14
- **Facebook** has used aspects of MQTT in Facebook Messenger for online chat
- The **EVRYTHNG IoT platform** uses MQTT as an M2M protocol for millions of connected products.

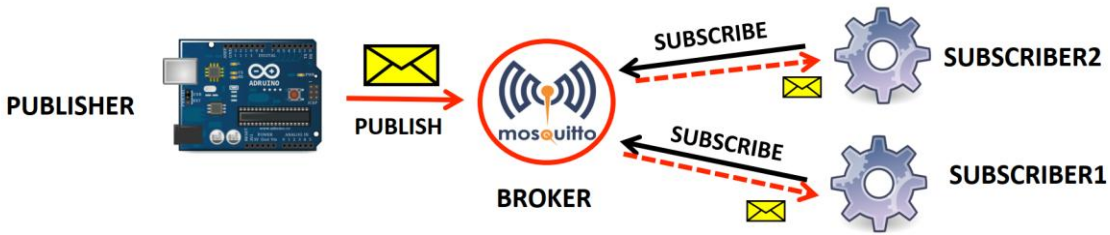
Publish-subscribe messaging

- MQTT system architecture involves three main actors:
  - A publisher sends (publishes) a message on a topic
  - A subscriber subscribes to a topic of interest, and receives notifications when a new message for the topic is available
  - A broker filters data based on topic and distributes them to subscribers
    - If no matches, the message is discarded
    - If one or more matches, the message is delivered to each matching subscriber

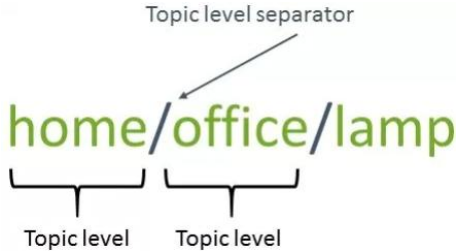


Publish-subscribe messaging

- Architectural remarks:
  - No direct communication between clients (the data messages are always forwarded via the broker)
  - Roles are purely logical: the same device can serve as publisher (on a topic) and subscriber (on a different topic).
    - Note: publishers and subscribers are both considered clients



Topic-based communication

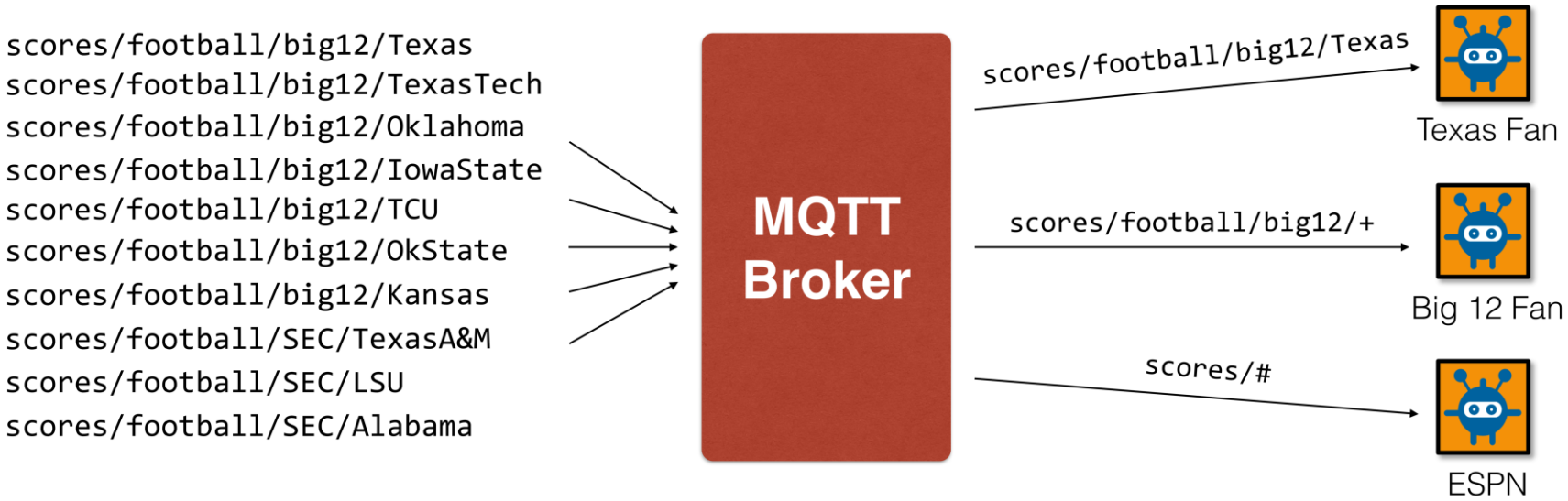


- Topics define the message context (e.g. temperature data)
- Namespace is hierarchical with each “sub-topic” separated by */*
- For example:
  - A house publishes information about itself on:
    - <country>/<region>/<town>/<postalcode>/<house no>/energyConsumption
    - <country>/<region>/<town>/<postalcode>/<house no>/solarEnergy
  - It subscribes for control commands:
    - <country>/<region>/<town>/<postalcode>/<house no>/thermostat/setTemp

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 there be a topic <sensors>/<variable>/<country>/<city>/<address>
    - Get data from all sensor types in London:
      - sensors/+/uk/london/baker_street
  - Multi-level wildcards “#” must appear at the end of the string
    - For example, get temperature data from all locations in UK:
      - sensors/temperature/uk/#
- Wildcards must be next to a separator
- Wildcards cannot be used when publishing

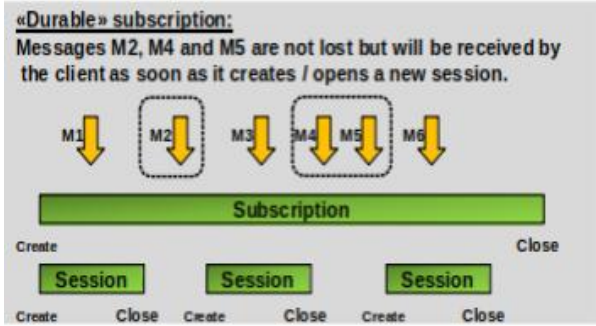
Multi-level subscriptions

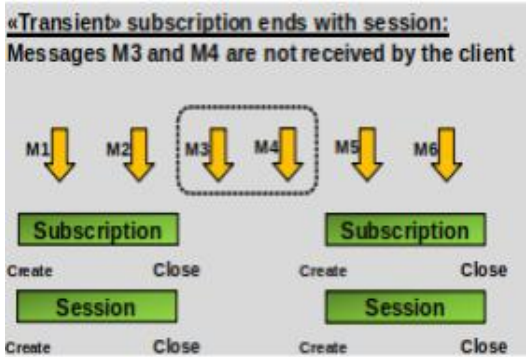


single level wildcard: **+** multi-level wildcard: **#**

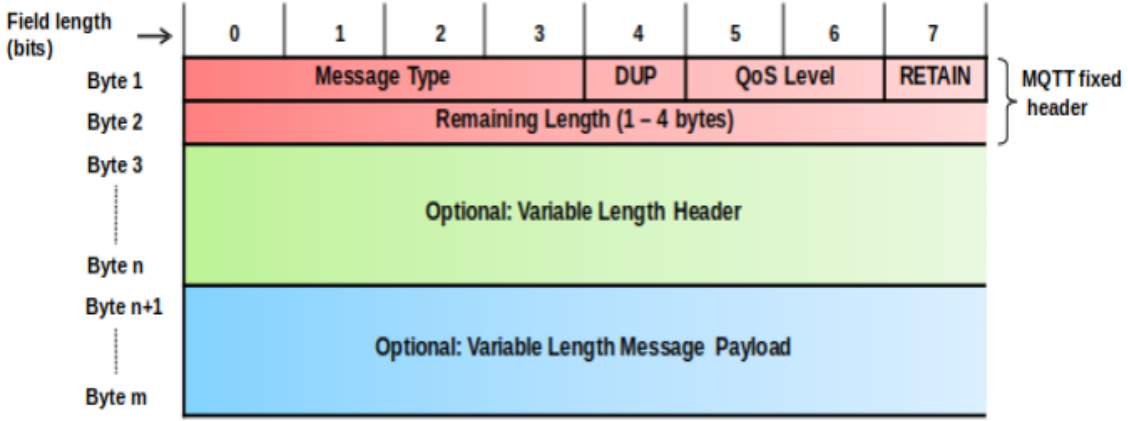
Types of subscriptions

- 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:
    - The subscription lifetime is the same as the time the subscriber is connected to the server / broker



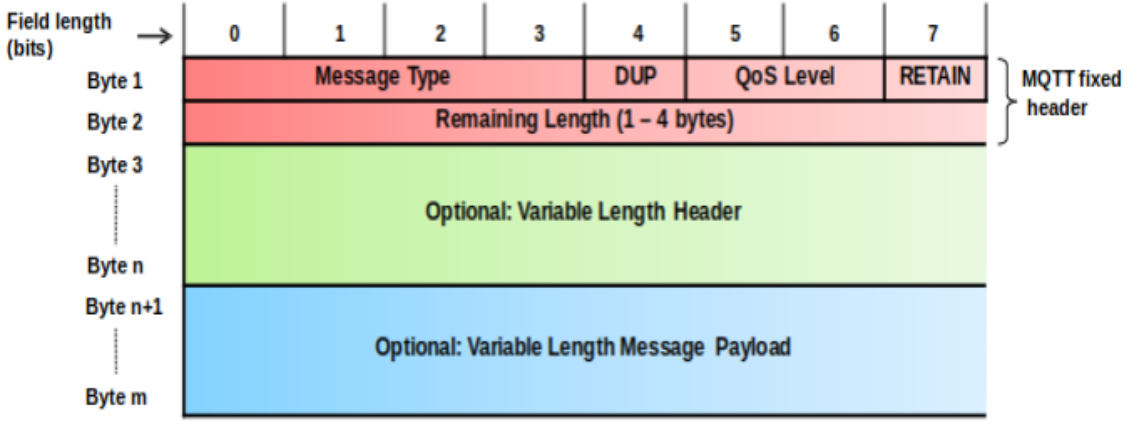


MQTT packet structure



- Fixed header:
  - 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: allows to select different QoS level
  - RETAIN: notifies the server to hold onto the last received PUBLISH message data
  - Remaining Length: specifies the size of optional fields

MQTT packet structure



- Fixed header is only 2 bytes
- Variable header contains the additional parameters based on the message type.
  - For instance, the header of the PUBLISH/SUBSCRIBE message contains the TOPIC field.

MQTT packet structure

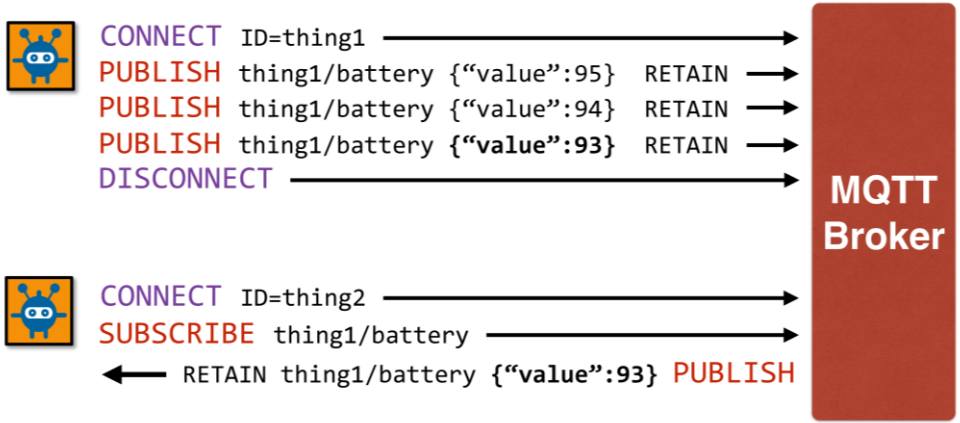
Message header field	Description / Values	
Message Type (4 bits) 14 message types, 2 are reserved	0: Reserved	8: SUBSCRIBE
	1: CONNECT	9: SUBACK
	2: CONNACK	10: UNSUBSCRIBE
	3: PUBLISH	11: UNSUBACK
	4: PUBACK (Publish ACK)	12: PINGREQ
	5: PUBREC (Publish Received)	13: PINGRESP
	6: PUBREL (Publish Release)	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.	

MQTT 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		

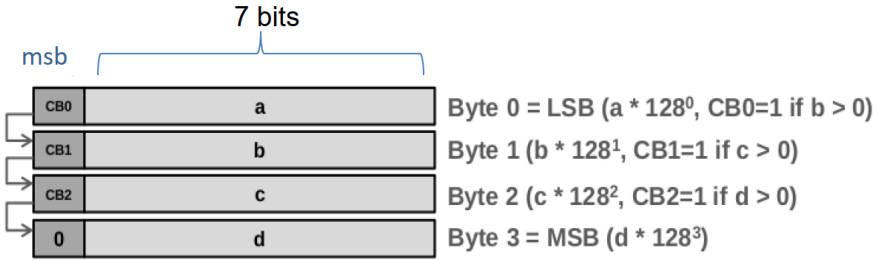
Retained messages

- 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.
  - Hence do not have to wait till next PUBLISH action
- For example:
  - Subscribers receive last known battery charge value from the battery data topic.
  - RETAIN=1 indicates to subscriber thing2 that the message may have been published some time ago.



Remaining Length (RL)

- The remaining length field encodes the sum of the lengths of:
  - (Optional) variable length header
  - (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: $a*128^0 + b*128^1 + c*128^2 + d*128^3$ and placed into the RL field bytes as depicted

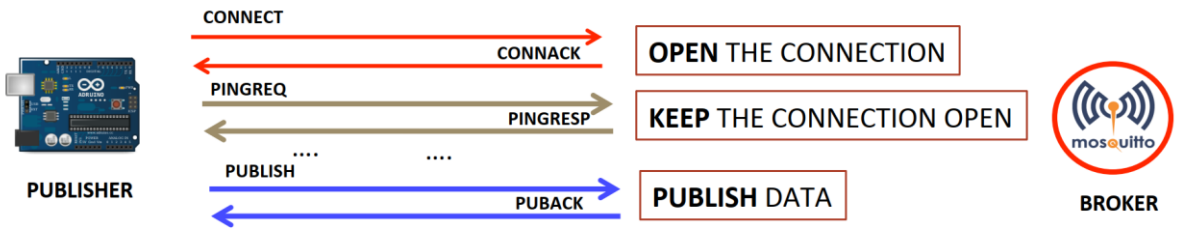


Example 1: **RL = 364 = $108*128^0+2*128^1$ → a=108, CB0=1, b=2, CB1=0, c=0, d=0, CB2=0**

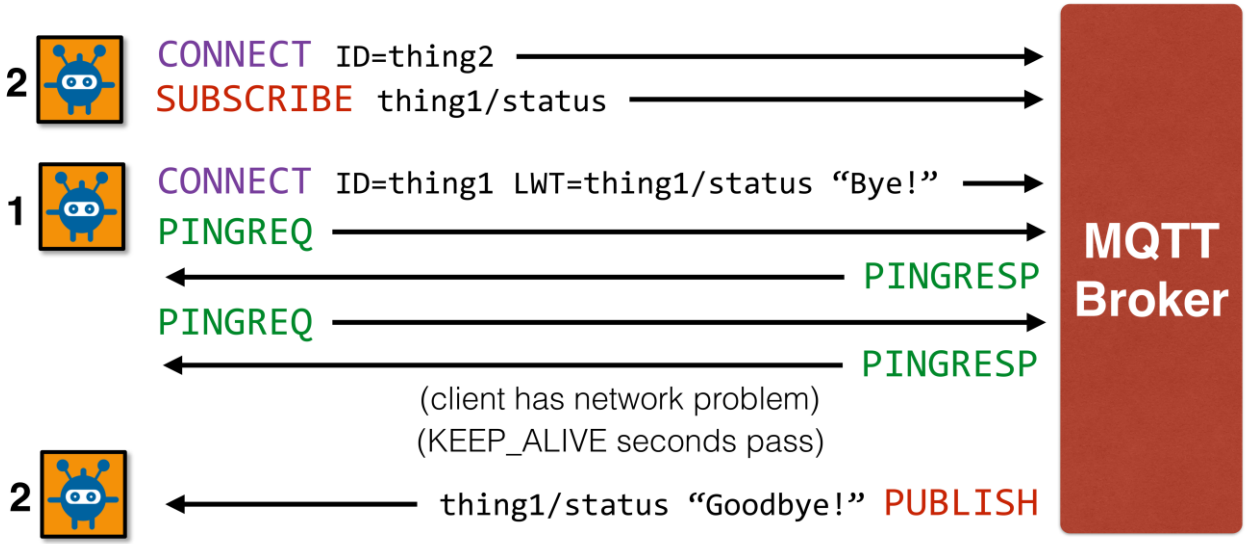
Example 2: **RL = 25'897 = $41*128^0 + 74*128^1 + 1*128^2$ → a=41, CB0=1, b=74, CB1=1, c=1, CB2=0, d=0**

Keep alive functionality

- MQTT keeps the TCP connection between a client and broker open as long as possible, by means of PINGREQ messages.
- The MQTT client is responsible of setting the right keep alive value.
- If the broker doesn't receive a PINGREQ or any other packet from a particular client, it will close the connection and send out a "last will and testament" message (if the client had specified one).
  - This message notifies other parts of the system that a node has gone down.

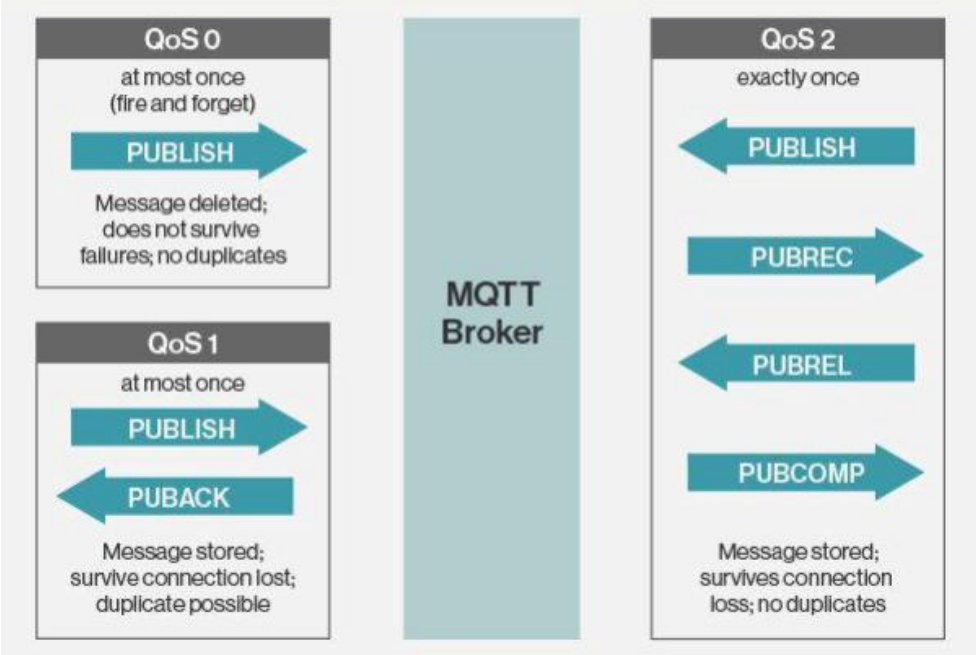


Last will and testament message



MQTT QoS

- Even though TCP provides guaranteed data delivery, data loss can still occur if a TCP connection breaks down and messages in transit are lost.
- Therefore, MQTT clients can request three level of Quality of Service (QoS) to the broker
  - QoS level 0 (default)
  - QoS level 1
  - QoS level 2

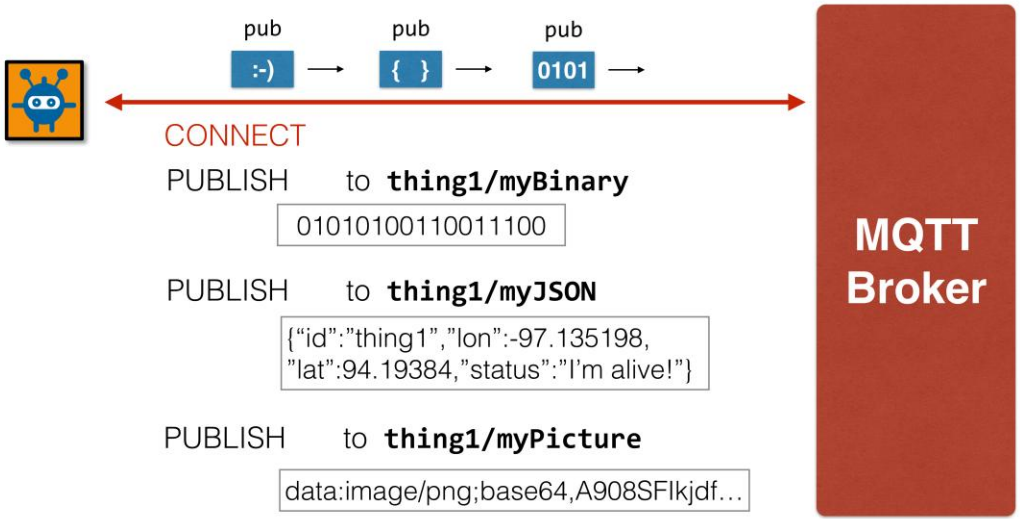


MQTT QoS

- QoS level 0 / “At most once”
  - 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”
  - 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”
  - This is the highest level that also incurs most overhead in terms of control messages and the need for locally storing the messages.
  - 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 duplicates.

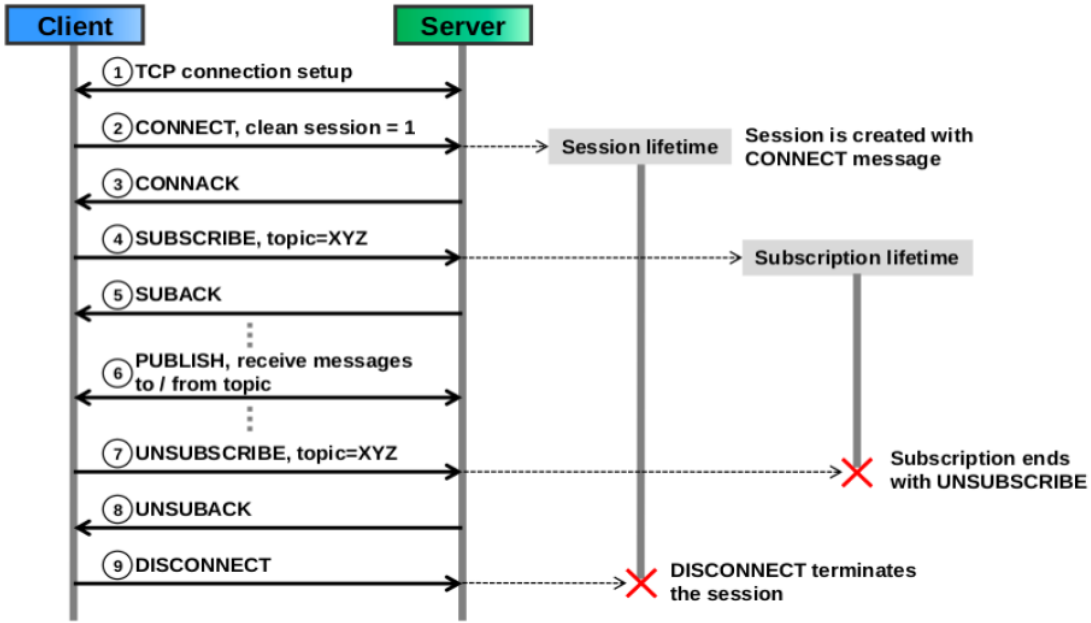
Agnostic payload

- MQTT is data-agnostic and it totally depends on the use case how the payload is structured.
- It's completely up to the sender if it wants to send binary data, textual data or even full-fledged XML or JSON.



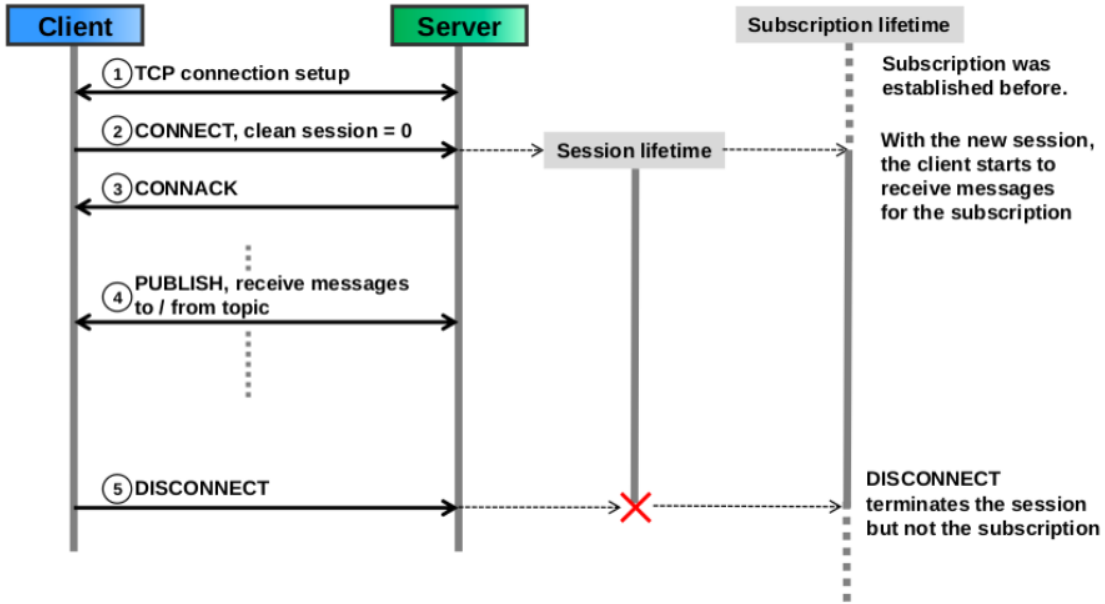
CONNECT & SUBSCRIBE message flow

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



CONNECT & SUBSCRIBE message flow

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



MQTT-SN: MQTT for sensor networks

- MQTT-SN is a variant of MQTT that has been optimized for use in low power environments such as sensor networks, as the name suggests.
- MQTT-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
    - Limited payload size
    - Uses UDP

MQTT security

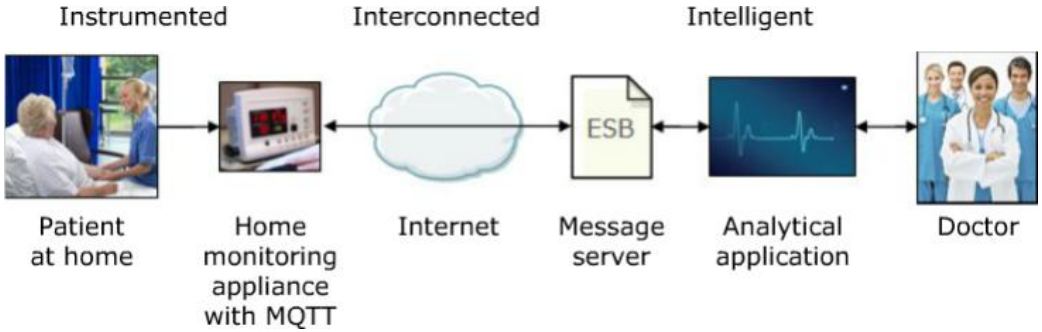
- MQTT provides security, but it is not enabled by default.
  - As a basic solution we can rely on the encrypted WiFi connection to provide a basic level of security.
- MQTT has the option for Transport Layer Security (TLS) encryption.
- MQTT also provides username/password authentication.
  - Note that the password is transmitted in clear text. Thus, be sure to use TLS encryption if you are using authentication.
- The broker can restrict access to specific topics based on client ID (aka topic ACL)

MQTT summary

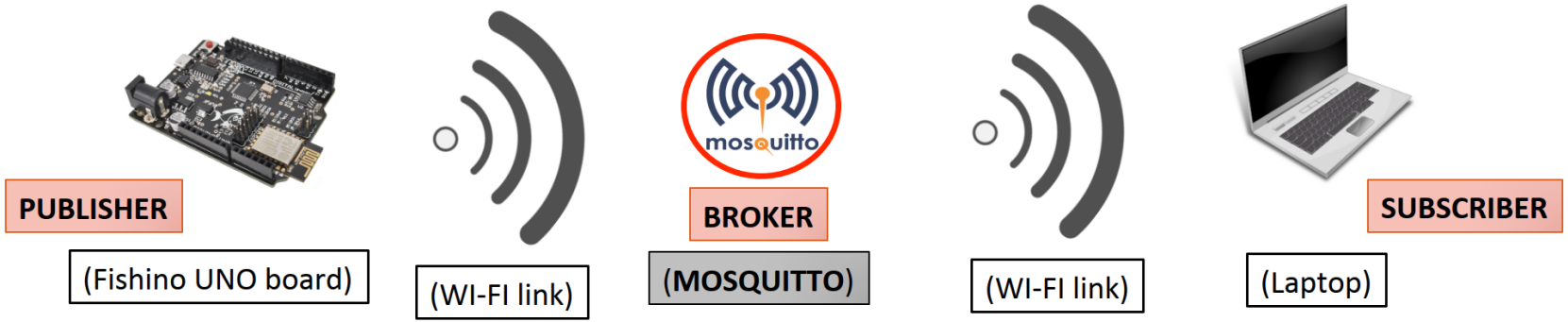
- Advantages
  - Push-based: no need to continuously look for updates
  - Useful for one-to-many, many-to-many applications
  - Small memory footprint protocol, with reduced use of battery
  - Built-in function useful for reliable behavior in an unreliable or intermittently connected wireless environments
  - Decoupling in multiple dimensions: space, time, synchronization
- Disadvantages
  - Does not define a standard client API
  - Does not have a point-to-point messaging pattern
  - Not much sense of history…
  - If the broker fails…

MQTT application example

- Home pacemaker monitoring solution
  - Sensors on patient
  - Collected by a monitoring equipment in home (broker) using MQTT
  - Subscribed by a computer in the hospital
  - Alerts the doctor if anything is out of order



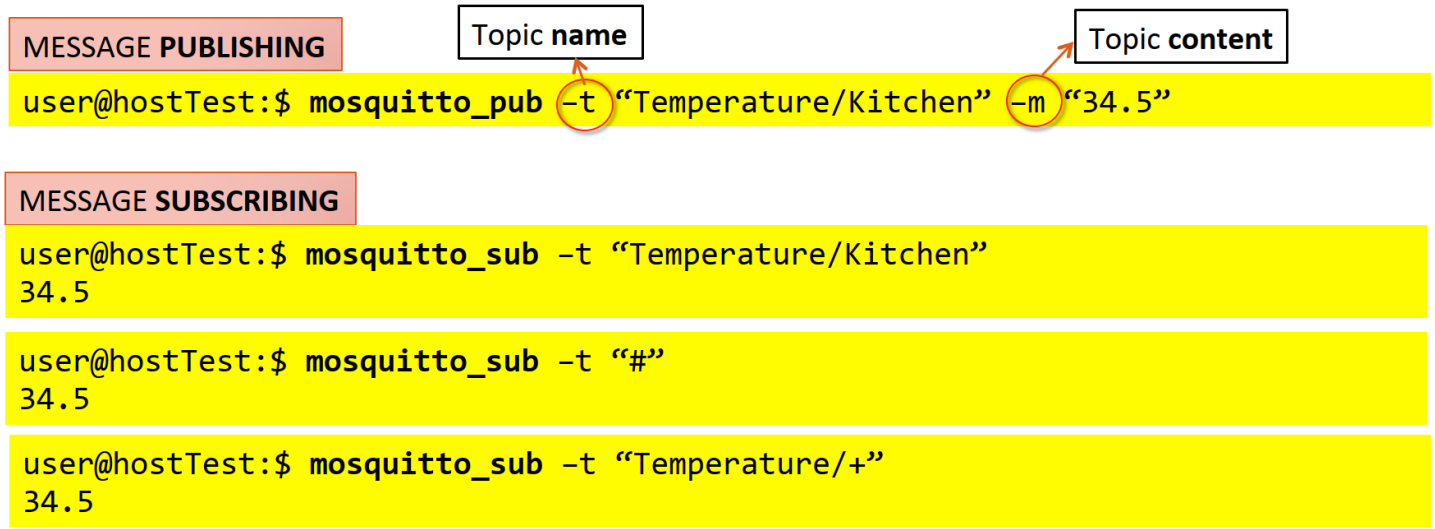
MQTT basic demo



- Mosquitto: Open source MQTT (1.3/1.3.1) broker implementation, multi-platform

MQTT basic demo

- MQTT client utilities: mosquitto_pub and mosquitto_sub



MQTT basic demo

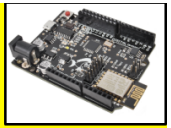
- Mosquitto configuration file: /etc/mosquitto/mosquitto.conf

```
#Enable/disable persistence, i.e. message savings on broker side
persistence true
persistence_location /var/lib/mosquitto/

#Enable/disable authentication
allow_anonymous false
password_files /etc/mosquitto/mosquitto_pwd

#Log broker activities
log_dest file /var/log/mosquitto/mosquitto.log

#Presence of duplicates (only for QoS 0 and 1)
allow_duplicate_messages false
```

MQTT basic demo

- MQTT at publisher side:

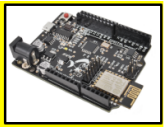
```
boolean publishData(char clientID, char* topic, char* payload) {
    boolean connected=clientMQTT.connected();
    if (!connected)
      connected=clientMQTT.connect(clientID);
    if (connected) {
        bool result=clientMQTT.publish(topic,payload);
        clientMQTT.loop();
        return result;
    } else
        Serial.println(F("MQTT Broker not available"));
    return(false);
}
```



MQTT basic demo

- MQTT at publisher side:

```
#include <PubSubClient.h>
PubSubClient clientMQTT;
void setup() {
    …
    clientMQTT.setClient(client);
    clientMQTT.setServer(”192.168.1.200",1883);
}

void loop() {
    …
    publishData(“MyClientID”,”MyTopic”,”MyMessage”);
}
```



MOSQUITTO Broker **IP Port**

MOSQUITTO Broker **IP Address**

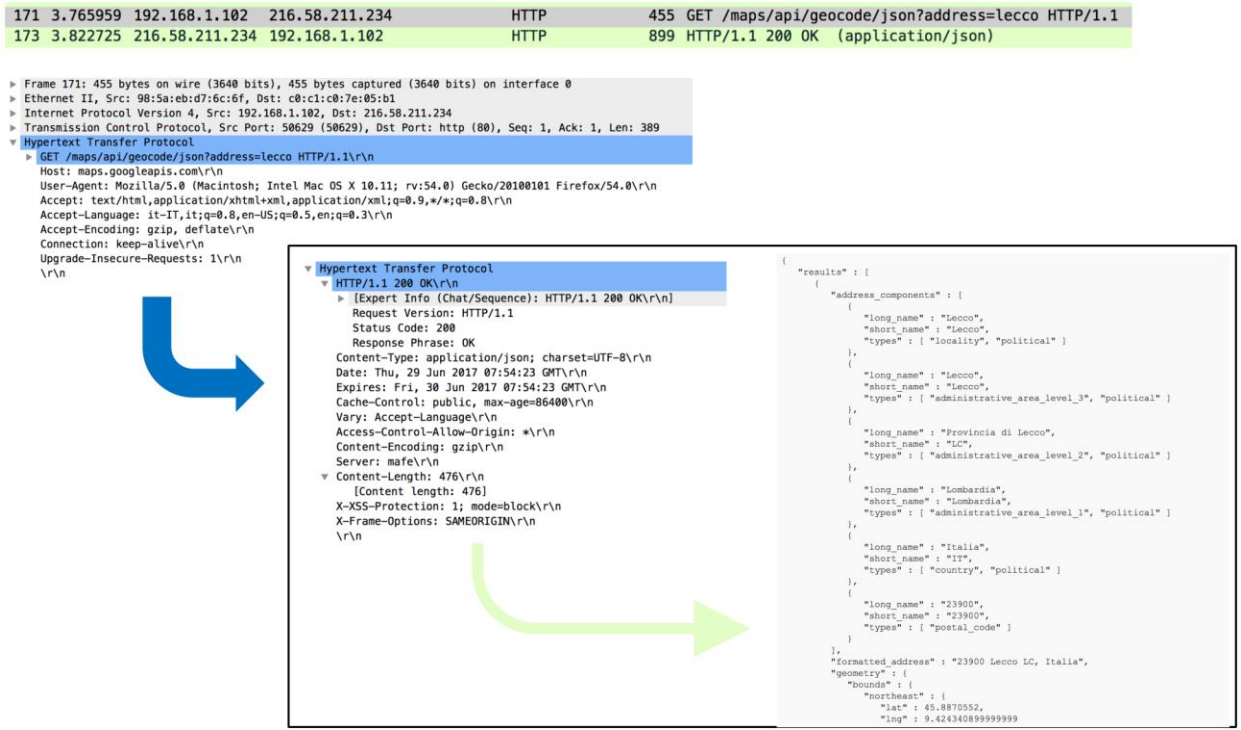
REST and HTTP

- REST stands for Representational State Transfer.
  - It basically leverages the HTTP protocol and its related frameworks to provide data services.
- The motivation for REST was to capture the characteristics of the Web which made the Web successful.
  - Pull-based: servers ask to gateways to send messages
    - Make a Request – Receive Response – Display Response
- REST is not a standard… but it uses several standards:
  - HTTP
  - URL
  - Resource Representations: XML/HTML/GIF/JPEG/etc
  - Resource Types, MIME Types: text/xml, text/html, image/gif, image/jpeg, etc

REST is widely used

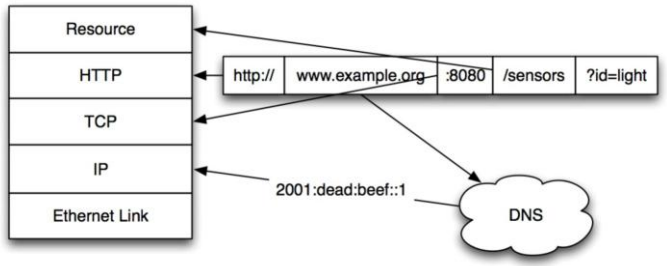
- Twitter:
  - https://dev.twitter.com/rest/public
- Facebook:
  - https://developers.facebook.com/docs/atlas-apis
- Amazon offers several REST services, e.g., for their S3 storage solution
  - http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
- The Google Glass API, known as "Mirror API", is a pure REST API.
  - Here is (https://youtu.be/JpWmGX55a40) a video talk about this API. (The actual API discussion starts after 16 minutes or so.)
- Tesla Model S uses an (undocumented) REST API between the car systems and its Android/iOS apps.
  - http://docs.timdorr.apiary.io/#reference/vehicles/state-and-settings

Wireshark view of HTTP request to Google Maps API



REST: the resources

- The key abstraction of information in REST is a resource.
- A resource is a conceptual mapping to a set of entities
  - Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person or a device), and so on
- Represented with a global identifier (URI in HTTP).
- For example:
  - http://www.acme.com/managed-devices/{device-id}
  - http://www.potus.org/user-management/users/{id}
  - http://www.library.edu/books/ISBN-0011/authors

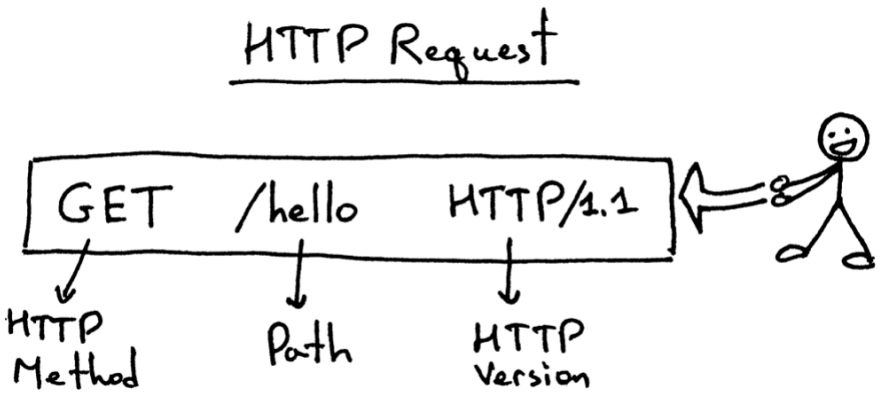


- As you traverse the path from more generic to more specific, you are navigating the data

Verbs

- Represent the actions to be performed on resources

- HTTP GET
- HTTP POST
- HTTP PUT
- HTTP DELETE



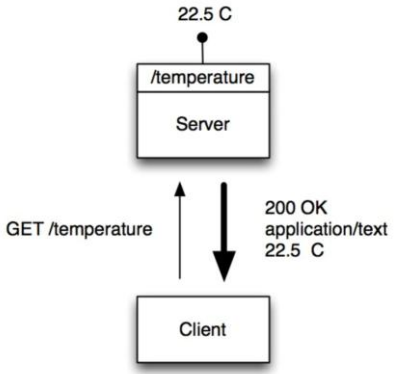
Running example of a resource: a ‘todo’ list

```
>>> tasks
[
{'id': 2345, 'summary': 'recipe for tiramisu’, 'description':
'call mom and ask...’},
{'id': 3657, 'summary': 'what to buy today’, 'description': '6
eggs, carrots, spaghetti’}
]

>>> tasks[1]['id']
2345

>>> tasks[1]
{'id': 2345, 'summary': 'recipe for tiramisu’, 'description':
'call mom and ask...’}
```

- JavaScript Object Notation (JSON) most spread syntax for formatting/serializing data, e.g., objects, arrays, numbers, strings, booleans

HTTP GET

- How clients ask for the information they seek.
- Issuing a GET request transfers the data from the server to the client in some representation (JSON, XML, …)



- GET /tasks/
  - Return a list of items on a todo list, in the format
  - {"id": <item_id>, "summary": <one-line summary>}
- GET /tasks/<item_id>/
  - Fetch all available information for a specific todo item, in the format
  - {"id": <item_id>, "summary": <one-line summary>, "description" : <free-form text field>}

HTTP POST

- Creates a resource

- POST /tasks/
  - Create a new todo item. The POST body is a JSON object with two fields: “summary” (must be under 120 characters, no newline), and “description” (free-form text field).
  - On success, the status code is 201, and the response body is an object with one field: the id created by the server (e.g., { "id": 3792 }).

HTTP PUT

- Updates a resource

- PUT /tasks/<item_id>/
  - Modify an existing task. The PUT body is a JSON object with two fields: “summary” (must be under 120 characters, no newline), and “description” (free-form text field).

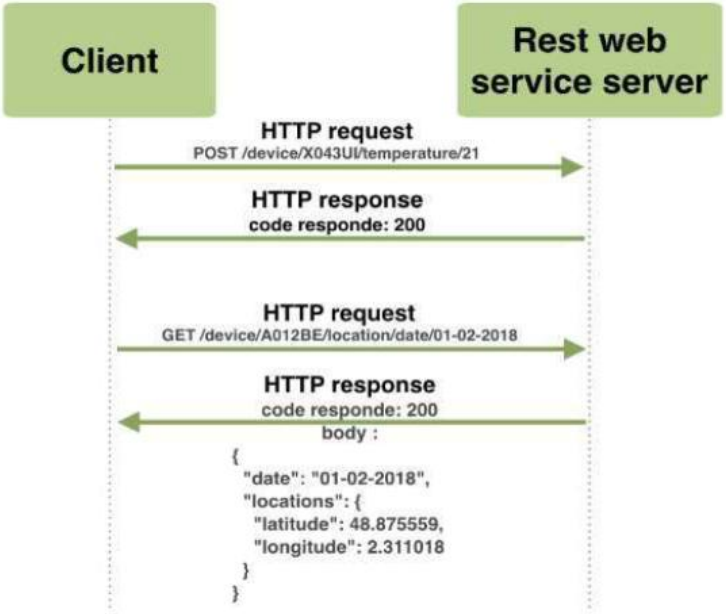
HTTP DELETE

- Removes the resource identified by the URI

- DELETE /tasks/<item_id>/
  - Mark the item as done. (I.e., strike it off the list, so GET /tasks/ will not show it.)
  - The response body is empty.

HTTP response codes

- The server also adds a three-digit HTTP response code to indicate the status of the response, which takes the following form:
  - 2xx indicates successful processing of the client request (e.g., 200 for OK)
  - 3xx redirects the client to another link
  - 4xx indicates an error in the client request (e.g., 404 for Not Found)
  - 5xx indicates a server error (e.g., 500 for Internal Server Error)

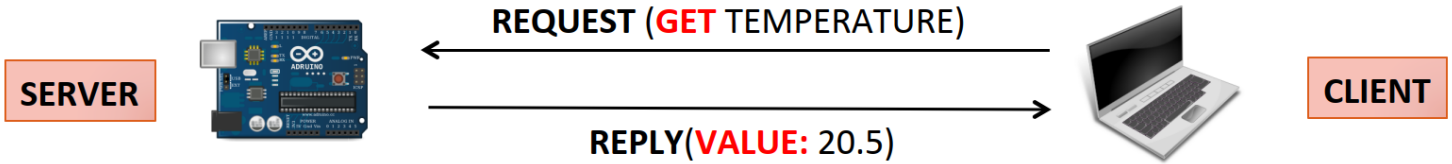


REST summary

- Advantages
  - Uniform and very widely adopted interface
  - The business logic is decoupled from the presentation
  - Allows to retrieve any historical data
- Disadvantages
  - Pull-based: being up to date requires polling, can produce unnecessary workloads and bandwidth consumption
  - One-to-one interaction
  - Too heavy for small devices (see Constrained Application Protocol, CoAP)

The CoAP protocol

- Constrained Application Protocol (CoAP)
- Messaging protocol based on a client-server architecture.
- Intended for use on constrained nodes and constrained networks.
- Lightweight version of REST designed for UDP communications.
- Does not replace HTTP



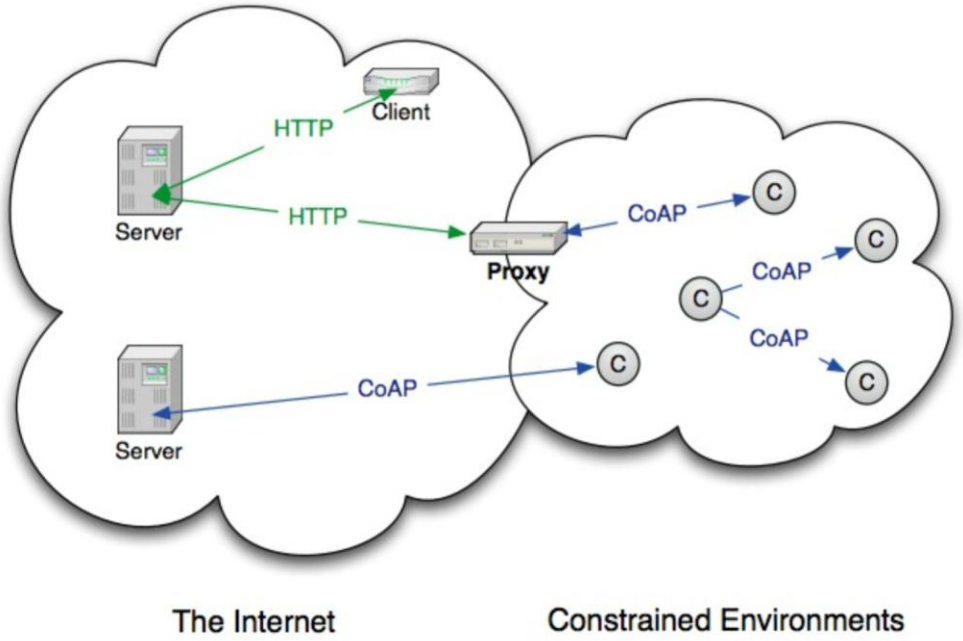
Why not HTTP?

- TCP overhead is too high and its flow control is not appropriate for short-lived transactions.
- UDP has lower overhead and supports multicast
- HTTP transactions are around 10 times higher than CoAP transaction bytes due to 6LoWPAN and CoAP header compression
- CoAP packet can be sent in single frame without fragmentation.
- Less bytes → lower power consumption and longer lifetime for CoAP.

Other CoAP features

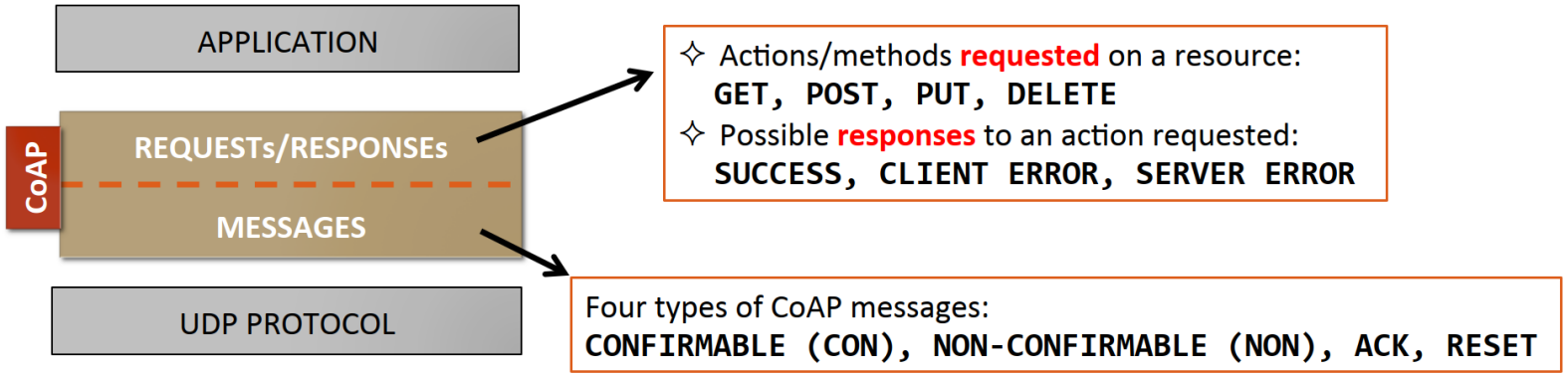
- Optional mechanisms can be used for enhanced reliability, i.e. confirmable messages + retransmissions
- Shorter, fixed-size (4 bytes) packet header
- Asynchronous request/response paradigm
- Resource discovery
- Proxy mechanisms

CoAP architecture



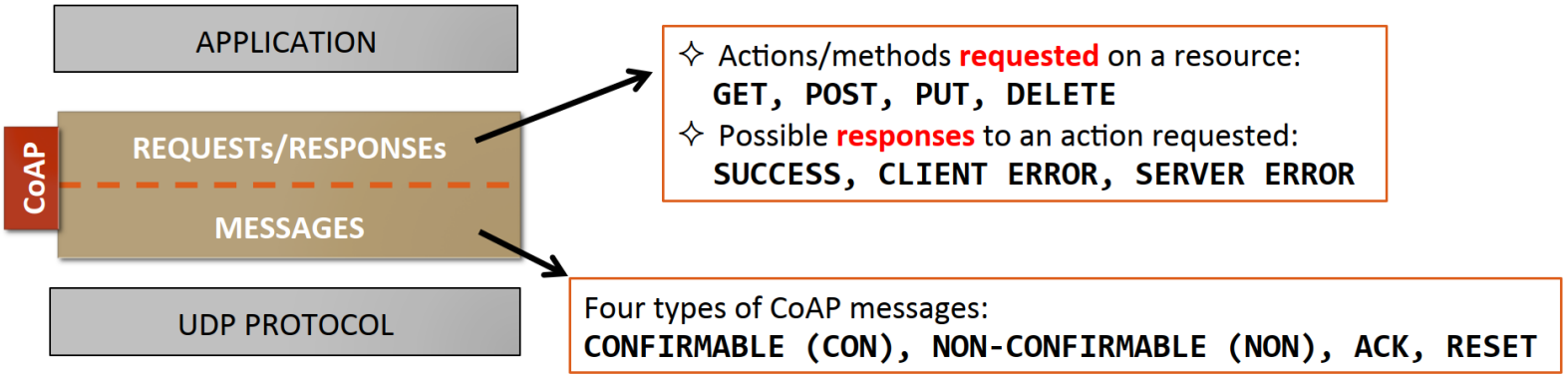
CoAP

- CoAP operations can be LOGICALLY split in two sub-layers:
  - Requests/responses → client-server RESTful interactions
  - Messages → paradigm implementation + reliability mechanisms



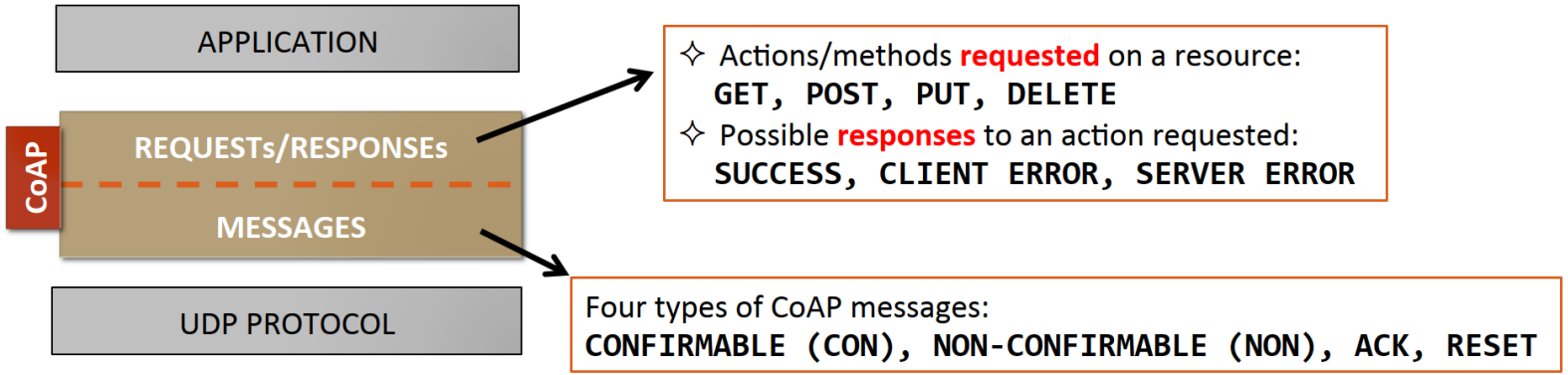
CoAP

- Four message types:
  - Confirmable (CON): Message sent with an acknowledgment request.
  - Non-Confirmable (NON): Message sent without an acknowledgment request.
  - Acknowledgment (ACK): Acknowledgment of the CON message.
  - Reset (RST): Acknowledgment of a message that cannot be processed.



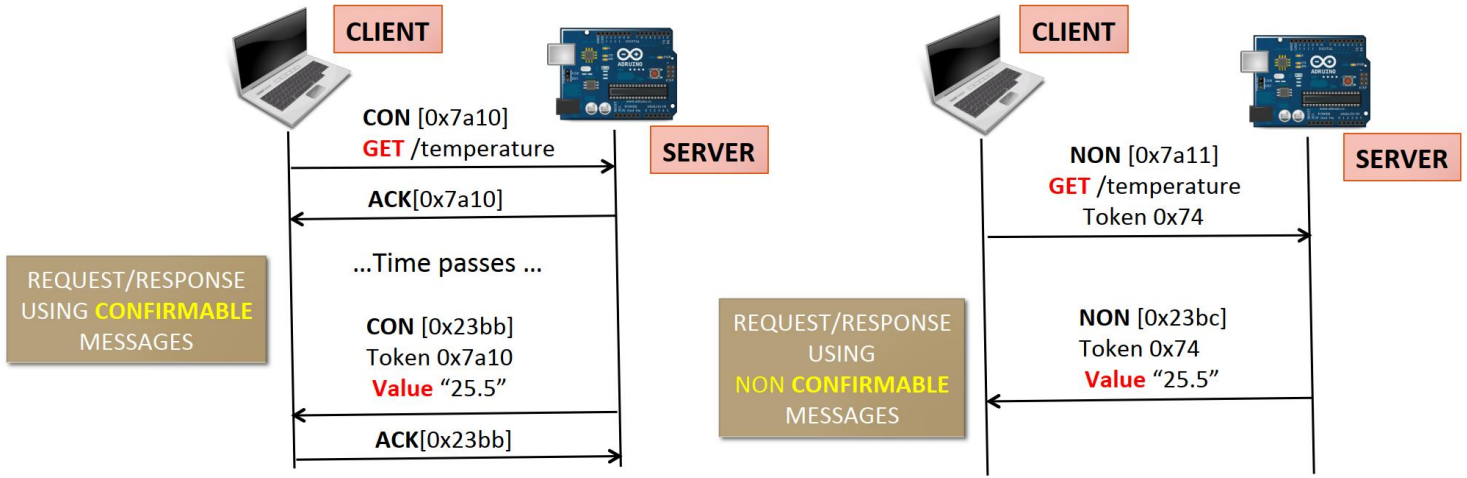
CoAP

- Three response codes:
  - Success (2.xx): Request was successfully received and processed
  - Client Error (4.xx): Client encountered an error
  - Server Error (5.xx): Server is unable to process the request



CoAP message exchanges

- For example, if the request is of type CON, the server returns a response containing the message type (ACK), the same message ID (mid) as the request, a response code (2.xx, 4.xx, or 5.xx), and a representation of the resource.

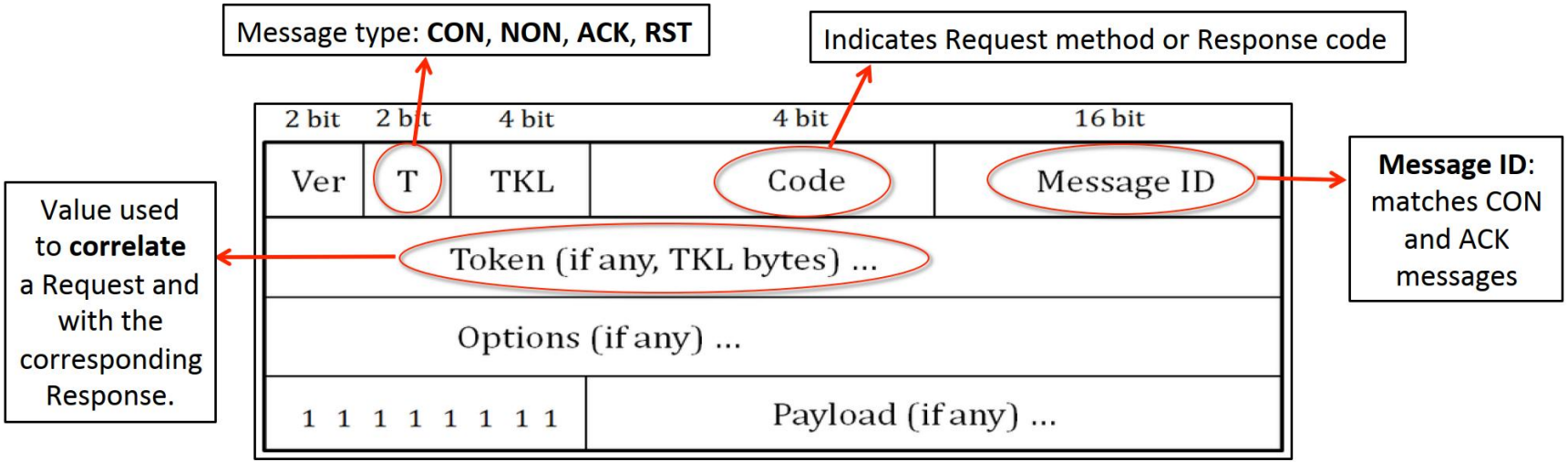


CoAP URI

- Each resource is addressed by an URI
- coap-URI = "coap:" "//" host [":" port] path ["?" query]

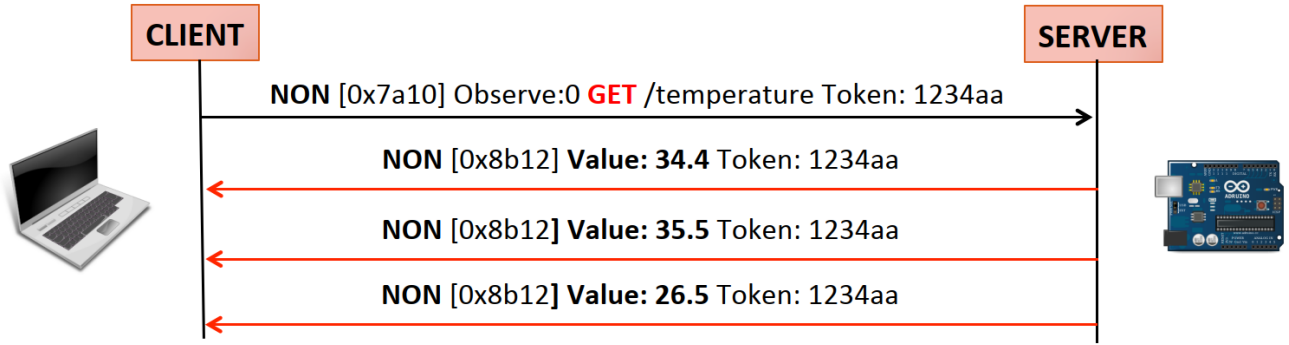
```
coap://dante.cs.unibo.it/temperature/serverRoom
```

CoAP message header



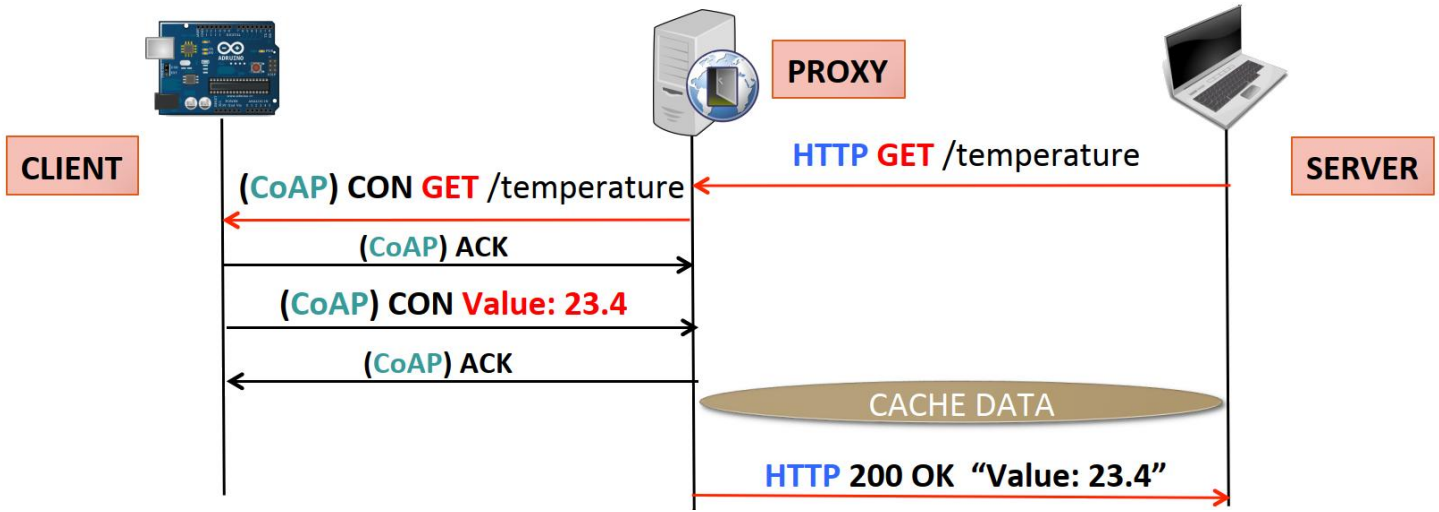
CoAP “observation”

- The OBSERVE mechanism allows implementing a data subscription mechanism (similar to MQTT, but without the broker).
  1. The client requests a resource (GET) with the Observe Option field.
  2. The server add the client to the list of observers of the resource
  3. At each change of the target resource, the server notifies all its observers

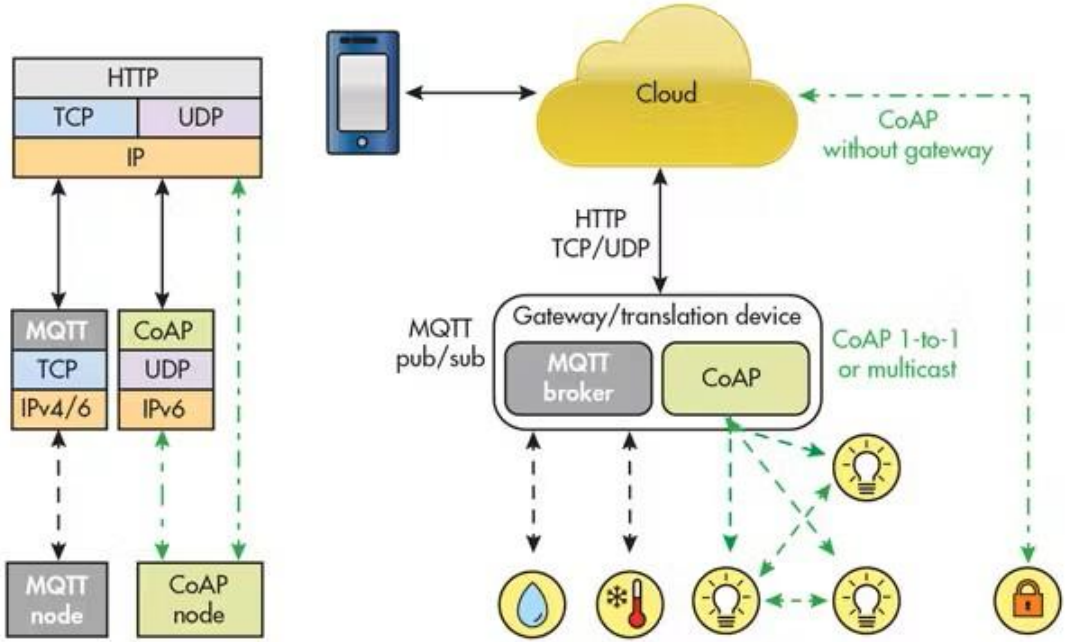


CoAP proxy

- CoAP only supports a limited subset of HTTP functionality; however, cross-protocol proxy mechanisms can guarantee seamless HTTP-CoAP interactions



MQTT / CoAP and HTTP deployment scenario

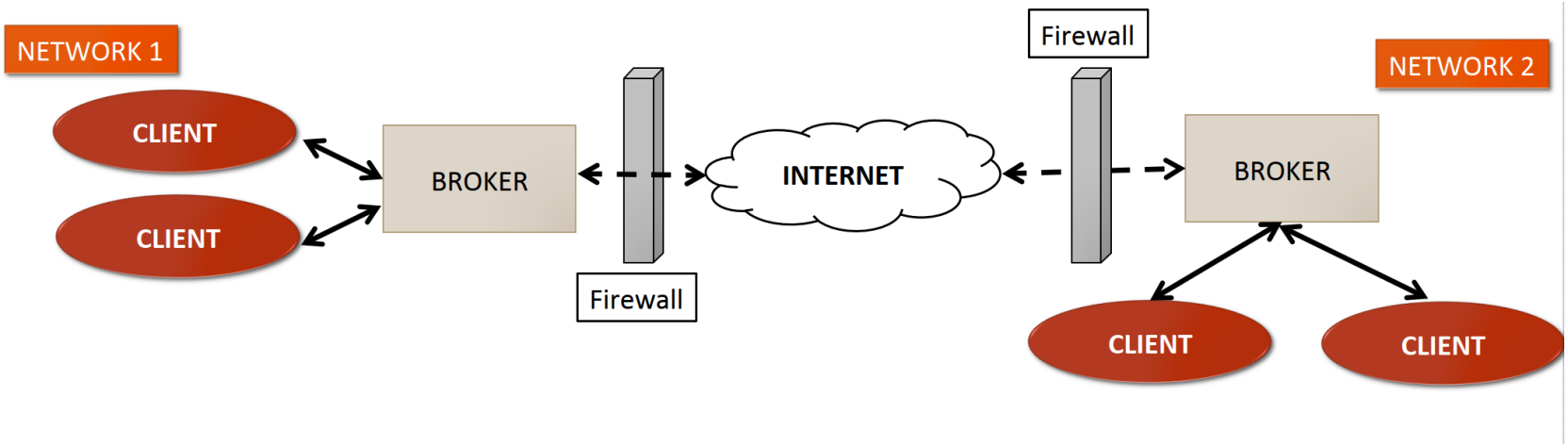


The AMQP Protocol

- Advanced Message Queuing Protocol (AMQP)
- Open-standard protocol for message-oriented applications.
- It supports system interoperability in distributed environments.
- Based on TCP protocol with additional reliability mechanisms (at-most-once, at-least-once or once-delivery).
- It supports both point-to-point communication and publish-subscribe communication paradigms (like MQTT).
- Programmable protocol: several entities and routing schemes are primarily defined by applications.

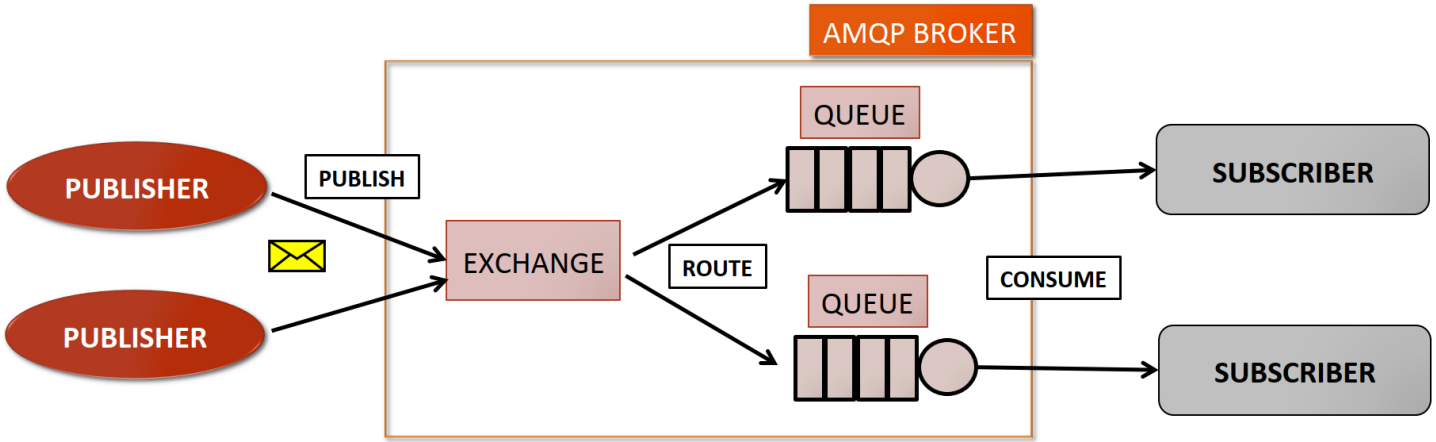
AMQP architecture

- AMQP natively supports system integration and message-oriented communication over the Internet.



AMQP actors

- The AMQP architecture involves three main actors: publishers, subscribers, and brokers.

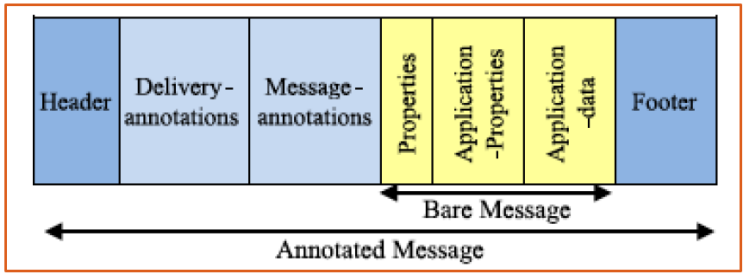


AMQP entities

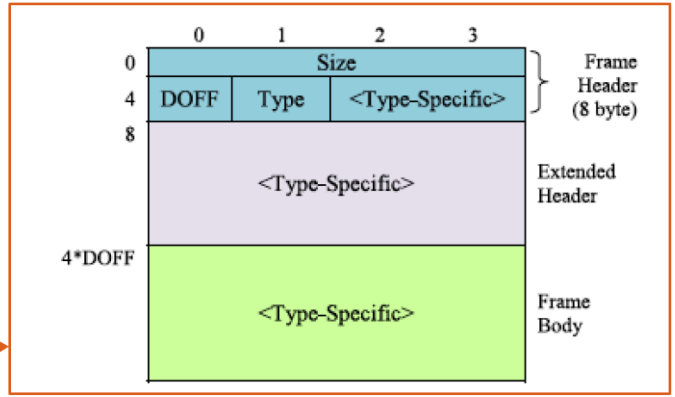
- Within the broker:
  - Queues: application-specific message buffers
  - Exchanges: often compared to post offices or mailboxes, take a message and route it into zero or more queues
  - Bindings: Rules followed by the exchange for the routing process

AMQP messages

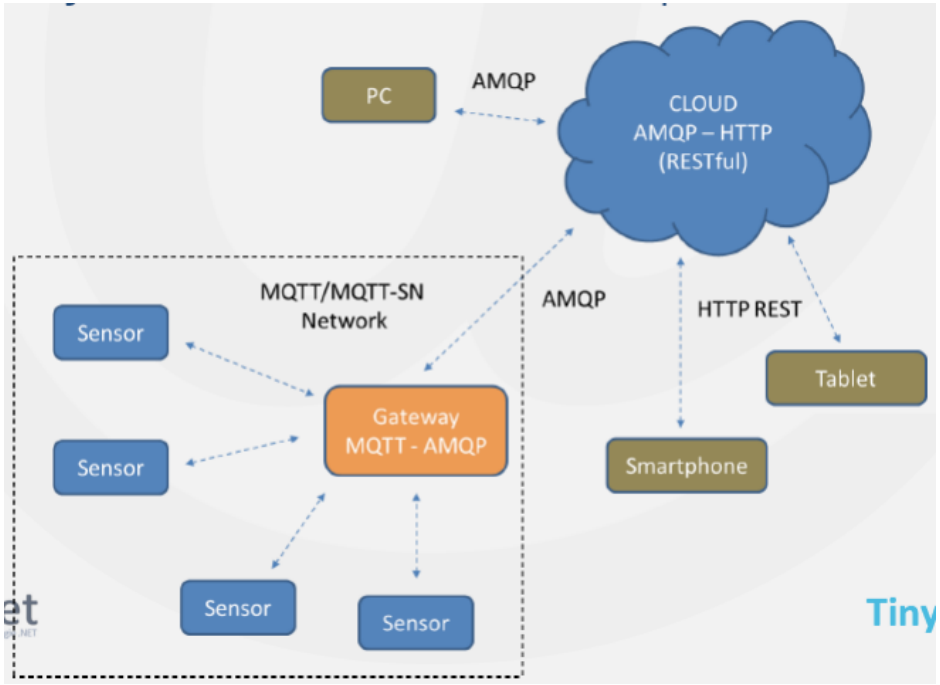
- The AMQP protocol defines two types of messages:
  - Bare messages, that are supplied by the sender.
  - Annotated messages, that are seen at the receiver



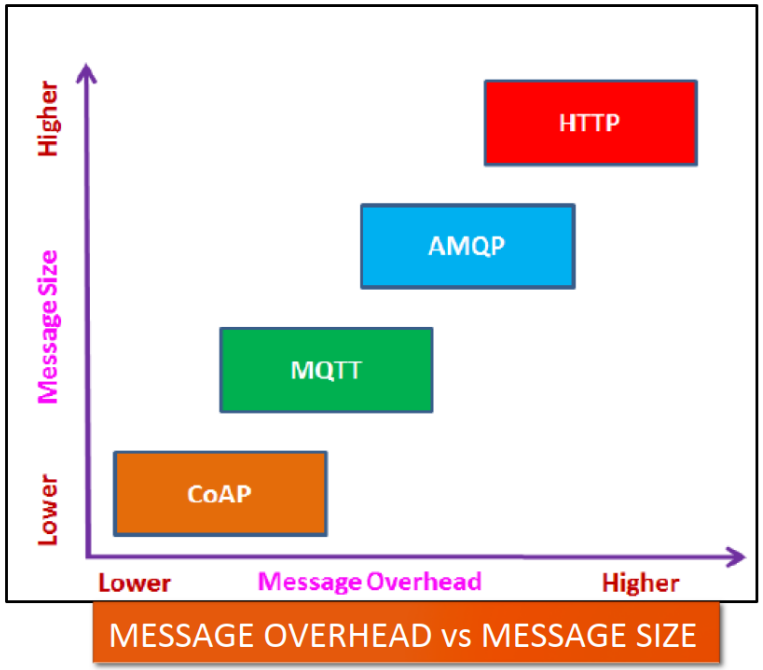
The **header** conveys the delivery parameters including: durability, priority, time to live, first acquirer, delivery count

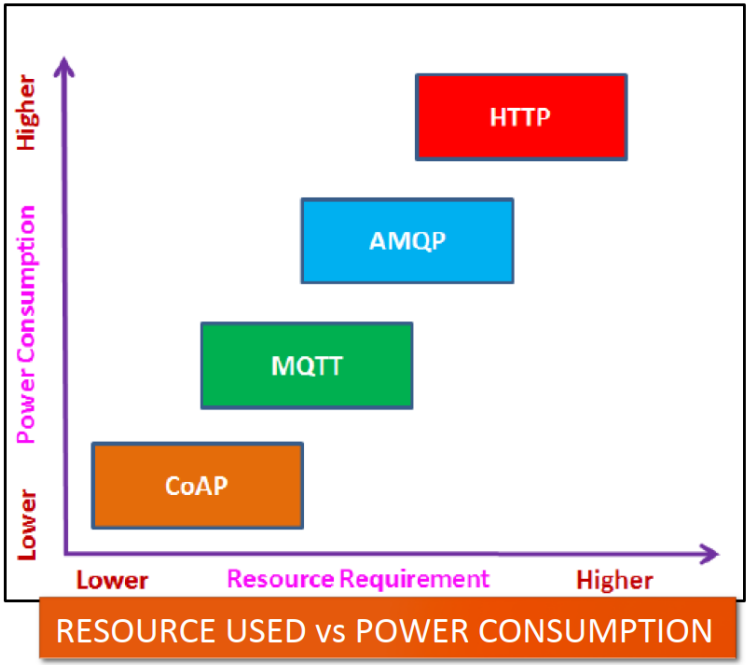


MQTT and AMQP deployment scenario

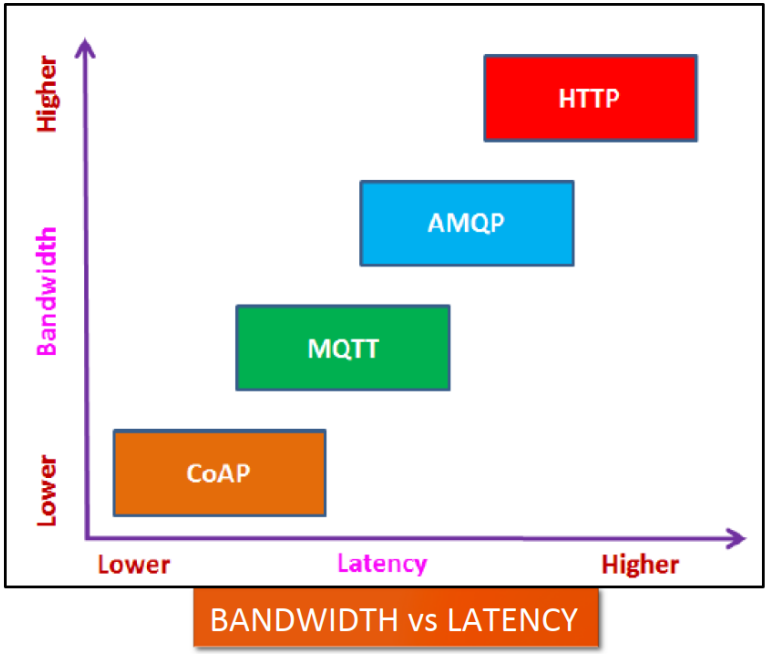


Protocol comparison

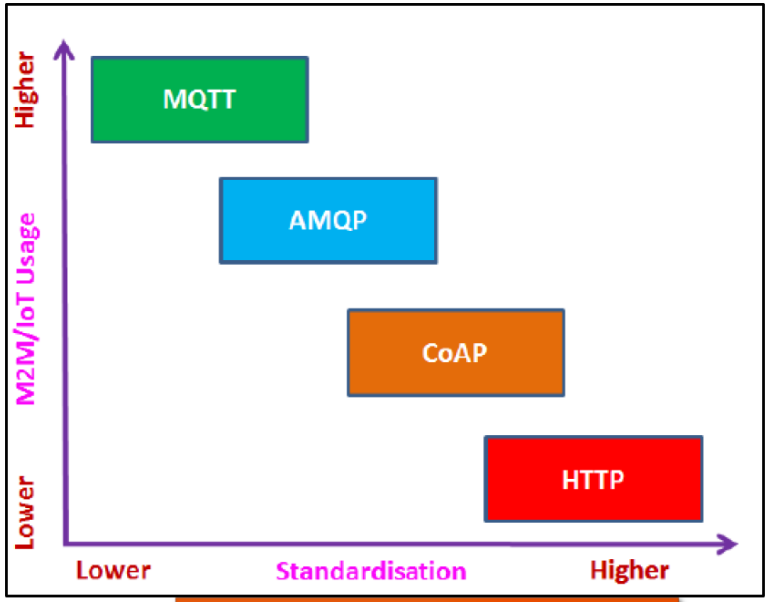




Protocol comparison

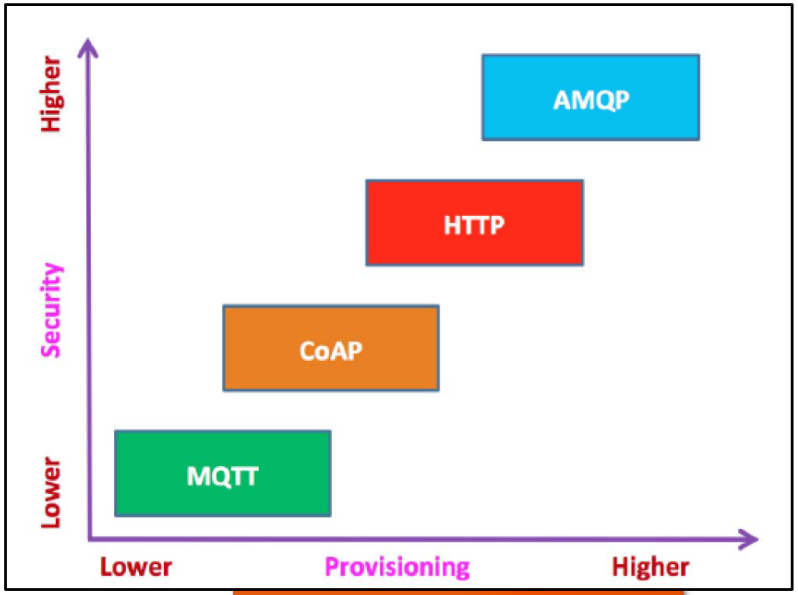


BANDWIDTH vs LATENCY

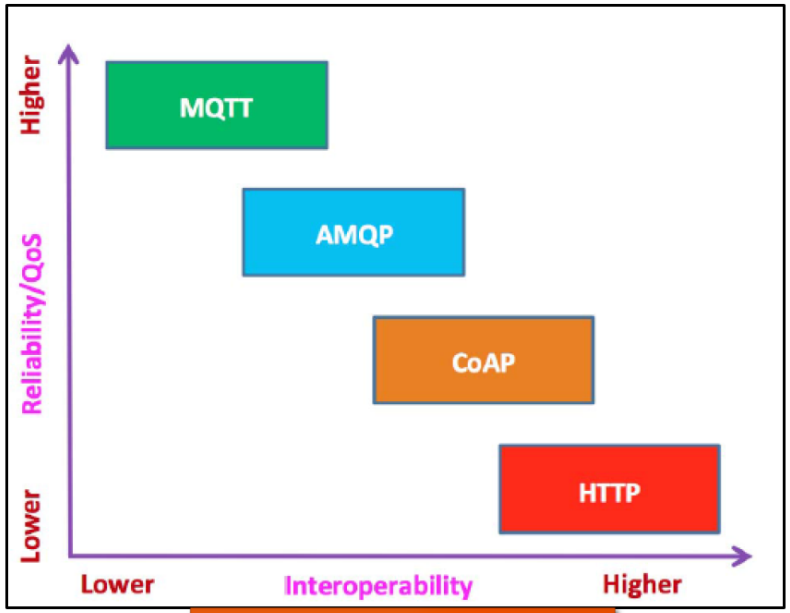


STANDADISATION vs IoT USAGE

Protocol comparison



PROVISIONING vs SECURITY



QoS vs INTEROPERABILITY

Protocol trends