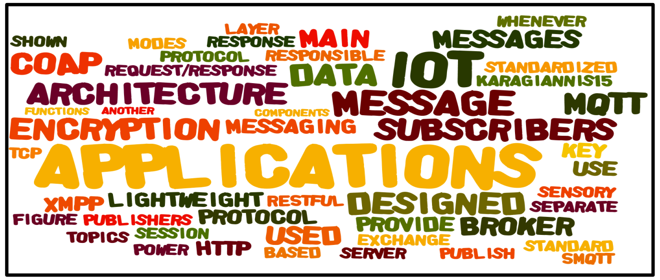


The Internet of Things: Messaging Protocols

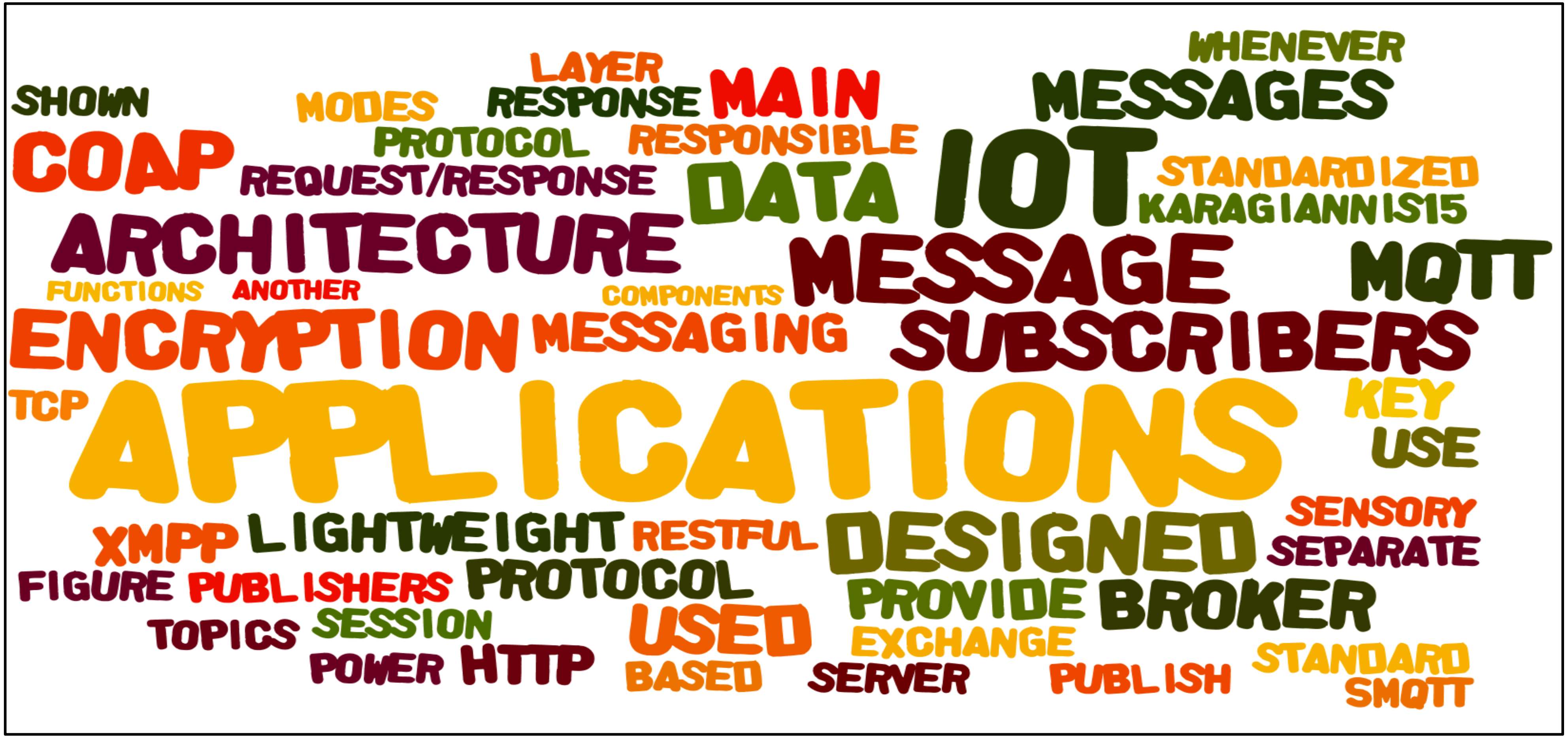
Course website: http://site.unibo.it/iot

Prof. Luciano Bononi
luciano.bononi@unibo.it

Prof. Marco Di Felice
marco.difelice3@unibo.it

MASTER DEGREE IN COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, UNIVERSITY OF BOLOGNA, ITALY

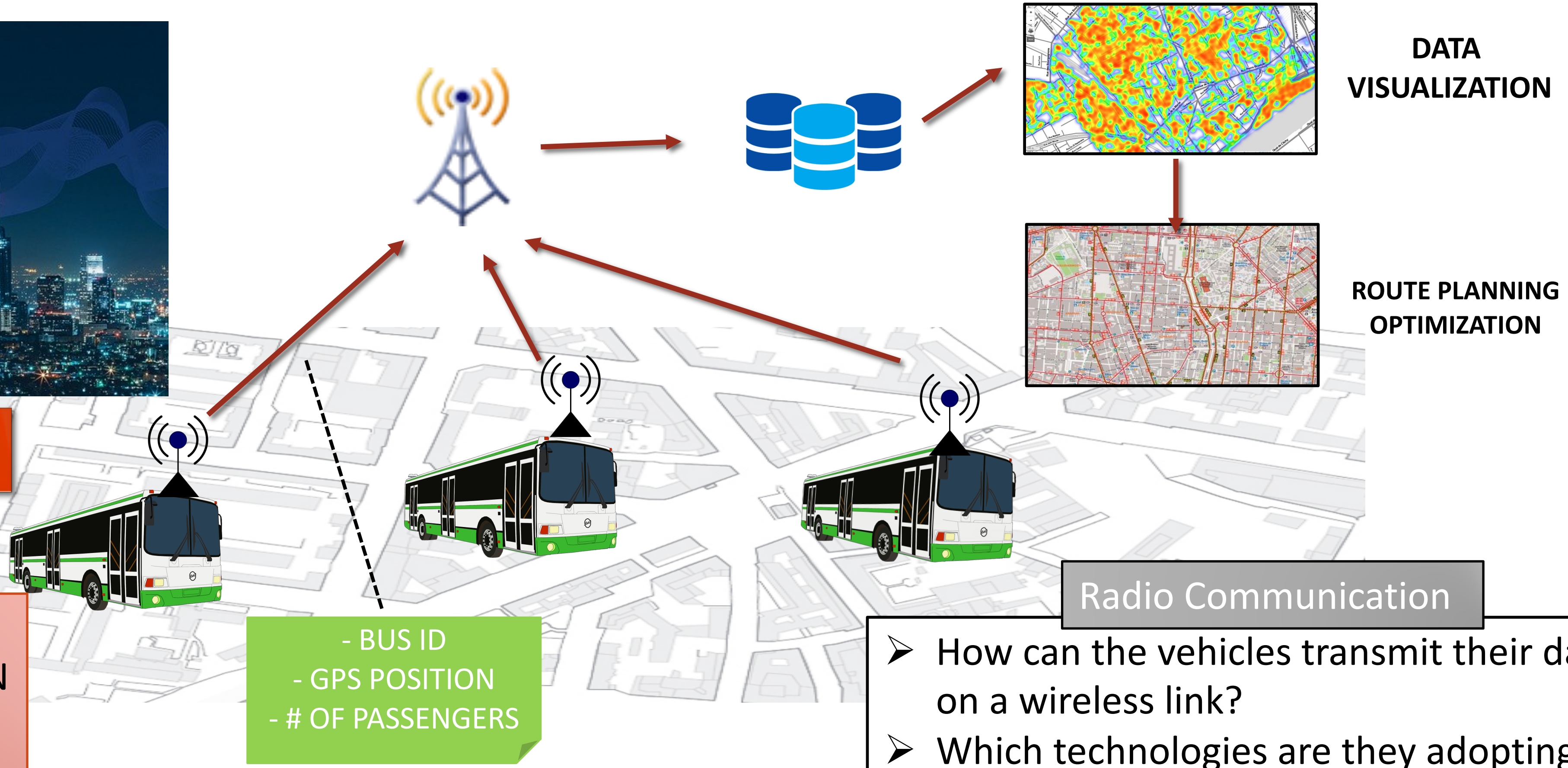
Overview


WHENEVER
LAYER
SHOWN
MODES
RESPONSE
MAIN
MESSAGES
PROTOCOL
RESPONSIBLE
COAP
REQUEST/RESPONSE
DATA
IOT
STANDARDIZED
KARAGIANNIS15
ARCHITECTURE
MESSAGE
MQTT
FUNCTIONS
ANOTHER
COMPONENTS
ENCRYPTION
MESSAGING
SUBSCRIBERS
TCP
APPLICATIONS
KEY
USE
SENSORY
XMPP
LIGHTWEIGHT
RESTFUL
DESIGNED
SEPARATE
FIGURE
PUBLISHERS
PROTOCOL
PROVIDE
BROKER
TOPICS
SESSION
USED
EXCHANGE
STANDARD
POWER
HTTP
BASED
SERVER
PUBLISH
SMQTT

Overview

SMART CITY

PUBLIC TRANSPORTATION SYSTEM MONITORING



Radio Communication

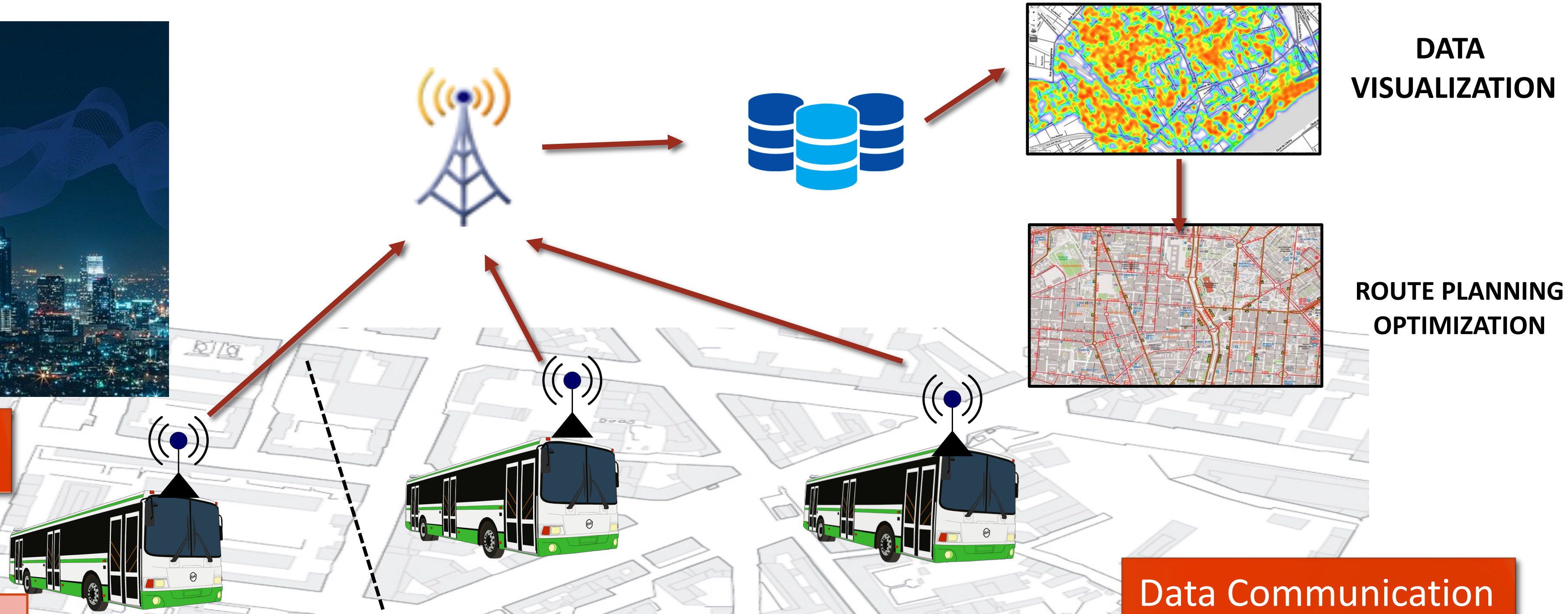
- How can the vehicles transmit their data on a wireless link?
- Which technologies are they adopting?

COVERED BY PREVIOUS LESSONS

Overview

SMART CITY

PUBLIC TRANSPORTATION SYSTEM MONITORING



- BUS ID
- GPS POSITION
- # OF PASSENGERS

Data Communication

- What about the **interaction** model?
- Let us consider the **protocol for sensor data acquisition** … How it works?

COVERED IN THIS LESSON

IoT Protocol Stack

MESSAGING PROTOCOLS	MQTT	CoAP	AMQP	HTTP	OTHERS
TRANSPORT PROTOCOLS	TCP		UDP		
NETWORK PROTOCOLS	IPv4 and IPv6 + 6LoWPAN				
PHY/MAC PROTOCOLS	IEEE 802.3	IEEE 802.11	IEEE 802.15	IEEE 802.16	OTHERS

IoT Messaging Protocols

✧ **Session/Application** Layer Protocols

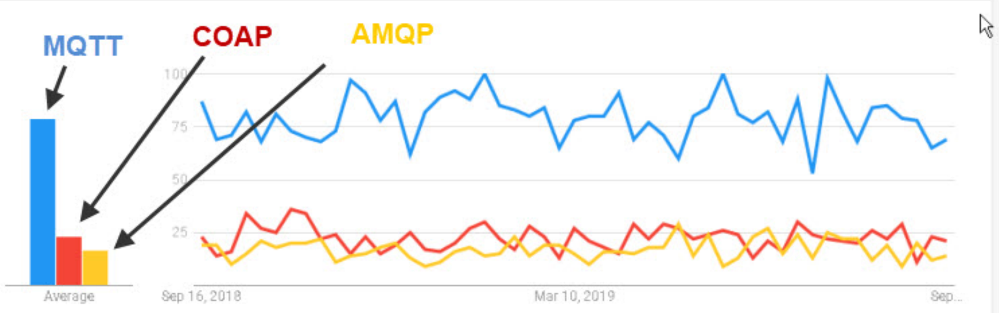
1. Providing the abstraction of "**message**" (elementary unit of data communication among IoT end-points).
2. Providing **primitives for data communication/message exchange** to the upper-layer IoT applications.
3. Implementing specific **networking paradigms** (e.g. publish-subscribe or request-response).
4. Providing additional **reliability** or **security** mechanisms.
5. Sometimes adaptation of **pre-existing** (not natively M2M) **solutions**

IoT Messaging Protocols

✧ Based on the interaction paradigm, IoT messaging protocols can be classified into two main categories:

❑ **Publish-subscribe protocols**

❑ Request-response protocols



Google Trends MQTT,COAP and AMQP

IoT Messaging Protocols

✧ **Session/Application** Layer Protocols … WHICH protocols?

- **HTTP**
- **MQTT**
- **CoAP**
- **AMQP**
- **XMPP** (not covered here)
- **DDS** (not covered here)
- …

We will talk more about HTTP for M2M communications later in this course (when discussing about the Web of Things). In any case, it cannot be considered an IoT-native protocol …

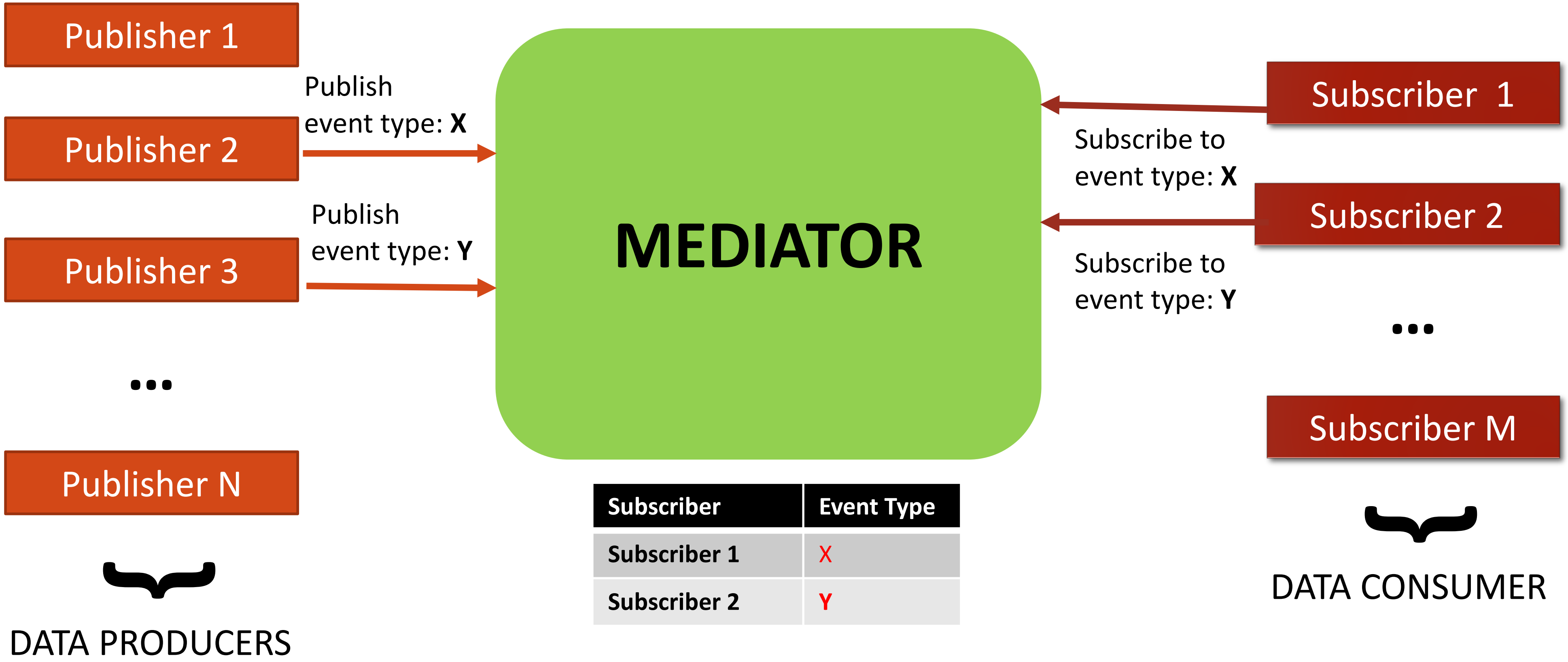
The MQTT Protocol

- **Message Queuing Telemetry Transport Protocol (MQTT)**
  - **Lightweight messaging protocol** designed for M2M (machine to machine) telemetry in resource-constrained environments.
  - Proposed initially by Andy Stanford-Clark (IBM) and Arlen Nipper in 1999 for connecting Oil Pipeline telemetry systems over satellite.
  - Released Royalty free in 2010 and as OASIS standard in 2014
    - **MQTT** (current specification 3.1/3.1.1)
    - **MQTT for Sensor Networks** (MQTT-SN)

Publish/subscribe paradigm: Overview

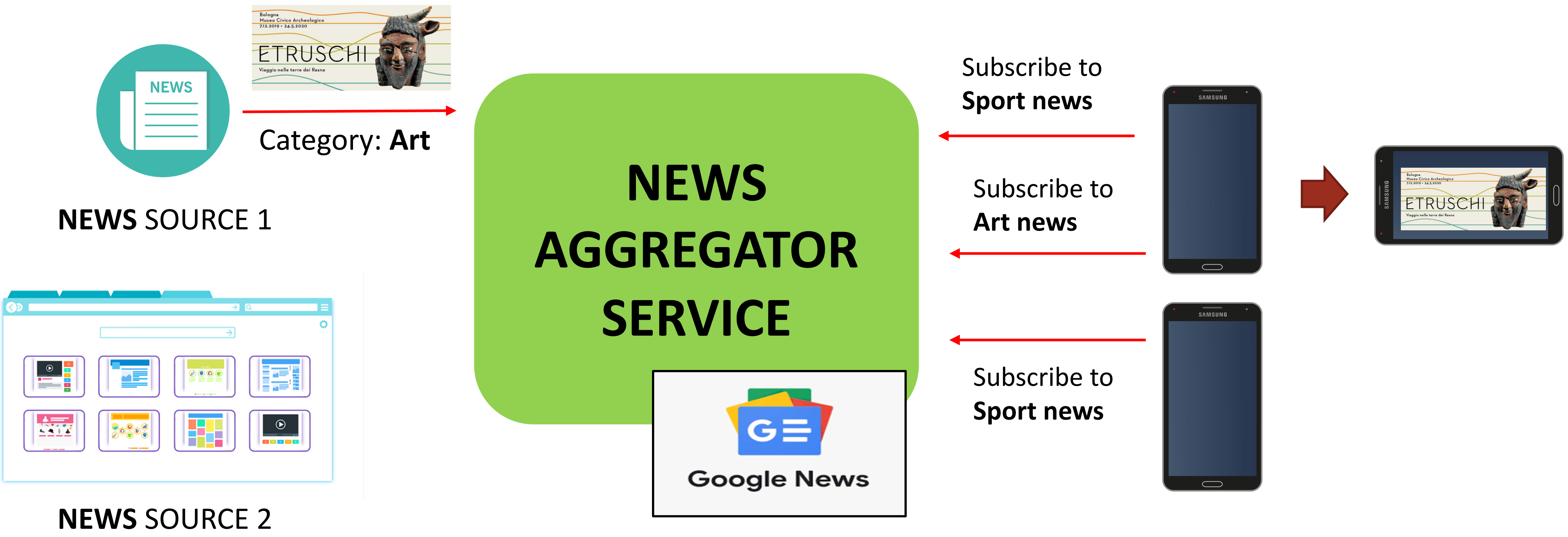
- Publish/subscribe is a popular communication paradigm involving the presence of three actors:
  - **Publishers**: produce data in forms of events
  - **Subscribers**: declare their interest in specific events
  - **Mediator**: notifies to each subscribers every published event that matches its subscription
- Roles of publishers/subscribers are purely logical
- The paradigm is general and can be applied on many different use-cases of distributed/networking systems.

Publish/subscribe paradigm: Overview



Subscriber	Event Type
Subscriber 1	X
Subscriber 2	Y

Publish/subscribe paradigm: Overview



Publish/subscribe paradigm: Overview

❑ Paradigm Characteristics

- **C1.** Many-to-many interactions

Same piece of information can be delivered at the same time to various consumers. Each consumer receives information from various producers.

- **C2.** Space **Decoupling**

Interacting parties do not need to know each other. Message addressing is based on their content.

- **C3.** Time **Decoupling**

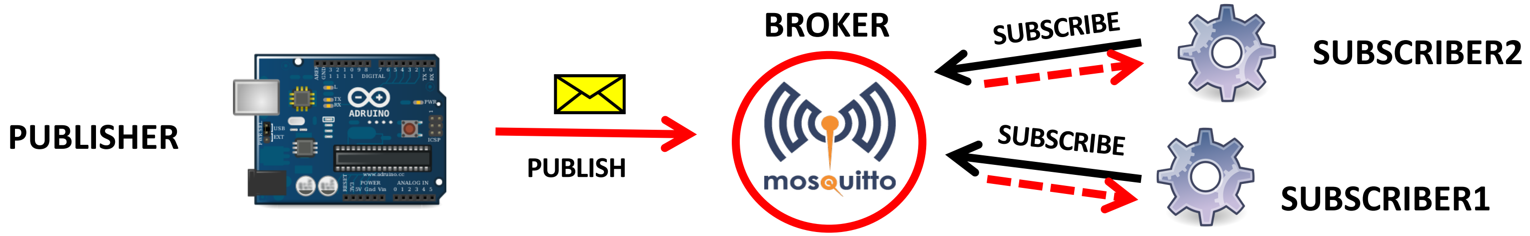
Interacting parties do not need to be actively participating in the interaction at the same time.

The MQTT Protocol

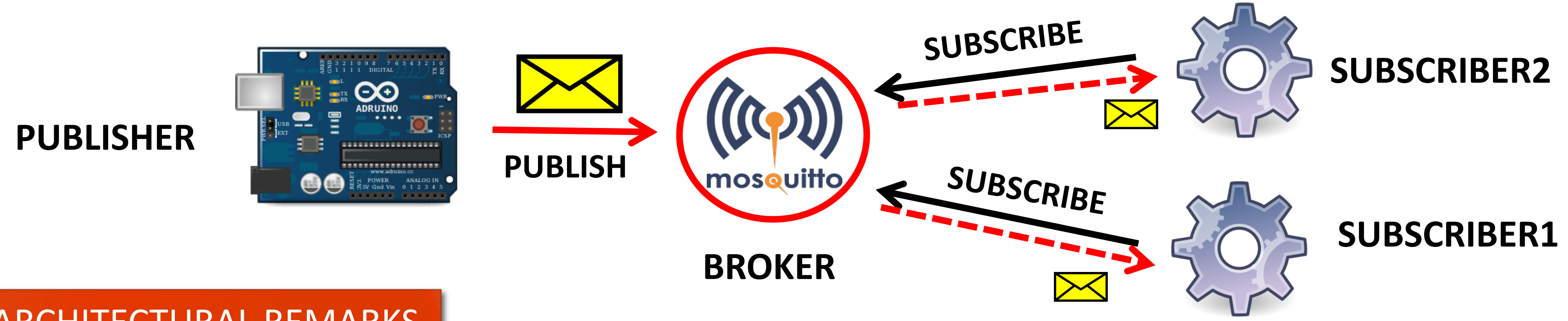
- ✧ The MQTT protocol implements a **publish-subscribe messaging** mechanism, involving three main actors:

CLIENTs

- ✧ **Publishers →** produce data and send them to a broker.
- ✧ **Subscribers →** subscribe to a topic of interest, and receive notifications when a new message for the topic is available.
- ✧ **Broker →** filter data based on topic and distribute them to subscribers.



The MQTT Protocol



ARCHITECTURAL REMARKS

- A topic defines the **message context** (e.g. temperature data).
- **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).

The MQTT Protocol

✧ MQTT **Control Packet Format**

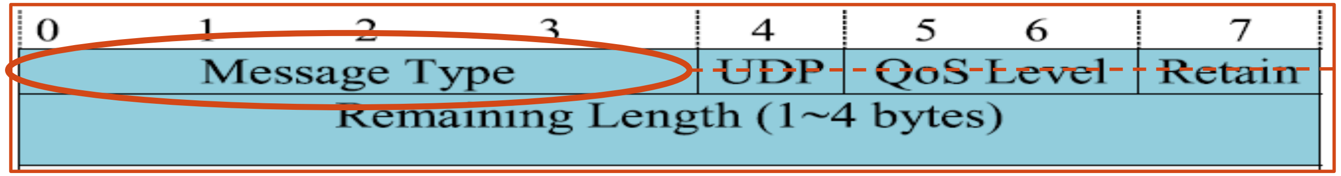
FIXED | VARIABLE | PAYLOAD

Fixed **Header**
(**only 2 bytes**!)

Optional, variable size

Optional, binary encoded, variable size **up to 256 MB**

✧ The FIXED header contains the **packet type**, the **size of the payload**, and the level of **Quality of Service** (see next slides).



14 different commands:

CONNECT, CONNACK, PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE, UNSUBSCRIBEACK, PINGREQ, PINGRESP, DISCONNECT

The MQTT Protocol

✧ MQTT **Control Packet Format**

FIXED	VARIABLE	PAYLOAD

✧ The Variable header contains the additional parameters based on the command type. For instance, the header of the PUBLISH/SUBSCRIBE message contains the **TOPIC** field.

✧ The TOPIC is a **string** field, without a specific format, just **naming conventions** (see [1]).

```
topic=temperature
```

→ **PLAIN NAMING**

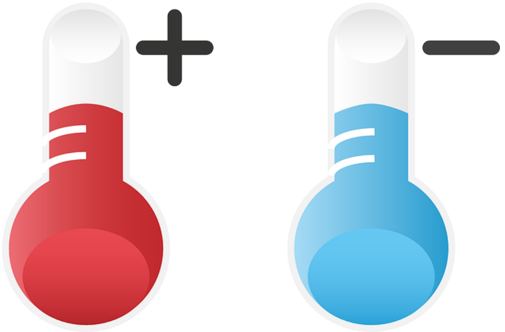
```
topic=temperature/kitchen/
topic=temperature/kitchen/sensor1
```

→ **HIERARCHIAL NAMING** (TOPIC **NAME** + TOPIC **FILTER**)

The MQTT Protocol

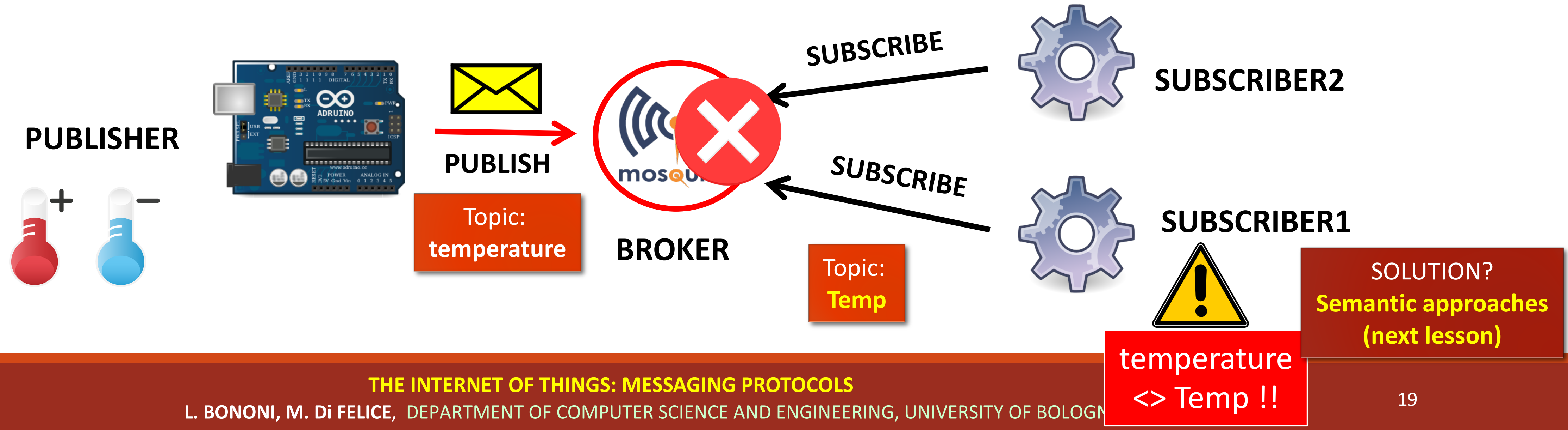
- The topic is a string, without any specific format
- Wildcard used for the topic definition:
  - + replaces one topic level
  - # replaces many topic levels

```
topic=data/temperature/kitchen/
topic=data/temperature/livingRoom/

topic=data/+/kitchen (SINGLE-LAYER)
topic=data/#     (MULTI-LAYERS)
```



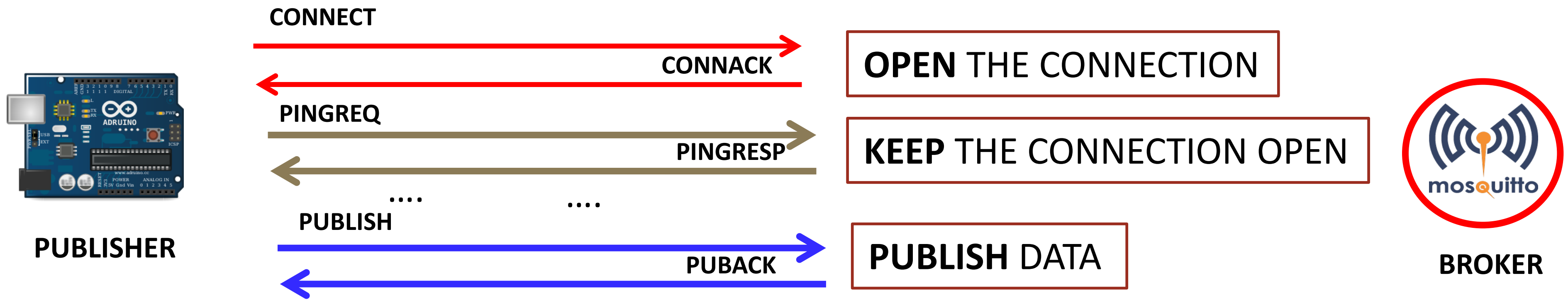
The MQTT Protocol

- ❑ The data consumer may not know the identity and location of the data producer …
- ❑ However, it must **match the topic name** used by data producers (i.e., it must know the string).



The MQTT Protocol

- MQTT is **built on top of the TCP protocol**
  - In-order delivery, connection-oriented, ACK and retranmissions.
  - .. but also longer TSP header size and higher complexity.
  - **MQTT-SN** → uses **UDP**, supports topic IDs (instead of names).
- MQTT keeps the **TCP** connection between a client and a broker open as long as possible, by means of **PINGREQ** messages.



The MQTT Protocol

✧ Despite using TCP, **MQTT messages can still be lost …**

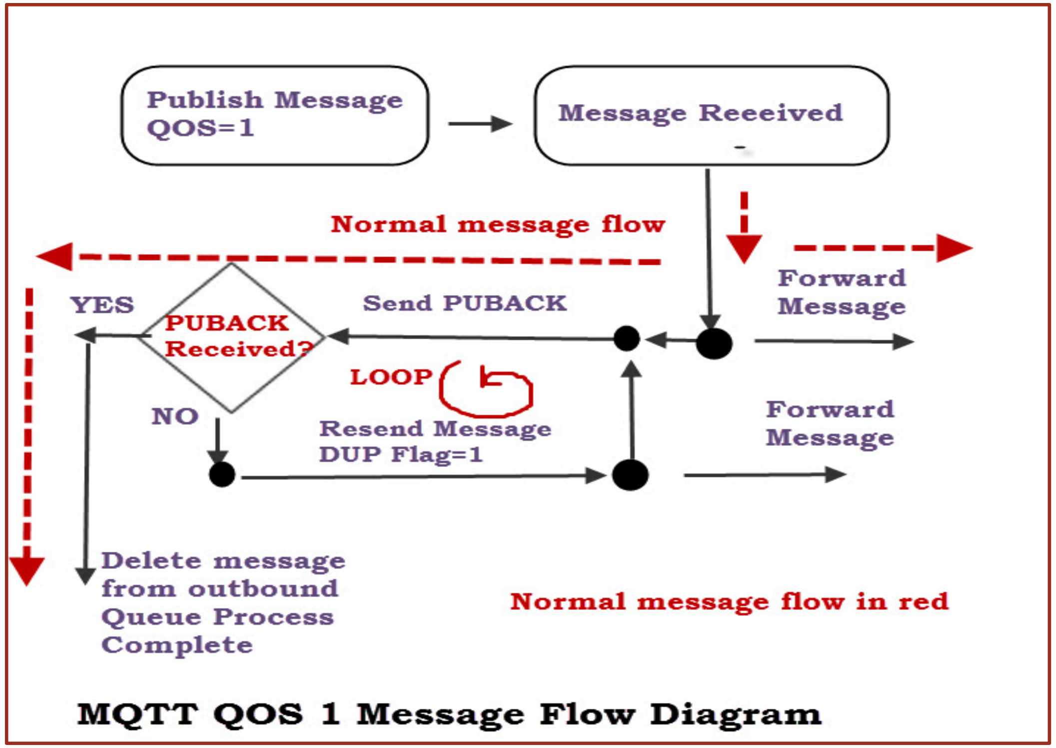
1. TCP guarantees delivery on a single link (agent →broker, or viceversa), what about publisher → subscriber delivery?
2. What about if the receiver is temporarily down while a sender is attempting to send a message?

✧MQTT clients can request three level of **Quality of Service** (**QoS**) to the broker:

- ✧**QoS Level 0** (**fire & forget**) → default QoS level, clients do not store messages and do not receive ACKs from broker, same delivery guarantees than TCP.
- ✧**QoS Level 1** (**deliver at least once**) → see next slide (10)
- ✧**Qos Level 2** (**deliver exactly once**) → see next slide (11)

The MQTT Protocol

✧ MQTT QoS Level 1 (Deliver at least one)

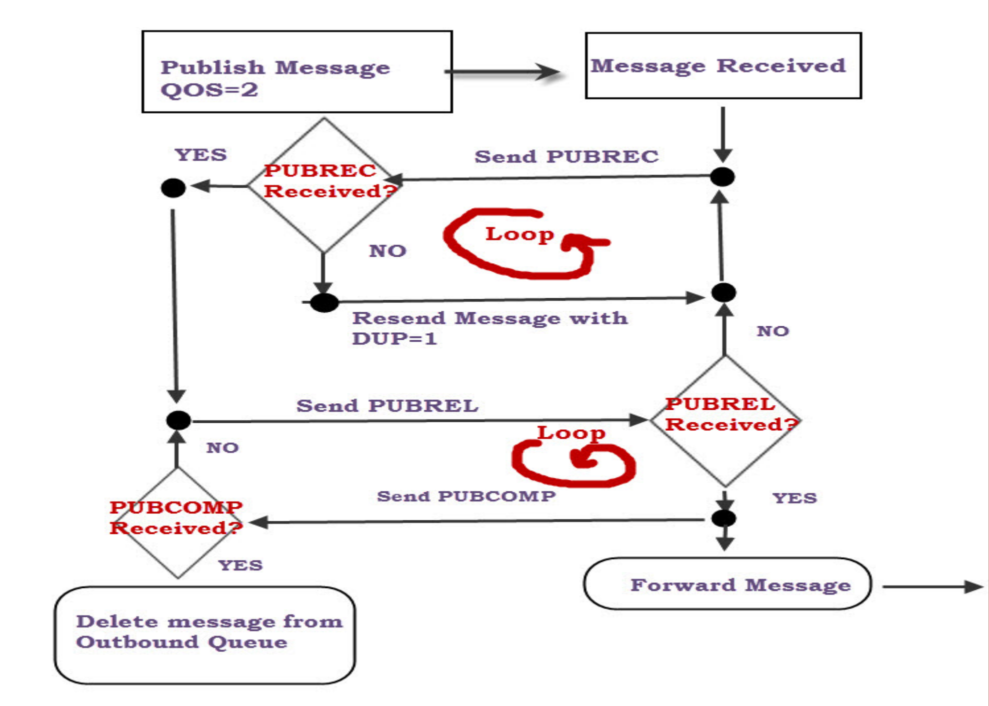


1. The client sends a message and waits for an acknowledgement (**PUBACK**) from the receiver.
2. If the PUBACK is received, the client deletes the message from the outbound queue.
3. Otherwise, it **resends the message at regular interval with the DUP flag set to 1, till a PUBACK is received**.

The receiver might receive the same data multiple times!

The MQTT Protocol

✧ MQTT QoS Level 2 (Deliver exactly once)



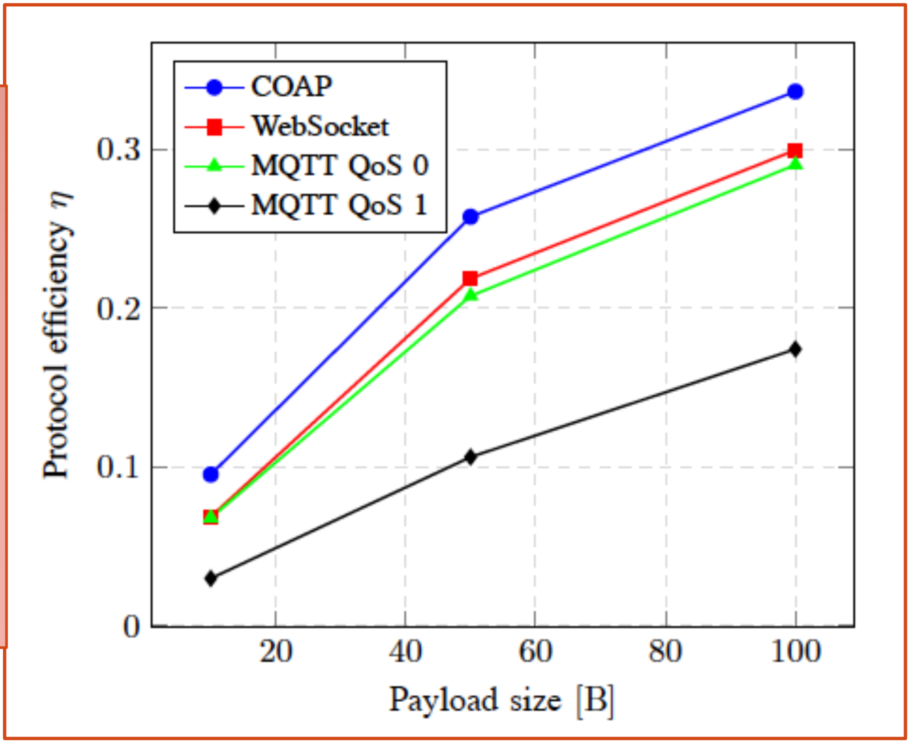
MQTT QOS 2 Message Flow Diagram

1. The sender sends a message and waits for an ACK (**PUBREC**)
2. The receiver sends a **PUBREC** message
3. If the sender doesn't receive an ACK(**PUBREC**) it will resend the message with the **DUP** flag set.
4. When the sender receives an ACK message PUBREC it then sends a message release message (**PUBREL**).
5. If the sender doesn't receive the PUBREL it will resend the PUBREC message
5. When the receiver receives PUBREL, it can now process the data.
6. The receiver then send a publish complete (**PUBCOMP**) .
7. If the sender doesn't receive the PUBCOMP message it will resend the PUBREL message.
8. When the sender receives the PUBCOMP the process is complete and it can delete the message from the outbound queue (finally!).

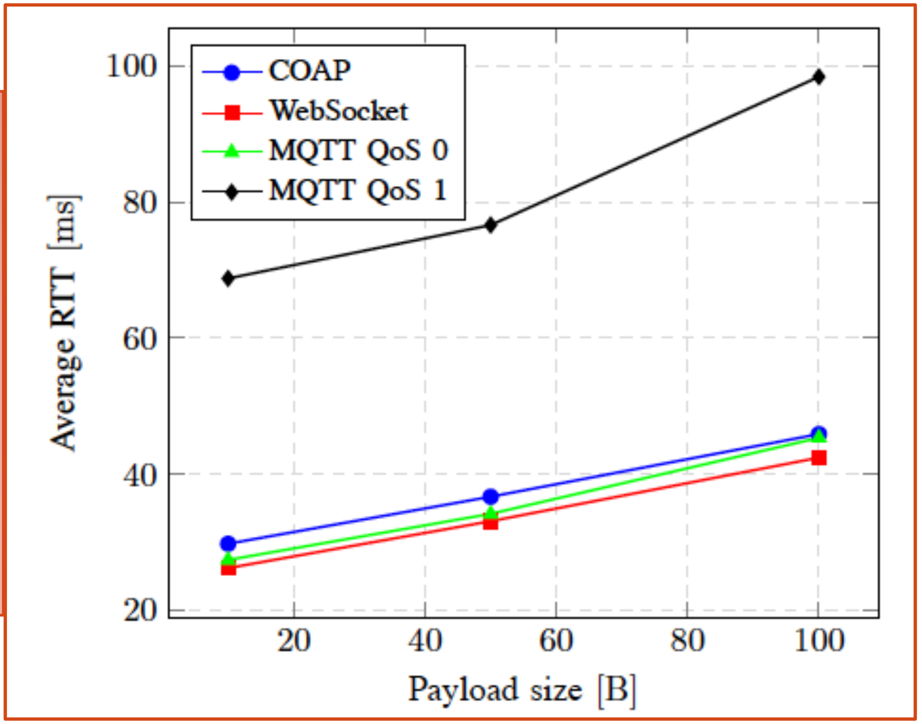
The MQTT Protocol

✧ MQTT QoS Level: evaluation results

PROTOCOL EFFICIENCY (*)



ROUND TRIP TIME (RTT)



(*) ratio between application bytes and overhead bytes

The MQTT Protocol

✧ MQTT QoS levels can be coupled with **additional settings at the broker side**, in order to ensure delivery of messages also in presence of client disconnections.

1. **RETAINED message**: The broker stores the last message for a specific topic. Each client that subscribes to that topic will receive the message immediately after subscribing. <u>For each topic only one retained message will be stored by the broker</u>.

(AIM) A newly connected subscribers will receive the latest update immediately and **shouldn't have to wait till next PUBLISH action**.

The MQTT Protocol

✧ MQTT QoS levels can be coupled with **additional settings at the broker side**, in order to ensure delivery of messages also in presence of client disconnections.

2. **PERSISTENT session**: The broker stores all the relevant information about clients, like: all subscriptions, or all QoS 1-2 data not confirmed since the client was offline.

(**AIM**) A client should explicitly get messages for the time it is offline …

clearly, increasing the resource utilization (CPU/HD) at the broker side!

The MQTT Protocol

- MQTT provides **some (basic) security mechanisms** for data **confidentiality** and **client authentication**, which mainly rely on external infrastructures or on lower layer solutions.

CLIENT AUTENTICATION

Client authentication can be performed in three ways:

- **Client IDs**: every MQTT client needs a univoque identifier.
- **Username and Password**: MQTT does not provide encryption mechanisms, need of transport layer (**TLS**) or network (**IPsec**) solutions.
- **Certificates**: provided/managed by third-party authorities.

(**TOPIC ACL**) Based on Client ID, the broker can restrict access to specific topics.

The MQTT Protocol

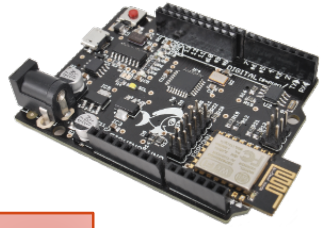
- MQTT provides **some (basic) security mechanisms** for data **confidentiality** and **client authentication**, which mainly rely on external infrastructures or on lower layer solutions.

DATA CONFIDENTIALIY

Data confidentiality can be implemented in two ways **(COMPLEMENTARY)**:

- **TSP-level Encryption:** not a part of MQTT**,** uses TLS/SSL protocol and encrypts TCP data segments→ refers only to the client → broker link.
- **APP-level Encryption**: not a part of MQTT**,** payload encryption must be provided by the application → can be useful for end-to-end security (publisher → subscriber), but does not protect the password needed for the broker access.

The MQTT Protocol: DEMO



PUBLISHER

(Fishino UNO board)

(WI-FI link)

BROKER

(MOSQUITTO)

(WI-FI link)

(Laptop)

SUBSCRIBER

MOSQUITTO → Open source MQTT (1.3/1.3.1) broker implementation
Multi-platform, Versions available for Linux/Ubuntu, MacOSX, Windows
Download at: https://mosquitto.org

```
apt-get install mosquitto mosquitto-client
```

The MQTT Protocol: DEMO

✧ MQTT client utilities: `mosquitto_pub` e `mosquitto_sub`

MESSAGE **PUBLISHING**

Topic **name**

Topic **content**

```
user@hostTest:$ mosquitto_pub -t "Temperature/Kitchen" -m "34.5"
```

MESSAGE **SUBSCRIBING**

```
user@hostTest:$ mosquitto_sub -t "Temperature/Kitchen"
34.5
```

```
user@hostTest:$ mosquitto_sub -t "#"
34.5
```

```
user@hostTest:$ mosquitto_sub -t "Temperature/+"
34.5
```

MOSQUITTO Broker: https://mosquitto.org

The MQTT Protocol: DEMO

✧ MQTT client utilities: `mosquitto_pub` e `mosquitto_sub`

MESSAGE **PUBLISHING**

Topic **name**

Topic **content**

```
user@hostTest:$ mosquitto_pub –t “Temperature/Kitchen” –m “34.5”
```

MESSAGE **SUBSCRIBING**

```
user@hostTest:$ mosquitto_sub –t “Temperature/Kitchen”
34.5
```

```
user@hostTest:$ mosquitto_sub –t “#”
34.5
```

```
user@hostTest:$ mosquitto_sub –t “Temperature/+”
34.5
```

WILDCARDS (in TOPIC name)

- ✧ + **single layer of hierarchy**
- ✧ # **all remaining levels of hierarchy** (only the final part)

- ✓ Temperature/1/2/value
- ✓ Temperature/+/+/value
- ✓ Temperature/#

The MQTT Protocol: DEMO

✧ Mosquitto configuration file: `/etc/mosquitto/mosquitto.conf`

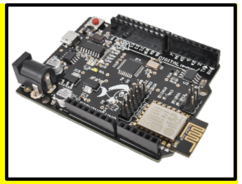
EXAMPLE, SEE [1] for the complete file format

```
#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
```

The MQTT Protocol: DEMO

✧ MQTT at **Publisher** side (Fishino UNO, but should work also on any Arduino* devices):

Complete code available at [1]

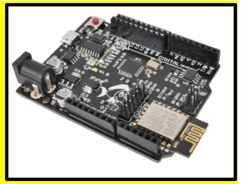


```
boolean publishData(char clientID, char* topic, char* payload) {
    boolean connected=clientMQTT.connected();
    if (!connected)
      connected=clientMQTT.connect(clientID);
    if (connected) {
        bool result=clientMQTT.publish(topic,payload);
        clientMQTT.loop();
        return result;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

The MQTT Protocol: DEMO

✧ MQTT at **Publisher** side (Fishino UNO, but should work also on any Arduino* devices):

Complete code available at [1]



```
#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**

IoT Messaging Protocols

✧ Session/Application Layer Protocols … WHICH protocols?

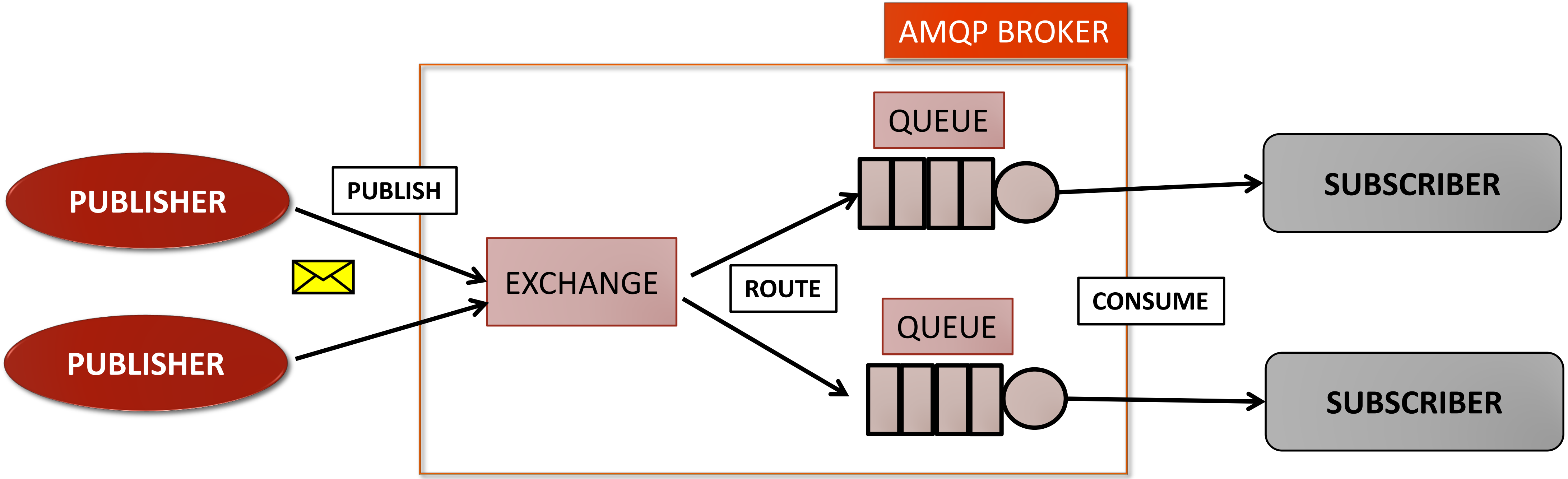
- **HTTP**
- **MQTT**
- **CoAP**
- **AMQP**
- **XMPP** (not covered here)
- **DDS** (not covered here)
- …

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.
  - Several functionalities: see [1] for a complete protocol illustration.

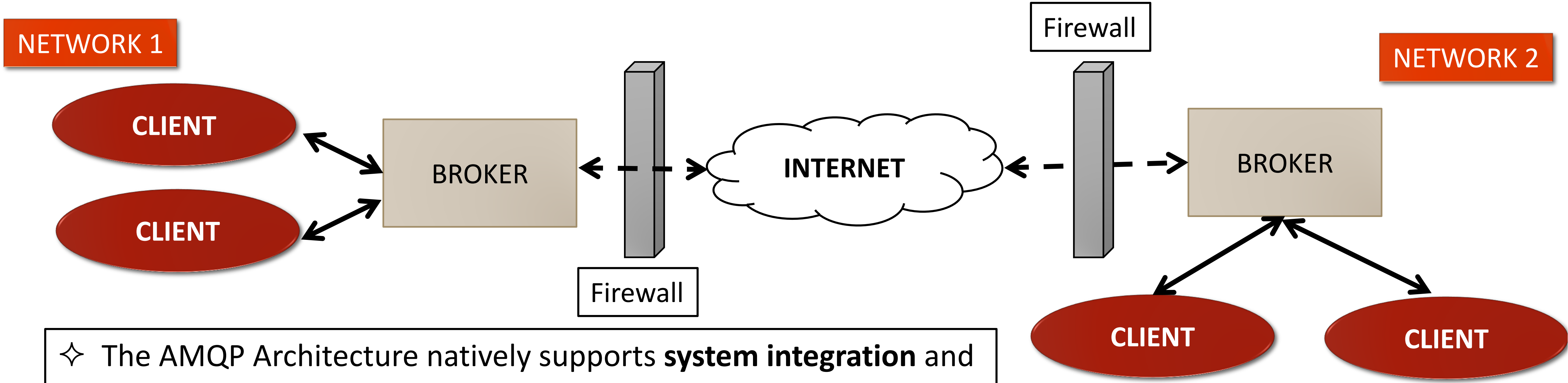
The AMQP Protocol

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



The AMQP Protocol

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



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

The AMQP Protocol

✧ 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

✧ Direct Exchange: delivers messages to queues based on the message routing key

✧ Fanout Exchange: delivers messages to all of the queues that are bound to it

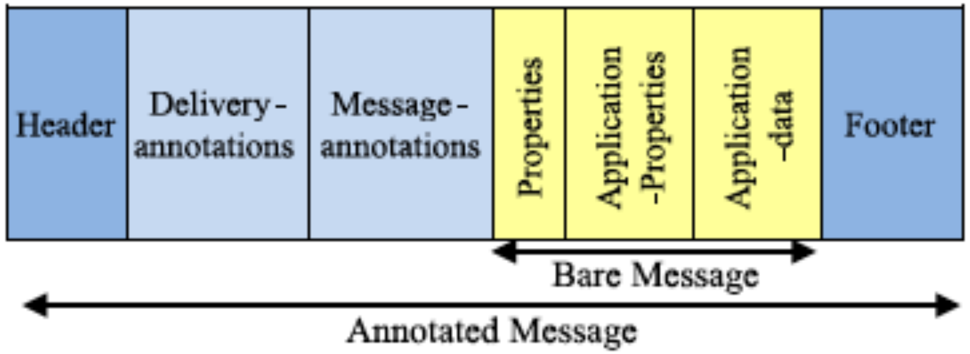
✧ Topic Exchange: delivers messages to one or many queues based on topic matching

✧ Headers exchange: delivers messages based on multiple attributes expressed as headers

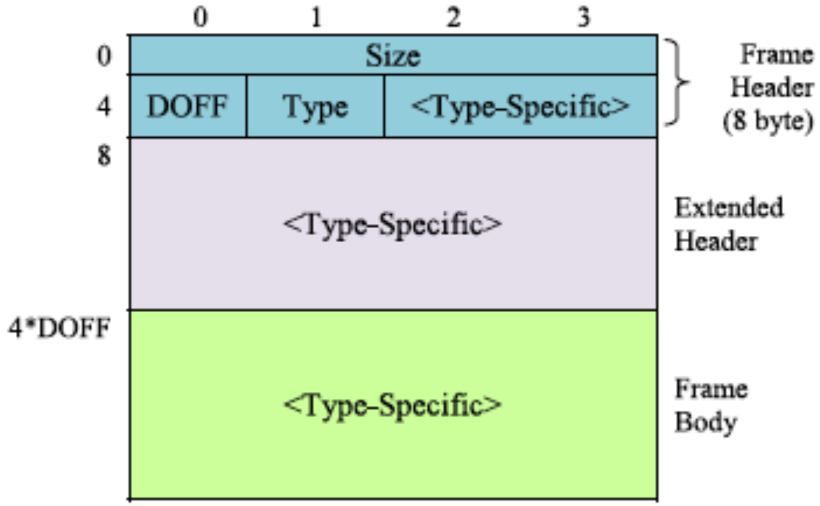
E. Ahmed, I. Yaqoob, A. Gani, M. Imrani and M. Guizani, Internet of things based smart environments: state of the art, taxonomy, and open research challenges, IEEE Wireless Communica]ons, 2016

The AMQP Protocol

- 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

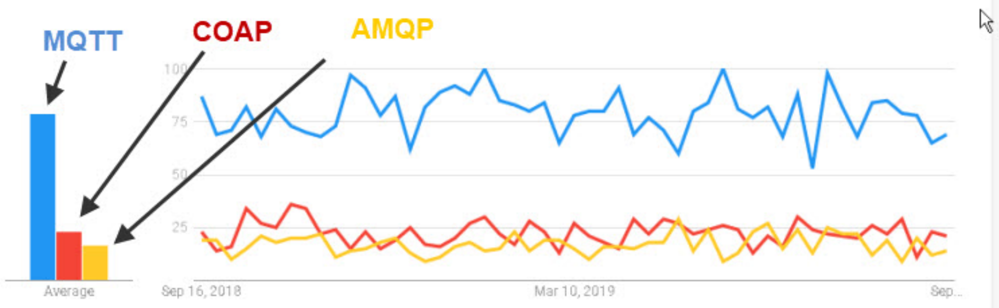


IoT Messaging Protocols

✧ Based on the interaction paradigm, IoT messaging protocols can be classified into two main categories:

❑ Publish-subscribe protocols

❑ **Request-response protocols**



Google Trends MQTT,COAP and AMQP

IoT Messaging Protocols

✧ **Session/Application** Layer Protocols … WHICH protocols?

- **HTTP**
- **MQTT**
- **CoAP**
- **AMQP**
- **XMPP** (not covered here)
- **DDS** (not covered here)
- …

Reference: REST Principles

- ❑ **Representational State Transfer (REST) →** set of architectural principles for distributed systems.

  1. Client Server → Interactions based on a request-response communication pattern.
  2. Uniform Interfaces →Unambigous standard interface for accessing the resources (e.g. the URI).
  3. Stateless → client context and state are not stored on the server.
  4. Cacheable → data are cached by clients and intermediaries.
  5. Layered System → intermediate components can hide what is behind them (e.g. content delivery networks).

Reference: REST Principles & the WEB

WEB SERVER

Manage the resources
(e.g. HTML documents)

REQUEST

RESPONSE

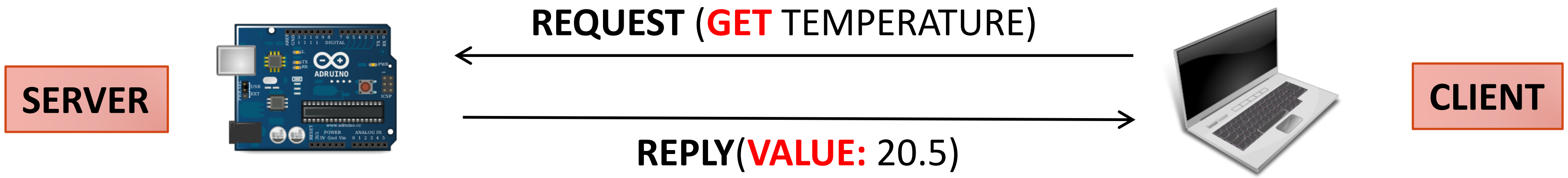
WEB CLIENT

Query the Server in order to perform operations on the resources (e.g. GET an HTML doc)

- Based on the **HTTP** (**Hypertext Transfer Protocol**) Protocol
  - ✧ Stateless, textual, request-response protocol
  - ✧ Versions: HTTP/1.1, HTTP/1.2, HTTP/2
  - ✧ Limited set of operations: GET, POST, HEAD, PUT, OPTIONS, …

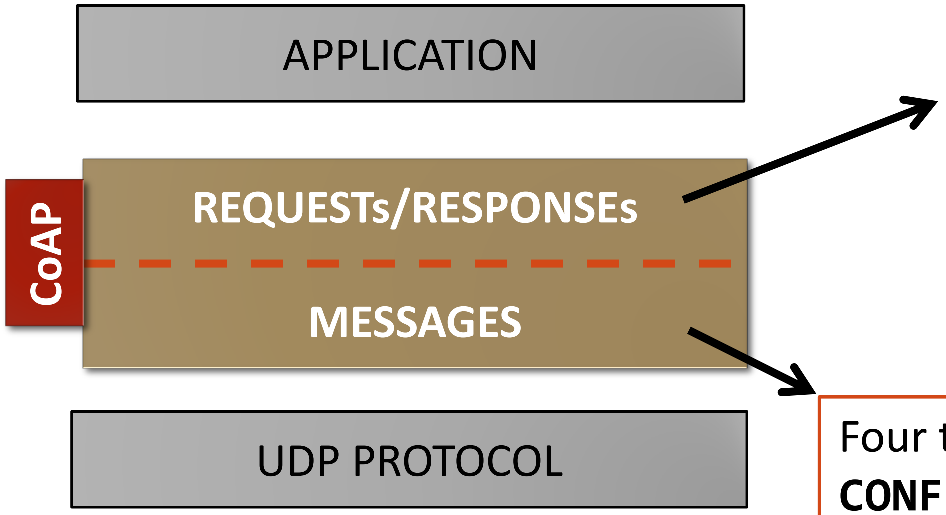
The COAP Protocol

- **Constrained Application Protocol (CoAP)**
  - Messaging protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks.
  - Differently from MQTT, CoAP implements a **request-response interaction** model (similar to the HTTP protocol).
  - **RESTful** architecture for Costrained Environments (**CoRE**).
  - Each resource is addressed by an **URI** (Uniform Resource Identifier).



The COAP Protocol

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

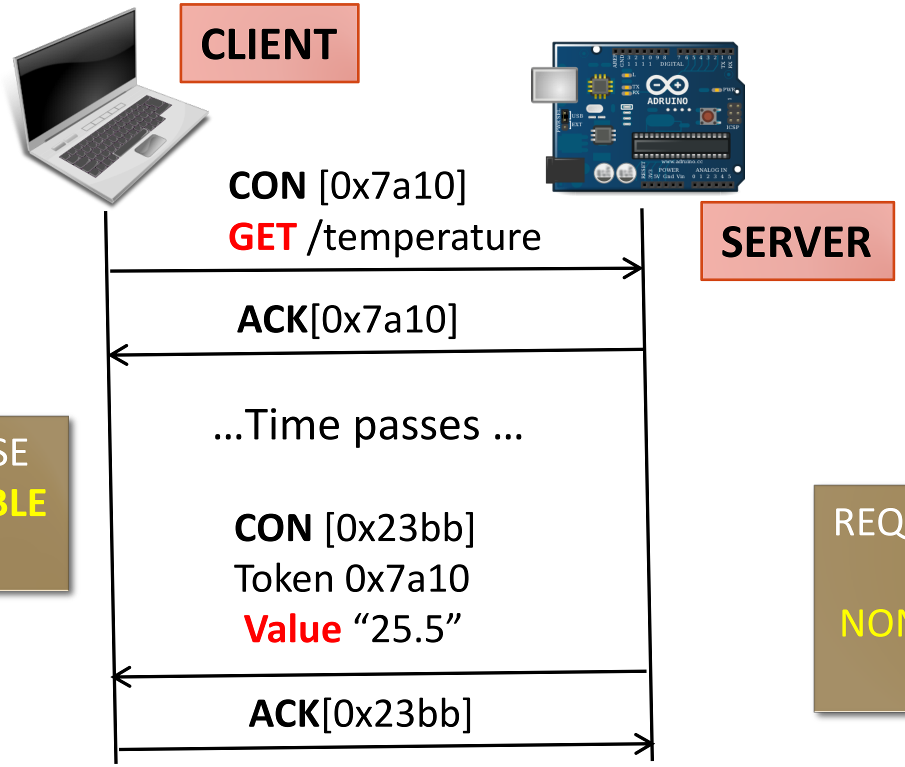


- Actions/methods **requested** on a resource:
  `GET, POST, PUT, DELETE`
- Possible **responses** to an action requested:
  `SUCCESS, CLIENT ERROR, SERVER ERROR`

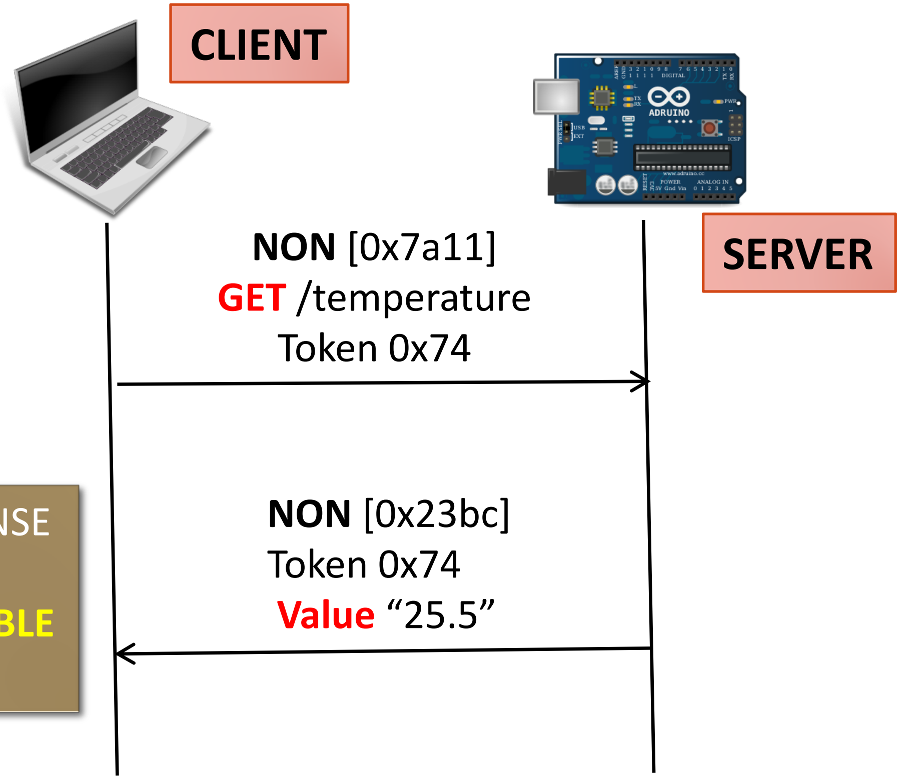
Four types of CoAP messages:
`CONFIRMABLE (CON), NON-CONFIRMABLE (NON), ACK, RESET`

The COAP Protocol

✧ Examples of CoAP message exchanges.



REQUEST/RESPONSE USING **CONFIRMABLE** MESSAGES



REQUEST/RESPONSE USING NON **CONFIRMABLE** MESSAGES

The COAP Protocol

✧ Each resource is addressed by an **URI** (Uniform Resource Identifier).

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

DIFFERENCES COMPARED TO THE HTTP PROTOCOL

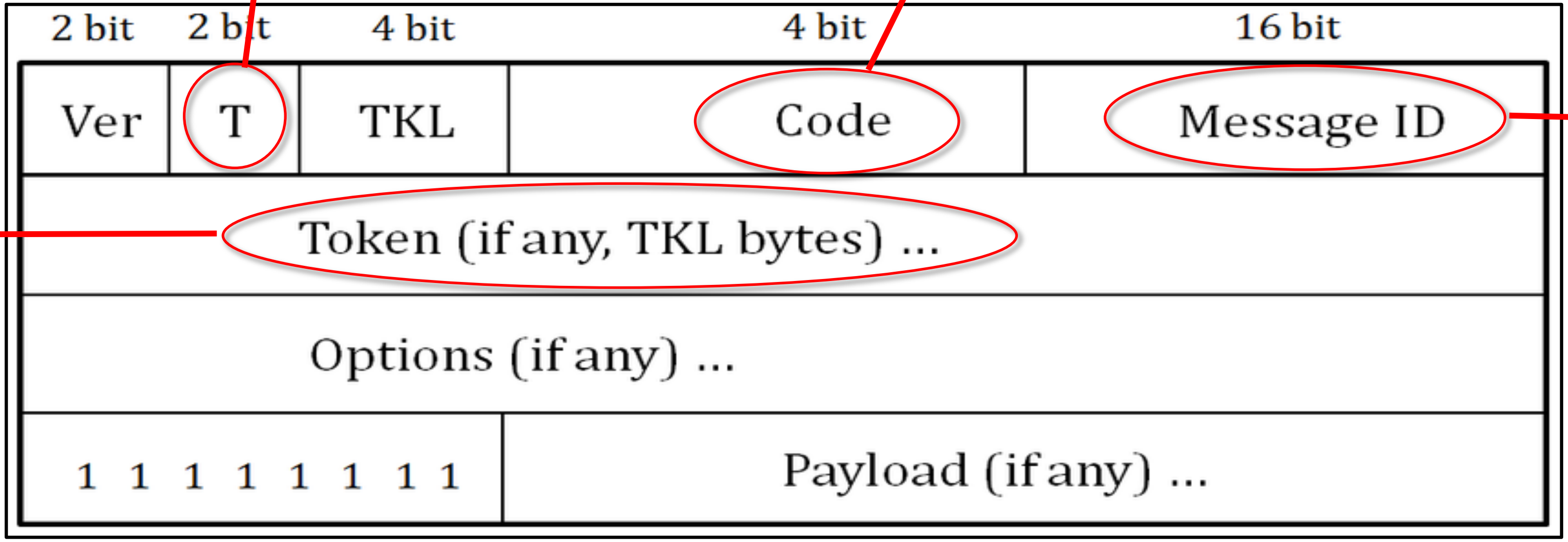
- ✧ Based on the **UDP** protocol (but optional mechanisms can be used for **enhanced reliability**, i.e. Confirmable messages + Retransmissions)
- ✧ Asynchronous Request/Response paradigm
- ✧ Different (Shorter) Packet Header (see next slide)
- ✧ Service Discovery and Proxy mechanisms

The COAP Protocol

✧ CoAP **Message Header** (fixed-size 4-byte header)

Message type: **CON**, **NON**, **ACK**, **RST**

Indicates Request method or Response code



Value used to **correlate** a Request and with the corresponding Response.

Message ID: matches CON and ACK messages

The COAP Protocol

✧ CoAP implements some lightweight reliability mechanisms:

- **Duplicate detection** for both Confirmable (CON) and Non-Confirmable (NON) messages
- Simple **stop-and-wait retransmission reliability** with **exponential back-off** for Confirmable messages

✧ The sender retransmits the Confirmable message at exponentially increasing intervals, until it receives an ACK (or RST message) or runs out of attempts.

Random Value →[ACK_TIMEOUT:ACK_TIMEOUT * ACK_RANDOM_FACTOR]

- ACK_TIMEOUT is doubled at each retransmission, till MAX_NUMBER_ATTEMPTS
- ACK_RANDOM_FACTOR is a node-specific value, used to avoid distributed synchronizations

The COAP Protocol

✧ 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

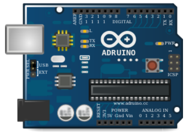
CLIENT | **SERVER**

NON [0x7a10] Observe:0 **GET** /temperature Token: 1234aa →

← **NON** [0x8b12] **Value: 34.4** Token: 1234aa

← **NON** [0x8b12] **Value: 35.5** Token: 1234aa

← **NON** [0x8b12] **Value: 26.5** Token: 1234aa

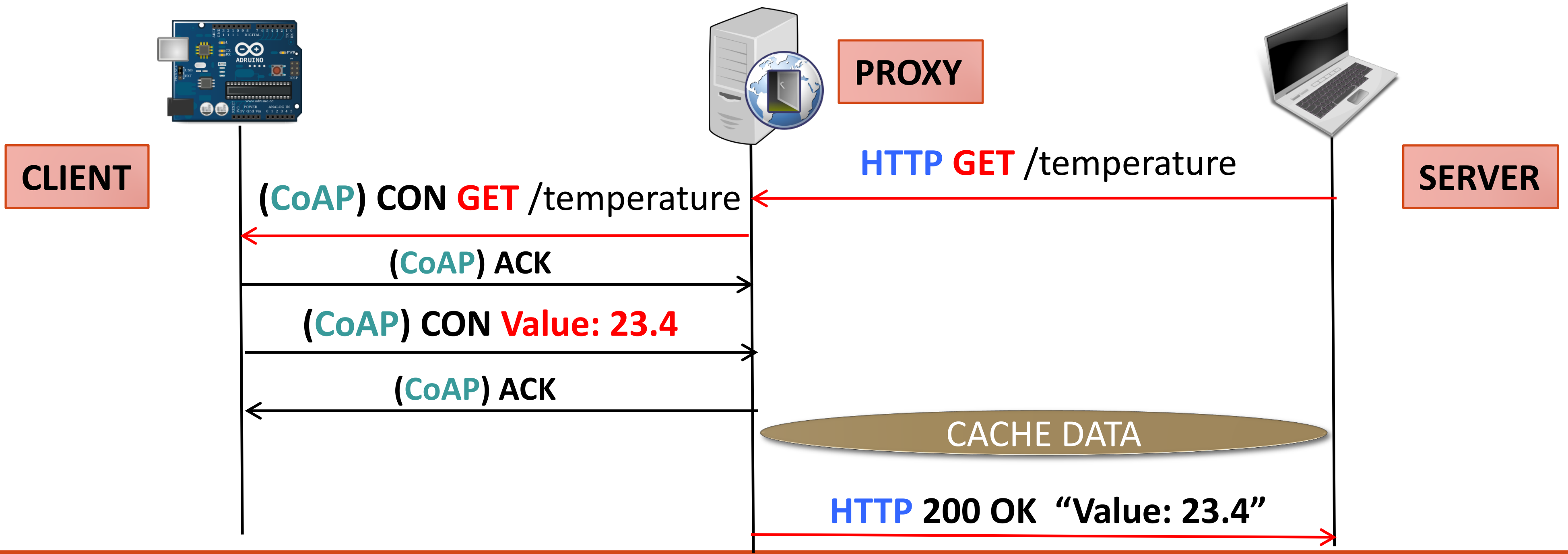


The COAP Protocol

- A **server** is used by a client knowing the URI that references a resource in the namespace of the server.
  - Alternatively, clients can use **multicast CoAP requests** (on the default port 5683) the "All CoAP Nodes" multicast address to find CoAP servers
  - Multicast requests are NOT Confirmable (i.e. no ACK messages are sent).
  - If a server does decide to respond to a multicast request, it should back-off (i.e. wait a random period before sending the reply)

The COAP Protocol

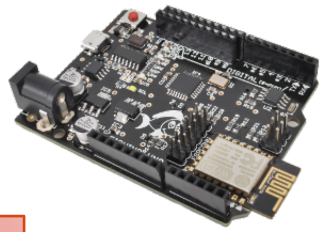
- ✧ CoAP only supports a limited subset of HTTP functionality,
- ✧ However, cross-protocol proxy mechanisms can guarantee seamless **HTTP-CoAP interactions** (beside providing data caching).



The COAP Protocol

- CoAP relies on lower-layer protocols for **securing the client-server communication**.
  - Message encryption provided at TSP Layer (**DTLS – Datagram Transport Layer Security**) or at the network Layer (**IPSec**).
  - As CoAP realizes a subset of the features in HTTP/1.1, the security considerations of HTTP are also pertinent to CoAP. In addition, CoAP presents some **unique vulnerabilities** (see [1] for details):
    1. Proxies are by their nature **man-in-the-middle**.
    2. Risk of message amplification and **DDoS attacks.**
    3. **IP spoofing** due to the lack of handshake in UDP.

The CoAP Protocol: DEMO



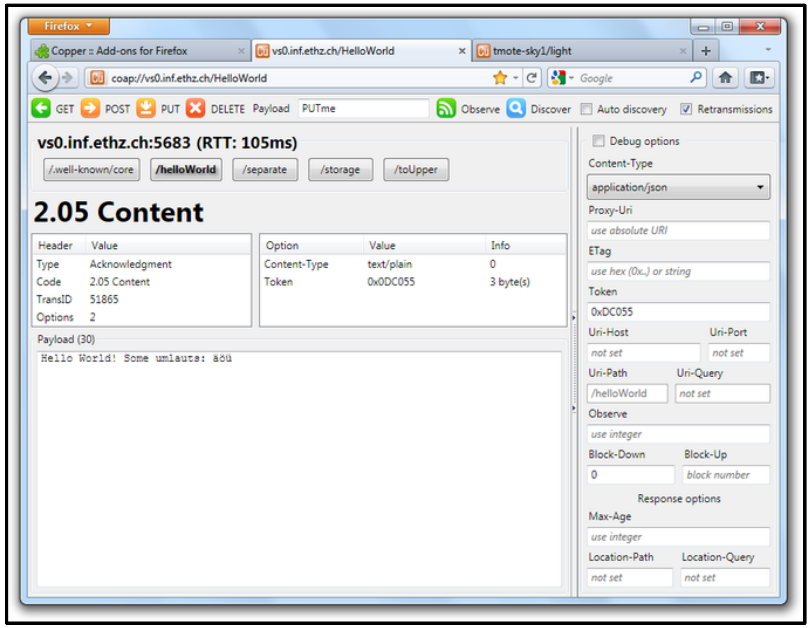
SERVER

(Fishino UNO board)

(WI-FI link)

CLIENT

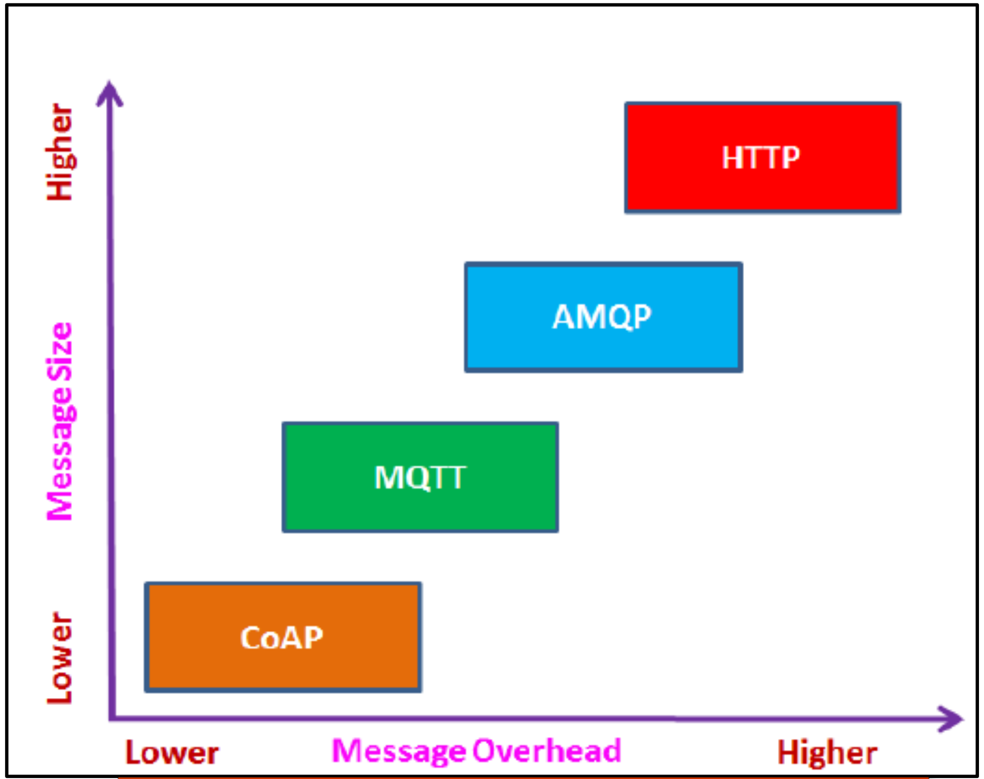
(COPPER)



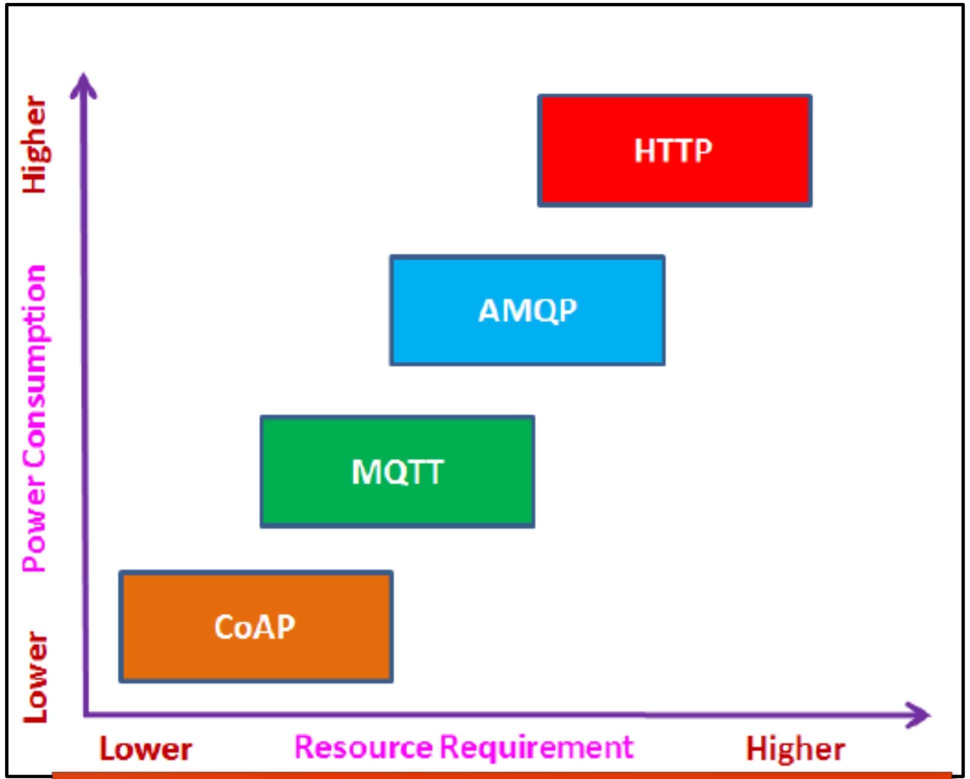
Firefox Plugin supporting CoAP URI scheme, and enabling CoAP Requests-Responses interactions via browser

https://addons.mozilla.org/en-US/firefox/addon/copper-270430/

Protocol Comparison

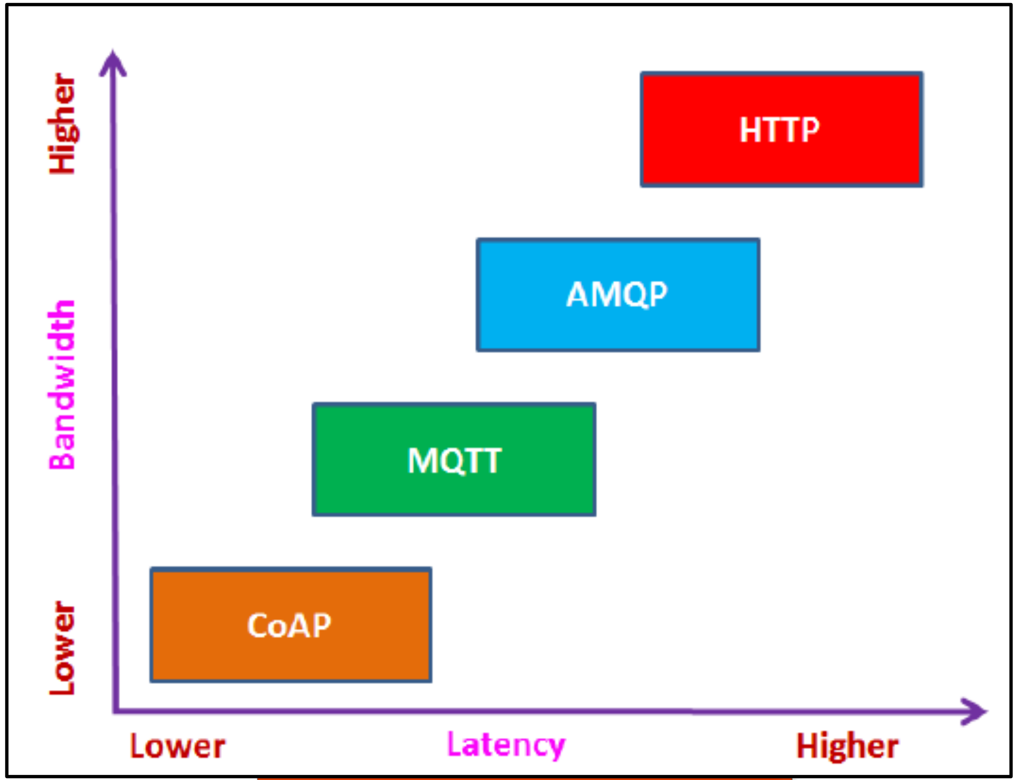


MESSAGE OVERHEAD vs MESSAGE SIZE

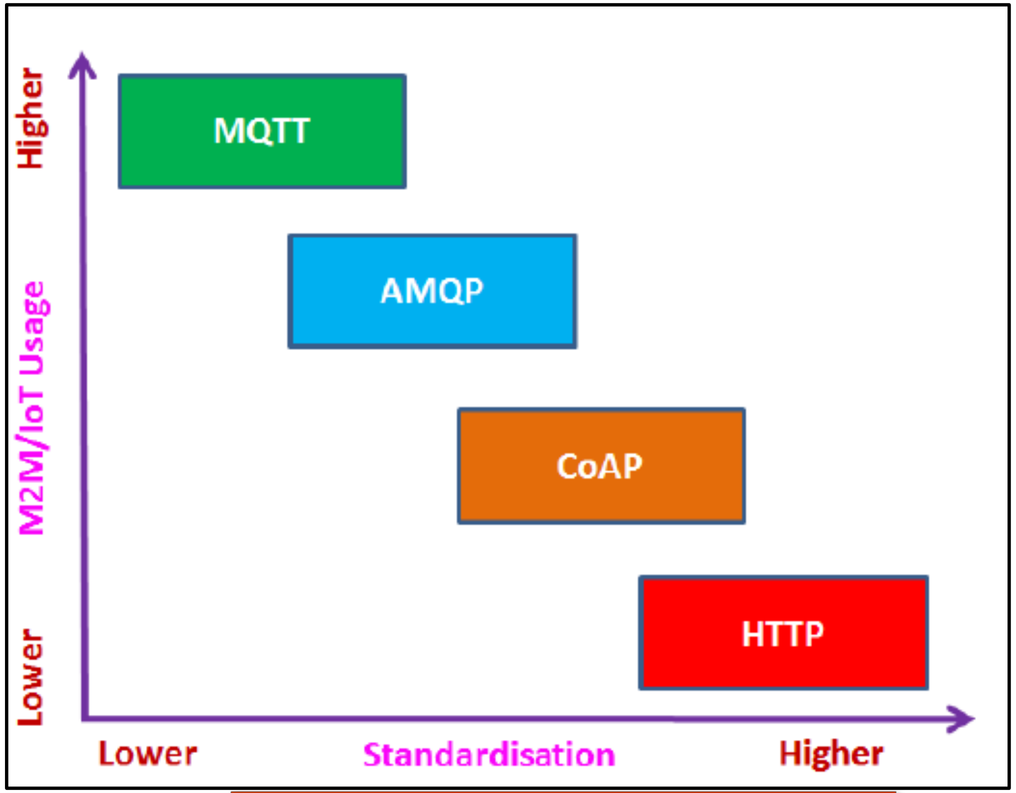


RESOURCE USED vs POWER CONSUMPTION

Protocol Comparison

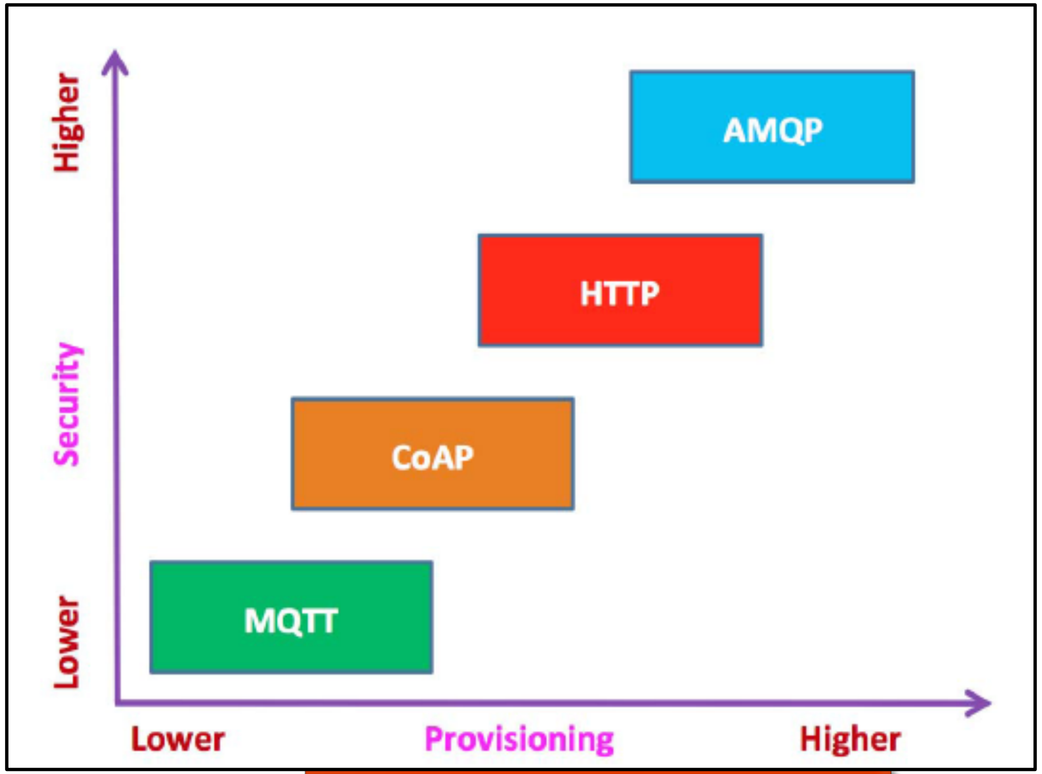


BANDWIDTH vs LATENCY

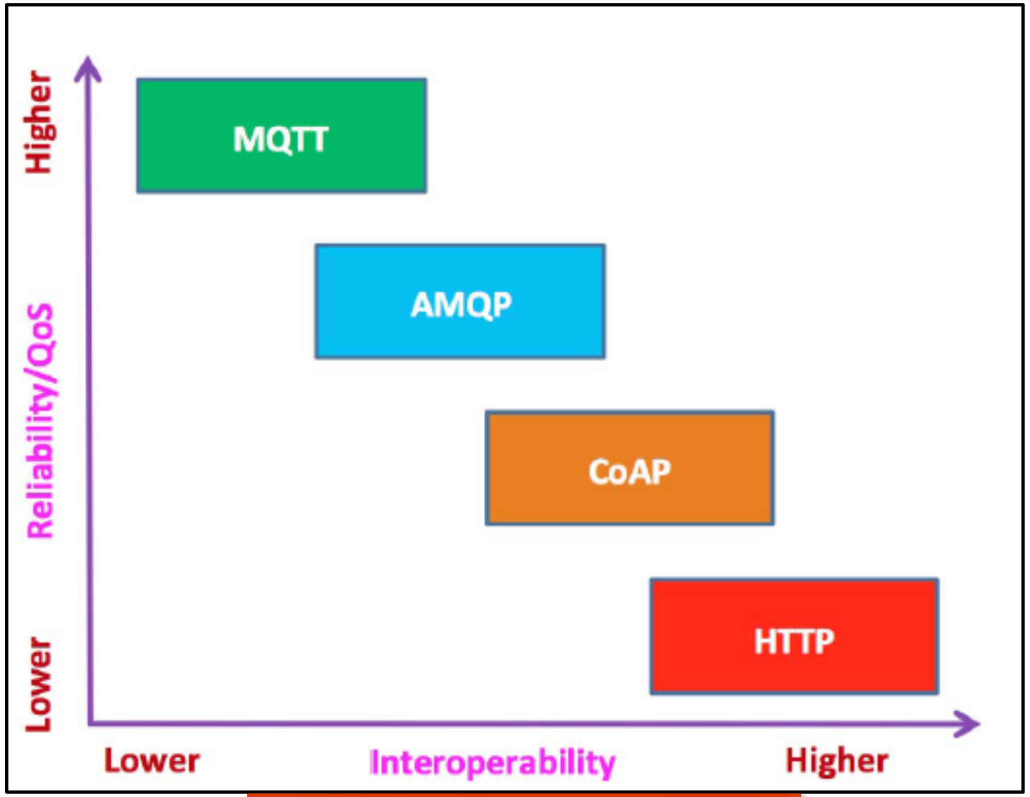


STANDADISATION vs IoT USAGE

Protocol Comparison



PROVISIONING vs SECURITY



QoS vs INTEROPERABILITY

Conclusions

- **CONCLUSIONS:**

  No one size fits all solutions

  Choose protocol/protocol paradigm based IoT on project assumptions and requirements. Some examples:

  - **Bandwidth** is the main issue → choose CoAP protocol
  - **Latency** is the main issue → choose CoAP protocol
  - **Energy** is the main issue → choose MQTT protocol
  - **QoS support** is the main issue → choose MQTT protocol
  - **Interoperability** is the main issue → choose HTTP protocol