UNIVERSITÀ DEGLI STUDI DI PADOVA

MIME
Master's degree ICT Internet Multimedia Engineering

Department of Information Engineering (DEI)
Master degree on ICT for Internet and Multimedia Engineering (MIME)

Internet of Things and Smart Cities
11 – IoT applications

Marco Giordani (marco.giordani@unipd.it)
Department of Information Engineering (DEI) – SIGNET Research Group
University of Padova – Via Gradenigo 6/B, 35131, Padova (Italy)

IoT applications

Introduction

- IoT technologies: mainly focused on PHY and MAC layers.
- Now, we focus on the upper layers: **transport** and **application** layers.
- For Internet networks: UDP and TCP.
  - UDP: connectionless, **simple** but **unreliable** service.
  - TCP: connection-oriented, guaranteed, **reliable** but **complex** service.
- UDP and TCP alone are not enough in complex scenarios.
  - Complexity: real-time requirements, large scale scenarios, etc.
- We need to **complement** UDP and TCP with other proptocols **at the application**.
- **Middleware**: application protocols and components which must be deployed to realize the full support of applications.

IoT applications

Motivations

Why middleware for IoT?

- Heterogeneous types of data (notifications, measurements, commands, etc.).
- Both (hard-)real-time and non-real-time traffic.
- Both client-server and publish-subscribe models.
- Constraints of enpoints (complexity, powering, bandwidth, etc.).
- Large-scale cyber-physical system.
- Complexity of storage.

No single solution exists satisfying the requirements of all IoT networking scenarios.

UNIVERSITÀ DEGLI STUDI DI PADOVA

Master's degree ICT Internet Multimedia Engineering

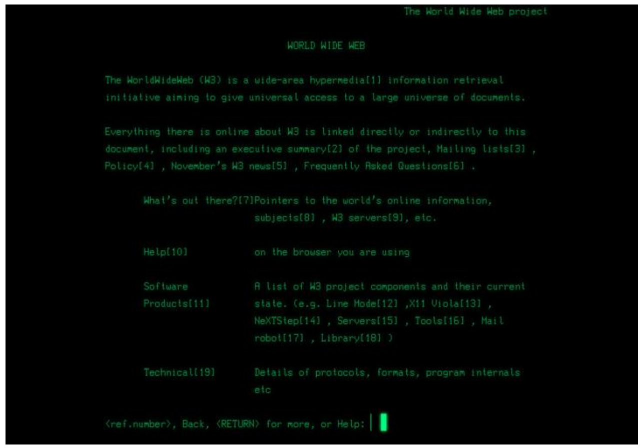
11 – IoT applications
HTTP / WebSocket

Marco Giordani (marco.giordani@unipd.it)
Department of Information Engineering (DEI) – SIGNET Research Group
University of Padova – Via Gradenigo 6/B, 35131, Padova (Italy)

HTTP

World Wide Web (WWW)

- The idea of the Web was proposed by Tim Berners-Lee in 1989 at CERN, to allow several researchers at different locations in Europe to access each others' research.
- The Web today is a repository of information in which the documents (**web pages**), are distributed all over the world and documents are linked together.
    - The linking of web pages was achieved using **hypertext**.
- The WWW today is a **distributed client-server service**, in which a client using a browser can access a service using a server.
    - The service is distributed over many locations called **sites**.
    - Each site holds one or more web pages.



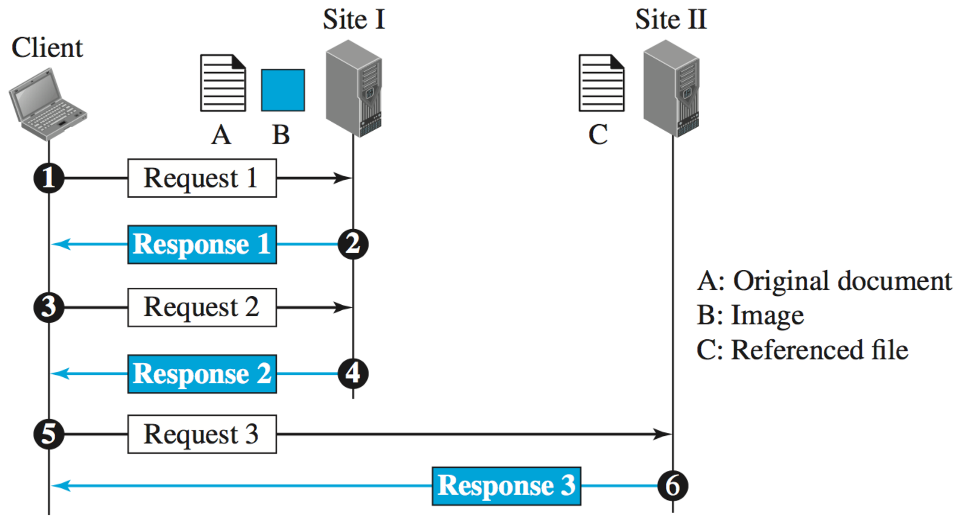
```
                                                  The World Wide Web project

                             WORLD WIDE WEB

The WorldWideWeb (W3) is a wide-area hypermedia[1] information retrieval
initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this
document, including an executive summary[2] of the project, Mailing lists[3] ,
Policy[4] , November's W3 news[5] , Frequently Asked Questions[6] .

     What's out there?[7]Pointers to the world's online information,
                    subjects[8] , W3 servers[9], etc.

     Help[10]       on the browser you are using

     Software       A list of W3 project components and their current
     Products[11]   state. (e.g. Line Mode[12] ,X11 Viola[13] ,
                    NeXTStep[14] , Servers[15] , Tools[16] , Mail
                    robot[17] , Library[18] )

     Technical[19]  Details of protocols, formats, program internals
                    etc

<ref.number>, Back, <RETURN> for more, or Help: |
```

Screenshot of the recreated page of the first website.
http://info.cern.ch

HTTP

World Wide Web (WWW): example

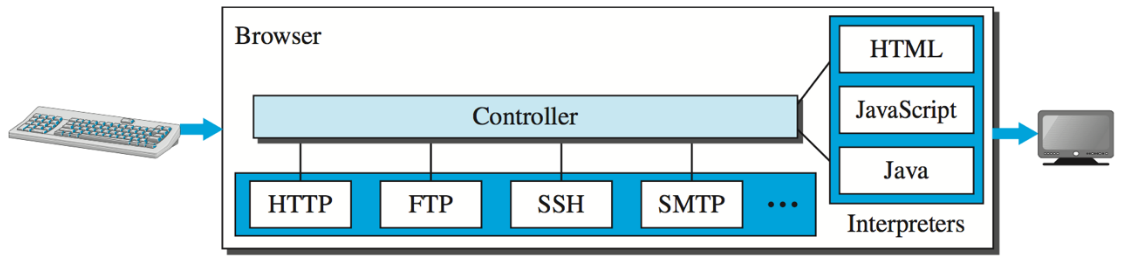
- Assume we need to retrieve a document (file A) that contains one reference to another text (file C) and one reference to a large image (file B). File A and file B are in the same site; file C is stored in another site.
  - Since we are dealing with 3 different files, we need 3 transactions to see the document.



HTTP

Web client and server

- A variety of vendors offer commercial **browsers** that interpret and display a web page, and all of them use nearly the same architecture. Each browser usually consists of three parts: a controller, client protocols, and interpreters.
- The web page is stored at the **server**. Each time a request arrives, the corresponding document is sent to the **client**.
  - To improve efficiency, servers normally store requested files in a **cache** in memory.
  - A server can also become efficient through **multithreading** or **multiprocessing**. In this case, a server can answer more than one request at a time.



HTTP

Uniform Resource Locator (URL)

- Web page has a unique **identifier** to distinguish it from other web pages.
  - We need three identifiers: host, port, and path.
- However, we also need to tell the browser what client-server application we want to use to get to the path, i.e., the protocol.
  - We need four identifiers to define the web page: host, port, path, protocol.
- The **uniform resource locator (URL)** combines these four pieces together:
  - protocol://host/path → Used most of the time.
  - protocol://host:port/path → Used when port number is needed.
  - Example: http://www.dei.unipd.it/en/department/map-department

HTTP

Uniform Resource Locator (URL)

```
protocol://host:port/path
```

- **Protocol**: Abbreviation for the client-server program that we need in order to access the web page, usually **HTTP (HyperText Transfer Protocol)**.
- **Host**: It can be the **IP address of the server**, or the unique name given to the server (the name is normally the domain name that uniquely defines the host).
- **Port**: A 16-bit integer, it is normally predefined for the client-server application. (e.g., HTTP uses port 80). However, the number can be explicitly given.
- **Path**: The path identifies the location and the name of the file in the underlying operating system. The format of this identifier normally depends on the operating system. In UNIX, a path is a set of directory names followed by the file name, all separated by a slash (e.g., /top/next/last/myfile).

HTTP

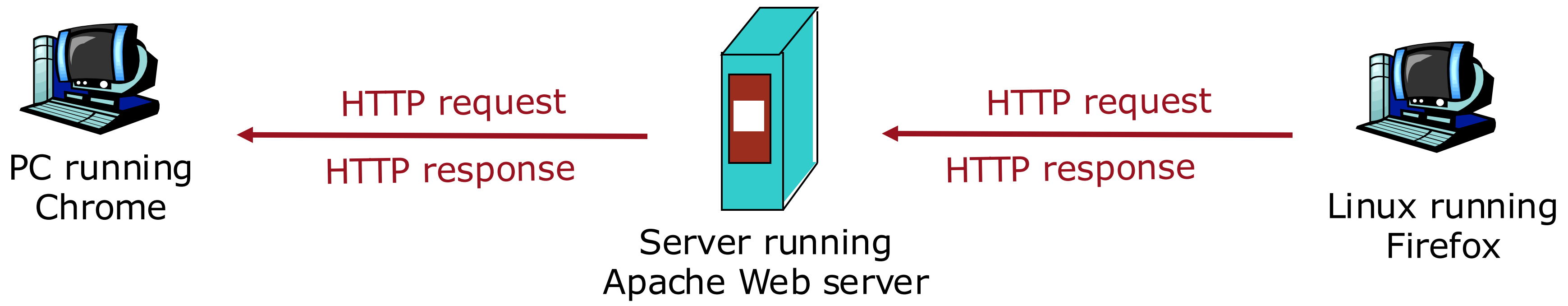
Overview of HTTP

- The **HyperText Transfer Protocol (HTTP)** is used to define how the client-server programs can be written to retrieve web pages from the Web. An HTTP client sends a **request**; an HTTP server returns a **response**.
  - Server uses port number 80; client uses a temporary port number.
  - HTTP uses the services of TCP (more recently, some applications use **QUIC**).
    - Connection establishment and termination phases are needed.
    - Client/server do not need to worry about errors in messages exchanged or loss of any message, because the TCP is reliable and will take care of this matter.
  - If Transport Layer Security is added on top of TCP: **HTTPS** (HTTP Secure).

HTTP

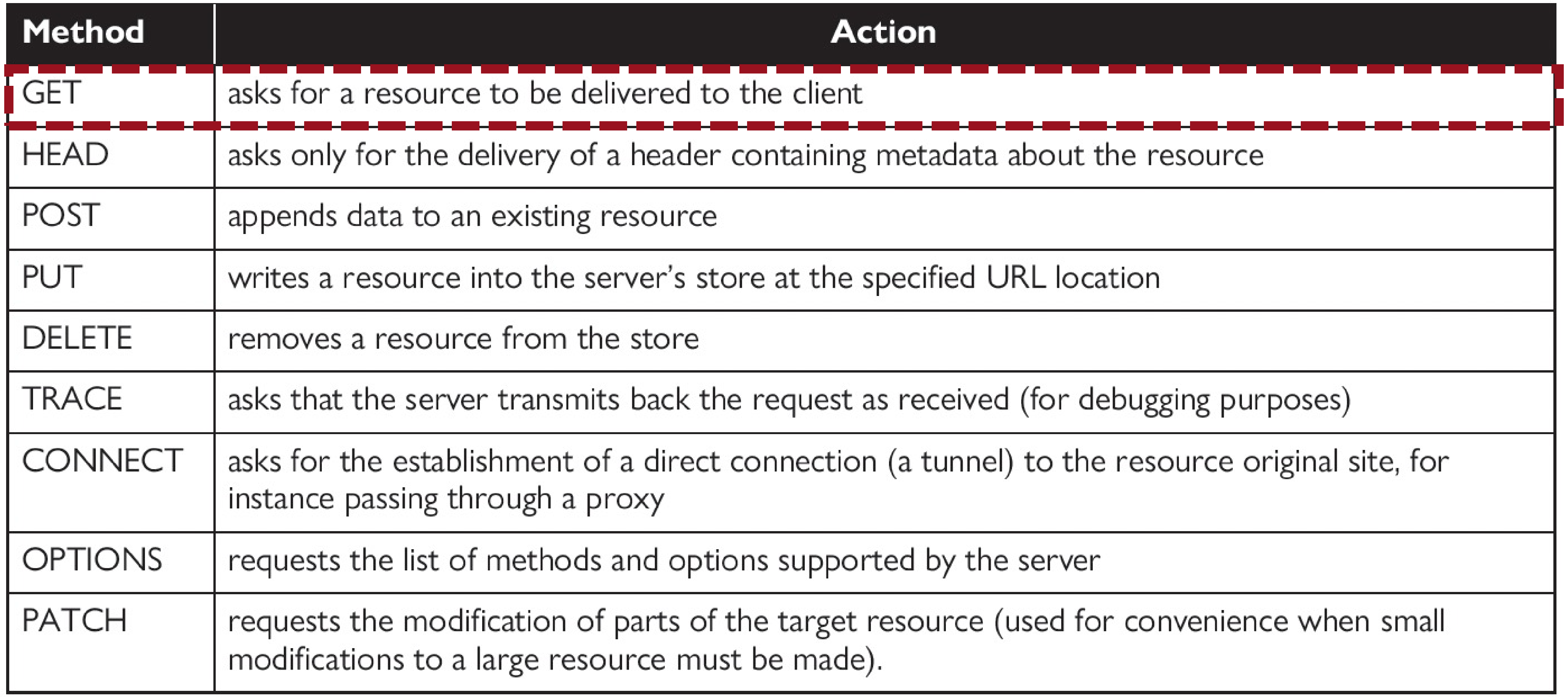
Overview of HTTP

- The client issues **requests** (called **methods**) towards a server hosting the objects.
- Depending on the method, the server's **response** may contain the resource data or the metadata only.
- Requests and responses may contain a large number of parameters, managing the many options available in HPPT.



HTTP

HTTP methods



Method	Action
GET	asks for a resource to be delivered to the client
HEAD	asks only for the delivery of a header containing metadata about the resource
POST	appends data to an existing resource
PUT	writes a resource into the server's store at the specified URL location
DELETE	removes a resource from the store
TRACE	asks that the server transmits back the request as received (for debugging purposes)
CONNECT	asks for the establishment of a direct connection (a tunnel) to the resource original site, for instance passing through a proxy
OPTIONS	requests the list of methods and options supported by the server
PATCH	requests the modification of parts of the target resource (used for convenience when small modifications to a large resource must be made).

HTTP

Problems for IoT traffic

- HTTP is a very **generic** protocol, which is used for many different application. In turn, IoT traffic has very specific characteristics and constraints.
- HTTP is a strict **client-server model:** it can’t accommodate event-based updates.
  - A client issues a request, and the server gives a response, but cannot receive unsolicited updates (e.g., when the state of the system changes), as in the IoT scenario.
  - Connection always starts from the client, so it cannot support **asynchronous** events.
- HTTP header is textual (using UTF-8 encoding), so it is “**verbose**” (size ~100 KB).
- HTTP implements **sophisticated options**, which may not be used for IoT traffic.
- <u>Some solutions</u>: HTTP/2, HTTP/3.
  - Still, “normal” HTTP does not seem to be the ideal solution for IoT gateway-based nets.

WebSocket

Overview

- For IoT traffic, **WebSocket** is defined as the combination of two elements:
  1. A mechanism to open a bidirectional TCP connection for data exchange using HTTP;
  2. A minimal set of messages which allow data transfer and maintainance of the connection.

- To open the connection, the client uses an HTTP **GET** request.
- The server agrees by sending a response: WebSocket communication can start.
  - If the server does not support WebSocket, it responds with an error code.
- WebSocket connection is **symmetrical**, so it can support unsolicited event transmissions from both sides.
- Limitation: It is still a client-server model.
  - There are more effective (tailored) solutions for IoT → **publish-subscribe** paradigm.

WebSocket

Frame structure

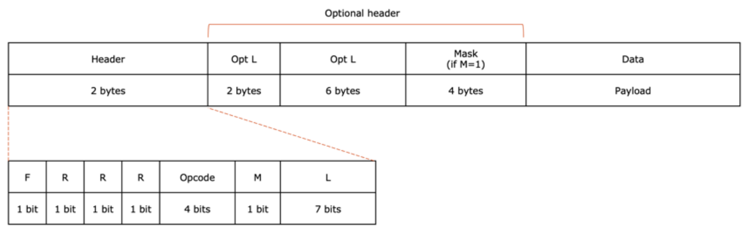
- WebSocket messages are characterized by a **minimal header** for efficiency.

Optional header

Header	Opt L	Opt L	Mask (if M=1)	Data
2 bytes	2 bytes	6 bytes	4 bytes	Payload

F	R	R	R	Opcode	M	L
1 bit	1 bit	1 bit	1 bit	4 bits	1 bit	7 bits

WebSocket



Frame structure

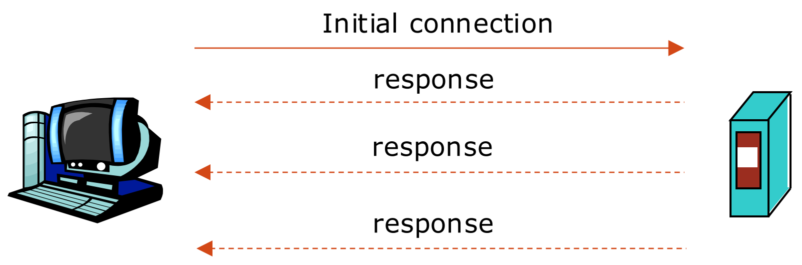
- **F** (FIN): If 1, it indicates that this is a complete message or the last fragment.
- **R**: Reserved.
- **Opcode**: it indicates the type of message.
- **M**: If 1, a “mask” is included in the payload.
- **L** (Length): it indicates the length of the message.
  - If L < 126 → Payload length.
  - If L = 126 → The following 2 bytes (Opt L) are the payload length (up to 65KB).
  - If L = 127 → The following 8 bytes (Opt L + Opt L) are the payload length (up to 9.22 EB).
- **Mask**: Mandatory only for client-to-server communications. It is used to “mark” packets with a key to avoid poisoning attacks.

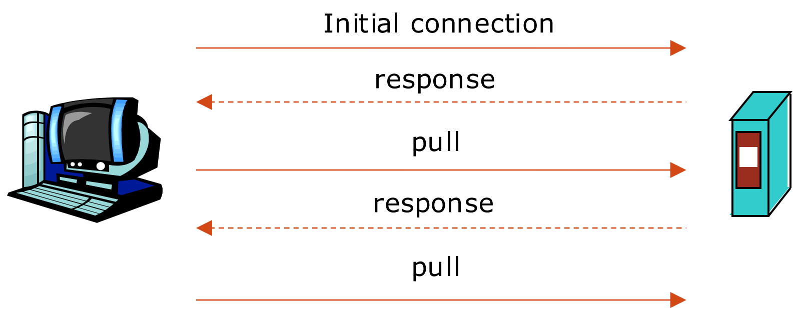
HTTP vs. WebSocket

Comparison

Feature	WebSocket	HTTP
Connection Type	Full-duplex, persistent connection.	Half-duplex, request-response model.
Communication Direction	Bidirectional	Unidirectional
Overhead	Minimal after the initial handshake	High for repeated requests
Efficiency	High	Moderate

https://sendbird.com/developer/tutorials/websocket-vs-http-communication-protocols





UNIVERSITÀ DEGLI STUDI DI PADOVA

Master's degree ICT Internet Multimedia Engineering

11 – IoT applications
MQTT

Marco Giordani (marco.giordani@unipd.it)
Department of Information Engineering (DEI) – SIGNET Research Group
University of Padova – Via Gradenigo 6/B, 35131, Padova (Italy)

MQTT

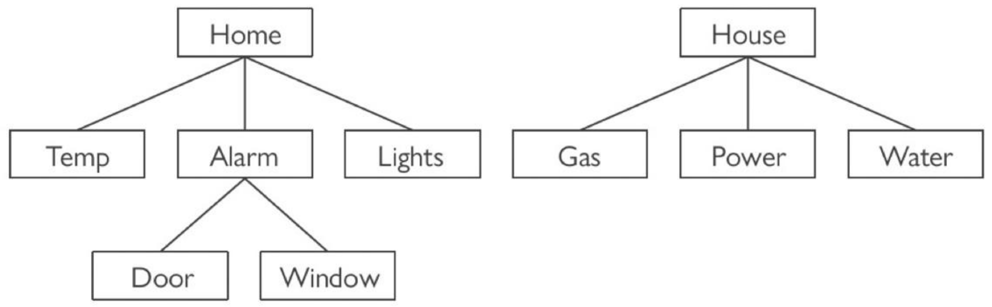
Overview

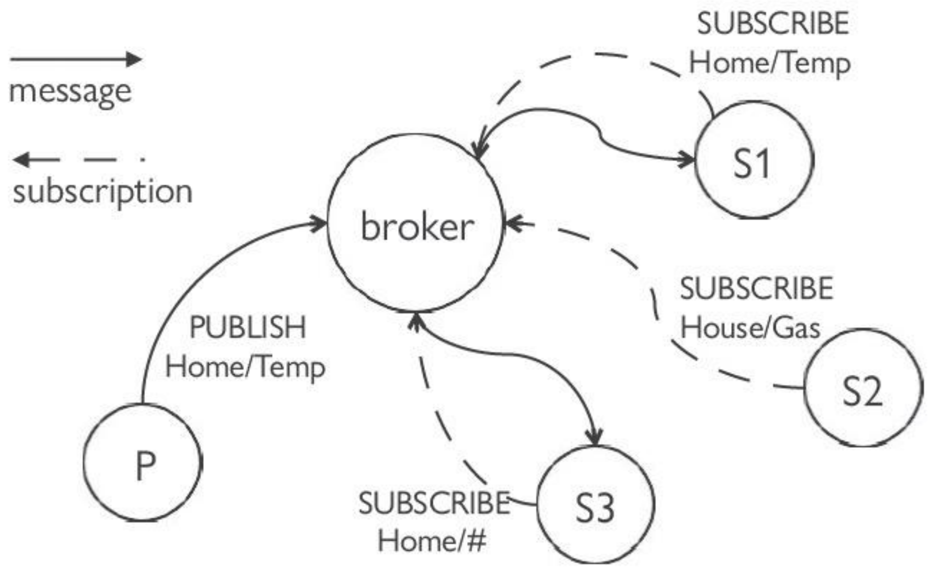
- **Message Queuing Telemetry Transport (MQTT)** is an example of an IoT-specific application protocol.
  - It was specifically designed for the collection of events from sensors.
- It is normally layered on top of TCP or TLS.
- It is implemented based on the **publish-subscribe** paradigm (vs. HTTP).
  - A central **publisher** distributes messages to a **broker**.
  - The broker fowards these **messages** to the **subscribers**.
  - Subscribers never create direct connections among them.

MQTT

Architecture

- **Publish-subscribe** paradigm
  - A central **publisher** distributes messages to a **broker**.
  - The broker fowards these **messages** to the **subscribers**.
  - Subscribers never create direct connections among them.
  - Subscription to **topics**.





MQTT

Frame structure

- MQTT messages are characterized by a **minimal header** for efficiency.
  - **Remaining length**: It represents the sum of the lengths of the Variable header and Payload fields. It is encoded using a data type called **Variable Byte Integer**, which employs a number of bytes (1÷4) depending on the represented integer.
  - **Variable header / Payload**: it depends on the type of message.

Fixed header **(even though the length is not fixed)**

Packet type	Control flags	Remaining length	Variable header (optional)	Payload
4 bits	4 bits	1-4 bytes	N bytes	M bytes

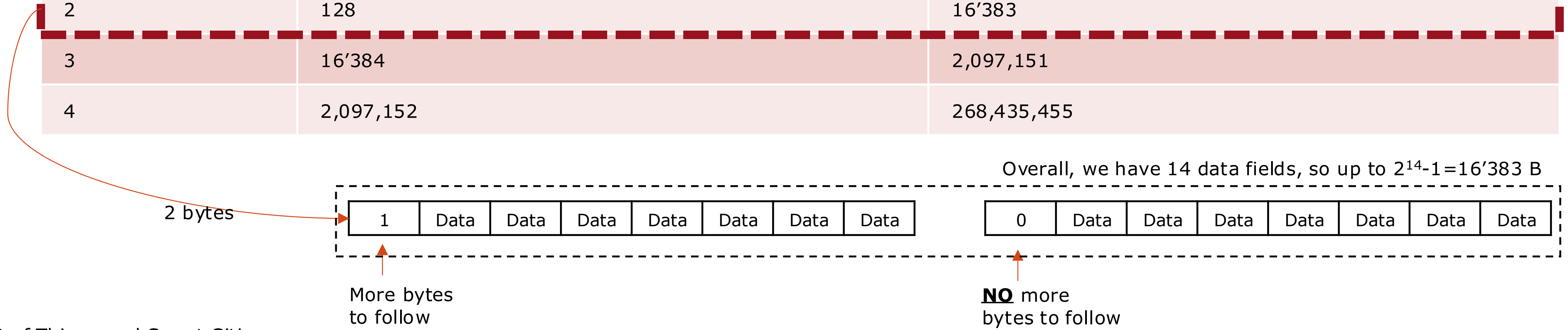
see next slide

MQTT

Frame structure

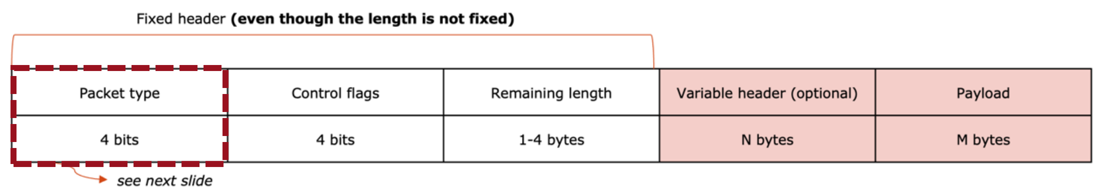
- **Variable Byte Integer:** utilize the lower 7 bits of each byte to encode data, while the highest bit indicates whether there are more bytes to follow.
  - Packet length <128 bytes: we only need 1 byte.
  - Maximum length (4 bytes): up to (2^{28} - 1) bytes, i.e., 256 MB of data.

Digits	From	To
1	0	127
2	128	16’383
3	16’384	2,097,151
4	2,097,152	268,435,455


Overall, we have 14 data fields, so up to 2^{14}-1=16’383 B
2 bytes
1
Data
Data
Data
Data
Data
Data
Data
0
Data
Data
Data
Data
Data
Data
Data
More bytes to follow
NO more bytes to follow

MQTT

Frame structure: packet type



The set of MQTT messages.

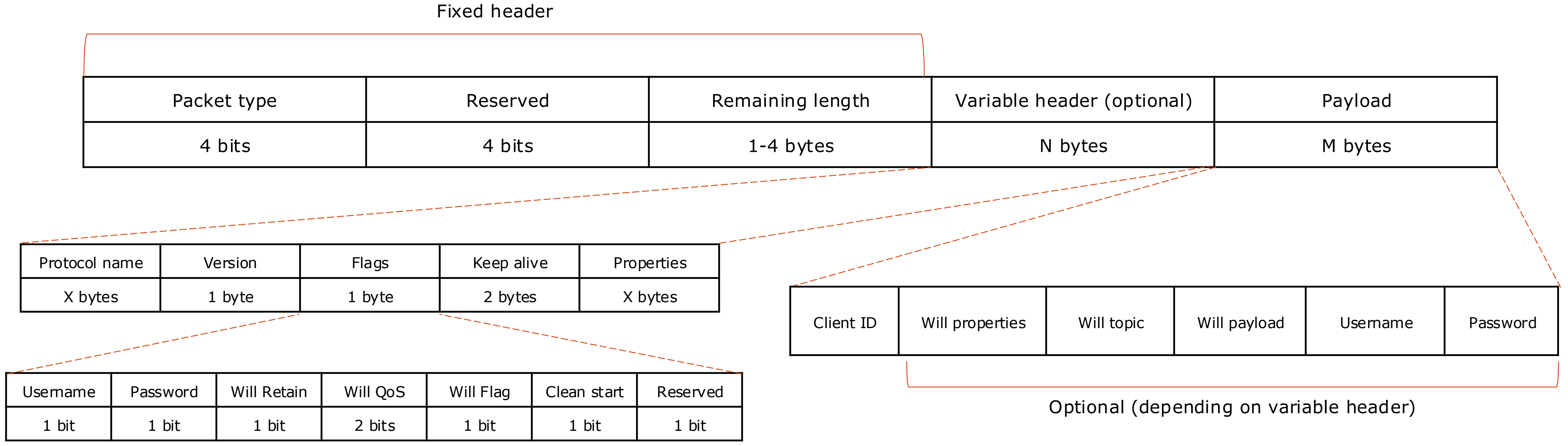
- We will focus on:
  - CONNECT;
  - CONNACK;
  - SUBSCRIBE;
  - PUBLISH.

Group	Message	Meaning
Connection Management	CONNECT	Connection open
	CONNACK	Connection acknowledgment
	AUTH	Authentication message
	PINGREQ	Keepalive
	PINGRESP	Keepalive response
	DISCONNECT	Connection close
Subscriptions	SUBSCRIBE	Subscribe to a subject
	SUBACK	Subscription acknowledgment
	UNSUBSCRIBE	Remove a subscriprion
	UNSUBACK	Removal acknowledgment
Publishing	PUBLISH	Message with topic and data
	PUBACK	Message acknowledgment (only for QoS=1)
	PUBREC	Message received (only for QoS=2)
	PUBREL	Release pending publish (only for QoS=2)
	PUBCOMP	Release acknowledgment

MQTT

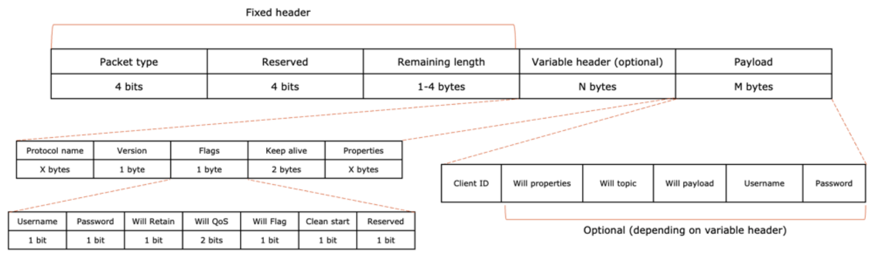
CONNECT

- It deals with connection **establishment**, **control**, and **removal**.
- **CONNECT**: open/reopens a session with the broker and carries the ID of the client.



MQTT

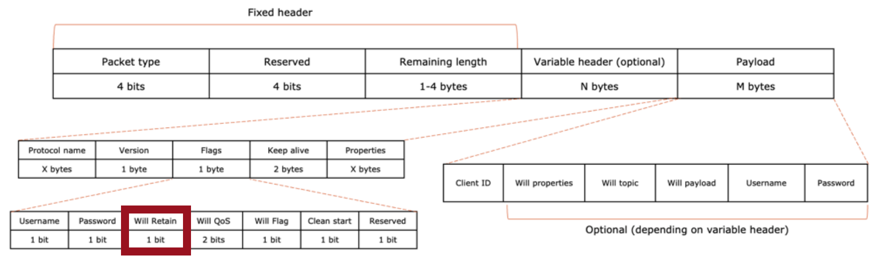
CONNECT



- **Username**: Used to indicate whether the Payload contains the Username.
- **Password**: Used to indicate whether the Payload contains the Password.
- **Will Retain**: Used to indicate if the Will Message is a Retained Message (we'll see).
- **Will QoS**: Used to indicate the QoS of the Will Message (we'll see).
- **Will Flag**: Used to indicate if the Payload contains fields of the Will Message.
- **Clean Start**: Used to indicate whether the current connection is a new session or a continuation of an existing session, which determines whether the server will directly create a new session or attempt to reuse an existing session.
- **Reserved** (for future use).
- **Keep Alive**: Used to indicate the maximum time interval (in seconds) between two adjacent control packets sent by the client.

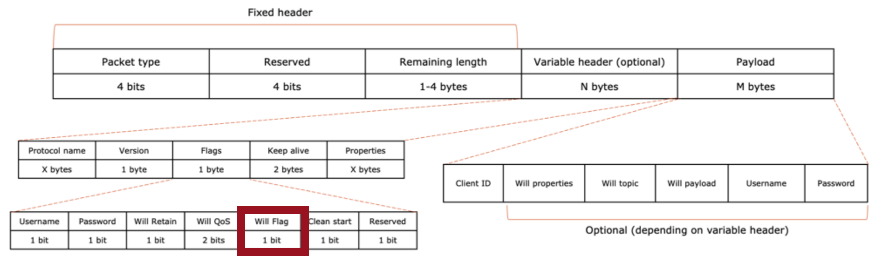
MQTT

Retained message



- MQTT has the disadvantage that subscribers cannot actively fetch messages from publishers (similar to HTTP client-server paradigm).
- New subscribers can get the latest data immediately without waiting using **Retained Messages**.
  - Upon receiving a message with the **Retain** flag set, the MQTT broker must store the message for the topic to which the message was published (store only the last message).
  - Subscribers of that topic can go offline, and **reconnect at any time to receive the latest message** instead of having to wait for the next message after the subscription.
  - Example: Sensor *monitoring a value that rarely changes*. The typical implementation is regularly publishing the latest value, but a better implementation is sending it only when the value changes in the form of a Retained Message. This allows *any new subscriber to get the current value immediately, without waiting for the sensor to publish again*.

MQTT

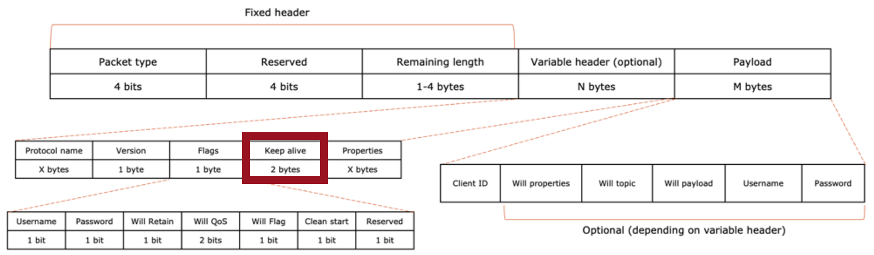


Last Will and Testament (LTW)

- **Will Messages** (if enabled) will be published if the connection terminates "ungrecefully," e.g., without a DISCONNECT or in case of protocol errors.
- When the client unexpectedly disconnects, the server sends a Will Message to other clients who have subscribed to the corresponding topic.
- Particularly useful in real-world IoT systems, where connectivity may be lost.
- Once the server publishes the Will Message, it will be removed from the session.
  - If the client who cares about this Will Message is offline, it will miss this Will Message.
  - To avoid this situation, we can set the Will Message **as a Retained Message**, so that after the Will Message is published, it will still be stored on the server in the form of a Retained Message, and the client can get this Will Message at any time.

MQTT

Keep Alive

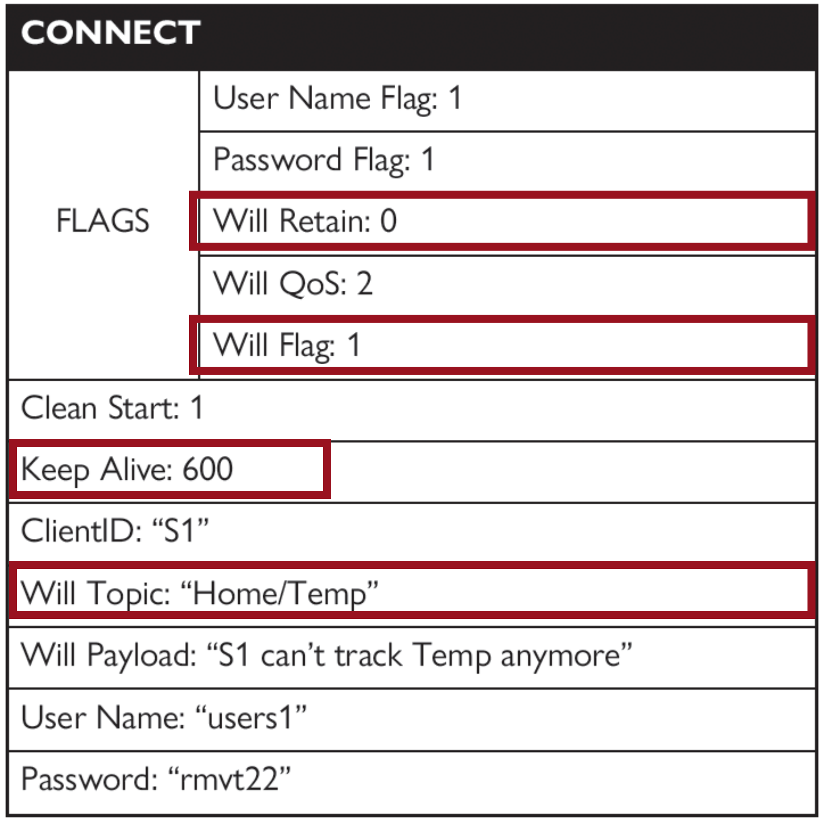


- MQTT protocol is based on TCP, which can have **half-connection** problems.
  - Half-connection: connection is disabled on one side, and active on the other side.
  - The half-connected party may send data, which never reach the other side.
- **Keep Alive** allows the client and MQTT server to determine whether there is a half-connection problem, and close the corresponding connection.
  - Integer from 0 to 65535 (2 bytes).
  - It represents the **maximum time** (in seconds) **allowed to elapse between MQTT pkts**.
  - The client needs to ensure that the interval between any two MQTT protocol packets it sends does not exceed the Keep Alive value.
  - Keep Alive is typically used in conjunction with Will Message: if the server does not receive any packets within the Keep Alive timer, it will send a **Will Message**.

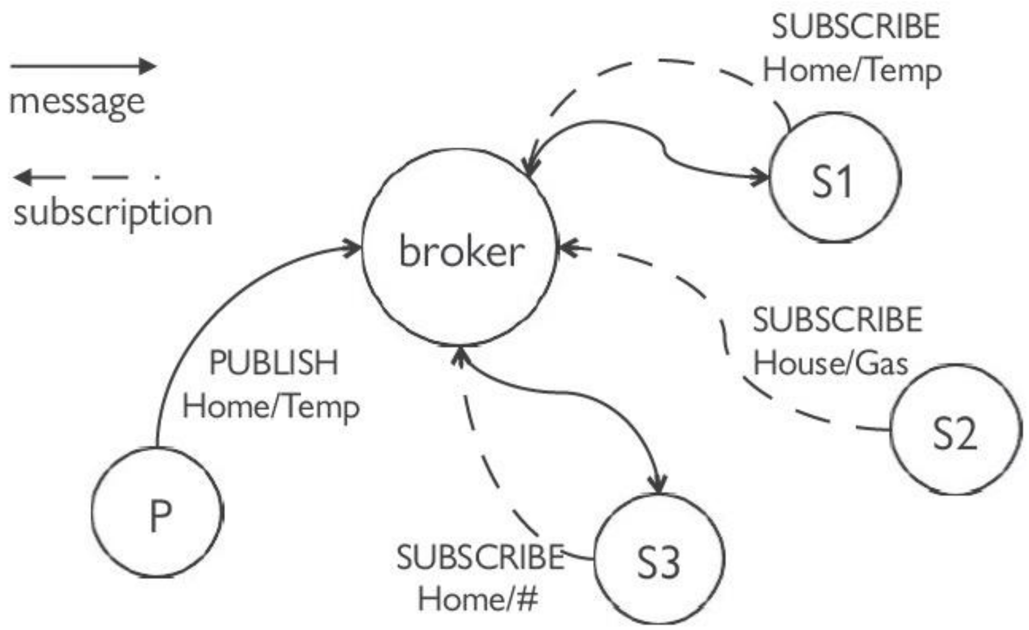
MQTT

CONNECT (example)

S1 opens a new "clean" session.

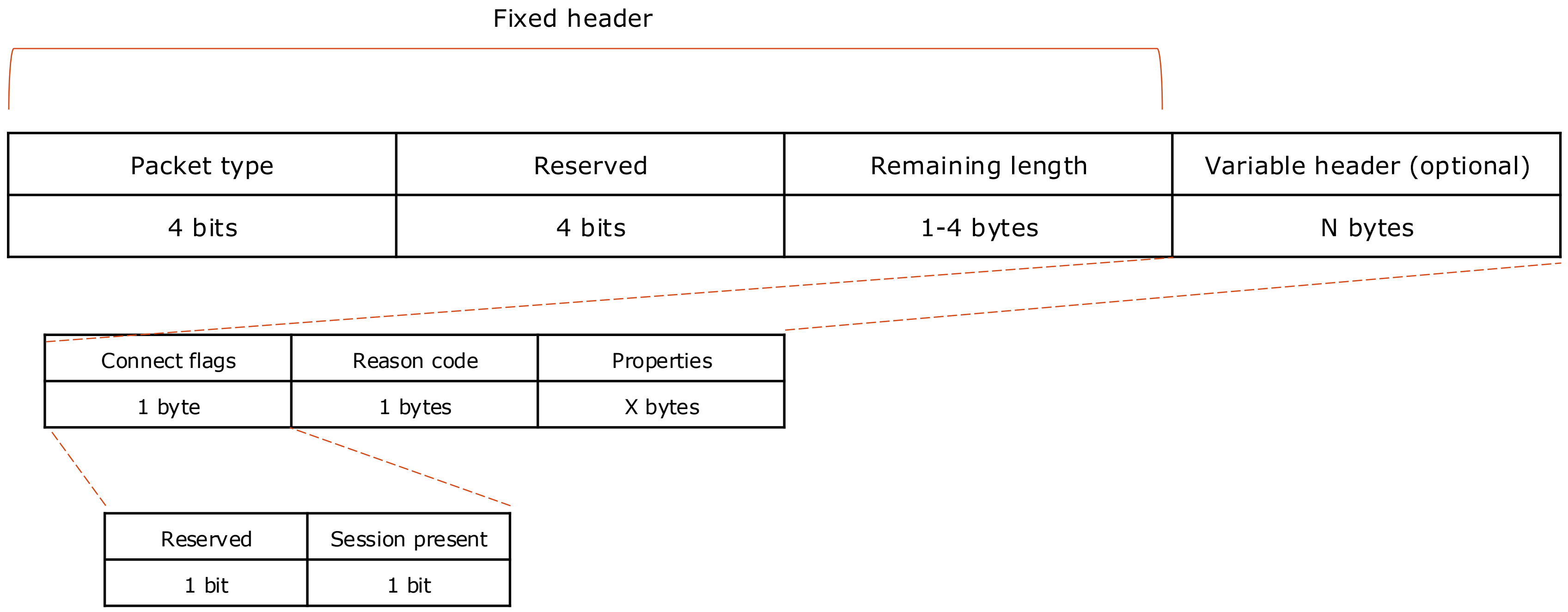


CONNECT	
FLAGS	User Name Flag: 1
	Password Flag: 1
	Will Retain: 0
	Will QoS: 2
	Will Flag: 1
Clean Start: 1	
Keep Alive: 600	
ClientID: "S1"	
Will Topic: "Home/Temp"	
Will Payload: "S1 can't track Temp anymore"	
User Name: "users1"	
Password: "rmvt22"	



CONNACK

- **CONNACK**: Sent by the server after CONNECT, to inform of the connection result.
- CONNACK has no payload.

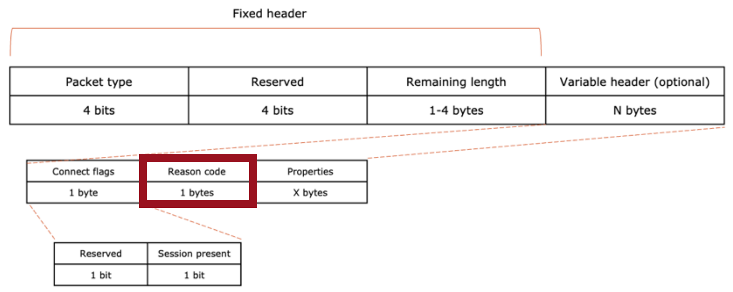


Packet type	Reserved	Remaining length	Variable header (optional)
4 bits	4 bits	1-4 bytes	N bytes

Connect flags	Reason code	Properties
1 byte	1 bytes	X bytes

Reserved	Session present
1 bit	1 bit

MQTT



CONNACK

- **Reason Code**: Used to indicate the result of the connection.

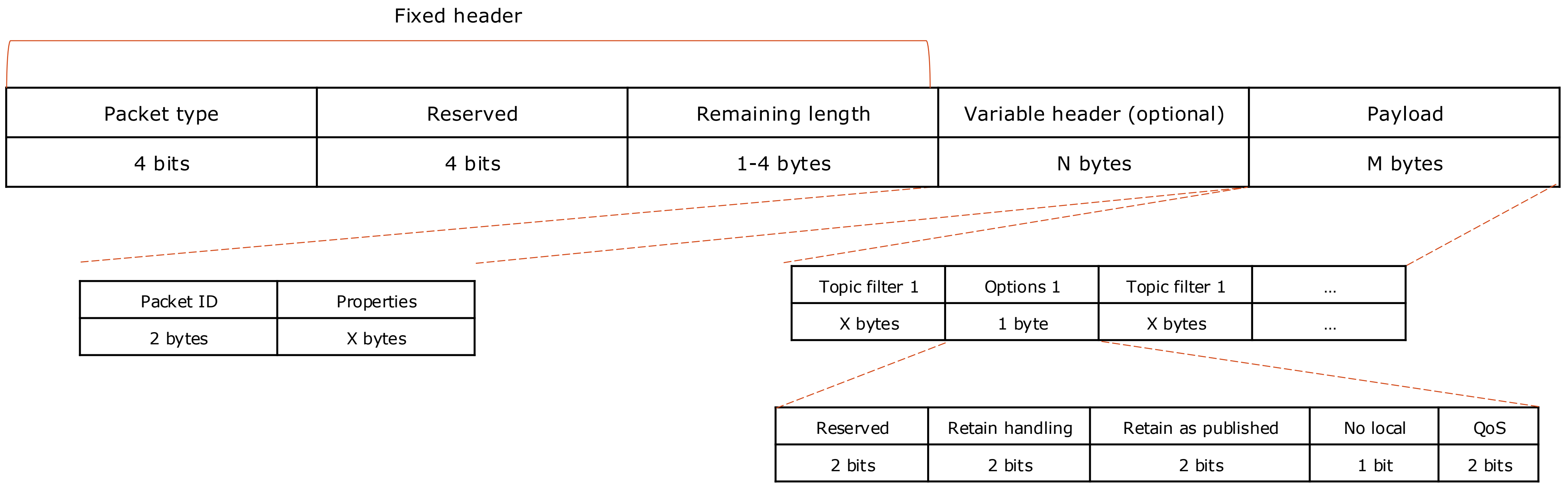
Valu e	Reason Code Name	Description
0x00	Success	The connection is accepted.
0x81	Malformed Packet	The server cannot correctly parse the CONNECT packet according to the protocol specification, for example, the reserved bit is not set to 0 according to the protocol requirements.
0x82	Protocol Error	The CONNECT packet can be parsed correctly, but the content does not conform to the protocol specification, for example, the value of the Will Topic field is not a valid MQTT topic.
0x84	Unsupported Protocol Version	The server does not support the MQTT protocol version requested by the client.
0x85	Client Identifier not valid	Client ID is valid, but is not accepted by the server. For example, the Client ID exceeds the maximum length allowed by the server.
0x86	Bad User Name or Password	The client was refused a connection because it used an incorrect username or password.
0x95	Packet too large	The CONNECT packet exceeds the maximum size allowed by the server, probably because it carries a large will message.
0x8A	Banned	Indicating that the client is prohibited from logging in. For example, the server detects the abnormal connection behavior of the client, so the Client ID or IP address of the client is added to the blacklist, or the background administrator manually blocks the client. Of course, the above usually depends on the specific server implementation.

More details: https://www.emqx.com/en/blog/mqtt5-new-features-reason-code-and-ack

MQTT

SUBSCRIBE

- It is used to initiate a subscription request for the topic(s) a client wants to subscribe to, each with a specified **QoS level** (we'll see).
- Followed by **SUBACK** (and **UNSUBSCRIBE + UNSUBACK**).



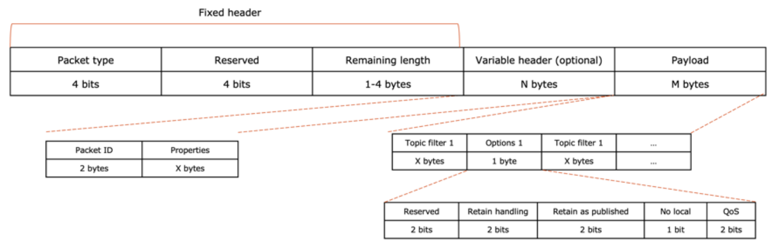
Packet type	Reserved	Remaining length	Variable header (optional)	Payload
4 bits	4 bits	1-4 bytes	N bytes	M bytes

Packet ID	Properties
2 bytes	X bytes

Topic filter 1	Options 1	Topic filter 1	…
X bytes	1 byte	X bytes	…

Reserved	Retain handling	Retain as published	No local	QoS
2 bits	2 bits	2 bits	1 bit	2 bits

MQTT



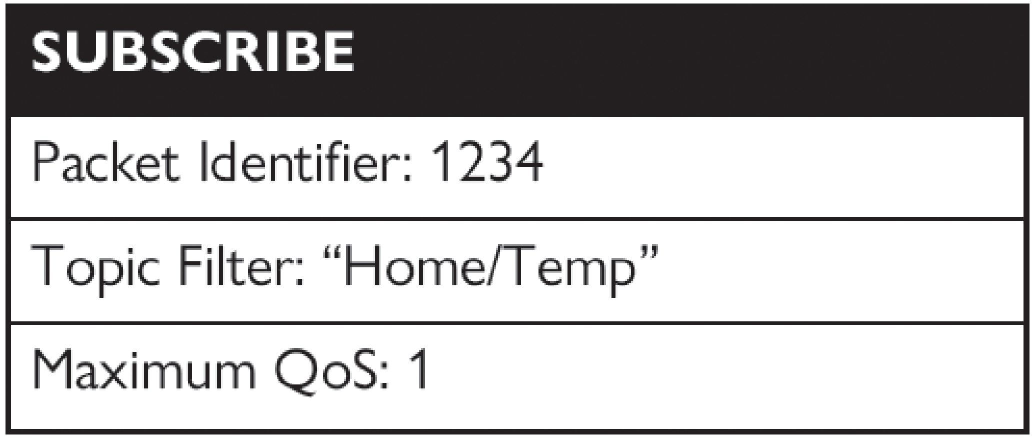
SUBSCRIBE

- **Packet ID**: Used to uniquely identify the subscription request. PUBLISH, SUBSCRIBE, and UNSUBSCRIBE packets use a set of packet identifiers, which means they cannot use the same packet identifier simultaneously.
- **Retain Handling**: Used to indicate whether the server needs to send Retained Messages to this subscription when the subscription is established.
- **Retain As Published**: Used to indicate if the server needs to keep the Retain flag in the message when forwarding the application message to this subscription.
- **No Local**: Used to indicate whether the server can forward the application message to the publisher of the message.
- **QoS**: Determines the maximum QoS level that the server can use when forwarding messages to this subscription (we'll see).

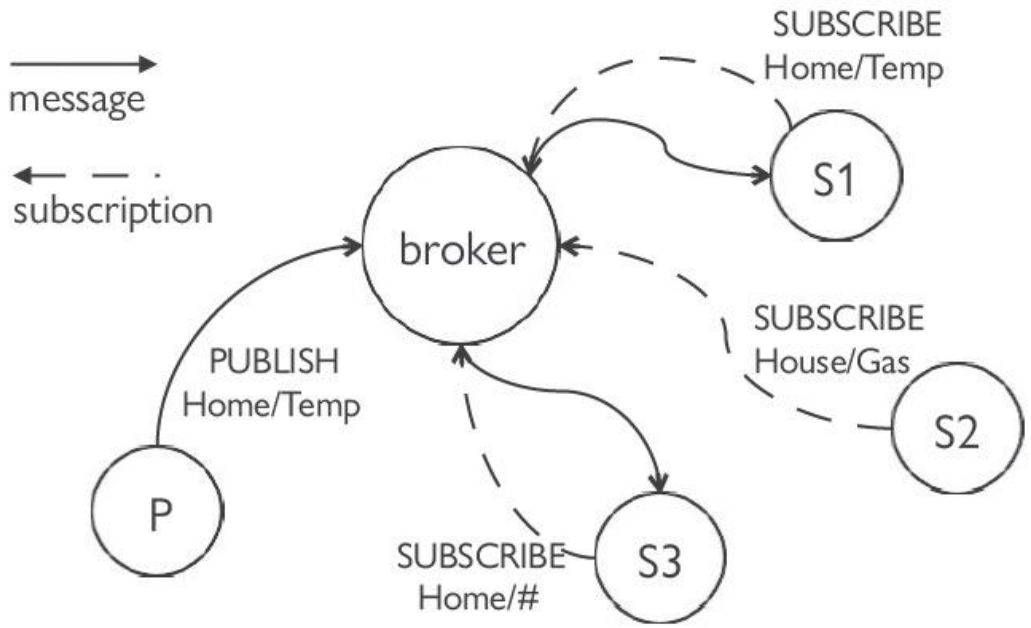
MQTT

SUBSCRIBE (example)

Client S1 sends the broker a SUBSCRIBE message with one or more topics to which it wants to subscrive.



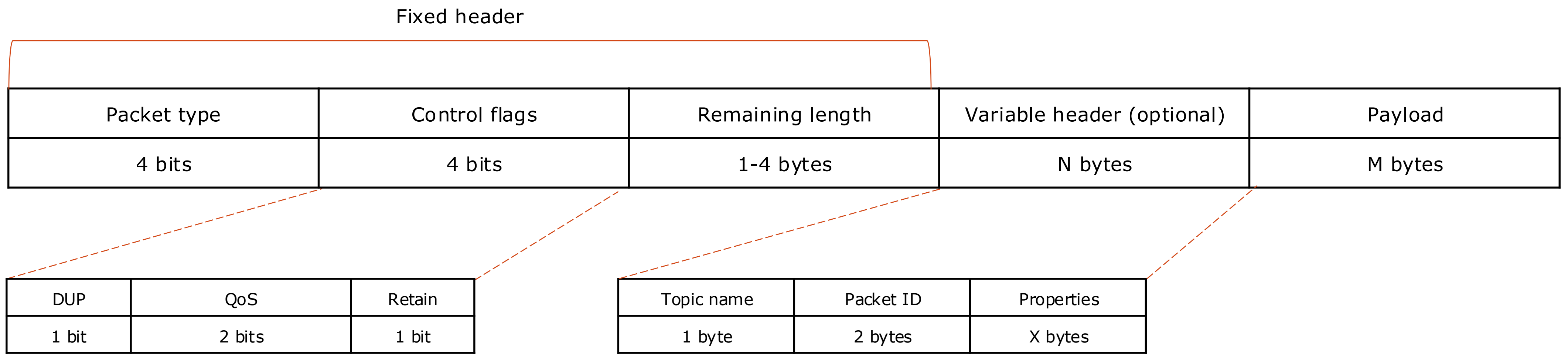
SUBSCRIBE
Packet Identifier: 1234
Topic Filter: “Home/Temp”
Maximum QoS: 1



MQTT

PUBLISH

- It is used for message distribution: the client publishing messages to the server, or the server forwarding messages to the subscriber.
- Followed by: **PUBACK**, **PUBREC**, **PUBREL**, and **PUBCOMP**.



Packet type	Control flags	Remaining length	Variable header (optional)	Payload
4 bits	4 bits	1-4 bytes	N bytes	M bytes

DUP	QoS	Retain
1 bit	2 bits	1 bit

Topic name	Packet ID	Properties
1 byte	2 bytes	X bytes

MQTT

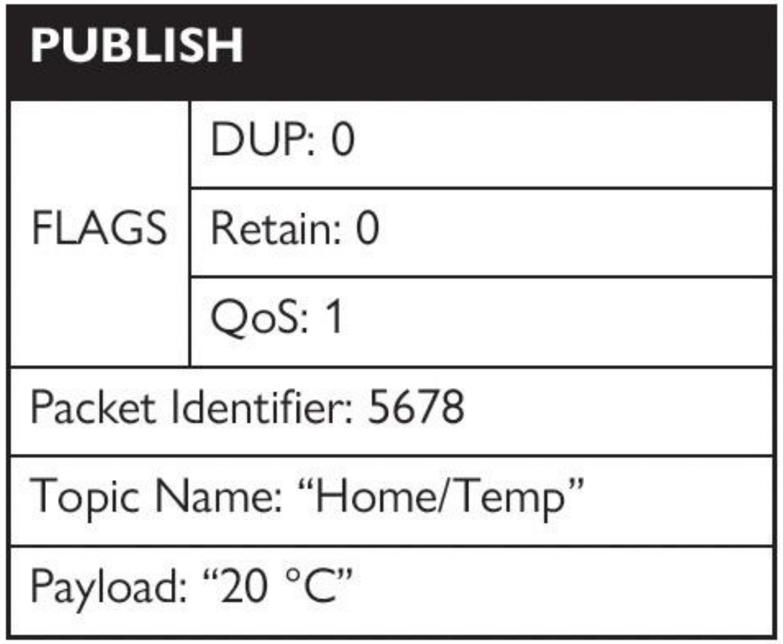
PUBLISH

- **DUP**: = 1 if it is a retransmitted packet. The number of packets with DUP set to 1 can reveal the quality of the current communication link.
- **QoS**: Determines the QoS level of the message (we'll see).
- **Retain**: = 1 if the current message is a Retained Message, meaning that it should be stored by the broker and delivered to any new subscriber.
  - It allows an immediate update of new subscribers.
- **Topic Name**: Used to indicate which channel the message should be published to.
- Packet Identifier: Used to uniquely identify the message currently being transmitted. It only appears in the PUBLISH packet when the QoS level is 1 or 2.
- **Payload**: The content of the application message we send.

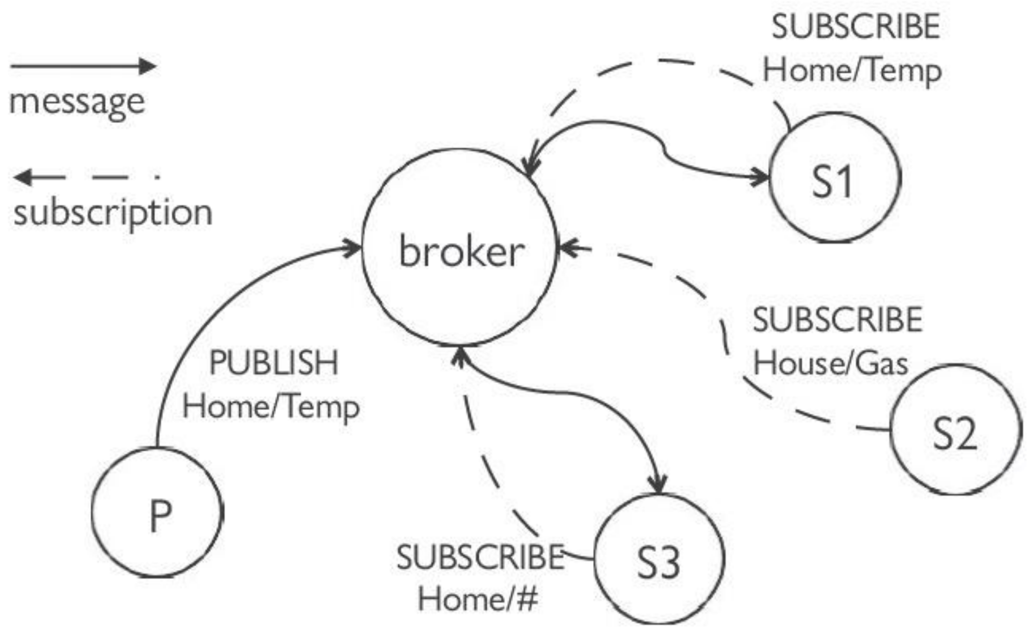
MQTT

PUBLISH (example)

Client P sends a PUBLISH message
with a certain payload.



PUBLISH	
FLAGS	DUP: 0
	Retain: 0
	QoS: 1
Packet Identifier: 5678	
Topic Name: “Home/Temp”	
Payload: “20 °C”	



MQTT

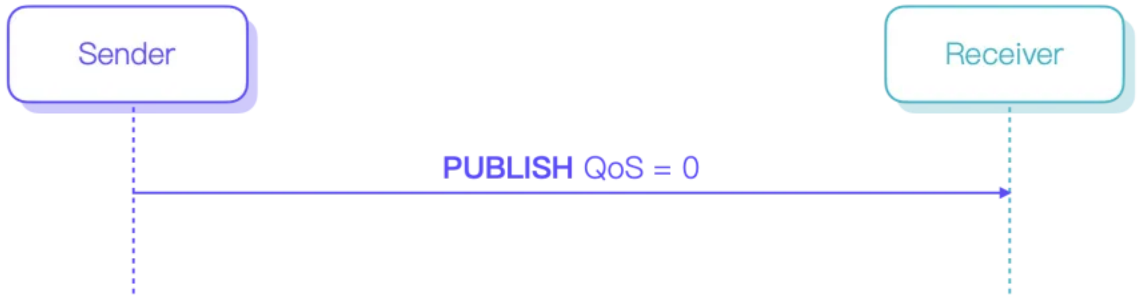
QoS levels

- In unstable network environments, MQTT devices may struggle to ensure reliable communication using only the TCP transport protocol.
- MQTT includes a **Quality of Service (QoS)** mechanism to provide the message with different levels of service, catering to the user's specific requirements.
  - MQTT 5.0: QoS should not be applied if the transport layer is working properly, as TCP should already provide the required level of reliability.
- QoS level is specified in the subscription request.

MQTT

QoS levels

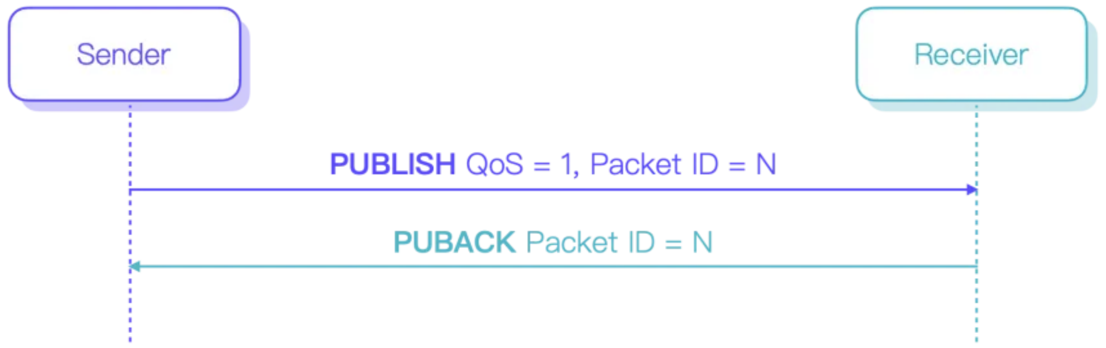
- **QoS 0**: At Most Once (aka ”fire and forget”).
  - The sender does not wait for acknowledgement or store and retransmit the message, so the receiver does not need to worry about receiving duplicate messages.



MQTT

QoS levels

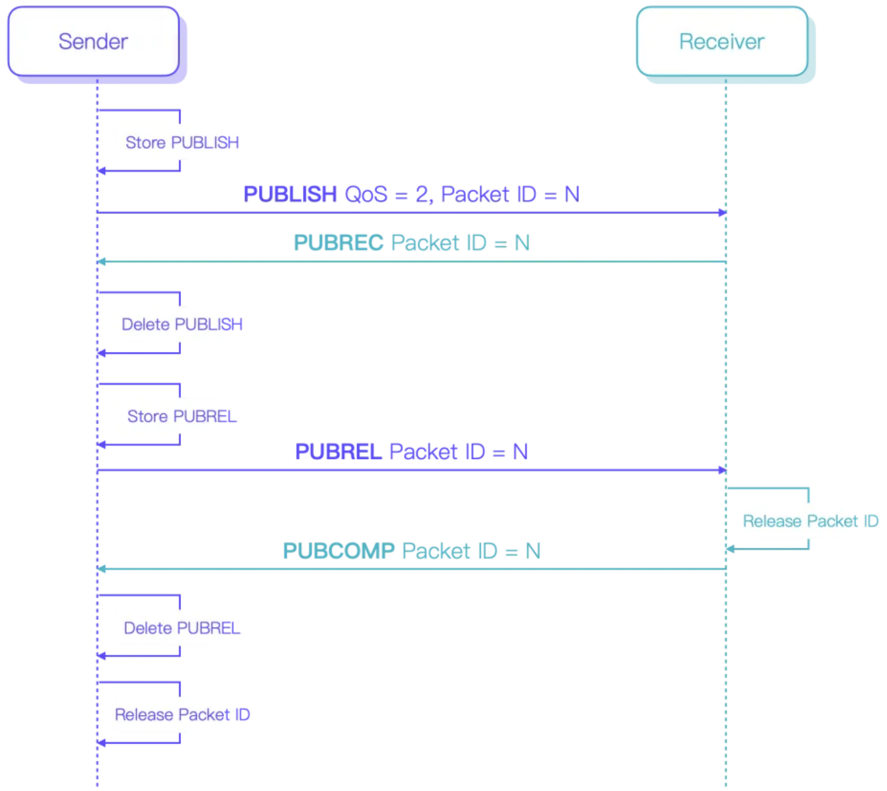
- **QoS 1**: At Least Once
  - When the sender receives a PUBACK packet from the receiver, it considers the message delivered successfully. Until then, PUBLISH packet is stored for potential retransmission.
  - Packet ID is used to match the PUBLISH packet with the corresponding PUBACK packet.



MQTT

QoS levels

- **QoS 2**: Exactly Once. Ensures that messages are not lost or duplicated, unlike in QoS 0/1.
  1. The sender stores and sends a PUBLISH packet with QoS 2 and then waits for a **PUBREC** response packet from the receiver. This process is similar to QoS 1, with the exception that the response packet is PUBREC instead of PUBACK.
  2. The sender can delete its locally stored copy.
  3. The sender sends a **PUBREL** packet to release the Packet ID. It is stored for potential retx.
  4. The receiver responds with a **PUBCOMP** packet.



MQTT

Final considerations

- It is a very good choice for IoT systems with constrained endpoints.
  - Lightweight protocol.
  - Can be implemented on many embedded platforms (Arduino, Raspberry Pi, …).
  - Several open-source implementations.
  - Retrain mechanism, to implement the seamless addition of new endpoints.
  - LTW, to ensure good management in case of connectivity outages.
- Main drawback: Centralized broker can create scalability and reliability issues.

UNIVERSITÀ DEGLI STUDI DI PADOVA

Master's degree ICT Internet Multimedia Engineering

11 – IoT applications
CoAP

Marco Giordani (marco.giordani@unipd.it)
Department of Information Engineering (DEI) – SIGNET Research Group
University of Padova – Via Gradenigo 6/B, 35131, Padova (Italy)

CoAP

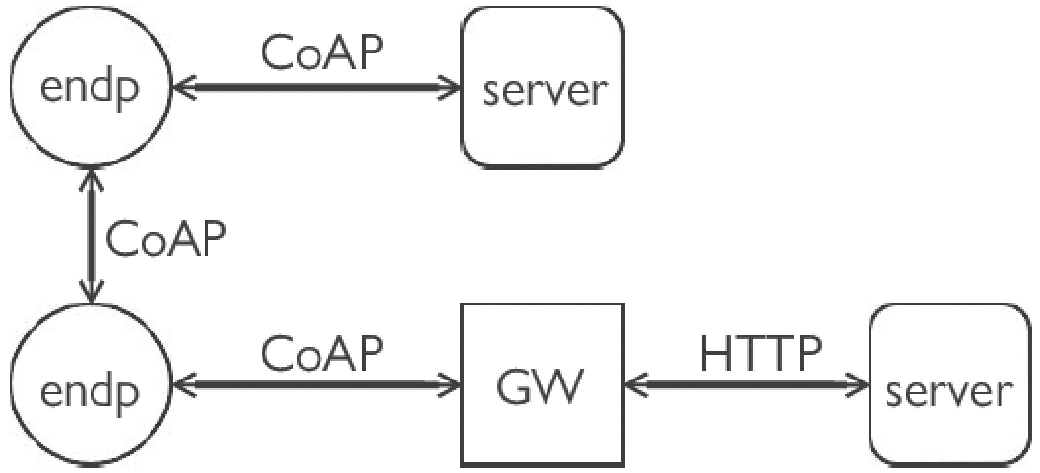
Overview

- **Constrained Application Protocol (CoAP)** is another example of an IoT-specific application protocol.
  - It was designed to allow for small, low-power devices with limited computational resources, scarce bandwidth, and long sleep times, to use the wider Internet.
- It is normally layered on top of UDP (vs. TCP for MQTT).
  - To provide **security**, it uses IPsec (at the network layer) or DTLS.
- It is implemented based on the **client-server** paradigm.
  - Messages are a sequence of **requests** (aka **methods**) and **responses**, like in HTTP.
  - It is not equivalent to HTTP, but internetworking across the two protocols is easy.
  - Unlike HTTP, CoAP devices can «**observe**» the state of resources, similar to the publish-subscribe paradigm (we'll see).

CoAP

Architecture

- Endpoints (clients) may talk to a CoAP server or an HTTP server through a gateway.
- They can also communicate directly in peer-to-peer.
  - The "lightweight" characteristics of the protocol make the configuration of a CoAP server feasible even on resource-constrained endpoints.
  - The CoAP server may act as a client too.



CoAP

Frame structure

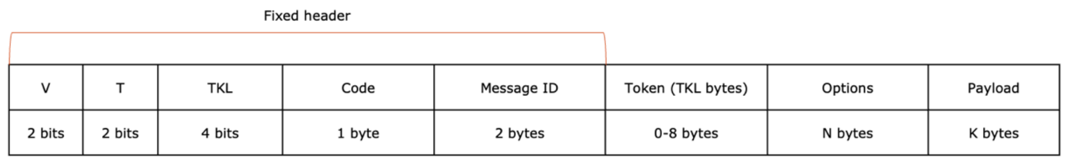
- Given the use of UDP, it requires dedicated **error recovery mechanisms**.
- The format (and header) of messages is **more compact than HTTP**.
- The format (and header) of messages may be slighlt **more complicated than MQTT.**

Fixed header

V	T	TKL	Code	Message ID	Token (TKL bytes)	Options	Payload
2 bits	2 bits	4 bits	1 byte	2 bytes	0-8 bytes	N bytes	K bytes

CoAP

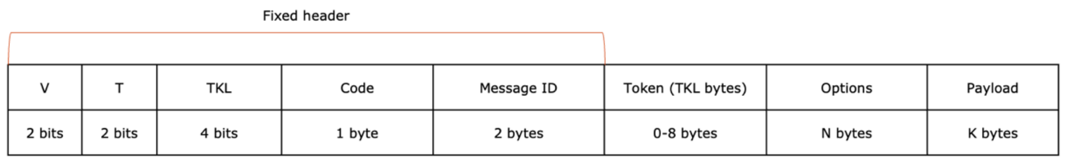
Frame structure



V	T	TKL	Code	Message ID	Token (TKL bytes)	Options	Payload
2 bits	2 bits	4 bits	1 byte	2 bytes	0-8 bytes	N bytes	K bytes

- **Version (V).**
- **Type (T)**: **Request** (Confirmable, Non-confirmable), **Response** (ACK, Reset).
  - Confirmable (0): This message expects a corresponding acknowledgement message.
  - Non-confirmable (1): This message does not expect a confirmation message.
  - Acknowledgement (2): A response that acknowledges a confirmable message
  - Reset (3): Indicates that the receiver cannot (or doesn't want to) process a request.
- **Token Length (TKL)**: Length of the Token field in the header (may be 0).
- **Code**: It is the code (ID) of the method/request or response.
  - It constists of a **class** (3 bits) and a **code** (5 bits).
- **Message ID**: Used to detect duplicates and/or retransmissions.
- **Token**: Used to match a response with its request (may be 0 if no ambiguity).

CoAP



Fixed header

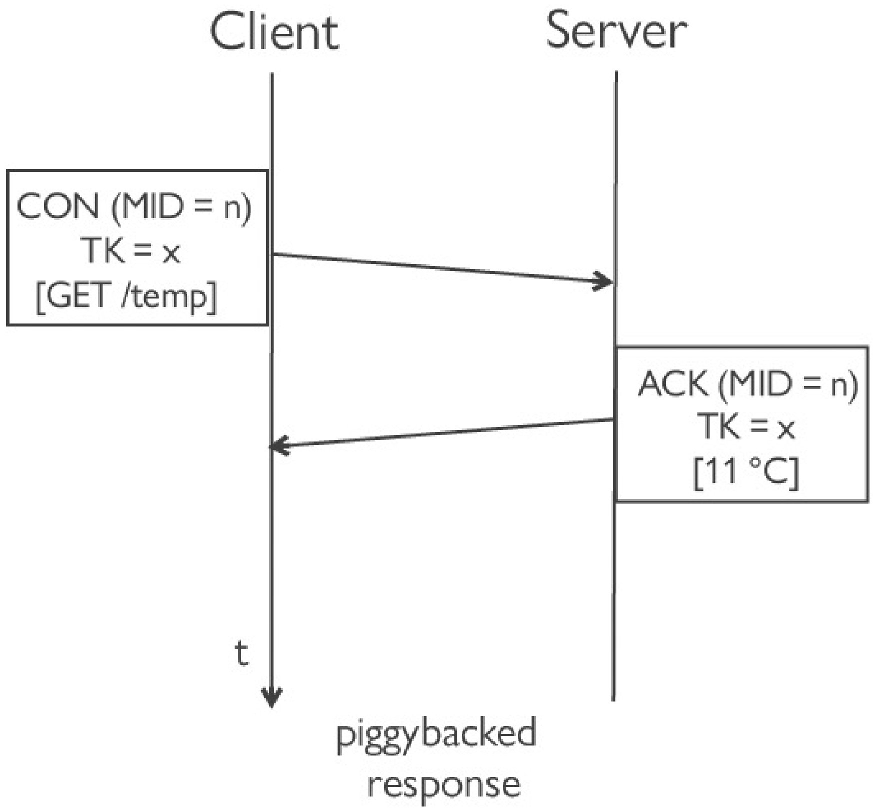
V	T	TKL	Code	Message ID	Token (TKL bytes)	Options	Payload
2 bits	2 bits	4 bits	1 byte	2 bytes	0-8 bytes	N bytes	K bytes

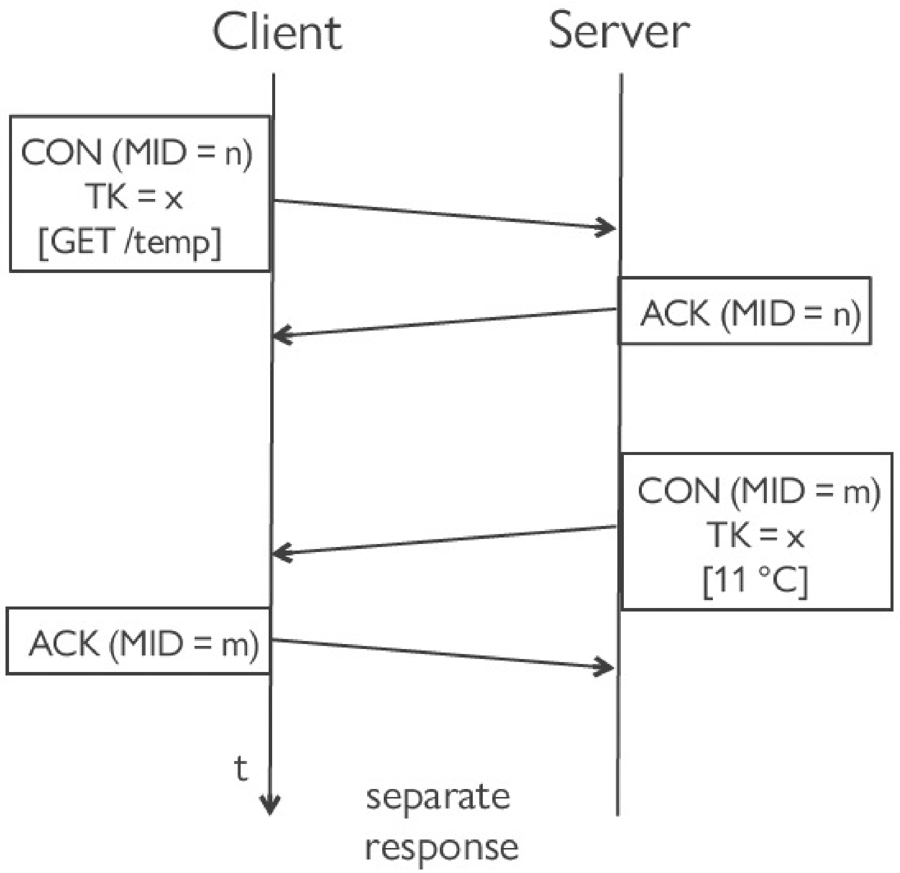
Frame structure: Token vs. Message ID

- **Message IDs** are unique values that are used to identify duplicate messages and match Confirmable messages to Acknowledgement messages.
  - Messages are paired to their ACKs by the Message ID
- **Tokens** are unique values used to match Requests to Responses.
  - Requests are paired to their responses by the Token.
  - Matching requests and responses is not done with the Message ID because a response may be sent in a different message than the ACK (which uses the Message ID).
  - Responses may be sent as **separate response** or **piggybacked**
    - Separate response: useful for low latency, if the ACK is generated after some time. Same Token, different ID. However, it requires an ACK for confirmation.
    - Piggyback: response is sent in the ACK relative to the (Confirmable) request. Same Token and ID. It does not require an ACK (if client in Confirmable mode already expects an ACK).
  - Token may not be used in Non-confirmable mode, or if there is a single pending request.

CoAP

Frame structure: Token vs. Message ID





CoAP

Requests (or methods)

- Some example of the most popular CoAP requests (aka methods).

Method/request (0.xx)	Code	Action
GET	0.1	Asks for (the state of) a resource
POST	0.2	Requests the processing of the message content, possibly resulting in the creation or update of a resource.
PUT	0.3	Substitutes the resource specified in the URI or, if non-existing, creates a new one
DELETE	0.4	Removes the resource

- Full list: https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#codes

CoAP

Responses

- Responses are characterized by some **codes**.
- Some example of the most popular CoAP responses.

Responses	Code	Action
		SUCCESS (2.xx)
Created	2.1	Resource has been created
Deleted	2.2	Resource has been delated
Content	2.5	Carries the content of the resource
		CLIENT ERROR (4.xx)
Unauthorized	4.1	The client is not authorized to perform the requested action.
Request Entity Too Large	4.13	The size of a client's request exceeds the server's limit.
		SERVER ERROR (5.xx)
Not implemented	5.1	The server cannot fulfill the client's request (e.g., the server does not recognize a certain method).

CoAP

Functionalities: resource observation

- **Resource observation**: Allows the possibility of a device to be asynchronously updated on the state of a resource.
  - Useful in IoT since sensory measurements may be unsolicited.
  - **GET** message (with "**Observer**" option set to 0): the "observer" will receive a sequence of responses with the same Token, all carrying information/measurements about the state of the resource.
  - Sequence number is increased at each response for the observer to do reordering.
  - Observation ends (1) after a Reset, or (2) after a GET message with "**Observer**" option 1.
  - The rate of update depends on the nature of the resource (e.g., resolution of measures).
  - Rate-limiting mechanisms may be in place to reduce flooding of messages.

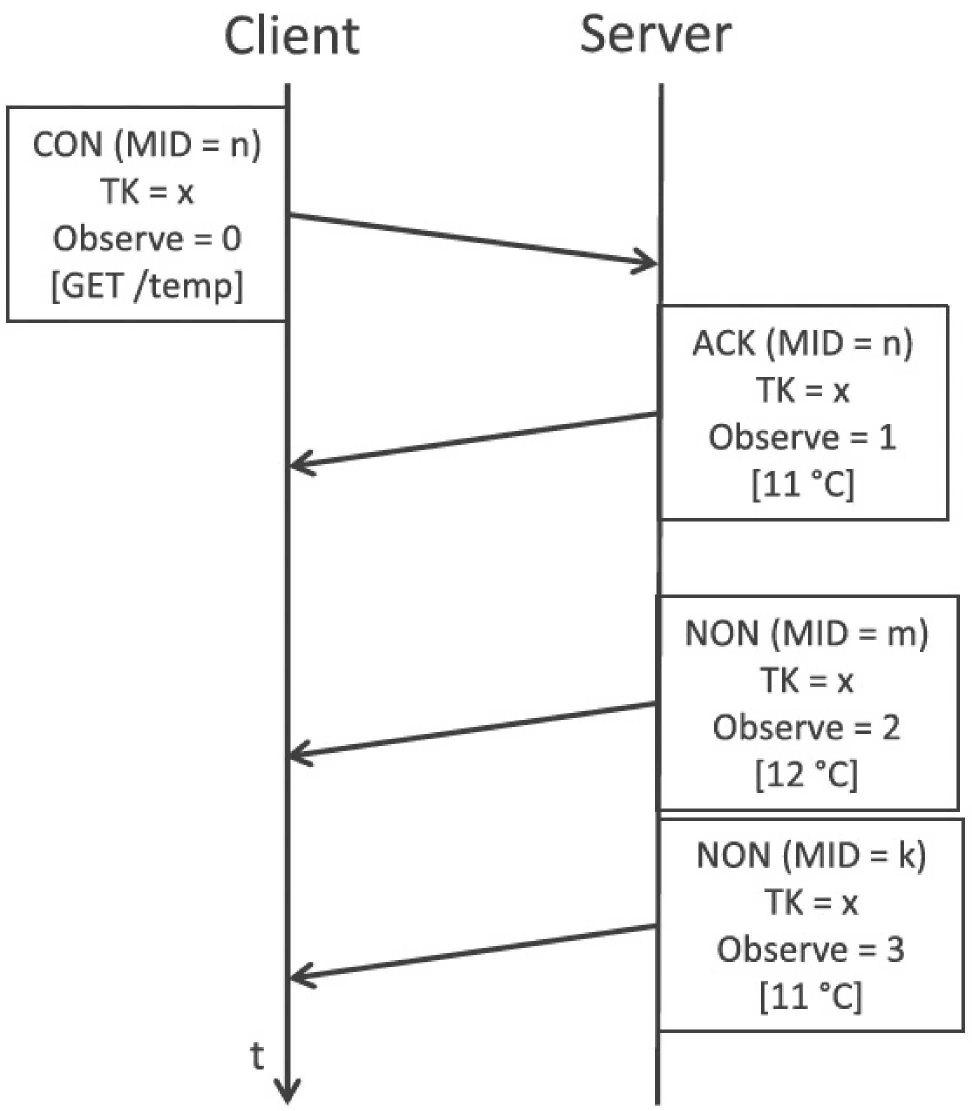
CoAP

Functionalities: resource observation

Resource observation allows for the server to **send unsolicited messages**.

The rate of update depends on the nature of the resource (e.g., resolution of measures).

- For example, the resource representing a tempertaure sensor may be defined with a 1 °C resolution: in principle, the CoAP server will generate one-tenth of the updates of a 0.1 °C resource defined for the same sensor.


Client
Server
CON (MID = n) TK = x Observe = 0 [GET /temp]
ACK (MID = n) TK = x Observe = 1 [11 °C]
NON (MID = m) TK = x Observe = 2 [12 °C]
NON (MID = k) TK = x Observe = 3 [11 °C]
t

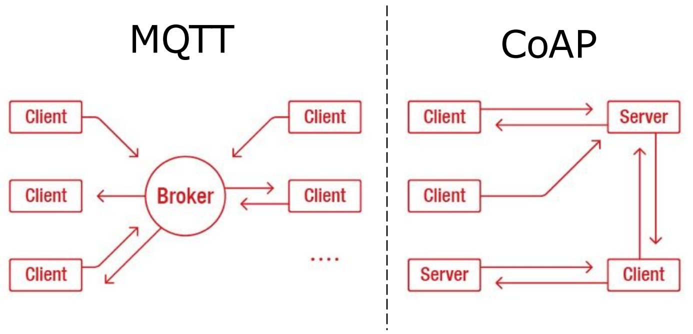
CoAP

Final condiserations

- It is a competitor of MQTT given some nice features:
  - Simple design (good for IoT networks with many resource-constrained end nodes).
  - Support for asynchronomous events with no strict real-time requirements.
  - Alignment and interoperability with HTTP.
  - REST model (easy for Web designers to build/implement CoAP applications).
- It comes with some drawbacks:
  - Even though it has an “Observer” method, it is still a client-server application. In order to implement a quasi-publish-subscribe paradigm, it is more convenient to use MQTT.

CoAP vs. MQTT

Comparison



Feature	MQTT	CoAP
Purpose	Messaging and communication in IoT	Designed for resource-constrained devices in IoT
Transport Protocol	TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
Communication Style	Publish/Subscribe model	Request/Response (Client-Server-like) model
Header Size	2 bytes fixed header	4 bytes fixed header
Payload Format	Supports binary and text payloads	Supports binary and text payloads
QoS (Quality of Service)	Levels 0, 1, and 2 for message delivery	Reliability through "confirmable" and "non-confirmable" messages
Message Types	Publish, Subscribe, Connect, Disconnect, etc.	GET, POST, PUT, DELETE, etc.
Resource Discovery	Not inherent, requires additional mechanisms	Built-in resource discovery through CoRE Link Format