MODULE-3 CHAPTER 6

Application Protocols for IoT

- **The Transport Layer**
- **IoT Application Transport Methods**

Generic Web-Based Protocols

- **Web-based protocols** have become common in consumer and enterprise applications and services.

- Hence, it makes **sense** to try to leverage these protocols when developing IoT applications, services, and devices in order to ease the integration of data and devices from prototyping to production.

- The level of familiarity with generic web-based protocols is high and programmers with basic web programming skills can work on IoT applications.

- The **scaling methods** for web environments is also well understood and it is crucial while developing consumer applications for potentially large number of IoT devices.

- In this case again we need have a look into a issue of constrained or non-constrained nodes and networks to design an appropriate web-based IoT protocol.

- When considering web services implementation on an IoT device, the choice between supporting the client or server side of the connection must be carefully weighed.

- On **non-constrained networks**, such as Ethernet, Wi-Fi, or 3G/4G cellular, where **bandwidth** is not perceived as a potential issue, **data payloads** based on a verbose data model representation, including XML or JavaScript Object Notation (JSON), can be transported over HTTP/HTTPS or WebSocket.

- On **constrained nodes,** one can deploy an embedded web server software with advanced features implemented in very little memory. This enables the use of embedded web services software on some constrained devices.

- Interactions between **real-time communication** tools powering collaborative applications, such as voice and video, instant messaging, chat rooms, and IoT devices, are also emerging.

- This is driving the need for simpler communication systems between people and IoT devices. One protocol that addresses this need is **Extensible Messaging and Presence Protocol (XMPP).**

- In summary, the **Internet of Things** greatly benefits from the existing web-based protocols. These protocols, including **HTTP/HTTPS and XMPP**, ease the integration of IoT devices in the Internet world through well-known and scalable programming techniques.

IoT Application Layer Protocols

- When considering **constrained networks** and/or a large-scale deployment of **constrained nodes**, verbose web-based and data model protocols, may be **too heavy for IoT applications.**

- To **address this problem**, the **IoT industry** is working on new lightweight protocols that are better suited to large numbers of constrained nodes and networks.

- Two of the **most popular protocols** are
- **CoAP and MQTT.**
- Figure 6.6 highlights their position in a common IoT protocol stack.
- In Figure 6.6, **CoAP and MQTT are naturally at the top of this sample IoT stack, based on an IEEE 802.15.4 mesh network.**
- We will almost always **find CoAP deployed over UDP and MQTT running over TCP.**

CoAP

- **Constrained Application Protocol (CoAP)** resulted from the IETF Constrained RESTful Environments (CoRE) working group's efforts to develop a generic framework for resource-oriented applications targeting constrained nodes and networks.

- The **CoAP framework** defines simple and flexible ways to manipulate sensors and actuators for data or device management.

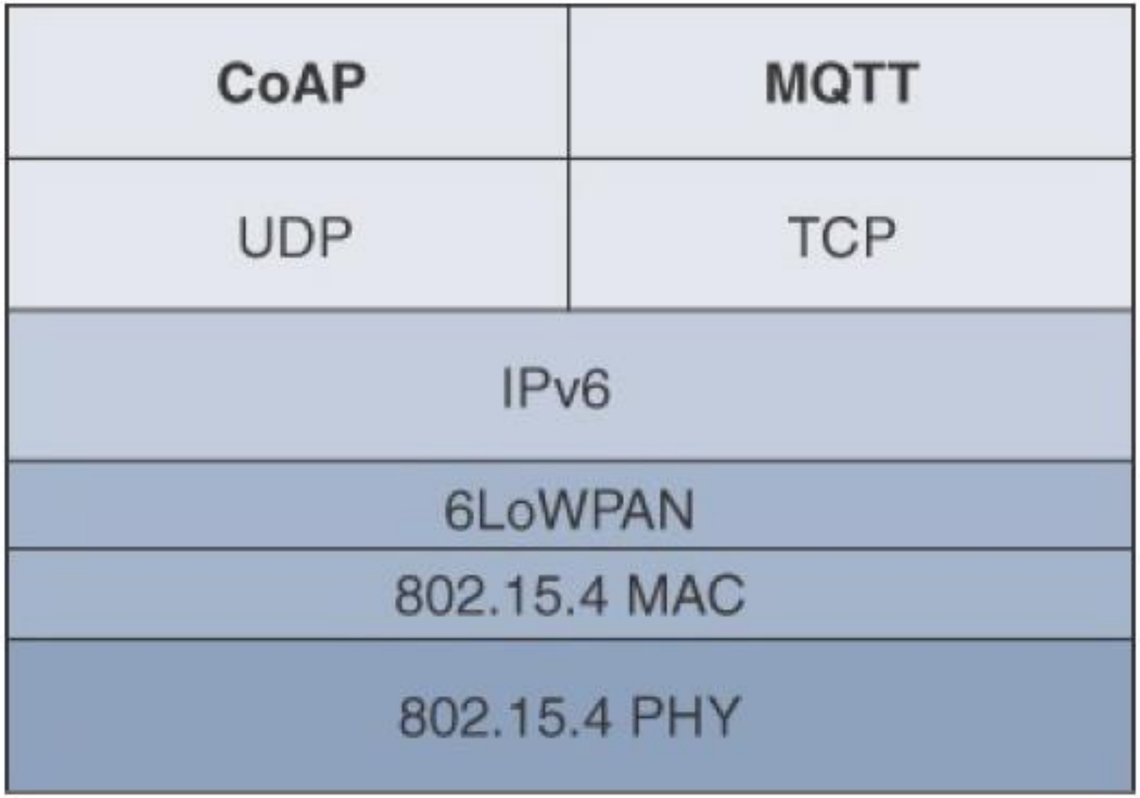

Figure 6.6: Example of a High-Level IoT Protocol Stack for CoAP and MQTT

- The **IETF CoRE** working group has published multiple standards-track specifications for CoAP, including the following:

- **RFC 6690**: Constrained RESTful Environments (CoRE) Link Format
- **RFC 7252**: The Constrained Application Protocol (CoAP)
- **RFC 7641**: Observing Resources in the Constrained Application Protocol (CoAP)
- **RFC 7959**: Block-Wise Transfers in the Constrained Application Protocol (CoAP)
- **RFC 8075**: Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP)

- The **CoAP messaging model** is primarily designed to facilitate the exchange of messages over UDP between endpoints, including the secure transport protocol Datagram Transport Layer Security (DTLS).

- The IETF CoRE working group is studying alternate transport mechanisms, including TCP, secure TLS, and WebSocket.

- **CoAP over Short Message Service (SMS)** as defined in Open Mobile Alliance for Lightweight Machine-to-Machine (LWM2M) for IoT device management is also being considered.

- **Four security modes** are defined: **NoSec, PreSharedKey, RawPublicKey, and Certificate.**

- The NoSec and RawPublicKey implementations are **mandatory**.

- From a formatting perspective, a **CoAP message** is composed of a short fixed length Header field (4 bytes), a variable-length but mandatory Token field (0–8 bytes), Options fields if necessary, and the Payload field.

- Figure 6.7 details the **CoAP message format**, which delivers low overhead while decreasing parsing complexity.
- The **CoAP message format** is relatively **simple and flexible**.
- It allows CoAP to **deliver low overhead**, which is critical for constrained networks, while also being easy to parse and process for constrained devices.

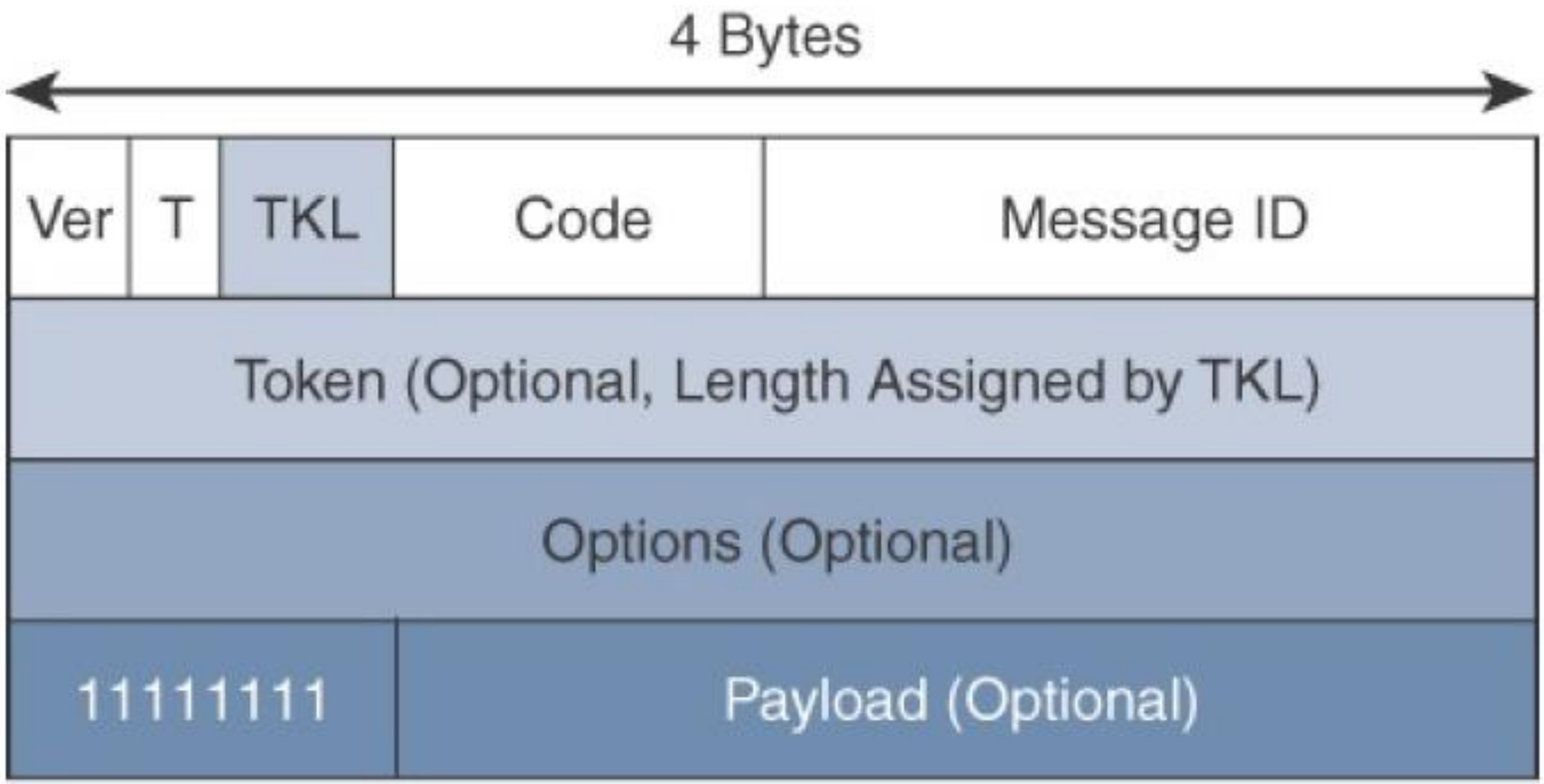

Figure 6.7: CoAP Message Format

CoAP Message Field	Description
Ver (Version)	Identifies the CoAP version.
T (Type)	Defines one of the following four message types: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK), or Reset (RST). CON and ACK are highlighted in more detail in Figure 6-9.
TKL (Token Length)	Specifies the size (0–8 Bytes) of the Token field.
Code	Indicates the request method for a request message and a response code for a response message. For example, in Figure 6-9, GET is the request method, and 2.05 is the response code. For a complete list of values for this field, refer to RFC 7252.
Message ID	Detects message duplication and used to match ACK and RST message types to Con and NON message types.
Token	With a length specified by TKL, correlates requests and responses.
Options	Specifies option number, length, and option value. Capabilities provided by the Options field include specifying the target resource of a request and proxy functions.
Payload	Carries the CoAP application data. This field is optional, but when it is present, a single byte of all 1s (0xFF) precedes the payload. The purpose of this byte is to delineate the end of the Options field and the beginning of Payload.

- **CoAP** can run over **IPv4 or IPv6.**
- It is recommended that the message fit within a **single IP packet** and UDP payload to avoid fragmentation.
- For **IPv6,** with the default **MTU size** being **1280 bytes** and allowing for **no fragmentation across nodes**, the maximum CoAP message size could be up to 1152 bytes, including 1024 bytes for the payload.

- As illustrated in Figure 6.8, CoAP communications across an IoT infrastructure can take various paths.

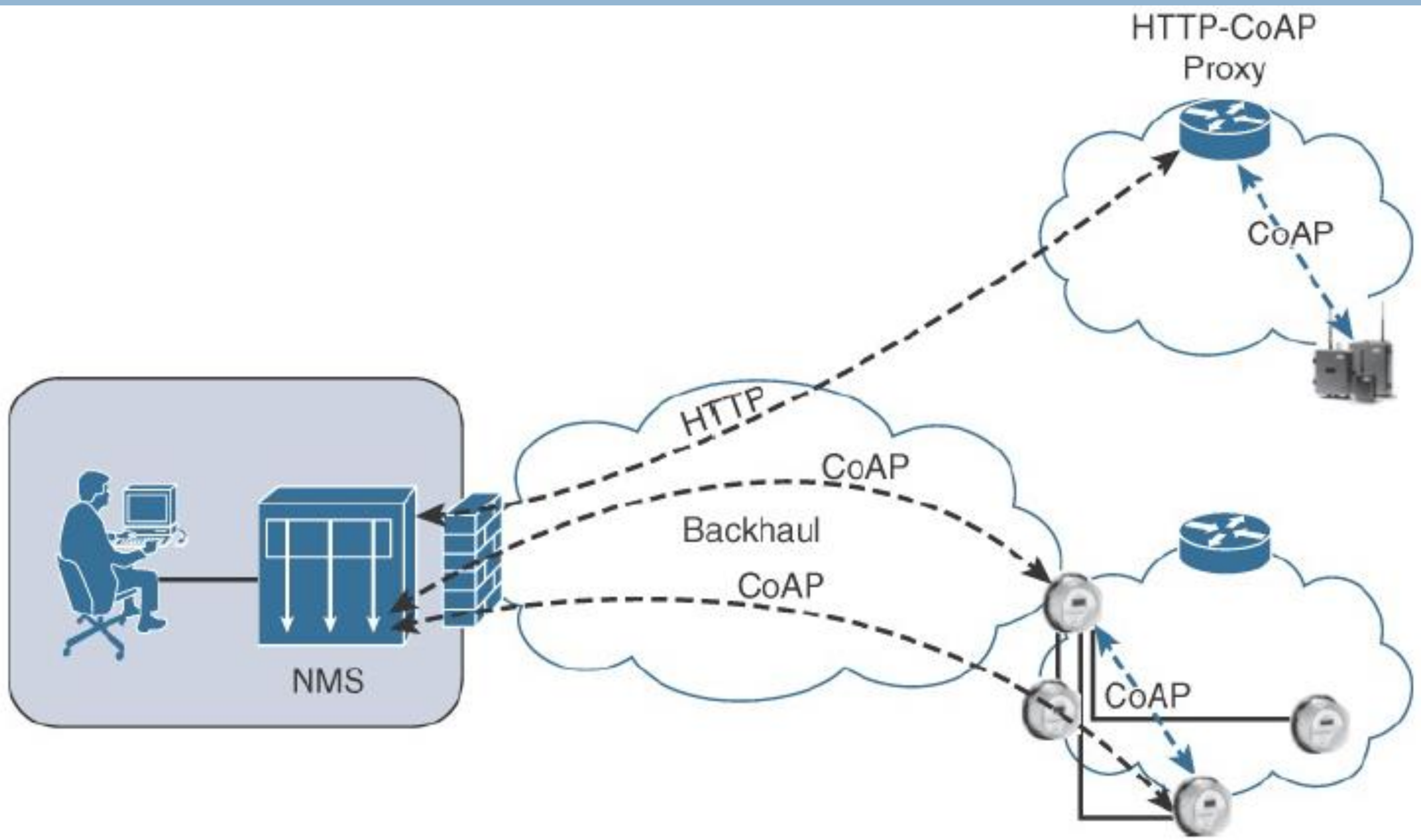

Figure 6.8: CoAP Communications in IoT Infrastrucutres

- Example 6.2 **shows the CoAP URI format**. We may notice that the CoAP URI format is similar to HTTP/HTTPS.

```
coap-URI = "coap:" "//" host [":" port] path-abempty ["?" query]
coaps-URI = "coaps:" "//" host [":" port] path-abempty  ["?" query]
```

Example 6.2 : CoAP URI Format

- The **coap/coaps URI scheme identifies a resource**, including host information and optional UDP port, as indicated by the **host and port** parameters in the **URI**.

- **Connections** can be **between devices** located on the same or different constrained networks or between devices and generic Internet or cloud servers, all operating over IP.

- **Proxy mechanisms are also defined**, and RFC 7252 details a basic HTTP mapping for CoAP.

- As both **HTTP and CoAP are IP-based protocols**, the proxy function can be located practically anywhere in the network, not necessarily at the border between constrained and non-constrained networks.

- Just like HTTP, CoAP is based on the REST architecture, but with a **"thing"** acting as both the **client and the server**.

- Through the **exchange of asynchronous messages, a client requests an action** via a method code on a server resource.

- A **uniform resource identifier** (URI) localized on the **server identifies this resource.**

- The **server responds with a response code** that may include a resource representation.

- The **CoAP request/response semantics** include the methods **GET, POST, PUT, and DELETE**.

- CoAP defines four types of messages: **confirmable**, **non-confirmable**, **acknowledgement**, and **reset**.
- **Method codes** and **response codes** included in some of these messages make them carry requests or responses.
- CoAP code, method and response codes, option numbers, and content format have been assigned by **IANA as Constrained RESTful Environments (CoRE) parameters.**
- While running over **UDP, CoAP** offers a reliable transmission of messages when a CoAP header is marked as **"confirmable."**

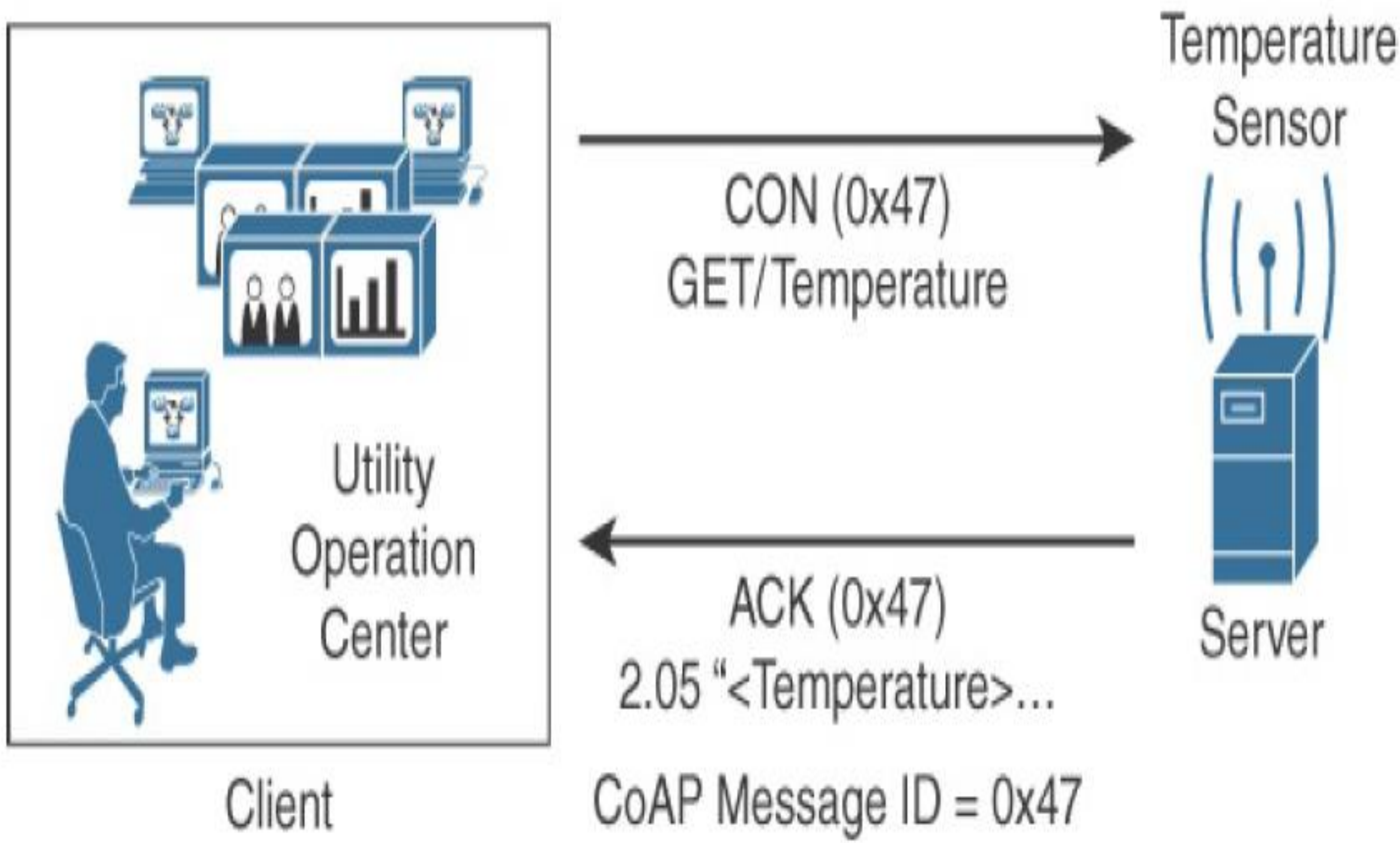

Figure 6.9: CoAP Reliable Transmission Example

- **CoAP supports** basic congestion control with a default time-out, **simple stop and wait retransmission** with exponential back-off mechanism, and detection of duplicate messages through a message ID.

- If a **request or response** is tagged as confirmable, the recipient must explicitly either **acknowledge or reject the message**, using the **same message ID** as shown in Figure 6.9.

- If a recipient can't process a **non-confirmable message**, a reset message is sent.

- Figure 6.9 shows a utility **operations center on the left**, acting as **the CoAP client**, with the CoAP server being a temperature sensor on the right of the figure.

- The **communication** between the client and server uses a **CoAP message ID of 0x47**.

- The **CoAP Message ID** ensures reliability and is used to detect duplicate messages.

- The client in Figure 6.9 sends a GET message to get the temperature from the sensor.
- The 0x47 message ID is present for this **GET message and that the message is also marked with CON.**
- A CON, or confirmable, marking in a CoAP message means the message will be retransmitted until the recipient sends an acknowledgement (or ACK) with the same message ID.

- In Figure 6.9, the **temperature sensor** does reply with an ACK message referencing the correct message ID of 0x47.
- In addition, this ACK message piggybacks a successful response to the GET request itself. This is indicated by the 2.05 response code followed by the requested data.
- CoAP supports data requests sent to a group of devices by leveraging the use **of IP Multicast.**
- **Implementing IP Multicast with CoAP** requires the use of all-CoAP-node multicast addresses.

- Therefore, **endpoints** can find **available CoAP services** through multicast service discovery.
- A typical use case for multicasting is deploying a firmware upgrade for a group of IoT devices, such as smart meters.
- With often no affordable manual configuration on the IoT endpoints, a CoAP server offering services and resources needs to be discovered by the CoAP clients.

- **Services from a CoAP server** can either be discovered by learning a URI in a namespace or through the "All CoAP nodes" multicast address.

- When utilizing the URI scheme for discovering services, the default **port 5683** is used for non-secured CoAP, or **coap**, while **port 5684** is utilized for DTLS-secured CoAP, or **coaps**.

- The CoAP server must be in listening state on these ports, unless a different port number is associated with the URI in a namespace.

Message Queuing Telemetry Transport

- At the end of the 1990s, engineers from IBM and Arcom (acquired in 2006 by Eurotech) were looking **for a reliable, lightweight, and cost-effective protocol.**

- They wanted to monitor and control a large number of sensors and their data from a central server location, as typically used by the oil and gas industries.

- These were some of the rationales for the selection of a **client/server and publish/subscribe framework** based on the TCP/IP architecture, as shown in Figure 6.10.

- An **MQTT client can act as a publisher** to **send data** (or resource information) to an MQTT server acting as an MQTT message broker.

- In the example illustrated in Figure 6.10, the **MQTT client** on the **left side is a temperature** (Temp) and **relative humidity (RH) sensor that publishes its Temp/RH data.**

- The **MQTT server (or message broker)** accepts the network connection along with application messages, such as Temp/RH data, from the publishers.

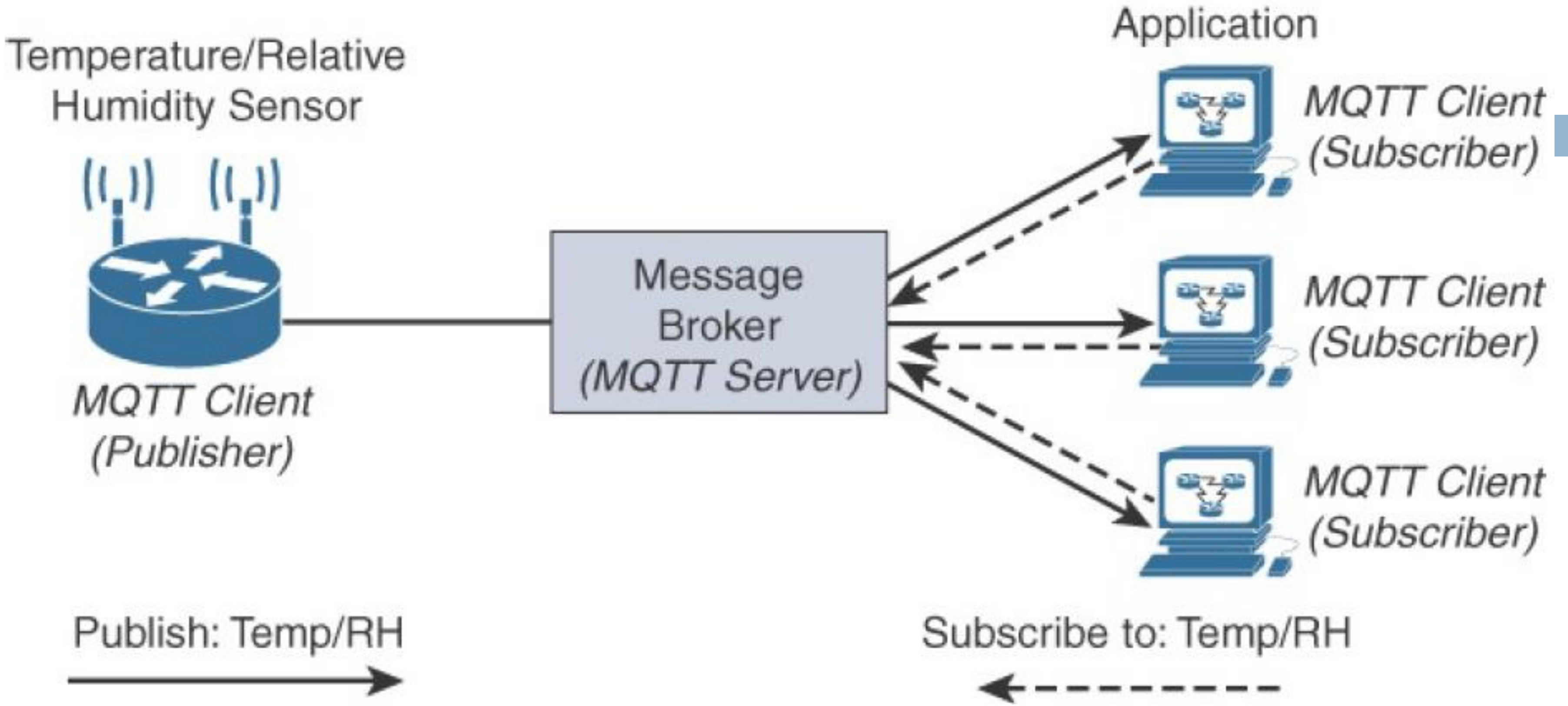

Figure 6.10: MQTT Publish/Subscribe Framework

- It also **handles the subscription and unsubscription process** and pushes the application data to MQTT clients acting as subscribers.

- The **application on the right side** of Figure 6-10 is an MQTT client that is a subscriber to the Temp/RH data being generated by the publisher or sensor on the left.

- This model, where subscribers express a desire to receive information from publishers, is well known.

- A great example is the **collaboration and social networking application Twitter.**

- With **MQTT,** clients can subscribe to all data (using a wildcard character) or specific data from the information tree of a publisher.

- In addition, the presence of a message broker in MQTT decouples the data transmission between clients acting as **publishers and subscribers**.

- In fact, **publishers and subscribers** do not even know (or need to know) about each other. A benefit of having this decoupling is that the MQTT message broker ensures that information can be buffered and cached in case of network failures.

- This also means that **publishers and subscribers** do not have to be online at the same time.
- **MQTT control packets** run over a **TCP transport** using port **1883.**
- TCP ensures an ordered, lossless stream of bytes between the MQTT client and the MQTT server.
- Optionally, MQTT can be secured using **TLS on port 8883**, and WebSocket (defined in RFC 6455) can also be used.

- **MQTT is a lightweight protocol** because each control packet consists of a 2-byte fixed header with optional variable header fields and optional payload.

- We should note that a **control packet** can contain a payload up to 256 MB. Figure 6.11 provides an overview of the **MQTT message format.**

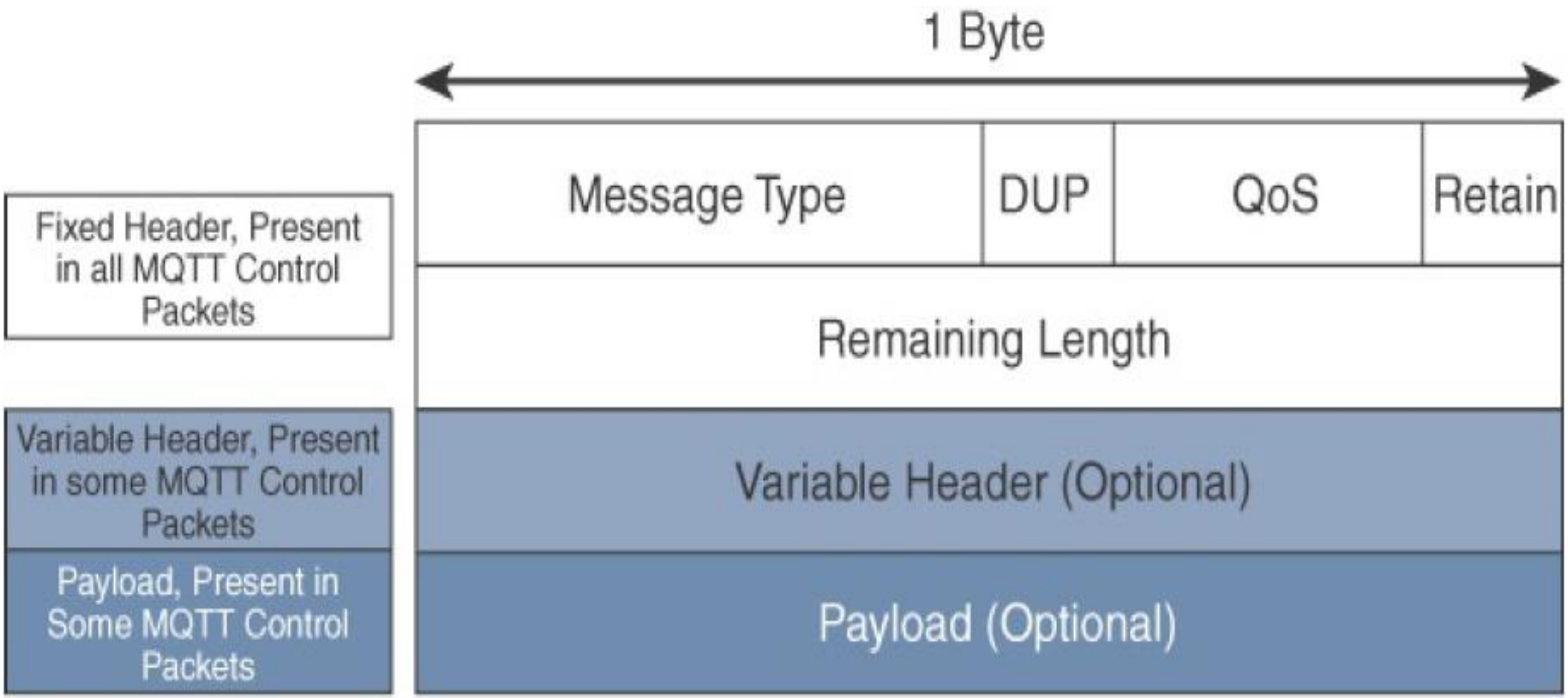

Figure 6.11: MQTT Message Format

- Compared to the **CoAP message format, MQTT contains** a smaller header of 2 bytes compared to 4 bytes for CoAP.

- The first **MQTT field in the header is Message Type**, which identifies the kind of MQTT packet within a message.

- **Fourteen different types** of control packets are specified in MQTT version 3.1.1.

- Each of them has a unique value that is coded into the Message Type field. Note that values 0 and 15 are reserved.

- MQTT message types are summarized in Table 6.2.

Message Type	Value	Flow	Description
CONNECT	1	Client to server	Request to connect
CONNACK	2	Server to client	Connect acknowledgement
PUBLISH	3	Client to server Server to client	Publish message
PUBACK	4	Client to server Server to client	Publish acknowledgement
PUBREC	5	Client to server Server to client	Publish received
PUBREL	6	Client to server Server to client	Publish release
PUBCOMP	7	Client to server Server to client	Publish complete
SUBSCRIBE	8	Client to server	Subscribe request
SUBACK	9	Server to client	Subscribe acknowledgement
UNSUBSCRIBE	10	Client to server	Unsubscribe request
UNSUBACK	11	Server to client	Unsubscribe acknowledgement
PINGREQ	12	Client to server	Ping request
PINGRESP	13	Server to client	Ping response
DISCONNECT	14	Client to server	Client disconnecting

Table 6.2 : MQTT Message Types

- The ***next field*** in the **MQTT header is DUP** (Duplication Flag).

- This flag, when set, allows the client to notate that the packet has been sent previously, but an acknowledgement was not received.

- The **QoS header field** allows for the selection of three different QoS levels.

- The next field is the **Retain** flag. Only found in a **PUBLISH** message, the **Retain** flag notifies the server to hold onto the message data

- This allows **new subscribers** to instantly receive the last known value without having to wait for the next update from the publisher.

- The last **mandatory field** in the MQTT message header is **Remaining Length**.
  - This field specifies the number of bytes in the MQTT packet following this field.

- **MQTT sessions** between each client and server consist of **four phases:** session establishment, authentication, data exchange, and session termination.

- Each **client connecting** to a **server** has a unique client ID, which allows the identification of the MQTT session between both parties.

- When the server is delivering an application message to **more than one client**, each client is treated independently.

- **Subscriptions** to resources generate **SUBSCRIBE/SUBACK** control packets, while **unsubscription** is performed through the exchange of **UNSUBSCRIBE/UNSUBACK** control packets.

- Graceful termination of a connection is done through a **DISCONNECT control packet**, which also offers the capability for a client to reconnect by re-sending its client ID to resume the operations.
- A **message broker** uses a topic string or topic name to filter messages for its subscribers. When subscribing to a resource, the subscriber indicates the one or more topic levels that are used to structure the topic name.
- **The forward slash (/) in an MQTT** topic name is used to separate each level within the topic tree and provide a hierarchical structure to the topic names.

- Figure 6.12 illustrates these concepts with **adt/lora.adeunis** being a topic level and **adt/lora/adeunis/0018B2000000023A** being an example of a topic name.

- Wide **flexibility** is available to clients subscribing to a topic name.

- An exact topic can be **subscribed to, or multiple topics** can be subscribed to at once, through the use of wildcard characters.

- A **subscription** can contain one of the wildcard characters to allow subscription to multiple topics at once.

- The **pound sign (#)** is a wildcard character that matches any number of levels within a topic.

- The multilevel wildcard represents the **parent** and any number of child levels.

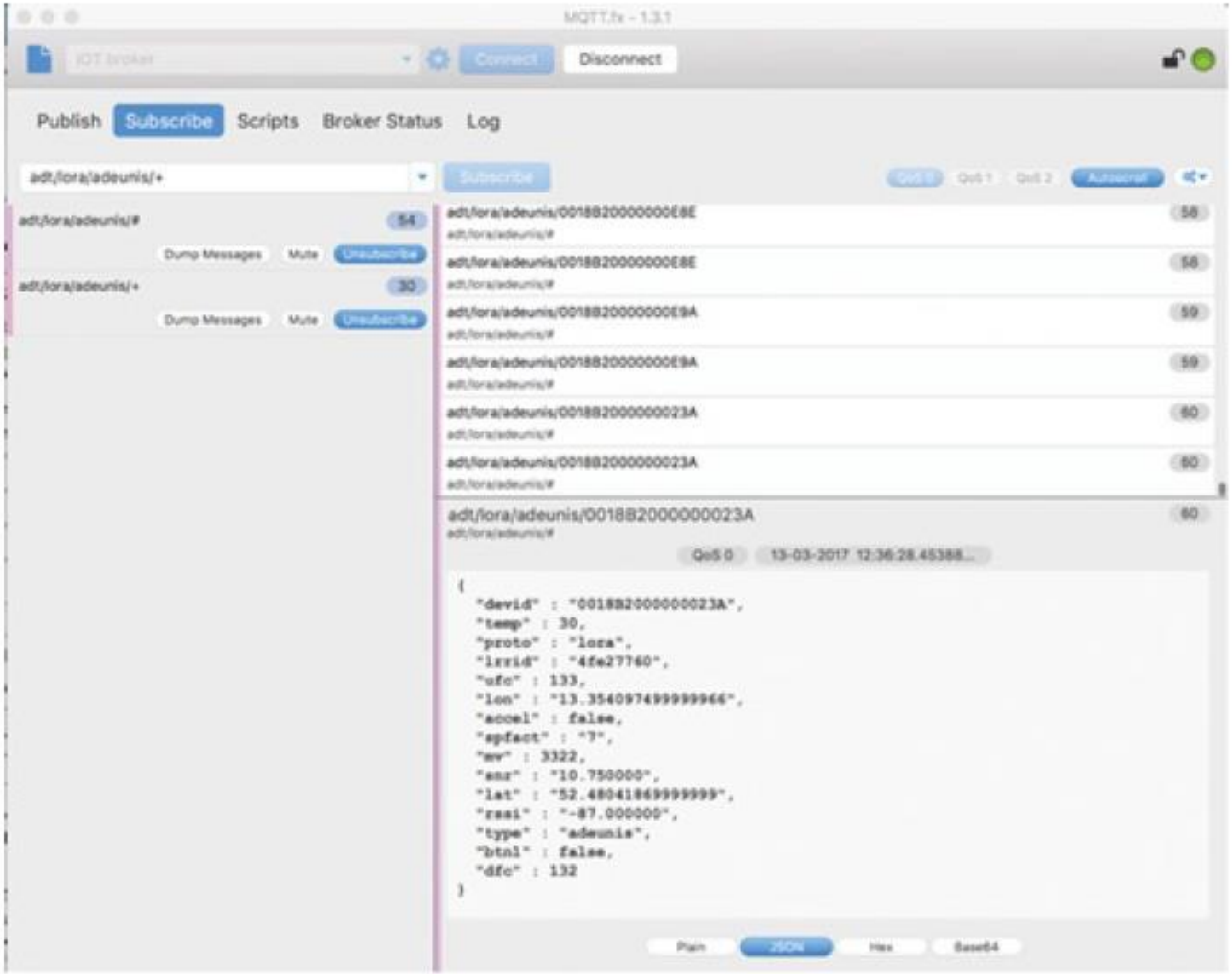

Figure 6.12: MQTT Subscription Example

- For ex : subscribing to **adt/lora/adeunis/#** enables the reception of the whole subtree, which could include topic names such as the following:

  - adt/lora/adeunis/0018B20000000E9E
  - adt/lora/adeunis/0018B20000000E8E
  - adt/lora/adeunis/0018B20000000E9A

- The plus sign (+) is a wildcard character that matches only one topic level.

- For ex : adt/lora/+ allows access to adt/lora/adeunis/ and adt/lora/abeeway but not to adt/lora/adeunis/0018B20000000E9E.

- **PINGREQ/PINGRESP** control packets are used to validate the connections between the client and server.

- Similar to **ICMP pings** that are part of IP, they are a sort of keepalive that helps to maintain and check the TCP session.

- **Securing MQTT** connections through TLS is considered optional because it calls for **more resources on constrained nodes.**

- When **TLS is not used**, the **client sends a clear-text username and password** during the connection initiation. MQTT server implementations may also accept anonymous client connections(with the username/password being “blank”).

- When **TLS is implemented**, a client must validate the server certificate for proper authentication.

- The **MQTT protocol offers** three levels of quality of service (QoS).

- **QoS for MQTT** is implemented when exchanging application messages with publishers or subscribers, and it is different from the IP QoS that most people are familiar with.

- The **delivery protocol** is concerned solely with the delivery of an application message from a single sender to a single receiver.

- These are the three levels of MQTT QoS:

- **QoS 0**:
  - This is a best-effort and unacknowledged data service referred to as **"at most once"** delivery.
  - The publisher sends its message one time to a server, which transmits it once to the subscribers.

- **QoS 1**:
  - This **QoS level** ensures that the message delivery between the publisher and server and then between the server and subscribers occurs at least once.
  - In **PUBLISH and PUBACK** packets, a packet identifier is included in the variable header.
  - If the message is not acknowledged by a PUBACK packet, it is sent again.
  - This level guarantees "at least once" delivery.

- **QoS 2**:
  - This is the highest QoS level, used when neither loss nor duplication of messages is acceptable.
  - There is an increased overhead associated with this **QoS level** because each packet contains an optional variable header with a packet identifier.

- Figure 6.13 provides an overview of the MQTT QoS flows for the **three different levels**.

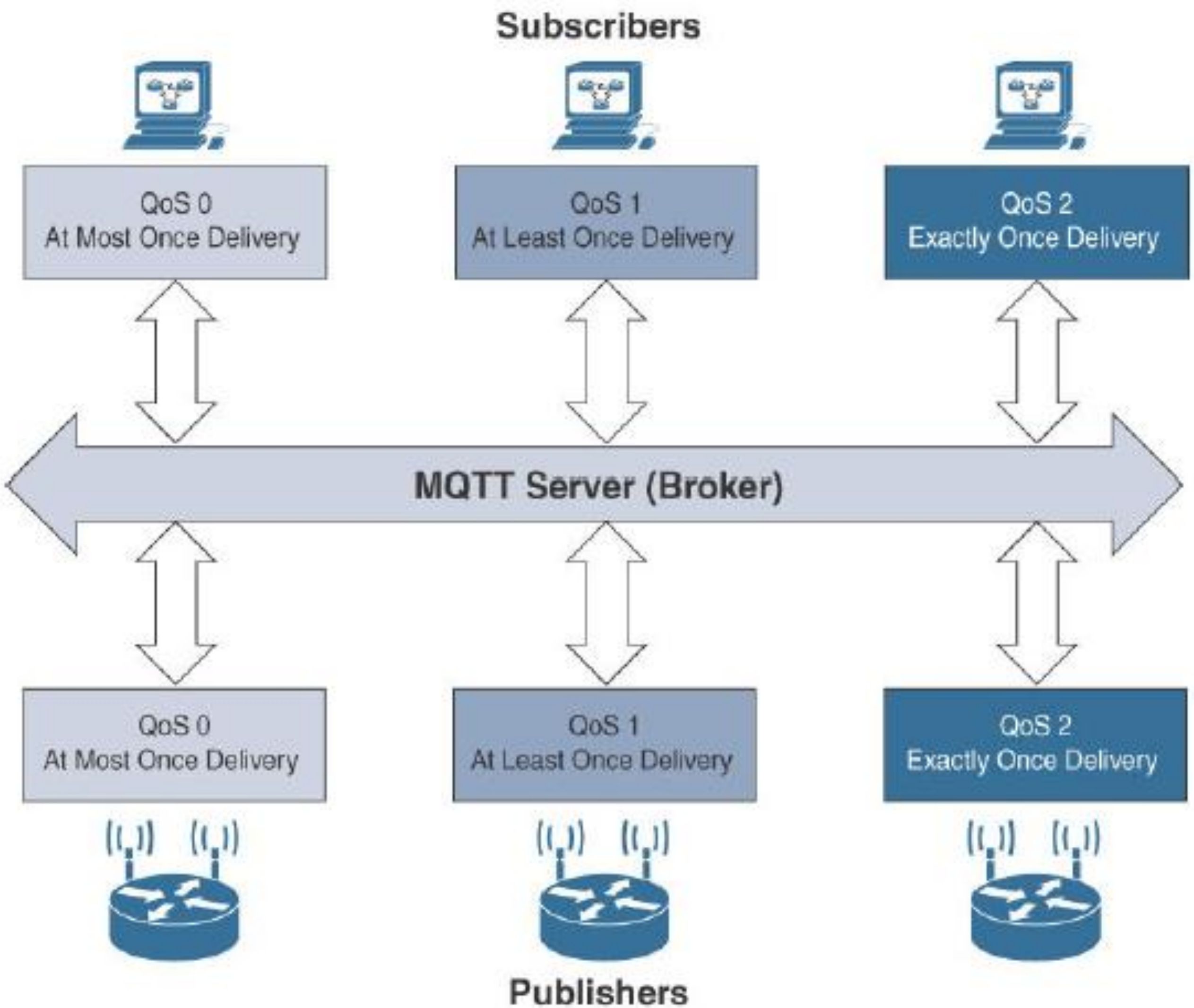

Figure 6.13: MQTT QoS Flows

Factor	CoAP	MQTT
Main transport protocol	UDP	TCP
Typical messaging	Request/response	Publish/subscribe
Effectiveness in LLNs	Excellent	Low/fair (Implementations pairing UDP with MQTT are better for LLNs.)
Security	DTLS	SSL/TLS
Communication model	One-to-one	many-to-many
Strengths	Lightweight and fast, with low overhead, and suitable for constrained networks; uses a RESTful model that is easy to code to; easy to parse and process for constrained devices; support for multicasting; asynchronous and synchronous messages	TCP and multiple QoS options provide robust communications; simple management and scalability using a broker architecture
Weaknesses	Not as reliable as TCP-based MQTT, so the application must ensure reliability.	Higher overhead for constrained devices and networks; TCP connections can drain low-power devices; no multicasting support

Table 6-3 *Comparison Between CoAP and MQTT*