

CS433: Internet of Things

NCS463: Internet of Things

Dr. Ahmed Shalaby

<http://bu.edu.eg/staff/ahmedshalaby14>

Communications

❑ Introduction

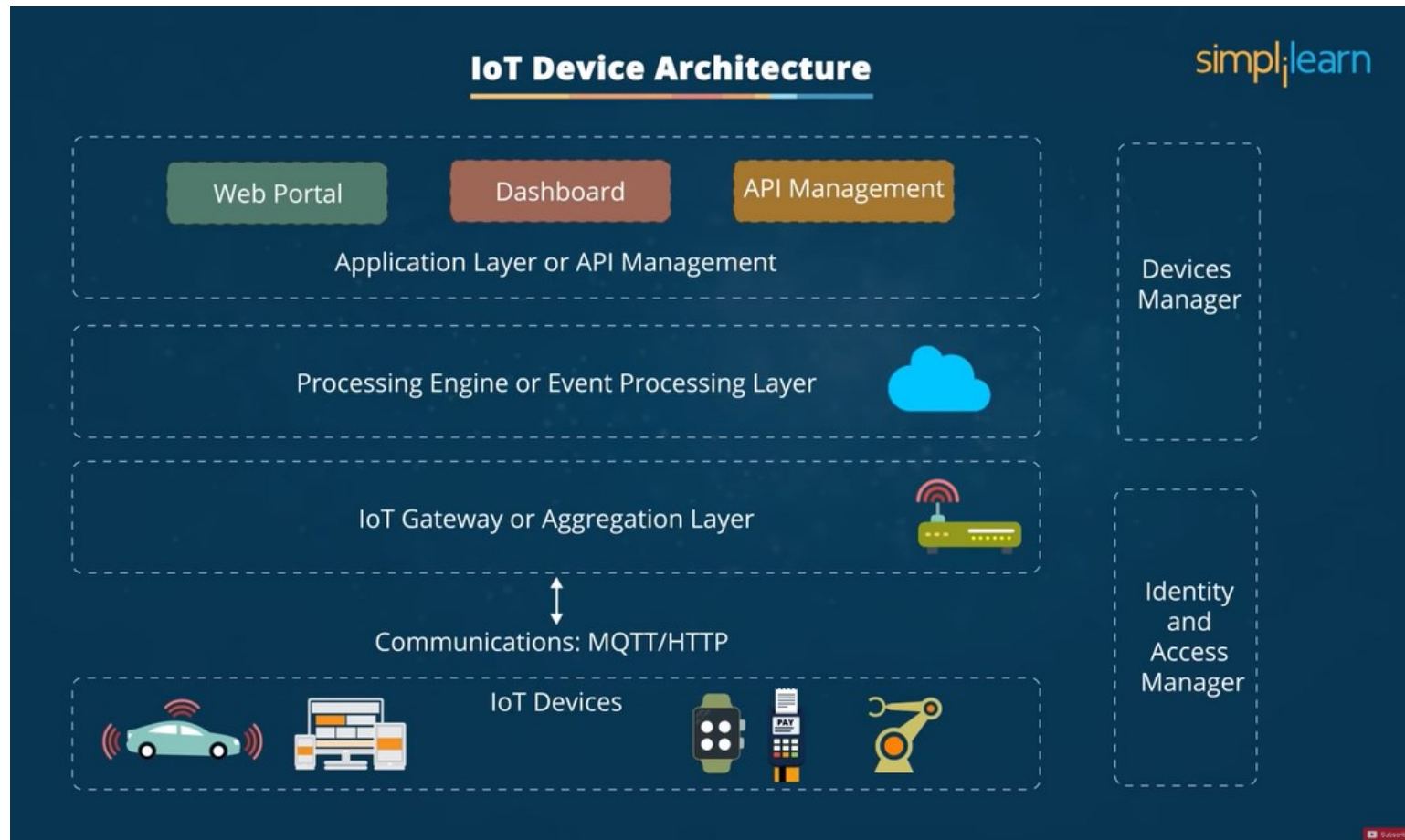
❑ MQTT

- Publish/Subscribe.
- Topics & Subscriptions.
- Quality of service levels.
- Sessions / Retained message / Wills.

❑ MQTT vs. HTTP

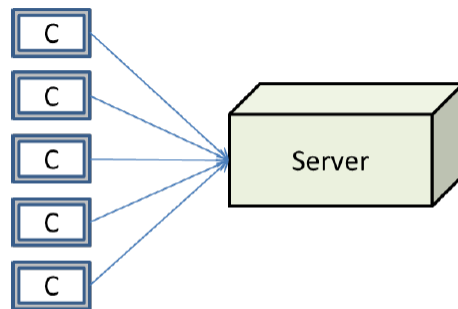
Source: [MQTT Broker Introduction](#)

Internet of Things – Architecture

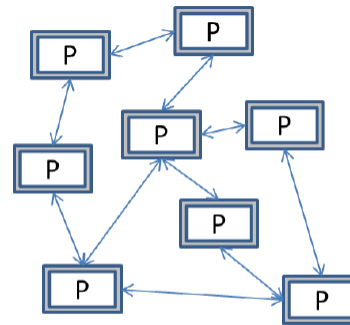


Communications: Intro

- ❑ IoT system interact and cooperate is seen as the “interaction paradigm”. **interaction paradigms** (such as: Client-Server, Peer-to-Peer, Message Passing, etc...)
- ❑ The C-S paradigm is based on a very simple interaction between the clients and the server; a client sends a request (essentially a structured message) to a server and expects (non-blocking) a response from the server.



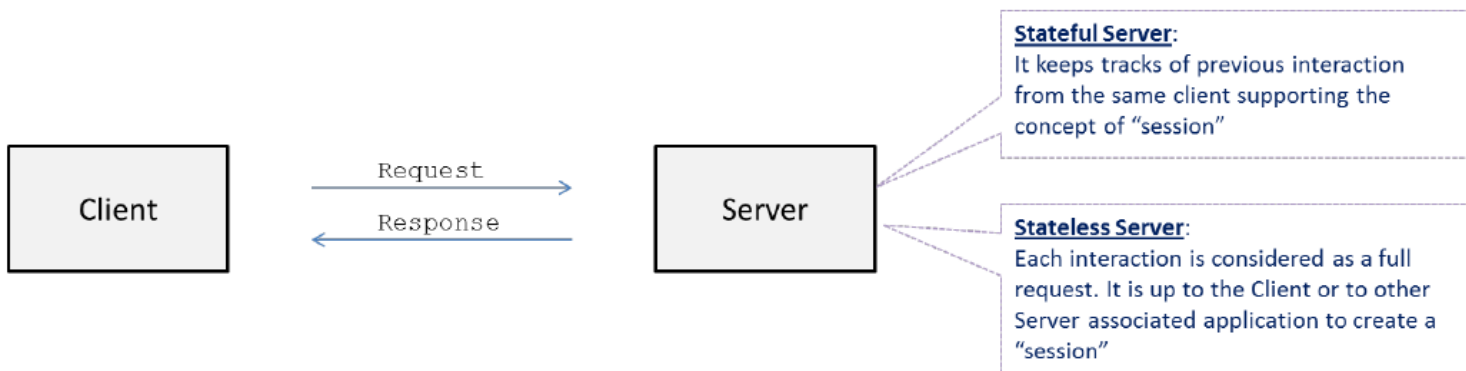
a) Client – Server
Interaction Paradigm



b) Peer to Peer
Interaction Paradigm

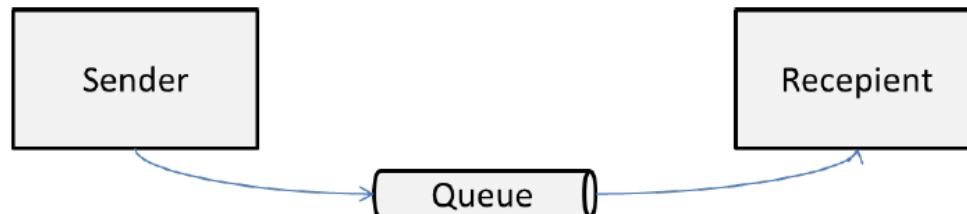
Communications: Intro

- ❑ The C-S paradigm is based on a very simple interaction between the clients and the server; a **client sends a request** (essentially a structured message) to a server and expects (non-blocking) a **response from the server**.
- ❑ The Server could be **Stateful** or **Stateless**; the difference is whether the system keep **track** of the previous interactions with clients and has a finite state machine associated to the interactions going on. A **stateful server is more complicated** to manage especially if many clients are requesting in parallel the functions of the server.



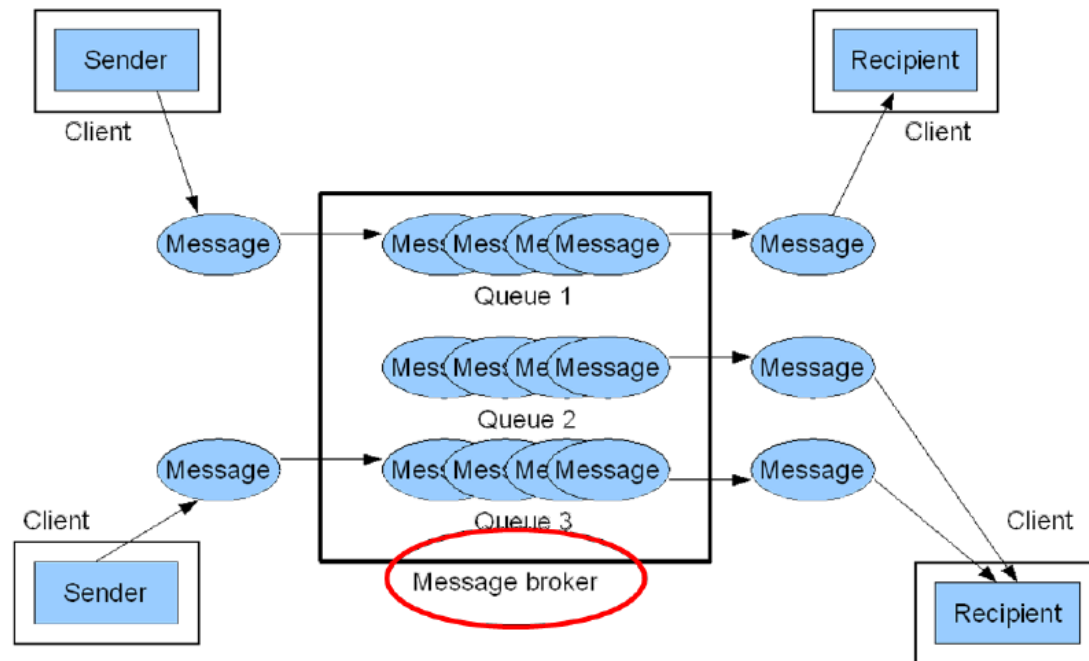
Communications: Intro

- ❑ Message Passing model is based on a very simple organization in which a sender **forwards messages** by means of a **queue** to a recipient.
- ❑ Message Passing: can be interpreted as an “***interrupt with data***”, i.e., events are queued for processing together with associated data. Receiving tasks can then **operate on data received when the message is taken from the queue**. However,
 - an **explicit treatment** of communication is needed (“send message to task n”)
 - **Assembling and disassembling** of messages can be cumbersome. Management of data (data are copied from data structures in the source of the message, transmitted to a queue and then copied to a data structure at the sink destination).



Communications: Intro

- In large Message Passing systems, senders and recipients share a common infrastructure made out of queues, these can be organized as a sort of Message Broker in charge for receiving and dispatching (in an asynchronous manner) messages between sources and sinks.



Communications: Intro

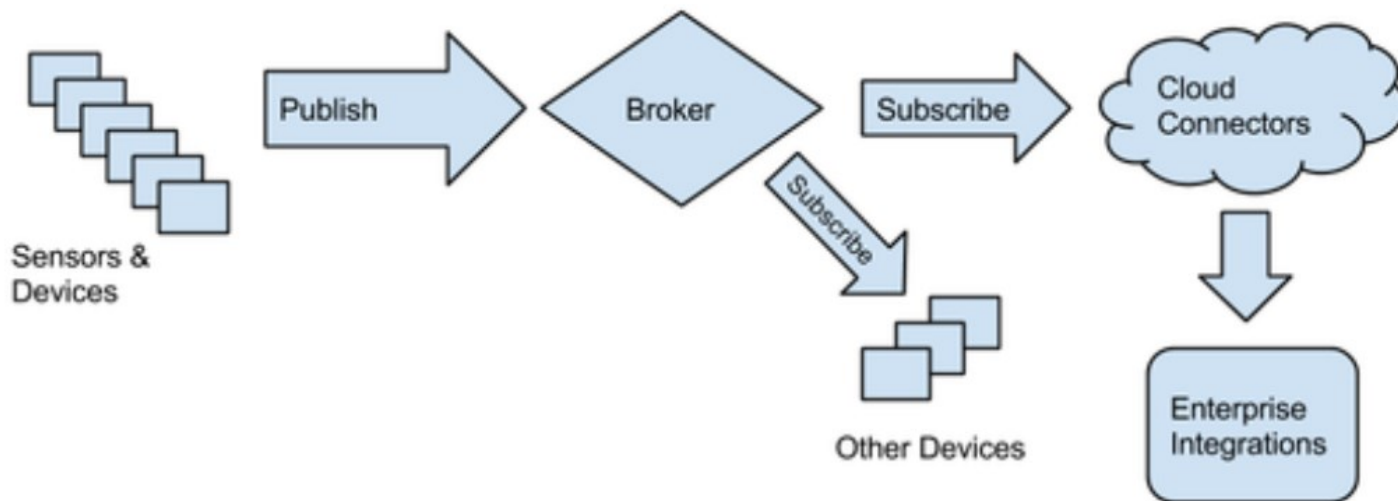
- There are at least *three important features*:
 - The model can be synchronous or asynchronous, i.e., in the first case, the sender ***waits for the receiver to deal with the message***, in the second case ***once a message has been sent, the sender can proceed with its computation***. The choice of synchronicity and asynchronicity depends on the applications requirements. The introduction of queues in order to store messages for later processing is a case in which asynchronous mechanisms are implemented.
 - Routing, the system (especially by means of the Broker Functions) can **route the messages even if the sender has not provided a full path to the destination**.
 - Conversion, the Broker functions can also make compatible different **messages format** in such a way to **support the interoperability between different messaging systems**.

Communications: Intro

- ❑ IoT network traffic falls into two categories: telemetry and telecommand.
 - Telemetry is the act of gathering telemetrics, or sending data over long distances. Usually telemetry involves sending data from many dumb sensors to a smart hub of some sort.
 - Telecommand is the act of sending commands across a network.
- ❑ Most telemetry protocols are modeled as **publish/subscribe** architecture. Sensors connect **to a broker and periodically publish** their readings to a topic. A central cluster of servers (the cloud) will then **subscribe to the topic and process** sensor readings in real-time.

Communications: Intro

- A typical enterprise arrangement will have thousands or millions of sensors sending telemetrics to a handful of servers that split up the task of processing the data.



Communications: MQTT

- ❑ **MQTT** (Message Queue Telemetry Transport) is more and more becoming the standard messaging protocol for IoT messaging.
- ❑ MQTT was developed by IBM in 1999.
- ❑ Since 2014 MQTT is an OASIS standard messaging protocol.
- ❑ MQTT is very lightweight. As all workload is done by the broker.
 - The term 'workload' addresses the amount of load the CPU of either the broker or client has to work off.
 - Also, the data exchange using MQTT doesn't require a lot of CPU.
 - Also, MQTT can be very easily implemented on the client-side. There is a large variety of use cases and plenty of client code libraries.
- ❑ MQTT technology can be found in automotive, manufacturing, telecommunications, and many more.

Communications: MQTT

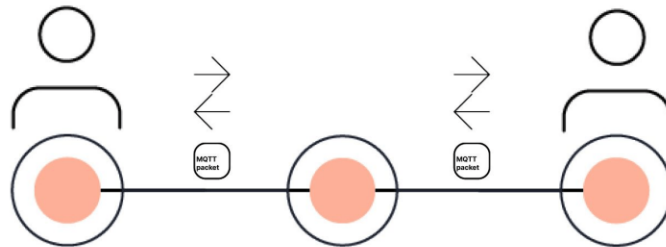
- ❑ In MQTT the *clients* (the communication participants such as sensors, machines, and applications) *do not directly communicate* with each other but via a so-called *broker*.
- ❑ This is possible because of the *efficient protocol format*.

Communications: MQTT

- ❑ The MQTT broker **receives** the data from the data senders, filters the data packets, and **forwards** them to the receiving clients.
- ❑ Clients **sending** data are called "**Publishers**".
- ❑ Clients which **receive data** are called "**Subscribers**".

Communications: MQTT

- ❑ In fact, an MQTT system enables receiving clients to become publishers as well.
- ❑ As MQTT messages can be **very small** depending on the payload size, they can be sent easily even by small devices.



Communications: MQTT

Quality of Services

- ❑ MQTT can send messages with different *"Quality of Service" levels*. This means the publisher and subscriber can choose how reliably a message will be sent.
- ❑ QoS is between the client and broker, or broker and client only. It is **not between client and client**.
- ❑ A receiver sets its QoS level when connecting. For a publisher, the broker is the receiver.
- ❑ For a publisher, the broker is the receiver. When the broker, then, forwards the message, the subscriber is the receiver.



Final QoS		Subscriber QoS		
		0	1	2
Publisher QoS	0	0	0	0
	1	0	1	1
	2	0	1	2

Communications: MQTT

Quality of Services

❑ QoS Levels

- ❑ **QoS level 0** means messages are sent without any confirmation from the receiver. This means it is technically possible for a message to get lost, given an unreliable connection. “fire and forget”-level.
- ❑ **QoS level 1** means the receiver must send a confirmation to let the sender know that the message was received. However, with QoS 1, it is possible that the receiver gets a single message multiple times. This QoS level ensures that a message makes it from sender to receiver but does not ensure that it is received exactly once.
- ❑ **QoS level 2** uses a four-step communication process to ensure a message is sent exactly once only, which can be important depending on the use case.



Communications: MQTT

- ❑ **MQTT** is designed especially for **IoT** - the Internet of things, or machine-to-machine messaging with low transmission capacity.
- ❑ **MQTT** leaves a small footprint on the network. For example, the **HTTP** header uses about **8,000 bytes**, while the **MQTT** protocol uses only **two bytes and a few lines of code**. Consider it in a factory setting where many IoT devices are interacting and generating workloads.
- ❑ Comparing HTTP and MQTT shows, e.g., that **MQTT delivers messages many times faster and more efficiently** than the HTTP protocol.
- ❑ Using **QoS** levels, to ensure data delivery, and the ability to queue messages, **MQTT is a perfect fit for connections that cannot be 100% guaranteed to be stable**.

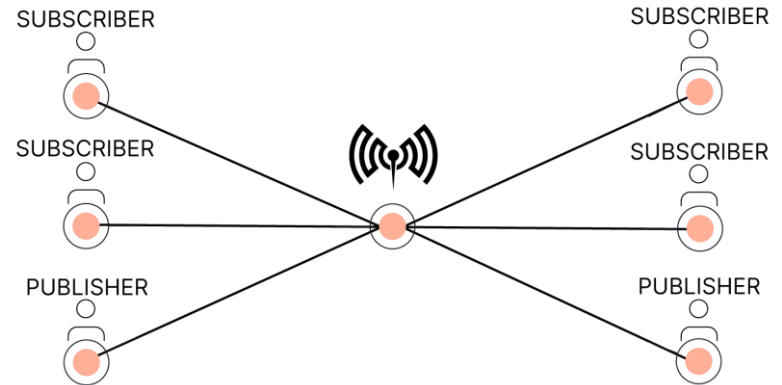
Communications: MQTT

MQTT Setup

- A typical MQTT setup includes one broker and as many clients as you want.

- **MQTT principle:**

- A publisher (client) **sends a message** (including data) to a broker related to a topic.
- Other clients **subscribe to this topic** to receive the messages.
- The **broker filters the incoming messages** and checks whether the subscribers have the necessary rights. The broker then **forwards the messages to the subscribers**.
- Several **publishers** can send messages on the **same topic**.
- Multiple **subscribers** can subscribe to the **same topic**.



Communications: MQTT

MQTT Setup

❑ *MQTT principle:*

- The broker **decouples the clients**. The clients, the publishers, and subscribers do not communicate with each other directly but always through the broker as an intermediary.
- As a result, with an increasing number of clients, the **connection count** still only *grows linearly*.

If all clients had to connect for communication, the connection count would grow **exponentially** making the system more complex, harder to manage, and more prone to errors and failures.

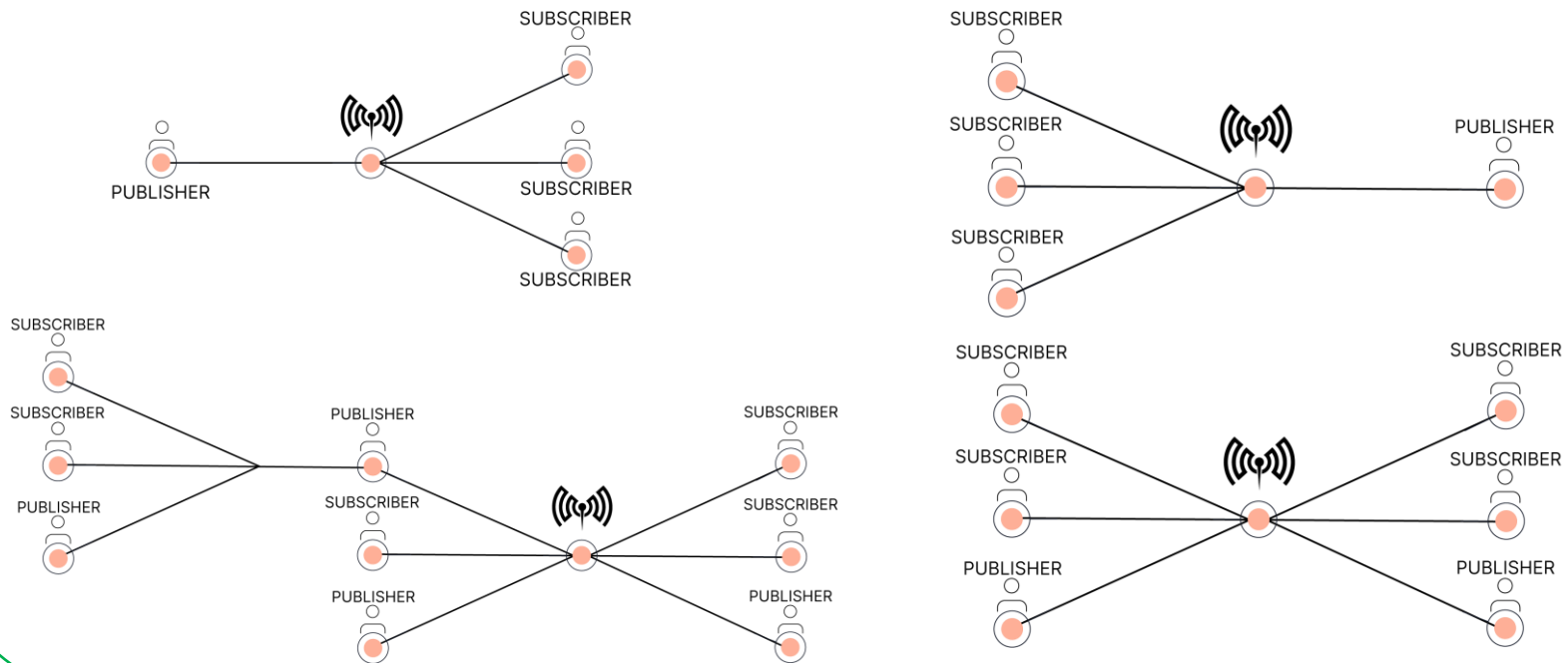
❑ *MQTT Concept:*

- PUBLISH / SUBSCRIBE
- CLIENT / BROKER

- ### ❑ *Message vs. packet:* messages and packets - are the same and the terms can be used interchangeably.

Communications: MQTT

- All connections are only with the broker. The broker is, thus, the central hub in MQTT communication. Consequently, ***most of the workload is carried by the broker as well.***
- The load balancer is the main character that moves messages/packets within the broker.

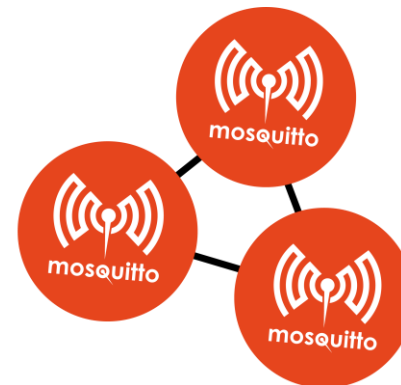


Communications: MQTT

- ❑ In an MQTT environment, the broker is the central entity and handles most of the workload. Therefore, **client devices have to do minimal processing** and use only minimal bandwidth.
- ❑ MQTT is very easy to implement on the client side. It's a perfect fit for constrained devices with **limited resources**.
- ❑ When considering how much an MQTT Broker is used and the work it has to perform, **it is not so much about the number of clients** that you should focus on but rather the **amount of data sent per second**, the "traffic".
- ❑ A different scenario that can incur a lot of traffic and workload for the broker. if you want to transfer pictures every millisecond using QoS level 2 (QoS2) - **a Quality of Service level occupies the resources** of a broker way more than other Quality of Service levels.
- ❑ The **Mosquitto broker** is the **most efficient broker offered worldwide**.
 - Hosting: You can install and operate the open-source Mosquitto broker yourself.

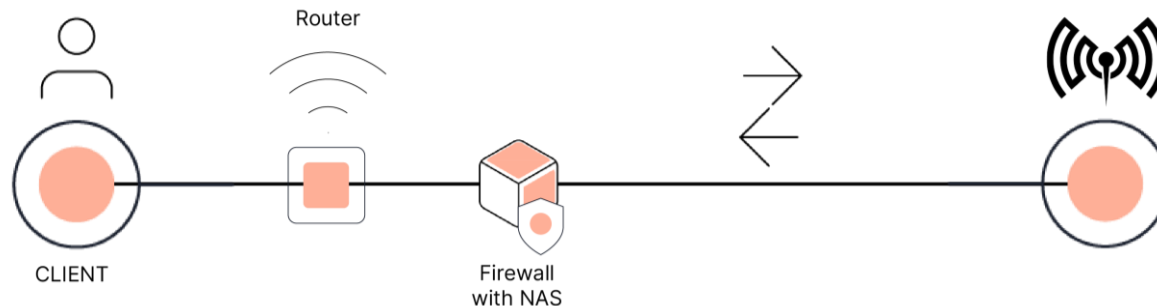
Communications: MQTT

- ❑ **High Availability** (HA) is the ability of a system to *cope with the loss of a central component*. This is typically achieved through **redundancies**. HA ensures that even if the (main) broker is lost, because of a hardware failure, the MQTT communication continues to work properly as another broker seamlessly takes over.
- ❑ This means that the systems should work without failure. The **MQTT Mosquitto** broker is designed to be **stable** and **consistently functional** by **clustering**.
- ❑ Clustering provides prevention of disadvantages whenever a broker might break down, e.g. hardware failure. Therefore **usually three or more brokers work together in a cluster**.
- ❑ In the case of a breakdown of broker number one, the so-called "**load balancer**" shifts the workload to a passive broker number two.
- ❑ To be safe, there is another **passive broker clustered**.



Communications: MQTT

- ❑ The **Payload** each message carries must not be **encoded nor decoded by the broker**. The broker simply receives messages from the publisher, filters the messages, and delivers them to the subscribing clients.
- ❑ Overall, it's possible to send up to **256MB** in each Payload.
- ❑ Setup: An MQTT client is located after a router, using the NAT (Network Address Translation) to transmit from a private network address to a public address.



Communications: MQTT

Security:

- ❑ it's always the **end user that has to make up his mind** about what level of security has to be chosen and how to implement it.
- ❑ The use of **ACLs (Access Control Lists)** allows restriction of subscriptions and publishing of clients.
- ❑ Usually, the broker supports common security measurements, e.g. TLS. Still, the end-user has to make sure that the chosen measurements fit the security environment the broker itself is part of.
- ❑ The default **secured MQTT** broker port is 8883.
- ❑ The standard **unsecured port** is 1883.
- ❑ TCP is embedding **TLS (Transport Layer Security)**, the successor of SSL (Secure Sockets Layer). Allowing MQTT packets to be transmitted via encrypted pipes.
- ❑ SASL Framework (**Simple authentication and security layer**) for example. SASL provides authentication options, data integrity checking, and encryption.

Communications: MQTT

Security:

- ❑ **Cryptology** ensures privacy and *integrity*. Also *authenticity* with the use of *certificates*.
- ❑ Advanced Encryption Standard [**AES**] is the most widely adopted encryption algorithm. There is hardware support for AES in many processors, but not commonly for embedded processors. The encryption algorithm ChaCha20 [**CHACHA20**] encrypts and decrypts *much faster in software*, but is not as widely available as AES.
- ❑ The highest security levels can be achieved using *client certificates (x509)*.

Communications: MQTT

Security:

- ❑ TLS **protects all parts of an MQTT packet**, not only the payload. Encrypting just the payload is also able. But again, encrypting a payload is done **at the application level, not the broker.**
 - Following **encrypted Payloads can be sent without broker configuration needed**. The broker just delivers packets. Subscribing clients on the other end must be able to decrypt the Payload.
 - The broker can identify the client on three different pieces of information given. The **clientId**, a **username**, and **password**, or a **certificate of the client**.
 - each client must have a **unique clientId**. The clientId can be set by the client, but it does not always have to. If a client connects to the broker and another client session already exists with the same clientId, the old session **will be kicked** out and taken over.

Communications: MQTT


Traditional Client / Broker Model:


- ❑ Publisher and subscriber don't exchange IP addresses and ports.
- ❑ There is no need for the publisher and subscriber to run at the same time. Downtimes do not mean that messages are lost.
- ❑ Fluency, publishing or receiving messages/packets the operations of the publisher and subscriber run.
- ❑ Any established connection is kept open by the broker until the client sends a DISCONNECT command or the connection breaks up.
- ❑ The architecture can be scaled easily without affecting existing client devices. This makes it easy to work and change architecture.
- ❑ Of course, MQTT can process messages event-driven. Most client libraries work asynchronously. While waiting for a message or publishing it, other tasks are not blocked.

Communications: MQTT

Client-Broker Connection:

- ❑ To publish messages, you must only know the hostname/IP and port of the broker.
- ❑ To receive messages, you must know the hostname/IP and port of the broker and the topic you want to subscribe to.
- ❑ To establish a connection there is always a client-initiated CONNECT packet needed, that is sent by the client to the broker.
- ❑ The broker in return responds by sending a so-called CONNACK packet (Acknowledge connection request) and the CONNACK return code.
- ❑ After the CONNECT and CONNACK packet are exchanged, the connection is enabled.

CONNACK 		MQTT-Packet
sessionPresent		true
return Code		0

CONNECT 		MQTT-Packet
clientId		„client1“
cleanSession		true
username		„user1“
password		„pw1“
lastWillTopic		„/car/door“
lastWillQos		0
lastWillMessage		„connection disabled“
lastWillRetain		false
keepAlive		120

Communications: MQTT

Client-Broker Connection:

- ❑ The *clientId* identifies each MQTT client that is connecting to an MQTT broker. each client must have a unique clientId. In that case, the client sends a blank clientId to the broker, and the *broker generates a unique id* for it whilst the session is open.

- ❑ Persistent session:
 - The client *must receive all messages*.
 - The broker *must queue the missed messages*.
 - The client must resume all QoS1 and QoS2 messages after reconnecting to the broker.

- ❑ Clean Session:
 - The client only needs to publish messages to topics.
 - It doesn't need to subscribe to topics.
 - The client *must not resume with messages* it missed.

Communications: MQTT

Client-Broker Connection:

- ❑ LastWillMessage feature allows you to keep clients of your IoT system informed of any unexpected disconnections. The broker distributes the lastWillMessage when one of these events happens:
 - The broker detects an *I/O error or network failure*.
 - There is *no communication* between client and broker within the defined keepAlive period.
 - The client *breaks off without sending a DISCONNECT* packet.
 - The broker *shuts down* the network connection when an error occurs.
- ❑ The lastWill message is sent by the broker on behalf of the client when a disconnect occurs without a DISCONNECT packet before the actual break-off.
- ❑ lastWillTopic: The MQTT topic that clients subscribed to, who will receive the lastWill message.
- ❑ lastWillRetain: Indicates whether the message will be a retained one or not.

Communications: MQTT


Client-Broker Connection:

- ❑ Retained messages should be considered to be a "**last known good**" value.
 - The typical example is a sensor that publishes when a door is opened/closed. If we didn't have retained messages, a subscriber would have to wait until the door was opened or closed, and hence a message published, before it knew the state of the door.
 - This could happen very infrequently. With a retained message, the subscriber gets to know the state of the door as soon as it subscribes - but it also knows that this is not a "fresh" message.
- ❑ As the client reconnects and publishes a message all subscribers will know that the client is connected again.
- ❑ **keepalive** feature can be enabled, it contains two functions:
 - Network outage or peer death recognition. The broker or client must disconnect.
 - Hold on to the connection in case **no interaction** took action for a specific time. A **maximum length of the time interval** is defined for each client request to connect with the broker. Within the interval, it's ok that the client and broker do not transmit messages.

Communications: MQTT

PUBLISH:

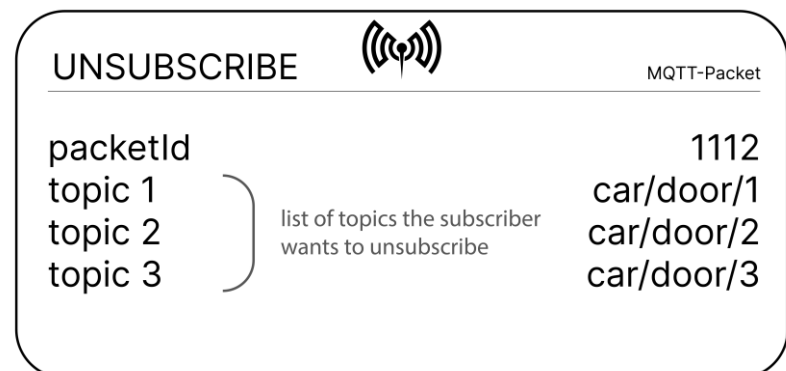
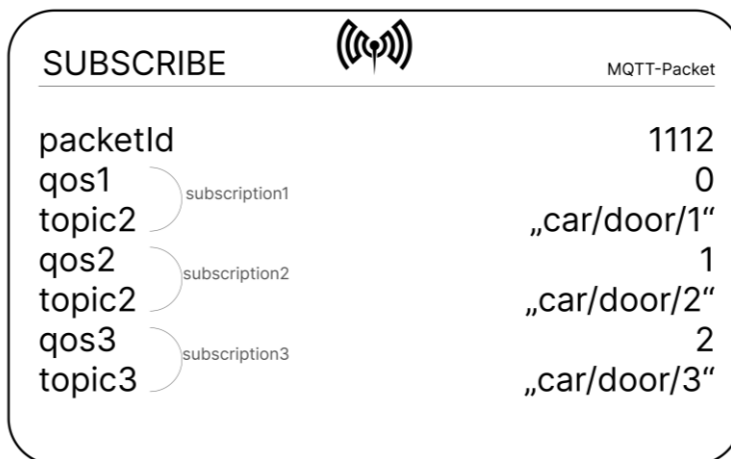
- ❑ **packetId**: Identifies a message. The packetId is set by the client library and broker.
- ❑ **topicName**: The topic is set using a simple string. Topics are treated hierarchically. To delimiter, a slash "/" symbolizes a separation. Allowing the client organizational structure, much like a common filesystem.
- ❑ **QoS**: (Quality of Service) defines a certain level of service that enables the publisher to make sure that the certainty and quantity of a subscriber receiving a sent message.
- ❑ **retainFlag**: With retained messages (retainFlag=true), you immediately find out, because the retained message gives you the last status.
- ❑ **dupFlag**: indicates the duplicate of a message. A message with dupFlag was resent.

PUBLISH		MQTT-Packet
packetId		1112
topicName		„car/door“
qos		1
retainFlag		false
payload		„open“
dupFlag		false

Communications: MQTT

SUBSCRIBE:

- ❑ **packetId**: Identifies a message.
- ❑ **subscriptions**: Each subscription consists of a topic and a QoS level.
 - Subscribing to several topics simultaneously is an option that's possible.
 - The broker must acknowledge each SUBSCRIBE packet. Therefore the broker sends a **SUBACK** packet (subscribe acknowledge) to the client.
- ❑ **Unsubscriptions** The UNSUBSCRIBE packet contains the topics you want to unsubscribe from.



Standard: MQTT

<https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf>

Section: 2.1.2 MQTT Control Packet type

Position: byte 1, bits 7-4.

Represented as a 4-bit unsigned value, the values are shown below

Table 2-1 MQTT Control Packet types

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Connection request
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment (QoS 1)

Standard: MQTT

Section: 2.1.2 MQTT Control Packet type

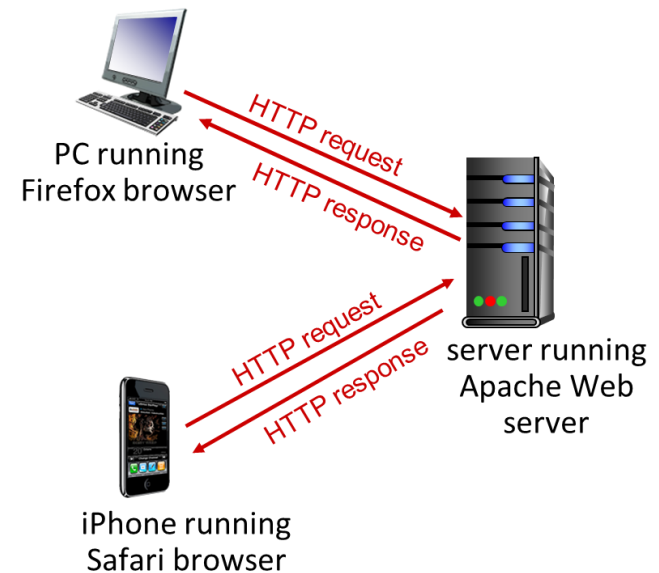
Position: byte 1, bits 7-4.

Represented as a 4-bit unsigned value, the values are shown below

PUBREC	5	Client to Server or Server to Client	Publish received (QoS 2 delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (QoS 2 delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (QoS 2 delivery part 3)
SUBSCRIBE	8	Client to Server	Subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server or Server to Client	Disconnect notification
AUTH	15	Client to Server or Server to Client	Authentication exchange

Communications: HTTP

- HTTP is the most widely used and available protocol. Almost all computing devices with a TCP/IP stack have it.
- HTTP: hypertext transfer protocol
- Web's application-layer protocol
- client/server model:
 - **client**: browser that requests, receives, (using HTTP protocol), and “displays” Web objects
 - **server**: Web server sends (using HTTP protocol) objects in response to requests



Communications: HTTP

HTTP uses TCP:

- ❑ client initiates TCP connection (creates socket) to the server, port **80**
- ❑ server accepts TCP connection from client
- ❑ HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- ❑ TCP connection closed

HTTP is “stateless”

- ❑ The server maintains no information about past client requests

HTTP connections: two types

- ❑ **Non-persistent** HTTP: at most one object sent over TCP connection.
- ❑ **Persistent** HTTP: multiple objects can be sent over a single TCP connection between the client, and the server.

Communications: HTTP

(containing text, references to 10 jpeg images)



1a. HTTP client initiates TCP connection to HTTP server (process) at `www.someSchool.edu` on port 80

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `someDepartment/home.index`

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects



1b. HTTP server at host `www.someSchool.edu` waiting for TCP connection at port 80 “accepts” connection, notifying client

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

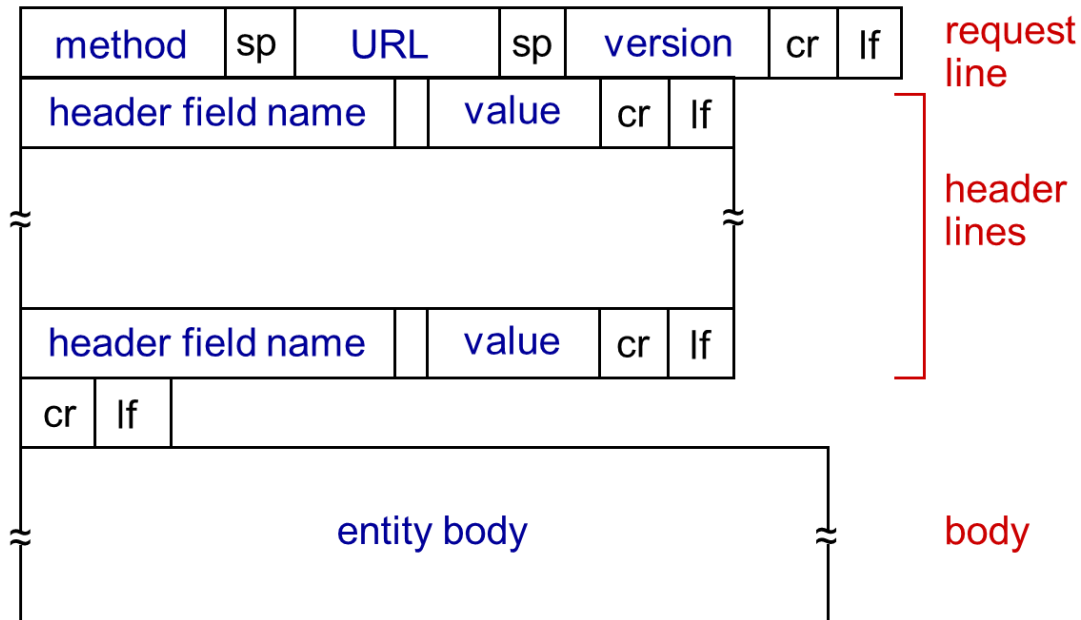
4. HTTP server closes TCP connection.

time

time

Communications: HTTP

HTTP request message: general format:



- ❑ Method: GET, PUT, HEAD, POST
- ❑ Status Codes: 200 OK (request succeeded), 404 Not Found (not found on this server)
- ❑ Header size: 8 k bytes

Communications: MQTT vs. HTTP

- HTTP is the most widely used and available protocol. Almost all computing devices with a TCP/IP stack have it. In addition, because HTTP and MQTT are both based on TCP/IP, developers need to choose between them.

	MQTT	HTTP
Design orientation	Data centric	Document centric
Pattern	Publish/subscribe	Request/response
Complexity	Simple	More complex
Message size	Small, with a compact binary header just two bytes in size	Larger, partly because status detail is text-based
Service levels	Three quality of service settings	All messages get the same level of service
Extra libraries	Libraries for C (30 KB) and Java (100 KB)	Depends on the application (JSON, XML), but typically not small
Data distribution	Supports 1 to zero, 1 to 1, and 1 to n	1 to 1 only