Stefano Chessa

Mobile and cyber-physical systems

IoT protocol stacks

Conventional Internet protocol suite		
Layer	Protocol	Features
Application	HTTP	Application-level access to information, client/server
Transport	TCP/UDP	Communication channels with guarantees (TCP); just an interface to IP (UDP)
Network	IP	Addressing & routing; best-effort

Internet protocol suite

IoT & internet

- Things must be connected to the Internet to become «IoT» devices
- To connect to Internet, they must adopt the internet protocol suite (TCP/IP + some application layer, e.g. HTTP)
- However, the Internet stack is thought for resource-rich devices
  - No power, memory, and connection constraints.
  - Too heavy for IoT devices
- Furthermore, IoT end nodes are also…
  - lossy
  - very low power
  - saddled with constrained resources
  - expected to remain alive for years.

IoT – requirements

IoT devices requirements	
Network requirements	Impact on networking
Scalability / redundancy	Multi-hop, mesh networking
Security	Configurable, with different security levels for different devices capabilities
Addressing	Scalability of address space, low overhead on network protocols
Device requirements	Impact on application-level
Low power / battery powered	Low duty-cycle communications
Limited capacity (memory/processor)	Small footprint, low complexity protocols
Low cost	Reliability of the device and further constraints…

Mobile and cyber-physical systems
-
MQTT

Introduction

- MQTT stands for «Message Queuing Telemetry Transport»
- It is a lightweight publish/subscribe reliable messaging transport protocol
- Lightweight:
  - Small code footprint
  - Low network bandwidth
  - Minimal packet overhead (guarantees better performances than HTTP)

Introduction

Highlights:

- Client-server architecture
- Simple to implement on the client-side
  - Complexity on the server-side
- Provides Quality of Service (QoS) data delivery
- Data agnostic
- Suitable for M2M and IoT applications

Introduction

- MQTT was created by Andy Stanford-Clark of IBM and Arlen Nipper of Arcom in 1999
- It is now an OASIS standard
  - OASIS = «Open Standards for the information Society»
  - released by the OASIS MQTT technical Committee
- It builds upon TCP/IP:
  - Port 1883
  - Port 8883 for using MQTT over SSL
    - … but SSL adds significant overhead!

Introduction

- Widely used:
  - Sensor to satellite
  - Home automation
  - E-health
  - Even Facebook messenger claims its use
  - …

Introduction

- These slides refer to MQTT version 3.1.1 (November 2014)

- You can visit the website:
  - http://mqtt.org/
- download the standard from:
  - http://mqtt.org/2014/11
- a tutorial is here:
  - https://www.hivemq.com/mqtt-essentials/
- A MQTT client for Arduino:
  - https://knolleary.net/2012/03/08/updated-arduino-mqtt-client-4/

Publish / subscribe paradigm

(pub/sub)

Publish/subscribe is a loosely coupled interaction schema

- Alternative to the classical client/server paradigm
  - See «The many faces of publish/subscribe», by P. TH. Eugster, P.A. Felber, R. Guerraoui, A. Kermarrec, ACM computing surveys 35(2):114-131, June 2003

Main actors:

- **publishers** and **subscribers**
  - Which are both «clients»
  - They don't know of each other
- The **event service** (AKA **broker**)
  - It's the server
  - Known both by publishers and subscribers

Pub/sub

- Publishers produce events
  - … or own data they wish to share by means of events
    - example: a sensor that shares sensed data
  - Interact with the broker
- Subscribers express the interest for an event (or a pattern of events)
  - Receive notifications whenever the event is generated
  - Interact with the broker
- Publishers and subscribers are fully decoupled in **time**, **space** and **synchronization**

Pub/sub

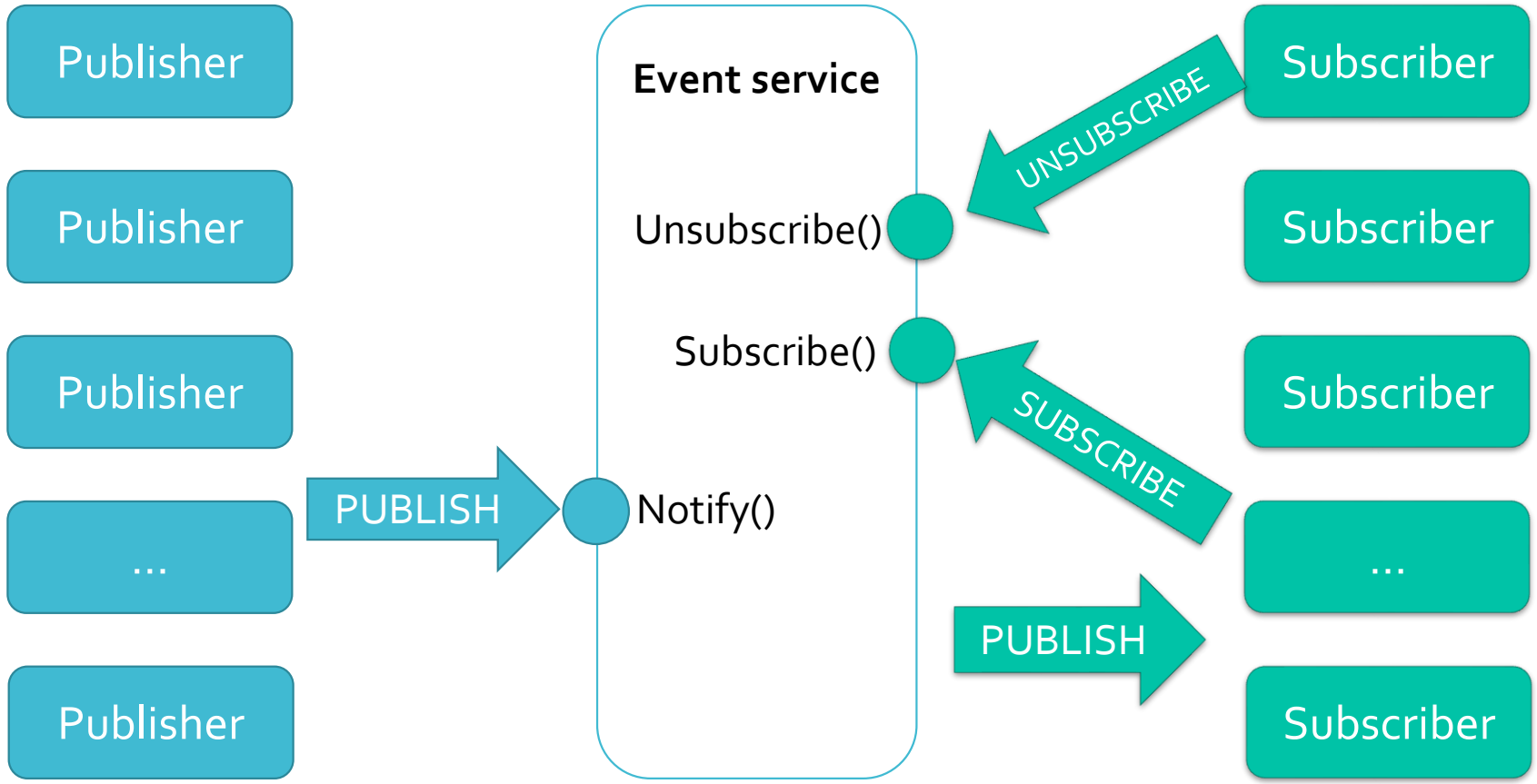
The Event service (AKA broker):

- Known to publishers and subscribers
- Receives all incoming messages from the publishers
- Filters all incoming messages
- Distributes all messages to the subscribers
- Manages the requests of subscription/unsubscription

Pub/sub

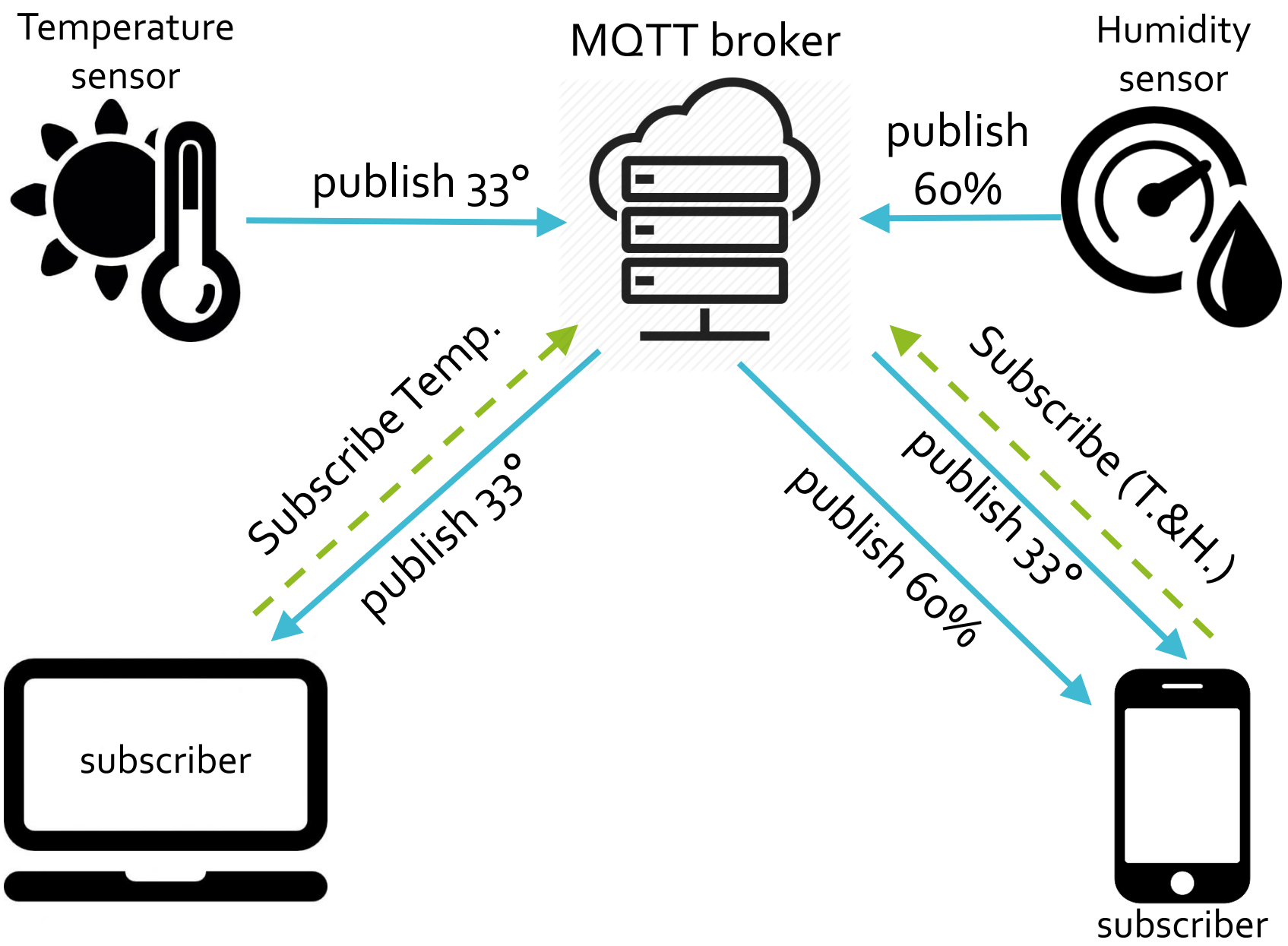
- A publish/subscribe interaction can be implemented in many different ways
- In the basic interaction schema the Event Service (Broker) is an independent agent
- Operations of:
  - Publish
  - Subscribe
  - Notify
  - Unsubscribe

Pub/sub



Event service (AKA broker): storage and management of subscriptions

Pub/sub in MQTT



Pub/sub

Space decoupling:

- Publisher and subscriber do not need each other
- For example, they don't know the IP address and port of each other

Time decoupling:

- Publisher and subscriber do not need to run at the same time.

Synchronization decoupling:

- Operations on both pub. and sub. are not halted during publish or receiving

Pub/sub

Scalability:

- Better than usual client/server approach
- The operations on the broker can be parallelized and are event-driven
- Scalability to a very large number of devices may require parallelization of the broker

Pub/sub

Filtering of messages at the broker:

- Based on a subject topic:
  - The subject (or topic) is a part of the messages
  - The clients subscribe for a specific topic
  - Typically topics are just strings (possibly organized in a taxonomy)
- Based on the content
  - The clients subscribe for a specific query
    - for example: temperature > 30°
  - The broker filters messages based on a specific query
  - Data cannot be encrypted!
- Based on type
  - Filtering of events based on both content and structure
  - The type refers to the type/class of the data
  - Tight integration of the middleware and the language (usually O.O.)

Pub/sub

You need to consider a few things however:

- Publishers and subscribers need to agree on the topics beforehand
- The publishers cannot assume that somebody is listening to the messages
  - i.e. a message may not read by any subscriber.

Pub/sub in MQTT – highlights

- Publishers and subscribers know the hostname/ip and port of the broker in order to publish/subscribe to messages.
- In most applications the delivery of messages is mostly in near-realtime. However,
  - the broker may also store messages for subscribers that are offline…
  - … but the offlline subscribers need to have been connected with a persistent session
  - … and they should have already subscribed the topic

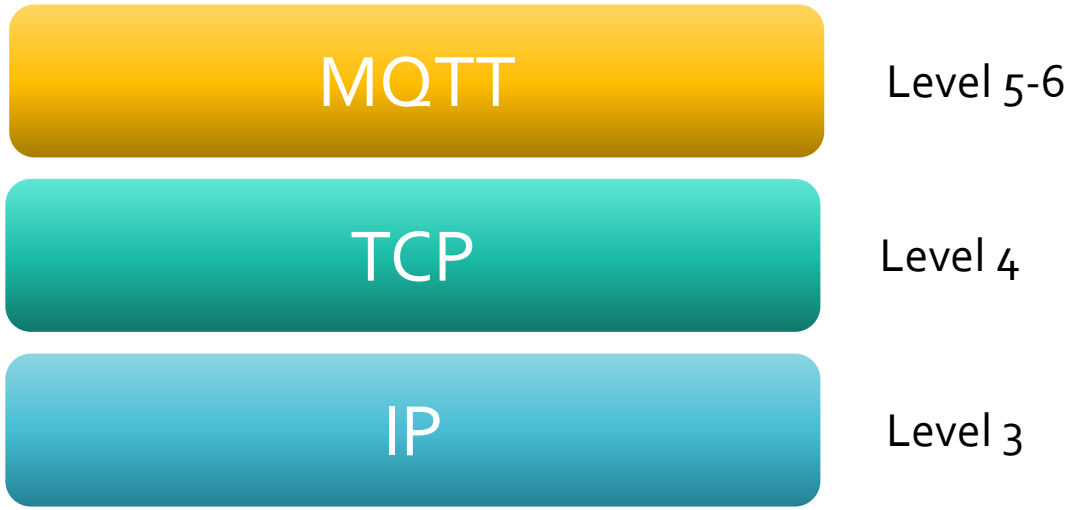
Pub/sub in MQTT – highlights

- MQTT decouples the synchronization:
  - most client libraries work asynchronously (for example, they are based on callbacks)
  - Synchronization is however possible if needed (by means of synchronous APIs)
- MQTT is especially easy to use on the client-side (either publisher or subscriber)
  - The complexity is on the broker-side
  - Suitable for low-power devices

Pub/sub in MQTT – highlights

- Subject-based filtering of messages.
  - i.e. each message contains a topic
  - In the design of a solution the hierarchy of topics should be designed carefully, also leaving space for future extensions
- QoS levels for the messages
  - Assures reliability in the delivery of messages
  - Three levels (0,1,2) of QoS in MQTT (see later)

MQTT in the ISO/OSI protocol stack



MQTT connection

A client connects to a broker by seding a CONNECT message.

The CONNECT message contains:

- Client ID
- Clean Session (optional)
- Username/Password (optional)
- Will flags (optional)
- KeepAlive (optional)

MQTT connection

- Client ID:
  - A string that **uniquely identifies the client** at the broker
  - Can be empty: the broker assigns a clientID and it does not keep a status for the client (the parameter Clean Session must be TRUE)
- Clean Session (optional)
  - FALSE if the client requests a **persistent session**:
    - the broker will store all subscriptions and missed messages for the client (QoS level must be `1` or `2`)
    - If there was a previous session (identified by the client ID) it is resumed
    - After disconnection the broker and the client will store the state of the session
  - TRUE otherwise
    - The broker cleans all information of the client of previous sessions

MQTT connection

- Username/Password (optional)
  - No encryption unless security is used at transport layer
- Will flags (optional)
  - If and when the client **disconnects ungracefully**, the broker will notify the other clients of the disconnection
- KeepAlive (optional)
  - The client committs itself to send a control packet (e.g. a ping message) to the broker within a keep-alive interval
  - The interval is expressed in seconds, in a 16-bits word
  - Serves to assure the broker that the client is still active (detect disconnections)
  - KeepAlive=`0` turns off this mechanism.

MQTT connection

The broker acknowledges the CONNECT message with the CONNECTACK message:

- Return code: confirm whether the connection was successful
  - Connection Accepted
  - Connection Refused: protocol violation / identifier rejected / bad username or passwd / server unavailable
- Session Present: indicates whether the broker already stores a persistent session of the client

Publish in MQTT

After connection, a client can publish messages

- Each message contains:
  - a topic – used to filter and forward messages to subscribers
  - a payload – which contains the data
- The actual format of the payload is application dependent, it can be:
  - Binary data
  - Text
  - XML or JSON documents

Publish in MQTT

NOTE that:

- The publish message is sent by the publisher to the broker…
- … and then forwarded by the broker to the subscribers

Publish in MQTT

PUBLISH message:

- packetId:
  - An integer
  - It is `0` if the QoS level is `0`
- topicName:
  - a string possibly structured in a hierarchy with «/» as delimiters
  - Example: «home/bedroom/temperature»
- qos:
  - `0,1` or `2`

Publish in MQTT

PUBLISH message:

- retainFlag:
  - tells if the message is to be stored by the broker as the last known value for the topic
  - If a subscriber connects later, it will get this message
- payload:
  - The actual message in any form
- dupFlag:
  - Indicates that the message is a duplicate of a previous, un-acked message
  - Meaningful only if the QoS level is >0

Publish in MQTT

When the broker receives a PUBLISH message, it will:

- Acknowledge the message (if requested, QoS>0)
- Process the message (identify its subscribers)
- Deliver the message to its subscribers

The publisher:

- Leaves the message to the broker
- Does not know whether there are any subscribers and whether or when they will receive the message

Subscribe in MQTT

A client subscribe a topic to receive the relevant messages

SUBSCRIBE message:

- packetId:
  - an integer
- topic1:
  - a string (see publish messages)
- QoS1:
  - `0,1 or 2`

Repeated in a list for all topics to subscribe

Subscribe in MQTT

The broker acknowledges the Subscriber with a SUBACK message

SUBACK message:

- packetId:
  - The same of the SUBSCRIBE message
- returnCode:
  - one for each topic subscribed
  - `128`: indicates failure (e.g. the subscriber is not allowed or the topic is malformed)
  - `0,1` or `2` indicates success, with corresponding maximum QoS granted (can be lesser than the QoS requested)

Unsubscribe in MQTT

A client can unsubscribe a topic to stop receiving the related messages

UNSUBSCRIBE message:

- packetId
- topic1
- topic2
- …

UNSUBACK message:

- packetId
  - the same of the UNSUBSCRIBE message

Topics

Topics are strings that are organized into a hierarchy (topic levels)

- Each level is separated by a «/»
- Example: home/firstfloor/bedroom/presence

Wildcards can be used at subscribe time to select a group of topics:

- home/firstfloor/+/presence
  - selects all presence sensors in all rooms of the first floor
- home/firstfloor/#
  - selects all sensors in the first floor

Topics

Topics that begins with a «**$**» are reserved for internal statistics of MQTT

- They cannot be published by clients

There is no standardization of such topics

Examples are (from HiveMQ):

- $SYS/broker/clients/connected
- $SYS/broker/clients/disconnected
- $SYS/broker/clients/total
- $SYS/broker/messages/sent
- $SYS/broker/uptime

Topics

There are no specific rules for the topics however, there are best practices:

- Don’t start a topic with a «/»
  - i.e. /home/livingroom
- Don’t use spaces
- Keep topics short enough
  - Long topics will add overhead to the communications
- Use only ASCII subset UTF-8 of characters
- Sometimes is useful to embed the clientID (or a unique identifier) in the topic
  - Es. sensor1/temperature
  - Only the client with the same clientId can publish such a topic

Topics

- Use hierarchies of topics that are easily extendible
- Specific topics are preferred to generic topics:
  - Ex. /home/livingroom VS /home/livingroom/temperature; /home/livingroom/humidity; …
- Do not subscribe all messages using #
  - The workload can be too much for a client (and for the network)
  - However, it could be useful to access to all messages to store them in a DB…

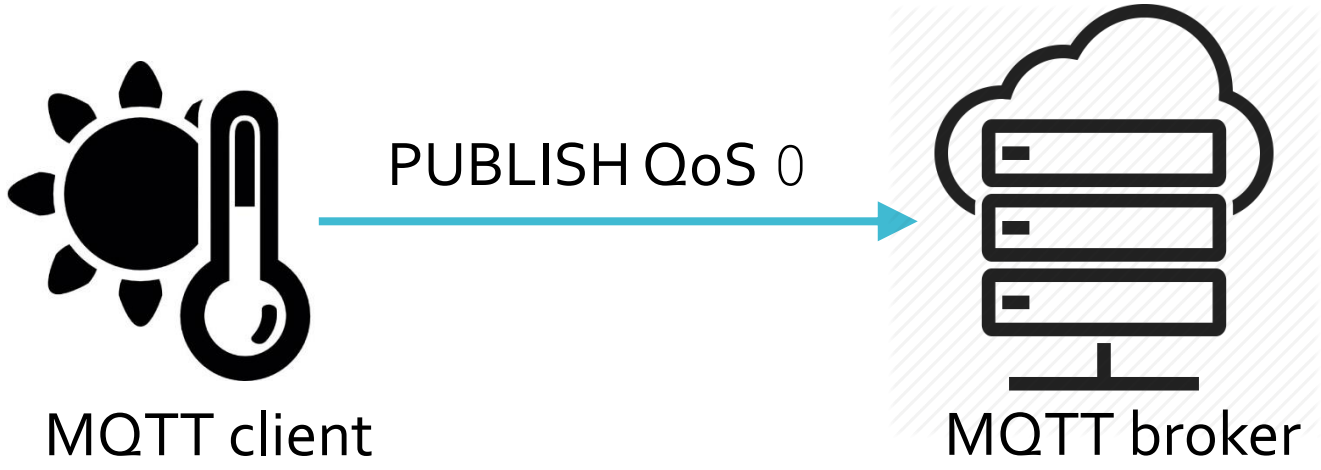
QoS in MQTT

- The QoS is an agreement between the sender and the receiver of a message
  - For example, in TCP the QoS includes guaranteed delivery and ordering of messages.
- In MQTT the QoS is an agreement between publishers and subscribers
- There are three levels of QoS:
  - At most once (level 0)
  - At least once (level 1)
  - Exactly once (level 2)
- NOTE: QoS is used both:
  - between publisher and broker
  - between broker and subscriber

QoS in MQTT

QoS level 0: at most once

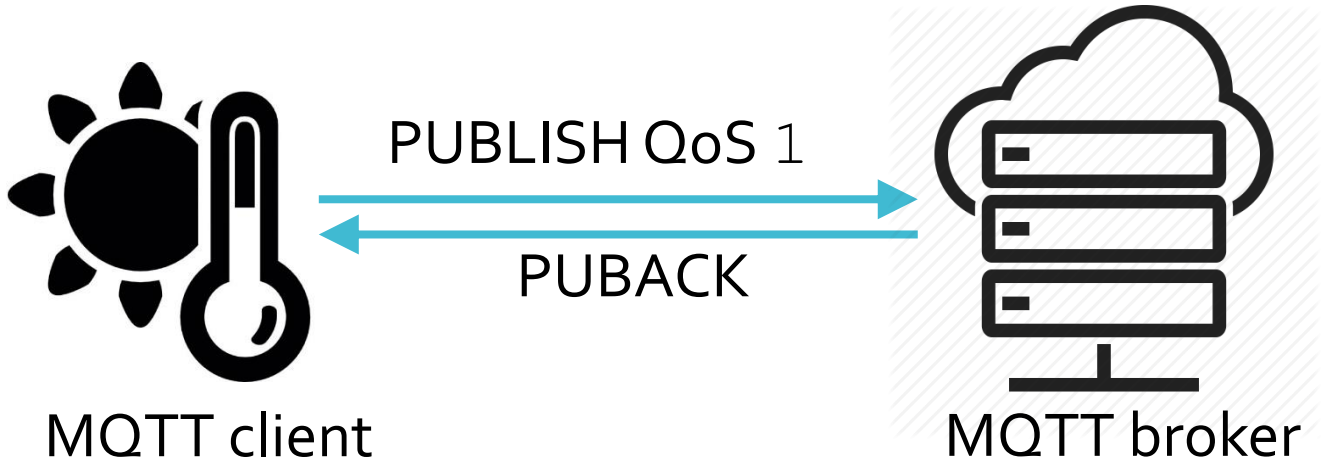
- It is a «best effort» delivery
- Messages are not acknowledged by the receiver
- Messages are not stored by the broker
  - Immediate forward
- Provides the same guarantee as the TCP protocol
  - Note: TCP guarantees delivery as long as the connection remains. If one of the two peers disconnect there's not guarantee anymore!



QoS in MQTT

QoS level 1: at least once

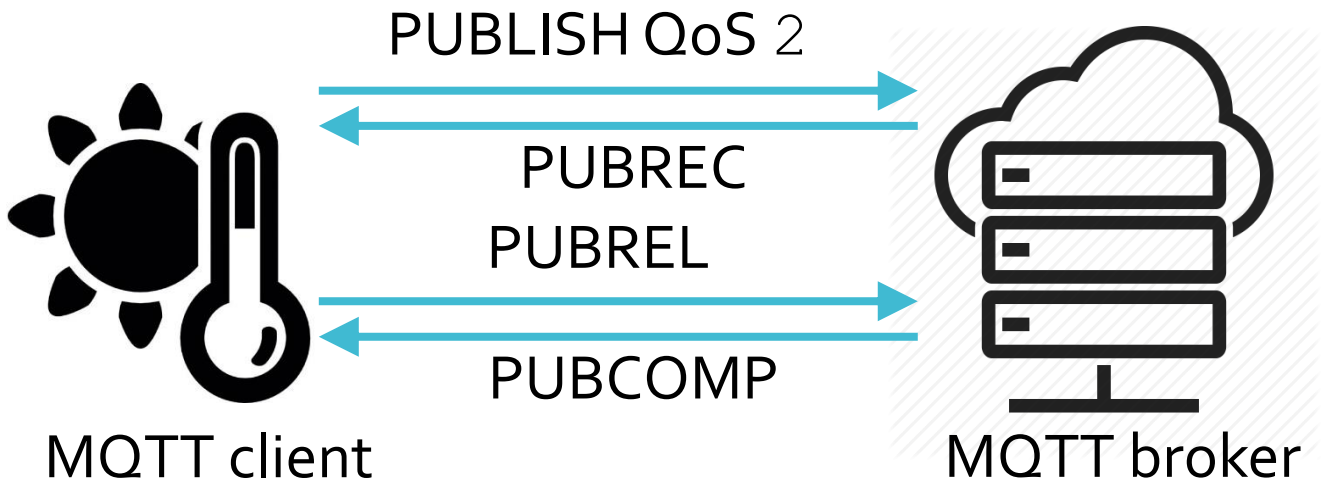
- Messages are numbered and stored by the broker until they are delivered to all subscribers with QoS level 1
- Each message is delivered at least once to the subscribers with QoS 1
- A message may be delivered more than once
- Subscribers send acknowledgements (PUBACK packets)



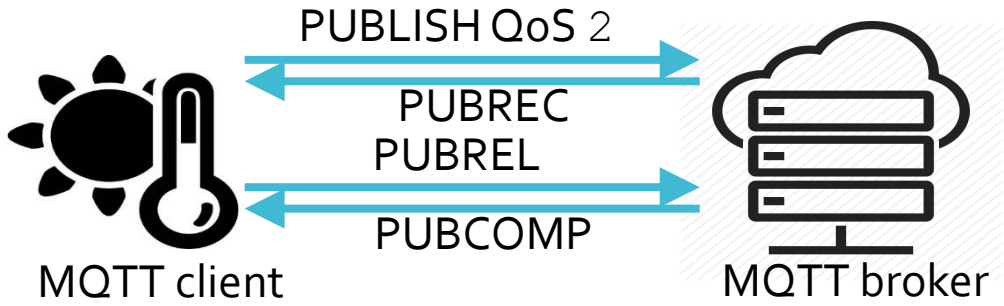
QoS in MQTT

QoS level 2: exactly once

- It's the highest QoS level in MQTT
- … and the slowest as well!
- It guarantees that each message is receaved exactly once by the subscriber
- It uses a double two-way handshake



QoS in MQTT



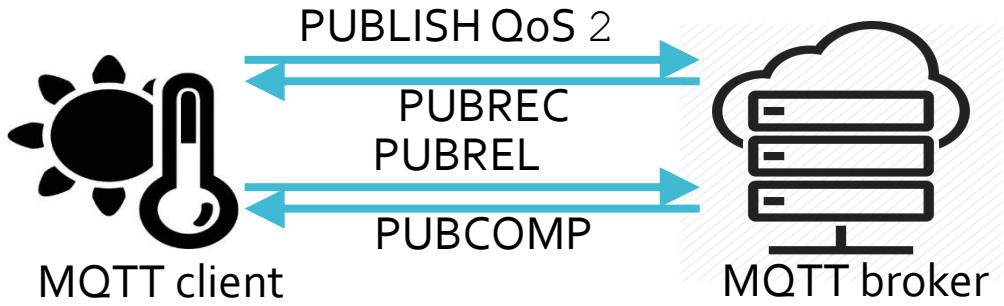
QoS level 2

The broker:

- Receives the PUBLISH with a message
- Processes the PUBLISH
- Sends back a PUBREC
- Keeps a reference to the message until it receives PUBCOMP

- The client:
  - Sends PUBLISH with a message
  - Waits for PUBREC and then store PUBREC and discards the message
  - Sends back a PUBREL to inform the broker
  - Waits for PUBCOMP and then discards the PUBREC

QoS in MQTT



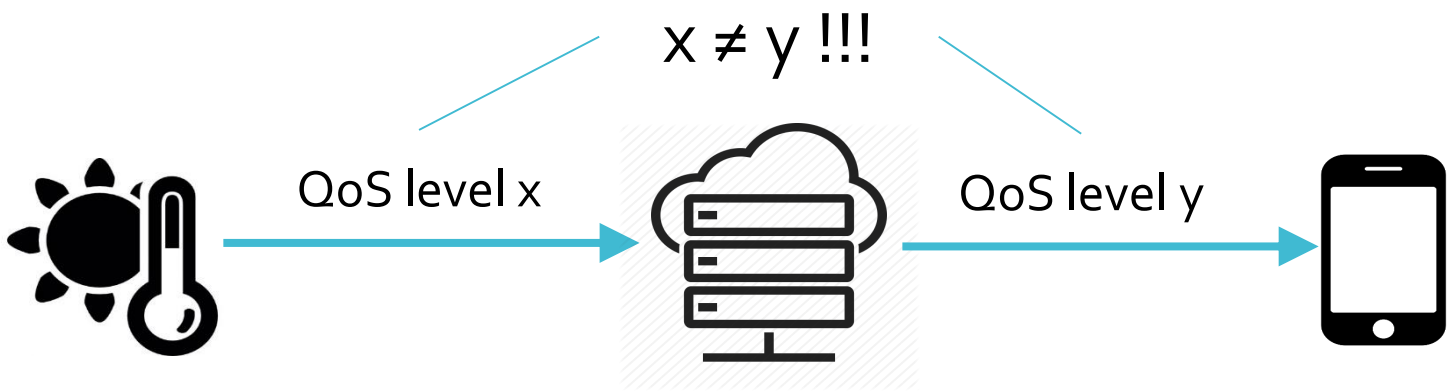
Note: this double handshake is necessary because:

1. The first (two-way) handshake is to send the message
2. The second is to agree to discard the state
3. PUBREC alone is not sufficient:
   - If PUBREC is lost, the client will send again the message.
   - Hence the broker need to keep a reference to the message to identify duplicates
   - With PUBCOMP the broker knows that it can discard the state associated to the message

QoS in MQTT

Some things to keep in mind about QoS:

- A flow publisher-broker is independent from the flow broker subscriber!
- Packet identifiers are unique per client:
  - Different clients (se. Subscribers) use different packet Ids, even for the same message
  - Packet Ids are 16 bits integers
- Messages sent with QoS 1 and 2 are always stored for off-line clients (that use persistent connections)



QoS in MQTT

Best practices:

- Use QoS level 0 when:
  - The connection is stable and reliable
  - Single message is not that important or get stale with time
  - Messages are updated frequently and old messages become stale
  - Don't need any queuing for offline receivers
- Use QoS level 1 when:
  - You need all messages and **you can handle duplicates**
- Use QoS level 2 when:
  - You need all messages and the **receivers cannot handle duplicates**
  - Has a much higher overhead!!!!

MQTT – persistent sessions

Persistent sessions keep the state between a client and the broker

- If a subscriber disconnects, when it connects again, it does not need to subscribe again the topics
- The session is associated to the clientId defined with the CONNECT

In a persistent session the data stored are:

- All subscriptions
- All QoS 1&2 messages that are not confirmed yet
- All QoS 1&2 messages that arrived when the client was offline

MQTT – persistent sessions

A persistent session is requested at CONNECT time

- See flag cleanSession
- The CONNACK message confirms whether the session is persistent

Also clients have to store state in a persistent session:

- Store all messages in a QoS 1&2 flow, not acked by the broker
- All received QoS 2 messages not confirmed by the broker

Messages of persistent sessions will be stored as long as the system allows it

MQTT – persistent sessions

Avoid persistent sessions for:

- Publishing-only clients that use only QoS 0
- If old messages are not important

Retained messages

- A publisher has no guarantee that its messages are actually delivered to the subscribers
  - You can only guarantee delivery to the broker (with QoS levels `1`, `2`)
- When a client connects to the broker and subscribes a topic, it does not know when it will get any message
  - It depends on when the publisher will publish messages
  - … can take ages

Retained messages

- A retained message is a normal message with retainFlag =True
- The message is stored by the broker
  - If a new retained message of the same topic is published, the broker will keep the last one
- When a a client subscribes the topic of the retained message…
- … the broker immediately sends the retained message for that topic
  - This works also with wildcards
- This way a new subscriber is immediately updated with the «state of the art»

Retained messages

- Note: retained messages are not inherent to persistent sessions:
  - They are kept by the server independent of the fact that they are delivered
- To delete a retained message is sufficient to publish a retained empty message of the same topic

Retained messages

- Retained messages make sense for unfrequent updates of a topic.
- For example, consider a device that update its status (ON/OFF) on topic home/devices/device1/status
  - If it publish «ON», this status will remain (hopefully) for long time
  - If the message is retained, all subscribers will easily know the device is on

Last will & testament

Last Will & testament is used to notify other clients about the **ungraceful disconnection** of a client.

- At CONNECT time, a client can request the broker a specific behavior about its last will
  - its last will is a normal message with topic, retained flag, QoS and payload
- The broker stores the last will message
- When the broker detects the client is abruptly disconnected, it sends the last will message to all subscribers of the topic specified in the last will message
- If the client instead DISCONNECT, the stored last will message is disregarded

Last will & testament

To this purpose the CONNECT message contains `4` optional fields:

- lastWillTopic — a topic
- lastWillQoS — a QoS level (either `0`, `1`, `2`)
- lastWillMessage — a string
- lastWillRetain — a boolean

Last will & testament

The broker sends a last will message if:

- Occurs an I/O network error
- The client does not send the KeepAlive message in time
- The client closes the network connection without sending a DISCONNECT
- The broker closes the connection with the client because of a protocol error

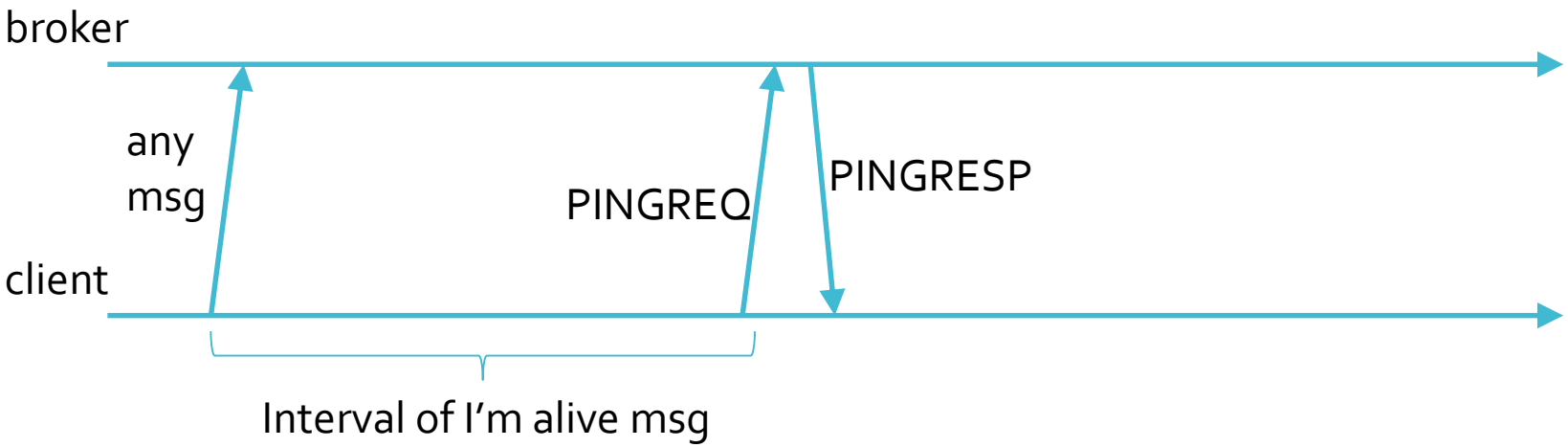
Last will & testament

Often the Last will message is used along with retained messages

- Consider the example of a device that updates its status (ON/OFF) on topic home/devices/device1/status
  - It publishes «ON» with a retained message
  - If it abruptly disconnect (thus it does not publishes «OFF»)
  - Last Will message is useful here…
  - … it could be a retained message with payload «OFF»
  - … this way the subscribers (and the future ones) are properly informed.

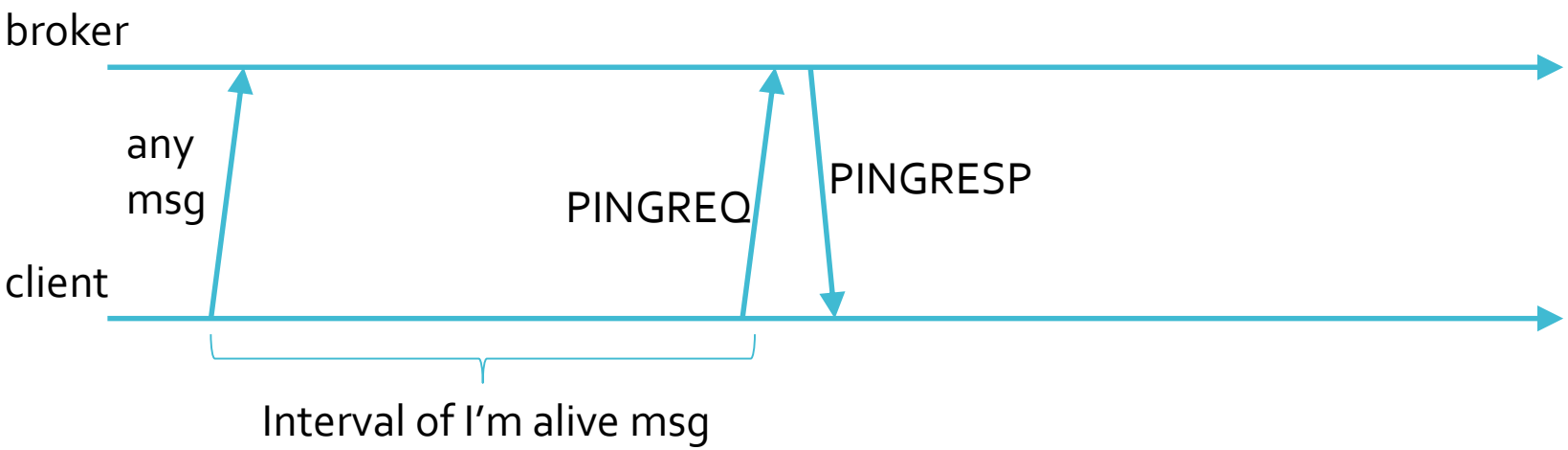
Keep Alive

- Keep alive is a simple mechanism to assures that a client (and its connection with the broker) is still active
- The client periodically sends sort of «I'm alive» messages to the broker
  - The frequency of these messages is declared in the CONNECT message
- The keep alive message must be sent by the client before the expiration of the interval set with the CONNECT



Keep Alive

- If the client does not sends PINGREQ in time (or any other message) the broker:
  - turns off the connection
  - Sends the last will and testament message



Packet format

Structure of an MQTT control packet:

Fixed header, present in all MQTT control packets
Variable header, present in some MQTT Control Packets
Payload, present in some MQTT Control Packets

Fixed header

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT control packet type				Flags specific to each MQTT control packet type			
byte 2…	Remaining length							

Remaining length is the length of the variable header and payload.

- 1 byte encodes length of up to 127 bits. The most significant bit specifies that there is another field of length (for packet longer than 127 bytes)

Packet format

Control packet type:

Name	value	direction of flow
reserved	0	forbidden
CONNECT	1	client to server
CONNACK	2	server to client
PUBLISH	3	client to server or server to client
PUBACK	4	client to server or server to client
PUBREC	5	client to server or server to client
PUBREL	6	client to server or server to client
PUBCOMP	7	client to server or server to client
SUBSCRIBE	8	client to server
SUBACK	9	server to client
UNSUBSCRIBE	10	client to server
UNSUBACK	11	server to client
PINGREQ	12	client to server
PINGRESP	13	server to client
DISCONNECT	14	client to server
reserved	15	forbidden

Packet format

Specific flags

- Reserved: it is for future use, must be set as shown.

Control packet	fixed header flags	bits:
CONNECT	reserved	0,0,0,0
CONNACK	reserved	0,0,0,0
PUBLISH	used in MQTT 3.1.1	DUP, QoS, Qos, Retain
PUBACK	reserved	0,0,0,0
PUBREC	reserved	0,0,0,0
PUBREL	reserved	0,0,1,0
PUBCOMP	reserved	0,0,0,0
SUBSCRIBE	reserved	0,0,1,0
SUBACK	reserved	0,0,0,0
UNSUBSCRIBE	reserved	0,0,1,0
UNSUBACK	reserved	0,0,0,0
PINGREQ	reserved	0,0,0,0
PINGRESP	reserved	0,0,0,0
DISCONNECT	reserved	0,0,0,0

Packet format

Variable header:

- Contains the **packet identifier type** (encoded with two bytes)
  - Only CONNECT and CONNACK control packets **do not** include this information
  - The PUBLISH packet contains this information only if QoS>0
- Contains other information depending on the control packet type
  - For example, CONNECT packets include the protocol name and version, plus a number of flags (see CONNECT)

Packet format

Payload:

- Contains additional information
- E.g. the payload of CONNECT includes:
  - client identifier (mandatory)
  - will topic (optional)
  - will message (optional)
  - Username (optional)
  - Password (optional)

Control packet	payload
CONNECT	required
CONNACK	none
PUBLISH	optional
PUBACK	none
PUBREC	none
PUBREL	none
PUBCOMP	none
SUBSCRIBE	reserved
SUBACK	reserved
UNSUBSCRIBE	reserved
UNSUBACK	none
PINGREQ	none
PINGRESP	none
DISCONNECT	none

Case study: MQTT in Arduino

An example: the MQTT client library for Arduino

- Only the core features of MQTT
  - No security with SSL/TLS
  - No QoS level `2`
  - Payload limited to `128` bytes

https://knolleary.net/2012/03/08/updated-arduino-mqtt-client-4/

Case study: MQTT in Arduino

Constructors

- **PubSubClient** ()
- **PubSubClient** (server, port, [callback], client, [stream])
  - Creates a fully configured client instance
  - The API includes other **PubSubClient** with lesser parameters

Parameters:

- server: IPadress of the broker (an IPAddress)
- port: port used by the broker (an int)
- callback: a pointer to a message callback function called when a message arrives for a subscription created by this client
- client: an instance of *Client* that can connect to a specified internet IP address and portClient (e.g. Ethernetclient)
- Stream: an instance of *Stream*, used to store received messages (e.g. mqtt_stream)

Case study: MQTT in Arduino

Functions

- boolean **connect** (clientID, username, password, willTopic, willQoS, willRetain, willMessage)
  - Connects the client, specifying the will message, username and password
  - The API includes other **connect** with lesser parameters
  - username, passrowd, willTopic, willMessage are const char []
  - willQoS, willretain are int
- void **disconnect** ()

Case study: MQTT in Arduino

- int **publish** (topic, payload, length, retained)
  - Publishes a message to the specified topic, with the retained flag as specified
  - Returns false if: publish fails, either due to connection lost, or message too large
  - Returns true if publish succeeds
  - The API includes other **publish** with lesser parameters
- boolean **subscribe** (topic, [qos])
- boolean **unsubscribe** (topic)
- Parameters:
  - topic is a const char []
  - payload is a byte[]
  - length is a byte
  - retained is a Boolean
  - qos is an int

Case study: MQTT in Arduino

- boolean **loop** ()
  - Sends keep alive messages
- int **connected** ()
  - Checks whether the client is connected to the server
- int **state** ()
  - Returns the current state of the client. If a connection attempt fails, this can be used to get more information about the failure

The following functions configure the parameters of the server (if not initialized at the construction):

- PubSubClient **setServer** (server, port)
- PubSubClient **setCallback** (callback)
- PubSubClient **setClient** (client)
- PubSubClient **setStream** (stream)

MQTT servers

There are many brokers available:

Mosquitto, http://mosquitto.org/

- An eclipse project (iot.eclipse.org)
- Compliant with MQTT versions 3.1 and 3.1.1

Mosca, http://www.mosca.io/

- It is a Node.js MQTT broker
- Compliant with MQTT versions 3.1 and 3.1.1
- Supports QoS levels 0 and 1

HiveMQ, https://www.hivemq.com/

- an enterprise MQTT broker

MQTT competitors

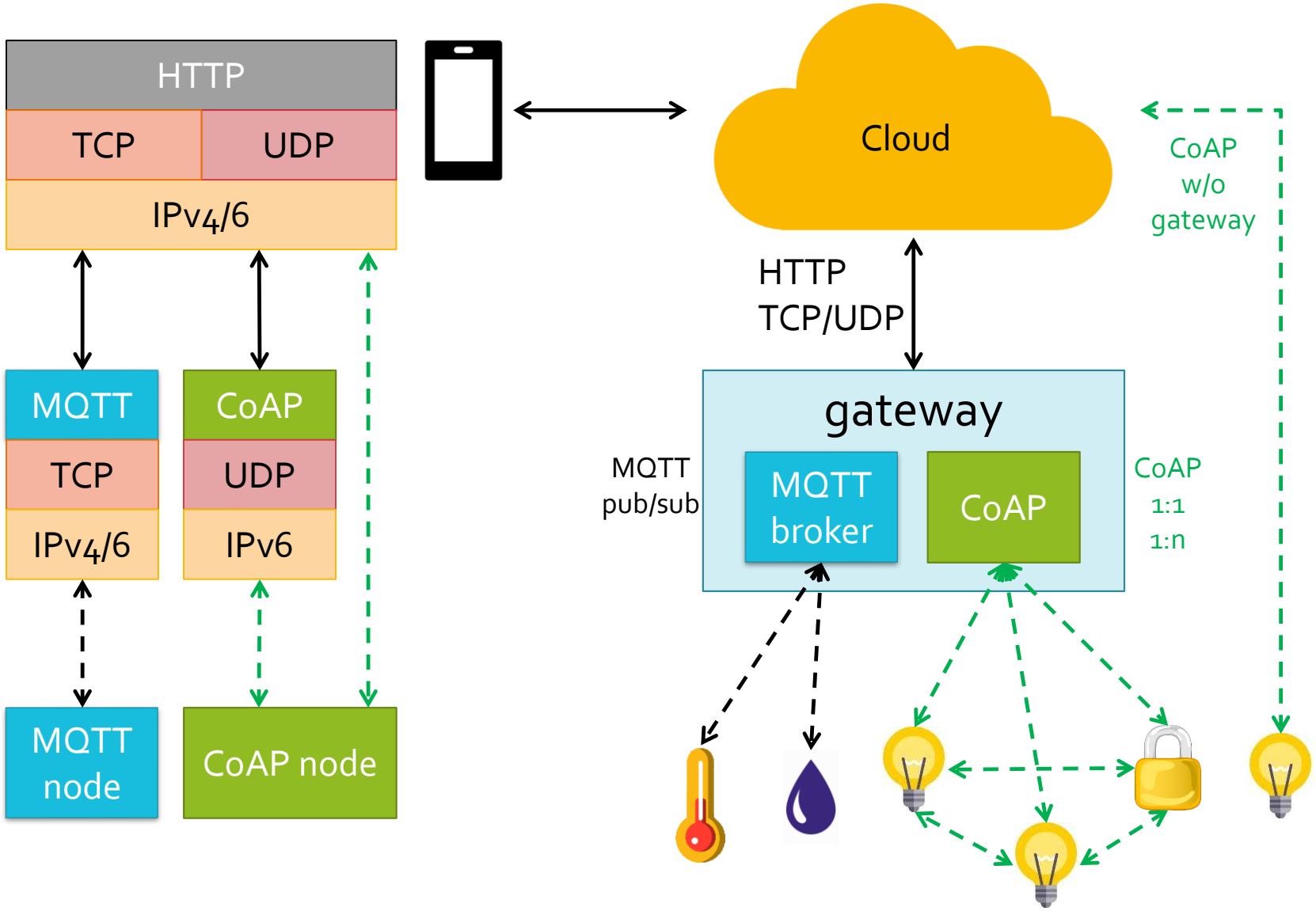
Issues with MQTT

- The need for a centralized broker can be limiting in distributed IoT appications with diffused point-to-point communications
  - The overhead of a broker may easily become not compatible with end-devices capabilities as the network scales up
- The broker is a single point of failure
  - It may fail or deplete its battery
  - … but it is essential to keep the network alive
- MQTT relies on TCP, which is not particularly cheap for low-end devices:
  - It requires much more resources than UDP
  - Long wake-up time due to establishment of TCP connection
  - TCP connections need to be established and maintained
  - It impacts on the batteries (and on the device lifetime)

MQTT vs HTTP

	MQTT	HTTP
pattern	publish/subscribe	client/server
focus of communication	single data (byte array)	single document
size of messages	small and small header	large, details encoded in text form
service levels	3 QoS levels	same service level for all messages
kind of interaction	1 to 0; 1 to 1; 1 to N	1 to 1

MQTT vs CoAP


HTTP
TCP
UDP
IPv4/6
MQTT
TCP
IPv4/6
CoAP
UDP
IPv6
MQTT node
CoAP node
Cloud
CoAP w/o gateway
HTTP TCP/UDP
gateway
MQTT pub/sub
MQTT broker
CoAP
CoAP 1:1 1:n

CoAP

Constrained Application Protocol (CoAP)

- Standardized in RFC 7252
- A specialized web transfer protocol
- For use with constrained nodes and constrained networks in the Internet of Things.
- The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation

- Supported by Eclipse
- Rapidly growing

CoAP highlights

- Based on a client/server paradigm
  - Sensors/actuators are servers
  - Applications are clients
- REST model:
  - Servers make resources available trough an URL
  - Clients access resources by means of GET, PUT, POST, DELETE requests
- Works similarly to HTTP, but:
  - It builds on top of UDP/IP
  - Small headers with a compact encoding of information
  - A resource directory provides resource discovery features
- Embeds strong security mechanisms
  - Equivalent to 3072-bits RSA
- Uses Uniform Resource Identifiers (URI) to describe network nodes

CoAP highlights

Designed for:

- Nodes with 8-bit microcontrollers
- Small amounts of ROM and RAM,
- Constrained networks such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)
  - With high packet error rates and a typical throughput of 10(s)s kbit/s.

CoAP strengths & weaknesses

Strengths:

- Native UDP
- Support also multicast (but 1:1 is the main comm. mechanism)
- Security
- Allows also for asynchronous communications as in MQTT

Weaknesses:

- Standard maturity
- Message reliability not quite sophisticated, similar to MQTT QoS

MQTT vs CoAP

MQTT vs CoAP:

- Both are suitable for IoT applications
- MQTT is particularly suitable for very low-power devices
- The publish/subscribe model of MQTT allows a full decopuling of producers and consumers
- Security in CoAP has no match in MQTT