LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 498R: Internet of Things

Lecture 06: IoT Application Layer, Integration Patterns, REST, and CoAp

Instructor: Haidar M. Harmanani

Fall 2017

Topics

- Two of the most promising protocols for small devices:
  - Constrained Application Protocol (CoAP)
  - Message Queuing Telemetry Transport (MQTT)

Internet Protocol Suite

We are here!

HTTP, Websockets, DNS, XMPP, MQTT, CoAp	Application layer
TLS, SSL	Application Layer (Encryption)
TCP, UDP	Transport
IP(V4, V6), 6LowPAN	Internet Layer
Ethernet, 802.11 WiFi, 802.15.4	Link Layer

Principles: Layering, modularity, separation of concerns
Each layer focuses on a particular set of concerns and abstracts these concerns from the layer above.

Where are we?

TCP/IP Protocol stack	IoT protocol stack
HTTP	CoAP
TCP, UDP	UDP
IPv4, IPv6	6LoWPAN
Ethernet MAC	ContikiMAC, X-MAC, TSCH
Ethernet PHY	IEEE 802.15.4 PHY, Low-power Wifi, 4G

CoAP: Background

- GOAL: to enable web-based services in constrained wireless networks
  - 8 bit micro-controllers
  - limited memory
  - low-power networks
- Problem: Web solutions are hardly applicable
  - Redesign web-based services for constrained networks!

How Does the Web Work?

- Resources in the Web are:
  - Managed by servers
  - Identified by URIs
  - Accessed synchronously by clients through request/response paradigms
- Is not that we call *Representational State Transfer (REST)*?

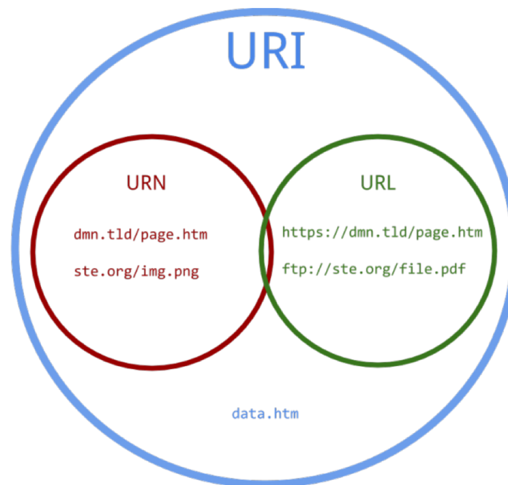
RESTful means …

- REpresentational State Transfer is a software architectural style for Web client-server
- Resources are represented as URL:
  - "example.com/profile/johnny"
  - "example.com/domain/sensor3/temp1"
- REST makes information available as resources that are identified by URIs.
- Resources can be retrieved and manipulated using VERBS:
  - GET, POST, PUT, DELETE

- *CoAP provides a request/response RESTful interaction like HTTP.*
- *Smaller messages than HTTP and with very low overhead.*
- *BLE nodes, for example, have limited memory and storage.*
- *Sensors and actuators on BLE nodes are simply CoAP REST resources.*
- *For example, to obtain a current temperature, send a GET request.*
- *To turn on/off or toggle LEDs we use PUT requests.*

CoAp is based on REST

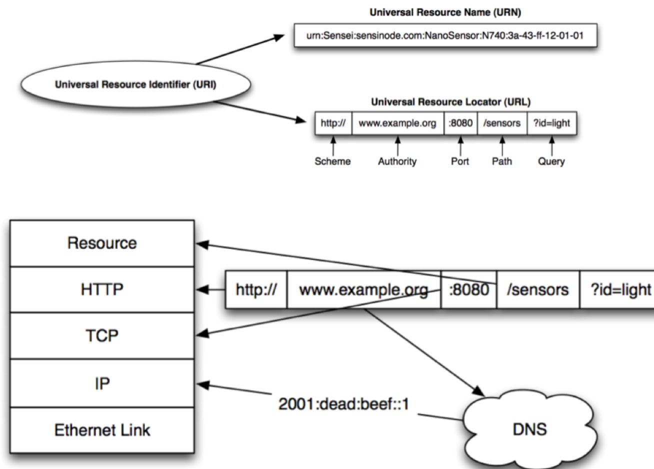
Difference Between URL and URI



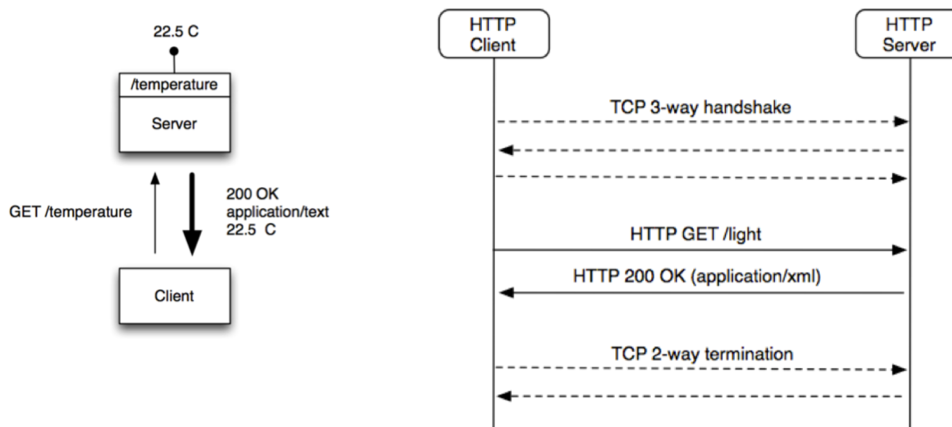
Tim Berners-Lee, et. al. in rfc 3986: uniform resource identifier (uri): generic syntax:

A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.
A URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location").

URL Resolution



Request/Response Transaction



Back to Our Initial Problem…

- GOAL: to enable web-based services in constrained wireless networks
  - 8 bit micro-controllers
  - limited memory
  - low-power networks
- Problem: Web solutions are hardly applicable
  - Redesign web-based services for constrained networks!

Recall the typical HTTP Interaction

- Connection oriented and synchronous
- TCP 3 way handshake with server
  - HTTP GET /kitchen/light
  - HTTP response with headers and {"setting" : "dim" }
  - TCP 2 way termination
- Too much work for simple IoT applications
- CoAP is not a general replacement for HTTP
- CoAP does not support all features of HTTP

CoAP at a Glance

- A key IoT standard
- Open IETF standard since June 2014
- Application level protocol over UDP or SMS on cellular networks
- Designed to be used with constrained nodes and lossy networks
- Designed for *M2M* applications, such as home and infrastructure monitoring
- Built-in resource discovery and observation ("push notification")
- Block-wise transfer

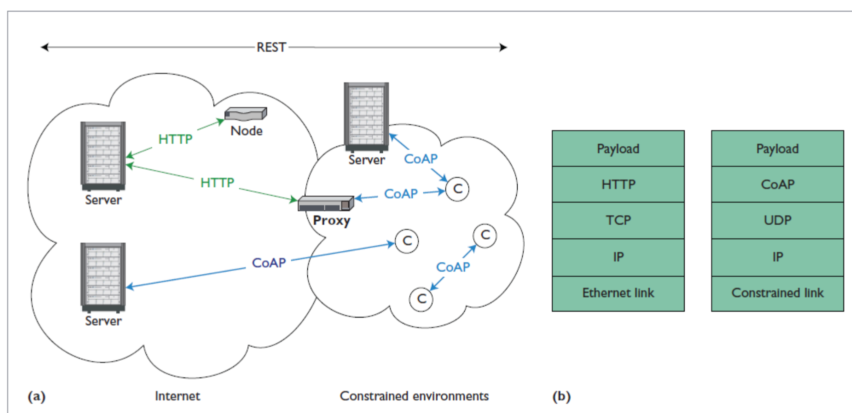
CoAP At a Glance

- Based on web standards, easily integrates with HTTP.
  – Not simply a compressed version of HTTP.
- Some built-in reliability
- May run over 6LoWPan
- Use on low power, low bandwidth, lossy networks
- DTLS for security
- Asynchronous subscriptions and notifications over UDP
- Built-in resource discovery

CoAP At a Glance

- RESTful for easy interfacing with HTTP
- Peer to peer or client server and multi-cast requests
- Low overhead and simple
- Should fit into a single UDP packet or IEEE 802.15.4 frame
- Uses compression techniques

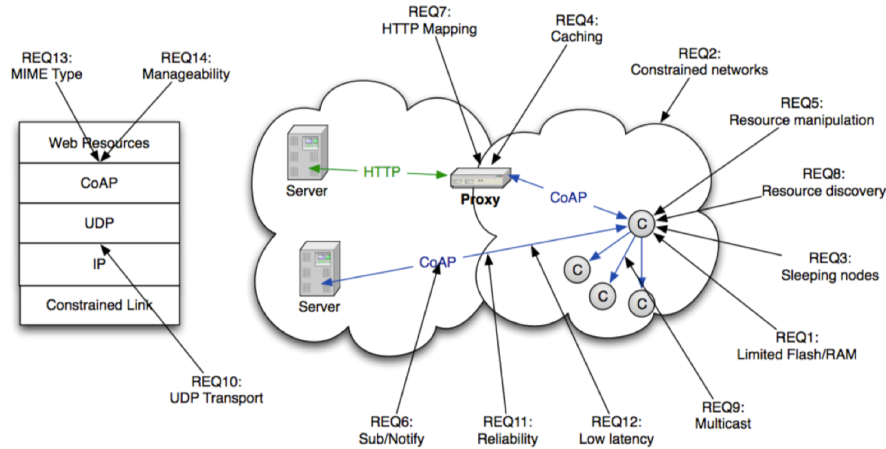
CoAP Protocol Stack and Interactions



Implementing the Web architecture with HTTP and the Constrained Application Protocol (CoAP). (a) HTTP and CoAP work together across constrained and traditional Internet environments; (b) the CoAP protocol stack is similar to, but less complex than, the HTTP protocol stack.

C. Bormann, A. P. Castellani, Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," IEEE Internet Computing, vol. 16, no. 2, pp. 62-67, Feb. 2012, doi:10.1109/MIC.2012.29

CoAP Design Requirements



CoAP Versus HTTP

- CoAP is a subset of HTTP functionality re-designed for low power embedded devices such as sensors (for IoT and M2M), occasionally sleeping devices
  – Why sleep?
- Why not HTTP?
  – TCP overhead is too high and its flow control is not appropriate for short-lived transactions.
  – UDP has lower overhead and supports multicast
  – HTTP transactions are around 10 times higher than CoAP transaction bytes due to 6LoWPAN and CoAP header compression
  – CoAP packet can be sent in single IEEE802.15.4 frame without fragmentation.
  – Less bytes ➔ lower power consumption and longer lifetime for CoAP.

An experimental CoAP power consumption evaluation shows a major improvement in power savings (see Integrating Wireless Sensor Networks with the Web by Colitto et al.)

Table 1. Comparison between CoAP and HTTP

	Bytes per-transaction	Power	Lifetime
CoAP	154	0.744 mW	151 days
HTTP	1451	1.333 mW	84 days

CoAP Versus HTTP

CoAP Messaging Basics

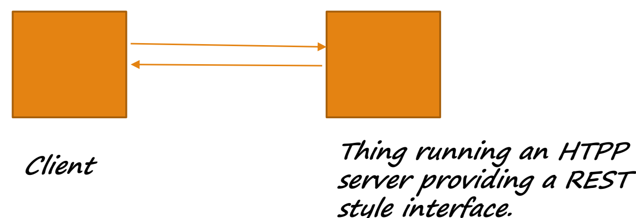
- Transport
  - (mainly) UDP binding
- All CoAP messages include:
  - A four-byte fixed length header followed by, depending upon the message type:
    - An optional header portion
    - Payload
- Each message includes a 16-bit message ID used to link requests to their accompanying acknowledgements (ACK) or error statuses where applicable.

CoAP Messaging Basics

- Message Exchange between Endpoints
  - Reliable exchange through *Confirmable Messages* which must be acknowledged (through ACK or Reset Messages).Simple Stop-and-Wait retransmission with exponential back-off.
  - Unreliable exchange through *Non-Confirmable Message*
  - Duplicate detection for both confirmable and non-confirmable messages (through message ID)

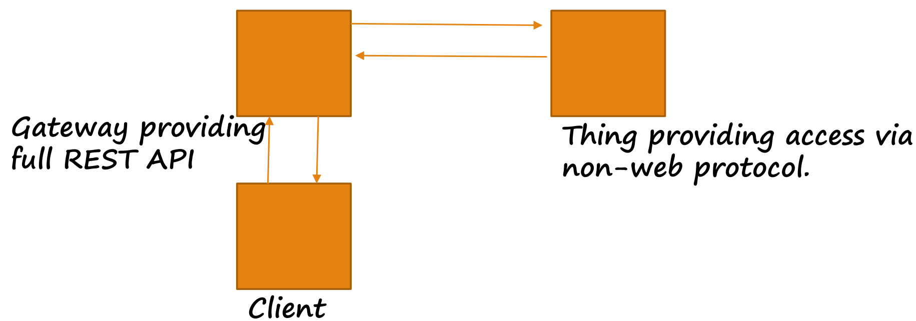
Direct Integration Pattern

- Some *Things* have full internet access.
  - These Things may provide an HTTP server running over TCP/IP and can directly connect to the internet – using, say, WiFi or Ethernet or cellular. Rasberry-Pi's and Photon's are examples. These may be used to implement a Direct Integration Pattern – REST on devices.
- Typical use cases
  - The Thing is not battery powered and communicates with low latency to a local device like a phone.
- Example: Use a phone to communicate via WiFi (with WiFi router) to an HTTP server on a device. Use web sockets for publish/subscribe, e.g., phone listens for doorbell events.



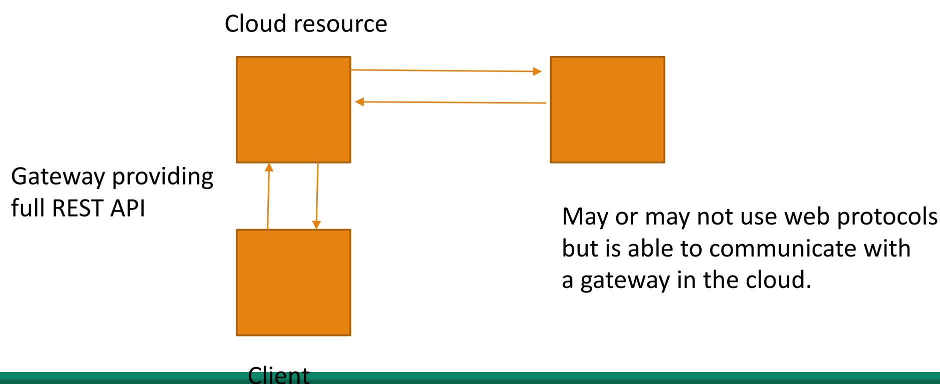
Gateway Integration Pattern

- Some Things do not have full internet access. These Things may support only Zigbee or Bluetooth or 802.15.4. We are not sending IP packets to these devices – they are constrained. This is the Gateway Integration Pattern.

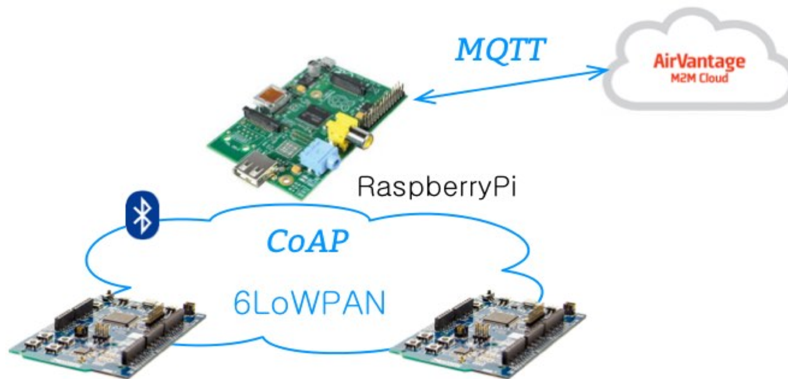


Cloud Integration Pattern

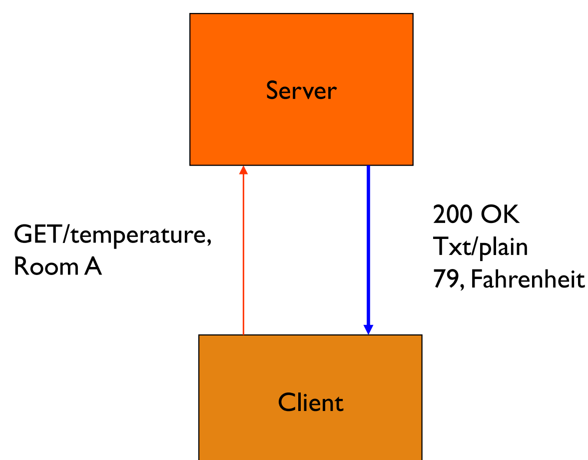
- Some Things have access to the cloud and need powerful and scalable cloud support. This is the Cloud Integration Pattern.



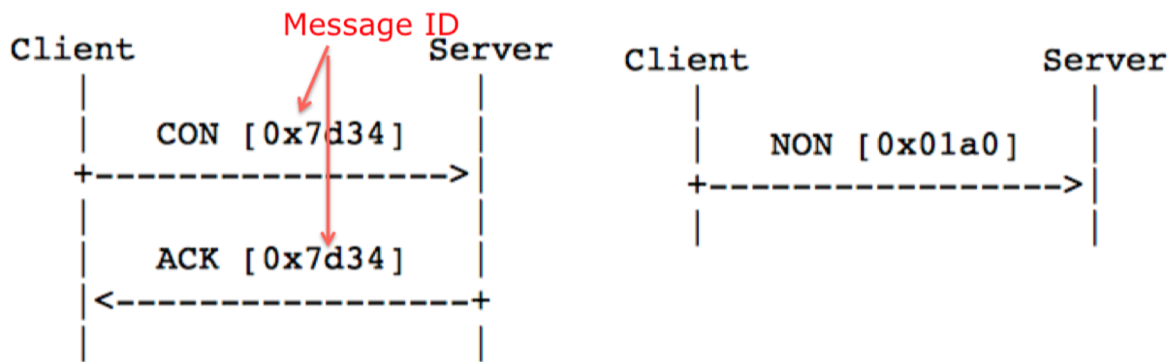
From an add from AirVantage



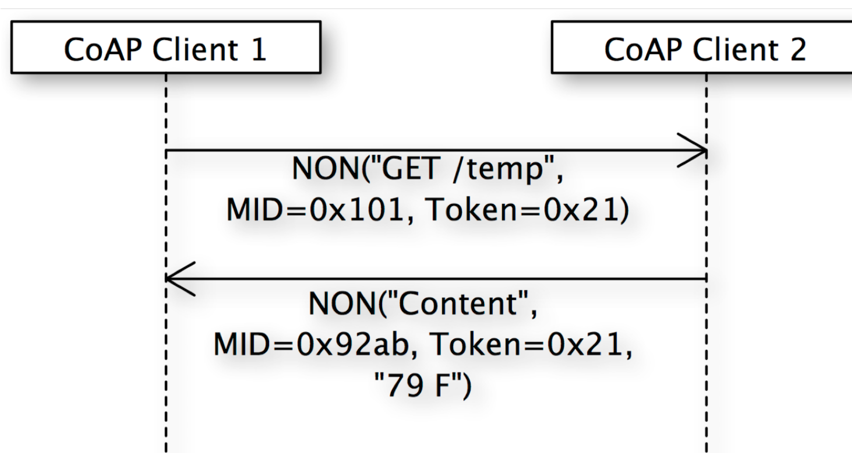
CoAP: Example 1



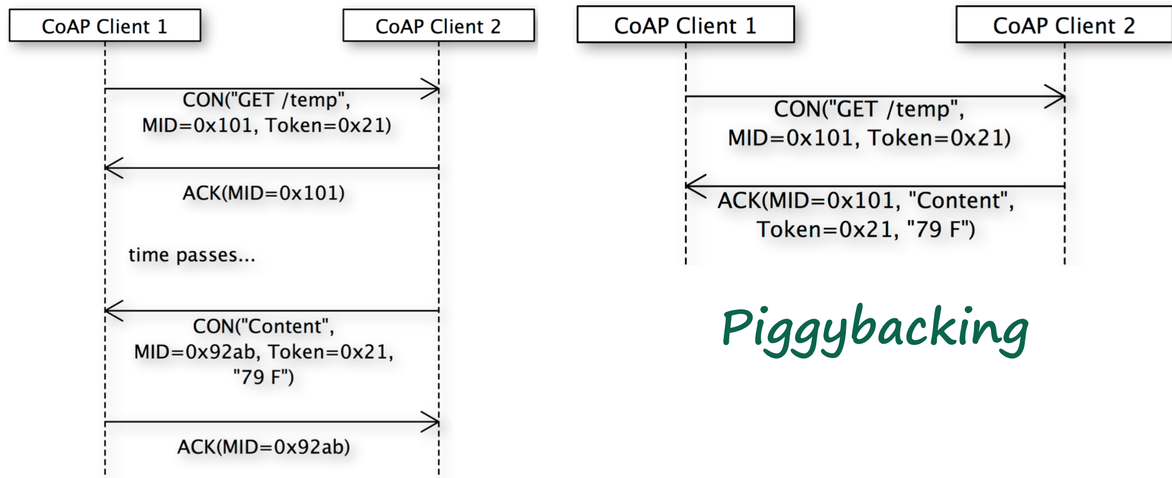
CoAP Messaging Basics [See rfc7252]



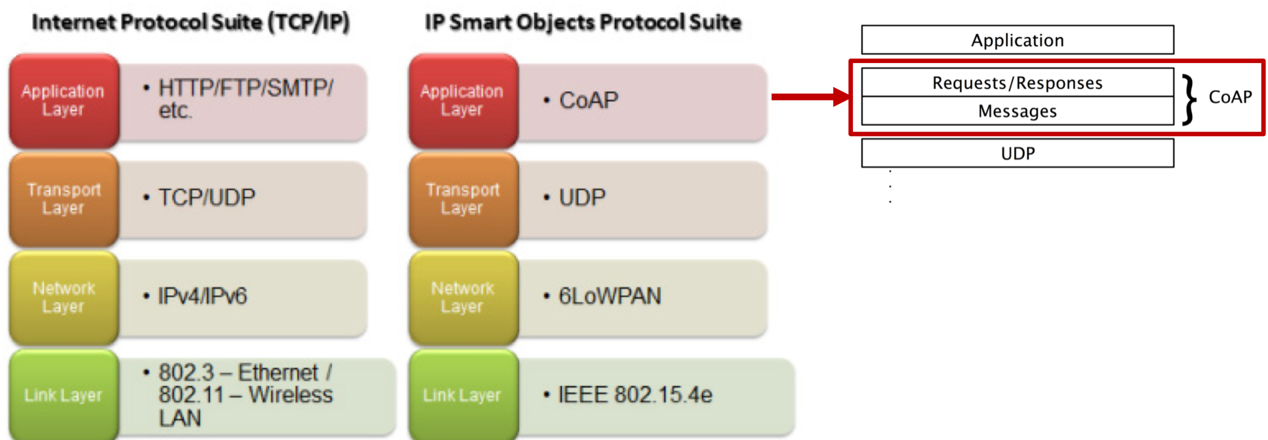
CoAP Messaging Basics



CoAP Messaging Basics



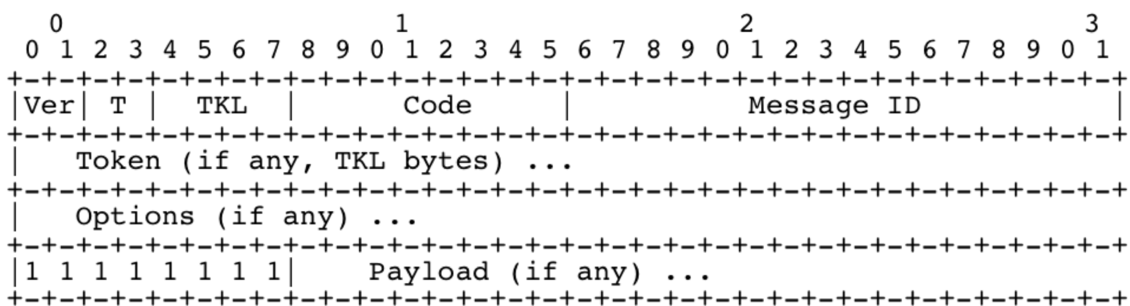
CoAP



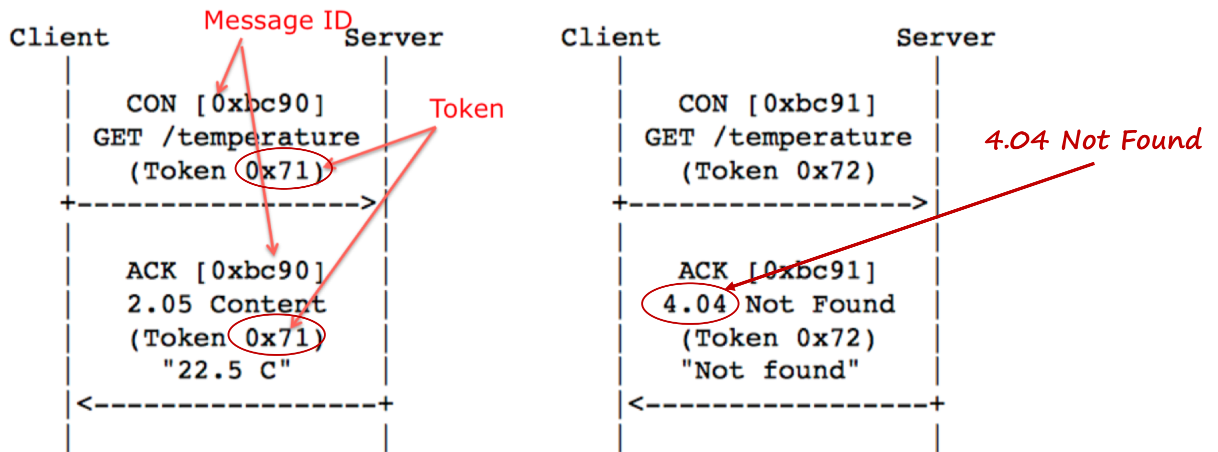
CoAP Request/Response Layer

- Responsible for transmission of requests and responses
- This is where REST-based communication occurs:
  - REST requests are piggybacked on *Confirmable* or *Non-confirmable* message.
  - REST responses are piggybacked on the related Acknowledgement message.
- CoAP uses tokens to match request/response in asynchronous communications.

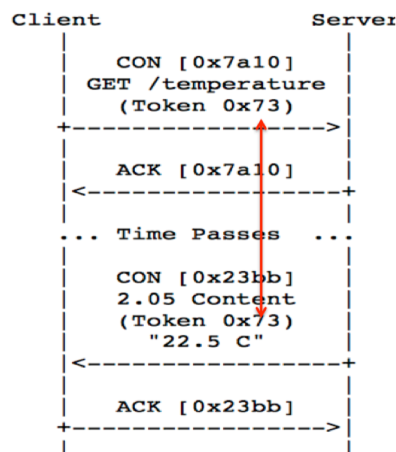
Message Header (4 bytes)



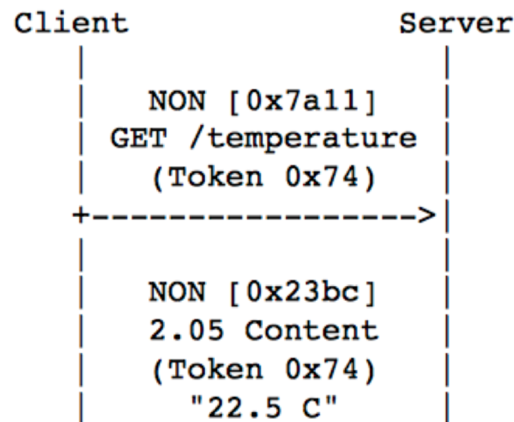
CoAP Request/Response Layer



A GET Request with Separate Response



A Request and a Response Carried in Non-confirmableMessages



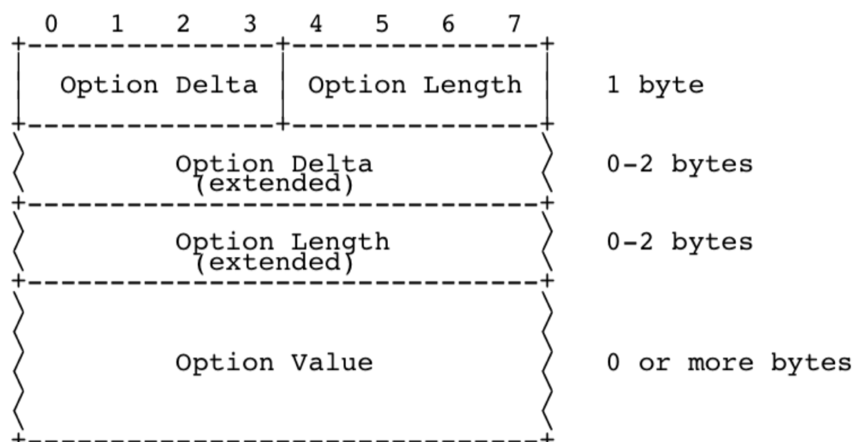
CoAP Transaction Layer

- Handles single message exchange between end points.
- Four message types:
  - Confirmable
    - Must be acknowledged by the receiver with an ACK packet
  - Non-confirmable [fire and forget]
    - No ACK needed.
  - Acknowledgement
    - ACKs a Confirmable.
  - Reset
    - Indicates a Confirmable message has been received but context is missing for processing
    - This condition is usually caused when the receiving node has rebooted and has forgotten some state that would be required to interpret the message.

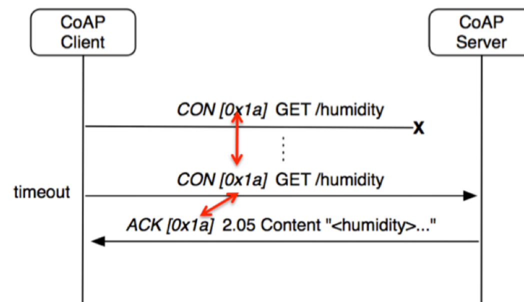
CoAP Reliability

- CoAP provides reliability without using TCP as transport protocol.
- CoAP enables asynchronous communication
  - For example, when CoAP server receives a request which it cannot handle immediately, it first ACKs the reception of the message and sends back the response in an off-line fashion
  - The transaction layer also supports multicast and congestion control.

Option Format



Dealing with Packet Loss



- Stop and Wait approach
- Repeat a request after a time-out in case ACK (or RST) is not coming back

Back-Off Details

- Initial time-out set to
  - Rand [ACK_TIMEOUT, ACK_TIMEOUT * ACK_RANDOM_FACTOR]
- When time-out expires and the transmission counter is less than MAX_RETRANSMIT
  - retransmit
  - Increase transmission counter
  - double the time-out value
- The procedure is repeated until
  - A ACK is received
  - A RST message is received
  - the transmission counter exceeds MAX_RETRANSMIT
  - the total attempt duration exceeds MAX_TRANSMIT_WAIT (93s)

COAP Observation

- PROBLEM:
  - REST paradigm is often ``PULL" type, that is, data is obtained by issuing an explicit request
  - Information/data in WSN is often periodic/triggered (e.g., get me a temperature sample every 2 seconds or get me a warning if temperature goes below 5°C)
- SOLUTION
  - Use Observation on COAP resources

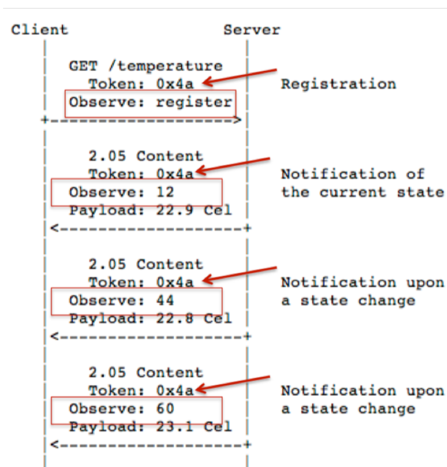
CoAP Efficiency

- Recall that CoAP design goals is to have small message overhead and limited fragmentation.
- CoAP uses compact fixed-length 4-byte binary header followed by compact binary options.
- Typical request with all encapsulation has a 10-20 byte header.
- CoAP implements an observation relationship whereby an ``observer" client registers itself using a modified GET to the server.
- When resource (object) changes state, server notifies the observer.

Observation

client makes a request to server with the OBSERVE option in the header set to 0

This indicates a registration request (value must be 0).



CoAP Resource Discovery

- Not the same as service discovery. Service discovery is at a lower level. We don't even know if services are available or how they communicate.
- We might register a printer, for example, with a discovery service and find it later on the fly.
- With resource discovery, we know we are looking for web resources.
- Links are returned. HATEOAS.
- Links may include a rel attribute.
- A well known resource is used to discover other resources.
- Perform a GET on the well known resource. Returned content is a list of links with REL attributes.
- Resource directories may be used to register services. Registrations are simply POSTs with links. PUTs are used for updates. GETs for discovery.

Getting Started with CoAP

- Open source implementations:
  - Java CoAP LibraryCalifornium
  - C CoAP LibraryErbium
  - libCoAPC Library
  - jCoAPJava Library
  - OpenCoAPC Library
  - TinyOS and Contiki include CoAP support
- Firefox has a CoAPplugin called Copper
- Wireshark has CoAP plugin

Internet Protocol Suite

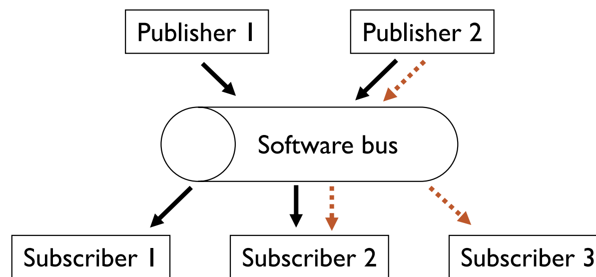
We are still here!

HTTP, Websockets, DNS, XMPP, MQTT, CoAp	Application layer
TLS, SSL	Application Layer (Encryption)
TCP, UDP	Transport
IP(V4, V6), 6LowPAN	Internet Layer
Ethernet, 802.11 WiFi, 802.15.4	Link Layer

Principles: Layering, modularity, separation of concerns
Each layer focuses on a particular set of concerns and abstracts these concerns from the layer above.

Background: Publish/Subscribe

- Achieved by publish/subscribe paradigm
  - Idea: Entities can publish data under certain names
  - Entities can subscribe to updates of such named data
- Conceptually: Implemented by a software bus
  - Software bus stores subscriptions, published data; names used as filters; subscribers notified when values of named data changes
  - Variations
    - *Topic-based* P/S – inflexible
    - *Content-based* P/S – use general predicates over named data



Message Queue Telemetry Transport (MQTT)

- A light-weight, open and scalable protocol for the *Internet of Things*
  - An ISO standard publish-subscribe-based messaging protocol for use on top of the TCP/IP protocol
  - Designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited
  - A variation of the main protocol aimed at embedded devices on non-TCP/IP networks, such as ZigBee

MQTT Pub/Sub Protocol

- MQ Telemetry Transport (MQTT) is a lightweight broker-based publish/subscribe messaging protocol.
- MQTT is designed to be open, simple, lightweight and easy to implement.
  - These characteristics make MQTT ideal for use in constrained environments, for example in IoT.
    - Where the network is expensive, has low bandwidth or is unreliable
    - When run on an embedded device with limited processor or memory resources;
- A small transport overhead (the fixed-length header is just 2 bytes), and protocol exchanges minimized to reduce network traffic
- MQTT was developed by Andy Stanford-Clark of IBM, and Arlen Nipper of Cirrus Link Solutions.

MQTT

- It supports publish/subscribe message pattern to provide one-to-many message distribution and decoupling of applications
- A messaging transport that is agnostic to the content of the payload
- The use of TCP/IP to provide basic network connectivity
- Three qualities of service for message delivery:
  - "At most once", where messages are delivered according to the best efforts of the underlying TCP/IP network. Message loss or duplication can occur.
    - This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  - "At least once", where messages are assured to arrive but duplicates may occur.
  - "Exactly once", where message are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

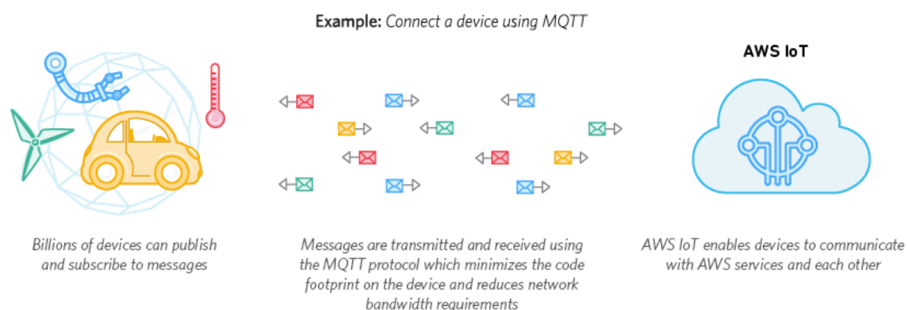
MQTT Message Format

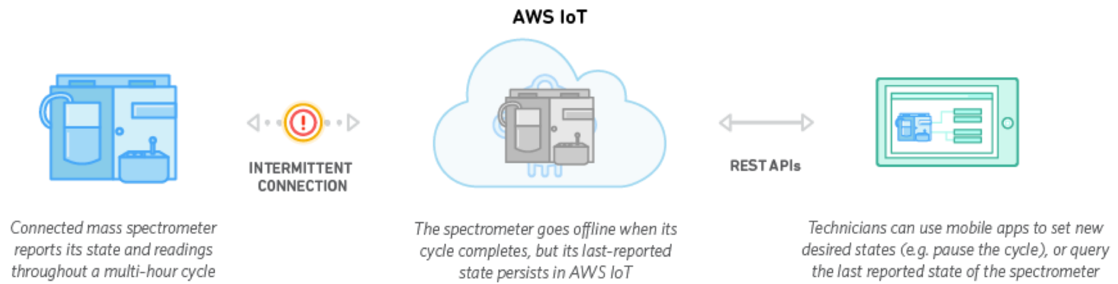
- The message header for each MQTT command message contains a fixed header.
- Some messages also require a variable header and a payload.
- The format for each part of the message header:

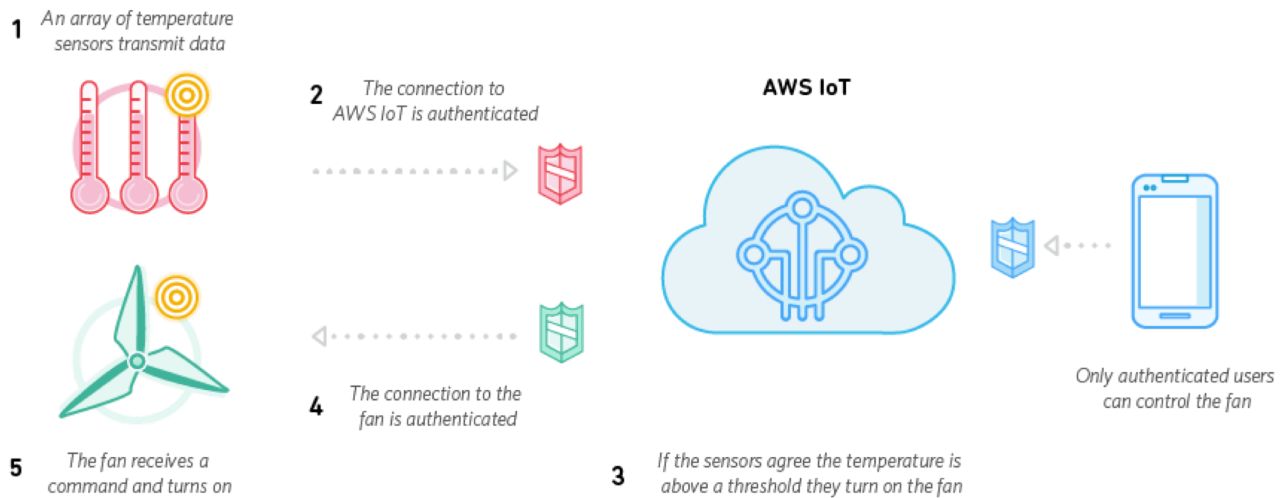
bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	Remaining Length							

— DUP: Duplicate delivery
— QoS: Quality of Service
— RETAIN: RETAIN flag
—This flag is only used on PUBLISH messages. When a client sends a PUBLISH to a server, if the Retain flag is set (1), the server should hold on to the message after it has been delivered to the current subscribers.
—This allows new subscribers to instantly receive data with the retained, or Last Known Good, value.

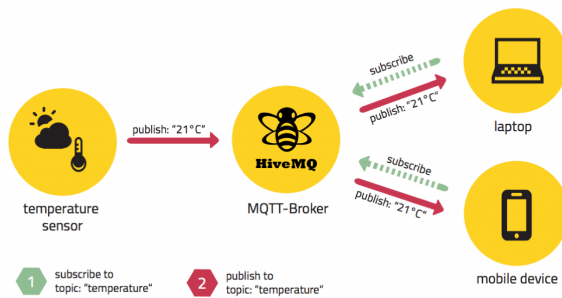
Source: MQTT V3.1 Protocol Specification, IBM, http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html




Example: Analyze chemicals in a sample with a mass spectrometer
AWS IoT
INTERMITTENT CONNECTION
REST APIs
Connected mass spectrometer reports its state and readings throughout a multi-hour cycle
The spectrometer goes offline when its cycle completes, but its last-reported state persists in AWS IoT
Technicians can use mobile apps to set new desired states (e.g. pause the cycle), or query the last reported state of the spectrometer


Example: Authenticate connections between sensors, a device and an application
1 An array of temperature sensors transmit data
2 The connection to AWS IoT is authenticated
AWS IoT
Only authenticated users can control the fan
4 The connection to the fan is authenticated
5 The fan receives a command and turns on
3 If the sensors agree the temperature is above a threshold they turn on the fan

Sensor Readings with pub/sub



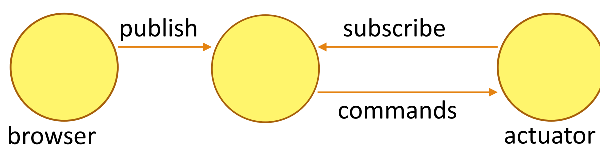
Decoupled in space and time.
The clients do not need each other's IP address and port (space) and They do not need to be running at the same time (time).

The broker's IP and port must be known by clients.

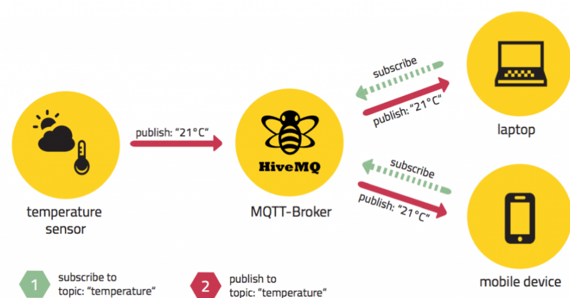
Namespace hierarchy used for topic filtering.

It may be the case that a published message is never consumed by any subscriber.

Actuators too!

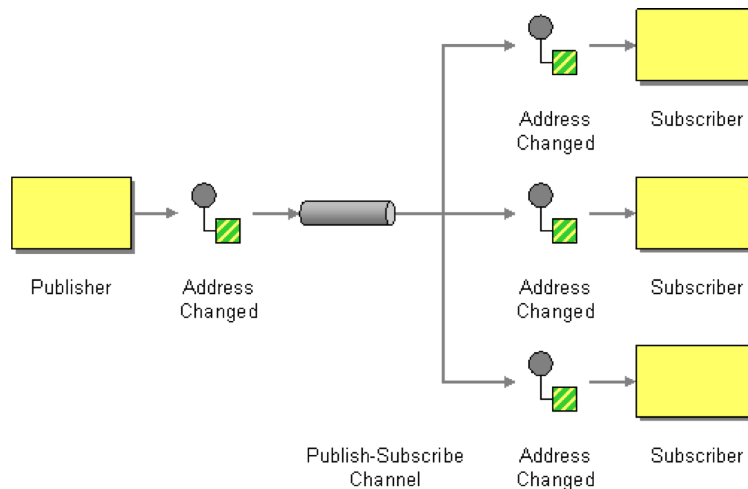


If my toaster is a command subscriber, I can control it over the web!



In the picture, replace the laptop with an actuator, subscribing to a command topic, say, device3/commands. Replace the sensor with a browser ending commands to device3/commands.

From: Enterprise Integration Patterns Book Hohpe and Woolf



MQTT

- Message Queuing Telemetry Transport (old acronym) since the 1990's
  – "Telemetry" is from the Greek remote measure
- Created by Andy Stanford-Clark (IBM) and Alan Nipper - now part of OASIS
- Version 3.1 released royalty free in 2010 (IBM)
- Originally built for oil pipeline monitoring over satellite connections.
- Satellites appropriate because pipelines are remote

MQTT

- Built for a proprietary embedded system now shifting to IoT
- You can send anything as a message, up to 256 MB.
- Built for unreliable networks
- Enterprise scale implementations down to hobby projects
- Decouples readers and writers
- Message have a topic, quality of service and retain status associated with them.

MQTT

- MQTT Runs over TCP or TLS.
- May use `Websockets` from within a browser.
- MQTT–SN uses UDP packets or serial communication rather than TCP
- MQTT-SN may run over Bluetooth Low Energy (BLE).
- Open, industry agnostic, no polling. What does it mean to be open?
- Hierarchical topic namespace and subscriptions with wildcards. MQTT-SN has simpler topics.
- As soon as you subscribe you may receive the most recently published message. One message per topic may be retained by the broker.
- This feature provides for devices that transmit messages only on occasion. A newly connected subscriber does not need to wait. Instead, it receives the most recent message.

MQTT Last Will and Testament

- Publishing clients may connect with a last will and testament message.
- If publishing client has no data to send, it sends ping requests to the broker to inform the broker that it is still alive.
- If a publisher disconnects in a faulty manner (the broker will miss the ping requests), the broker tells all subscribers the last will and testament. This is for an ungraceful disconnect. How could this happen? Battery failure, network down, out of reach, etc.
- In the case of graceful disconnect, the publisher sends a DISCONNECT message to the broker – the broker will discard the LWT message.

MQTT Clients

- A client may publish or subscribe or do both.
- An MQTT client is any device from a micro controller up to a full fledged server, that has an MQTT library running and is connecting to an MQTT broker over any kind of network. (from HiveMQ MQTT Essentials)
- A client is any device that has a TCP/IP stack and speaks MQTT. MQTT-SN does not require TCP.
- Client libraries widely available (Android, Arduino, iOS, Java, Javascript, etc.)
- No client is connected directly to any other client

MQTT Broker

- The broker is primarily responsible for receiving all messages, filtering them, decide who is interested in it and then sending the message to all subscribed clients. (From HiveMQ MQTT Essentials)
- May authenticate and authorize clients.
- Maintains sessions and missed messages
- Maintains a hierarchical namespace for topics and allows subscribers (but not publishers) to use wildcards (+ and #).

Topics are organized into a Hierarchical namespace

Suppose a client publishes to mm6House/Kitchen/Sensor/Temperature
Another client might subscribe to:
    mm6House/Kitchen/Sensor/Temperature
Or, using a single level wildcard (+)
    mm6House/Kitchen/+/Temperature   // All children of Kitchen that
    // have a child called
Temperature
Or, using a multi level wildcard (#)
    mm6House/Kitchen/#   // Goes deep
•The # must be the last character.
•To see every message, subscribe to #

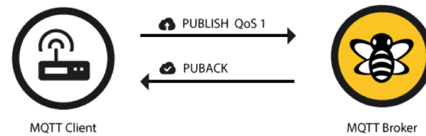
MQTT Qualities of Service (QoS)

- QoS defines how hard the two parties will work to ensure that messages arrive.
- Qualities of service: once and only once, at least once, at most once – fire and forget. These qualities of service exist between the client and a broker. More quality implies more resources.
- From publishing client to broker, use the QoS in the message sent by the publishing client.
- From broker to subscribing client, use the QoS established by the client's original subscription. A QoS may be downgraded if the subscribing client has a lower QoS than a publishing client.
- If a client has a subscription and the client disconnects, if the subscription is durable it will be available on reconnect.

MQTT Qualities of service (QoS 0)

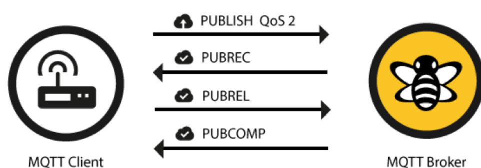
- From HiveMQ
  - QoS 0 implies at most once
  - Fire and forget. No acknowledgement from the broker. No client storage or redelivery. Still uses TCP below the scenes.
  - Use QoS 0 when you have a stable connection and do not mind losing an occasional packet. You are more interested in performance than reliability.

MQTT Qualities of service (QoS 1)



- From HiveMQ
  - QoS 1 implies at least once
  - Client will perform retries if no acknowledgement from the broker.
  - Use QoS 1 when you cannot lose an occasional message and can tolerate duplicate messages arriving at the broker. And you do not want the performance hit associated with QoS 2.

MQTT Qualities of service (QoS 2)



PUBREC = publish received
PUBREL = publish released
PUBCOMP = publish complete

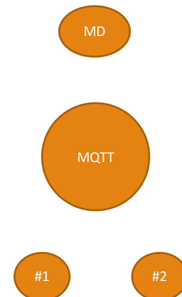
- From HiveMQ
- QoS 2 implies exactly once
- Client will save and retry and server will discard duplicates.
- Use this if it is critical that every message be received once and you do not mind the drop in performance.
- QoS 1 and QoS 2 messages will also be queued for offline subscribers - until they become available again. This happens only for clients requesting persistent connections when subscribing.

MQTT From OASIS

- MQTT is being used in sensors communicating to a broker via satellite links, Supervisory Control and Data Acquisition (SCADA), over occasional dial-up connections with healthcare providers (medical devices), and in a range of home automation and small device scenarios. MQTT is also ideal for mobile applications because of its small size, minimized data packets, and efficient distribution of information to one or many receivers (subscribers).

MQTT from Oracle (a drone application)

```
*Master Drone* successfully connected
[Drone #2] successfully connected
[Drone #1] successfully connected
[Drone #2] subscribed to the java-magazine-mqtt/drones/altitude topic
[Drone #1] subscribed to the java-magazine-mqtt/drones/altitude topic
*Master Drone* subscribed to the java-magazine-mqtt/drones/altitude topic
Message '[Drone #1] is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
Message '*Master Drone* is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
Message '[Drone #2] is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
[Drone #2] received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
[Drone #2] received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
Message 'COMMAND:GET_ALTITUDE:[Drone #1]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #1] altitude: 3746 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
Message 'COMMAND:GET_ALTITUDE:[Drone #2]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
[Drone #2] altitude: 4224 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
Message 'COMMAND:GET_ALTITUDE:[Drone #1]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #1] altitude: 433 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
```



MQTT From IBM

- The IBM Bluemix Internet of Things (IoT) service provides a simple but powerful capability to interconnect different kinds of devices and applications all over the world. What makes this possible? The secret behind the Bluemix IoT service is MQTT, the Message Queue Telemetry Transport.

MQTT From AWS

- The AWS IoT message broker is a publish/subscribe broker service that enables the sending and receiving of messages to and from AWS IoT. When communicating with AWS IoT, a client sends a message addressed to a topic like Sensor/temp/room1. The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as publishing.
- MQTT is a widely adopted lightweight messaging protocol designed for constrained devices. For more information, see MQTT.
- Although the AWS IoT message broker implementation is based on MQTT version 3.1.1, it deviates from the specification as follows: (several deviations from the standard are listed.)

IoT Hub enables devices to communicate with the IoT Hub device endpoints using the MQTT v3.1.1 protocol on port 8883 or MQTT v3.1.1 over WebSocket protocol on port 443. IoT Hub requires all device communication to be secured using TLS/SSL (hence, IoT Hub doesn't support non-secure connections over port 1883).
Microsoft lists several deviations from the standard as well.

MQTT From Microsoft Azure

Building Facebook Messenger

One of the problems we experienced was long latency when sending a message. The method we were using to send was reliable but slow, and there were limitations on how much we could improve it. With just a few weeks until launch, we ended up building a new mechanism that maintains a persistent connection to our servers. To do this without killing battery life, we used a protocol called MQTT that we had experimented with in Beluga. MQTT is specifically designed for applications like sending telemetry data to and from space probes, so it is designed to use bandwidth and batteries sparingly. By maintaining an MQTT connection and routing messages through our chat pipeline, we were able to often achieve phone-to-phone delivery in the hundreds of milliseconds, rather than multiple seconds.

From Facebook

The future?

- Ask Alexa to subscribe to kitchen/oven/temperature and kitchen/oven/timer