Internet of Things

DR. RAIHAN UL ISLAM

ASSOCIATE PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ROOM NO# AB3-1003
EMAIL: RAIHAN.ISLAM@EWUBD.EDU

MOBILE: +8801992392611

What is REST ?

REST means
REpresentational
State
Transfer

REpresentational ? State ? Transfer ?

- It represent the state of database at a time. But how???
- REST is an architectural style which is based on web-standards and the **HTTP** protocol.
- In a REST based architecture everything is a **Resource**.
- A resource is accessed via a common interface based on the HTTP standard methods.
- You typically have a REST server which provides access to the resources and a REST client which accesses and modifies the REST resources.

REpresentational ? State ? Transfer ?

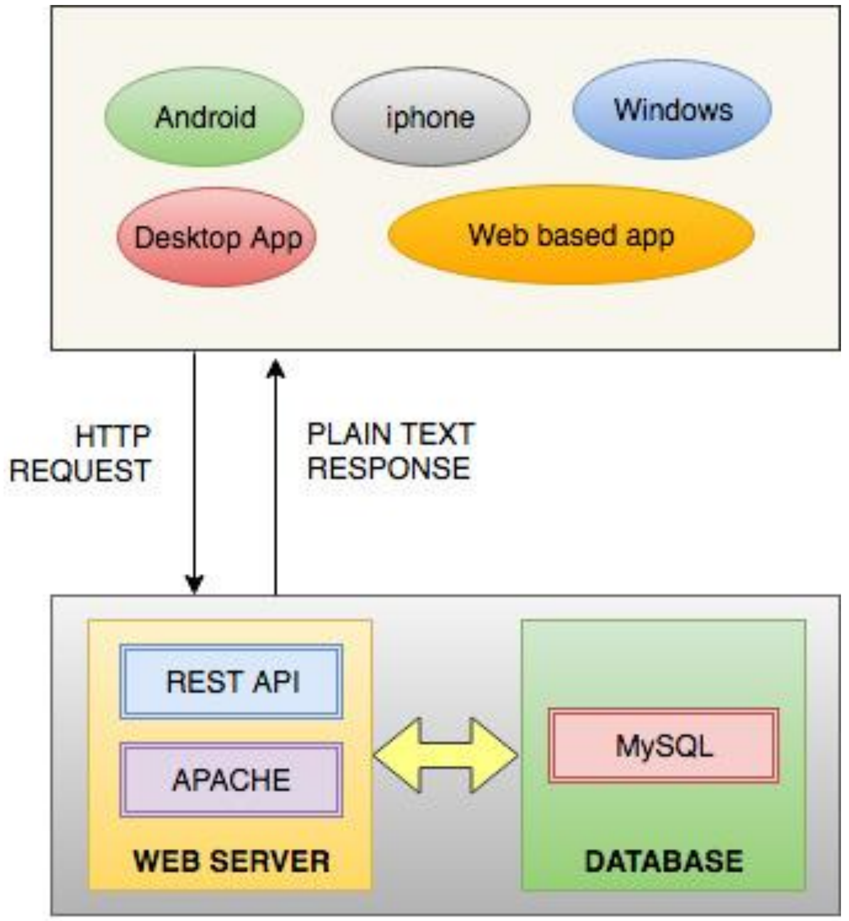
- ❑ Every resource should support the HTTP common operations.
- ❑ Resources are identified by global IDs (which are typically **URIs** or **URLs**).
- ❑ REST allows that resources have different representations, e.g., text, XML, JSON etc.
- ❑ **Stateless** in nature. **Excellent** for **distributed system.**
- ❑ Stateless components can be freely redeployed if something fails, and they can **scale** to accommodate load changes.
- ❑ This is because any request can be directed to any instance of a component.

HTTP Methods

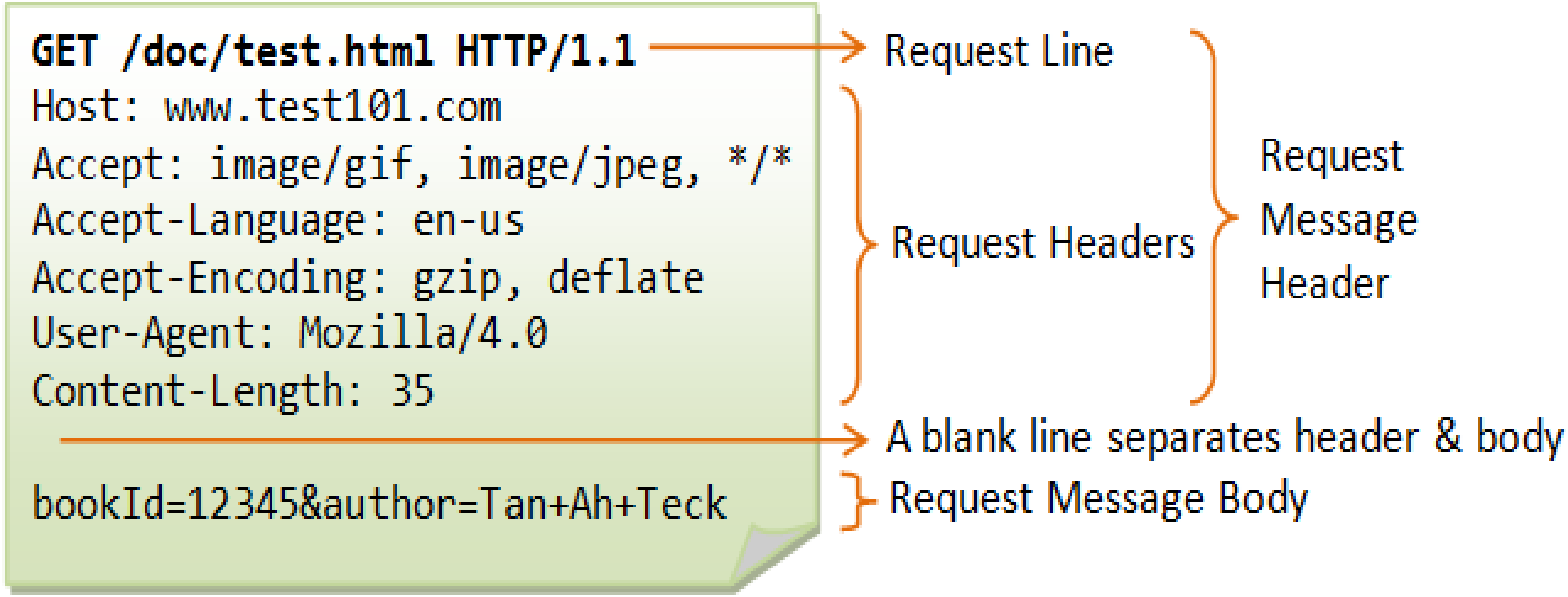
The *PUT*, *GET*, *POST* and *DELETE* methods are typically used in REST based architectures. The following table gives an explanation of these operations:

HTTP Method	CRUD Operation	Description
POST	INSERT	Addes to an existing resource
PUT	UPDATE	Overrides existing resource
GET	SELECT	Fetches a resource. The resource is never changed via a GET request
DELETE	DELETE	Deletes a resource

Architecture:



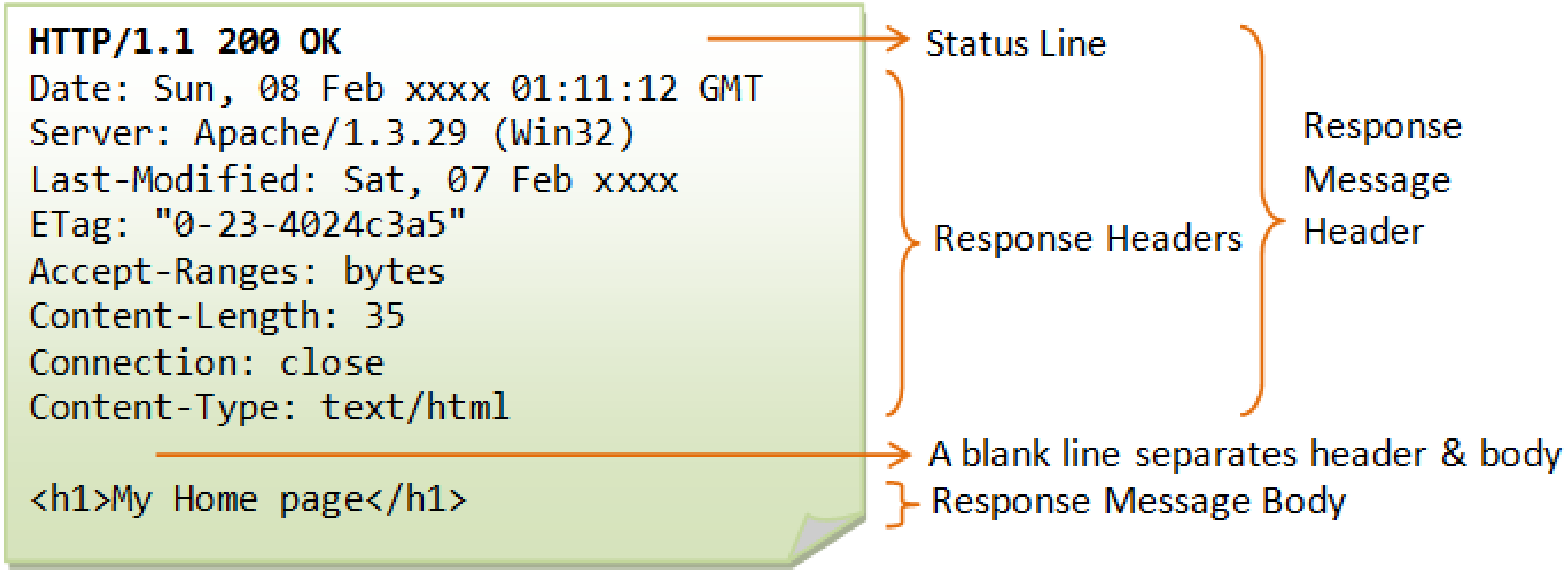
HTTP Request Example:



```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

HTTP Response Example:



```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

HTTP REST Request:

GET https://www.myhost.com/api/v1/user/1/cities

Read, All the cities for user whose id is 1

```
GET /user/1/cities http/1.1
host: https://www.myhost.com/api/v1
Content-Type: application/json
Accept-Language: us-en
state_id: 2
```

HTTP REST Response:

```
HTTP/1.1 200 OK (285ms)
Date: Fri, 21 Apr 2017 10:27:20 GMT
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.1e-fips PHP/7.0.16
X-Powered-By: PHP/7.0.16
Content-Length: 109
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json; charset=UTF-8
```

```
{"status":"success","message":"City
List","data":[{"city_name":"Visakhapatnam"},{"city_name":"Vijayawada"}]}
```

HTTP Response Status Code:

1xx	Informational Codes
2xx	Successful Codes
3xx	Redirection Codes
4xx	Client Error Code
5xx	Server Error Codes

List at here: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Points to be noted

FOR REST APIS

Use Nouns but no verbs in path/URI:

Purpose	Method	Incorrect	Correct
Retrieves a list of users	GET	/getAllCars	/users
Create a new user	POST	/createUser	/users
Delete a user	DELETE	/deleteUser	/users/10
Get balance of user	GET	/getUserBalance	/users/11/balance

Use plural nouns:

Do not mix up singular and plural nouns. Keep it simple and use only plural nouns for all resources.

/cars instead of /car

/users instead of /user

/products instead of /product

/settings instead of /setting

GET method should not alter the state:

Use **PUT, POST** and **DELETE** methods instead of the **GET** method to alter the state.
Do not use **GET** method or Query parameters for state changes:

```
GET /users/711?activate or

GET /users/711/activate
```

Use sub-resources for relation:

If a resource is related to another resource use subresources

GET /cars/711/drivers/ (Returns a list of drivers for car 711)

GET /cars/711/drivers/4 (Returns driver #4 for car 711)

Use HTTP headers for serialization formats:

- Both, client and server need to know which format is used for the communication. The format has to be specified in the HTTP-Header.
- *Content-Type* defines the request format.
- *Accept* defines a list of acceptable response formats.

Versioning is important:

- Make the API Version mandatory and do not release an unversioned API.
- Use a simple ordinal number and avoid dot notation such as 2.5
- We are using the url for the API versioning starting with the letter “v”

  /blog/api/v1

Handle Errors with HTTP Status code:

200	OK (Everything is working)	403	Forbidden (The server understood the request, but is refusing it or the access is not allowed)
201	OK (New resource has been created)	404	Not found (There is no resource behind the URI)
204	OK (Resource successfully deleted)	405	Method not allowed
400	Bad Request (The request was invalid or cannot be served. The exact error should be explained in the error payload. E.g. „The JSON is not valid“)	408	Request timeout
401	Unauthorized (The request requires an user authentication)	500	Internal server error

Filtering:

- Use a unique query parameter for all fields or a query language for filtering.
  - GET /cars?color=red (Returns a list of red cars)
  - GET /users?name=tom (Returns a list of users whose name matches tom)

Sorting:

- Allow ascending and descending sorting over multiple fields.

GET /cars?sort=-manufacturer,+model

This returns a list of cars sorted by descending manufacturers and ascending models)

Paging:

- Use limit and offset. It is flexible for the user and common in leading databases.

The default should be limit=20 and offset=0

GET /cars?offset=10&limit=5

Aliases for common queries:

- To make the API experience more pleasant for the average consumer, consider packaging up sets of conditions into easily accessible RESTful paths.

For example consider following use case:

POST /users (User Register: Creates a new user)

POST /users/login (User Login: Creates auth token for user authentication)

Want to develop REST APIs easily???

Checkout Slim - A microframework for php.

Documentation: https://www.slimframework.com/docs/

Introduction To CoAP

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained devices(such as microcontrollers) and constrained networks in the **Internet of Things.**

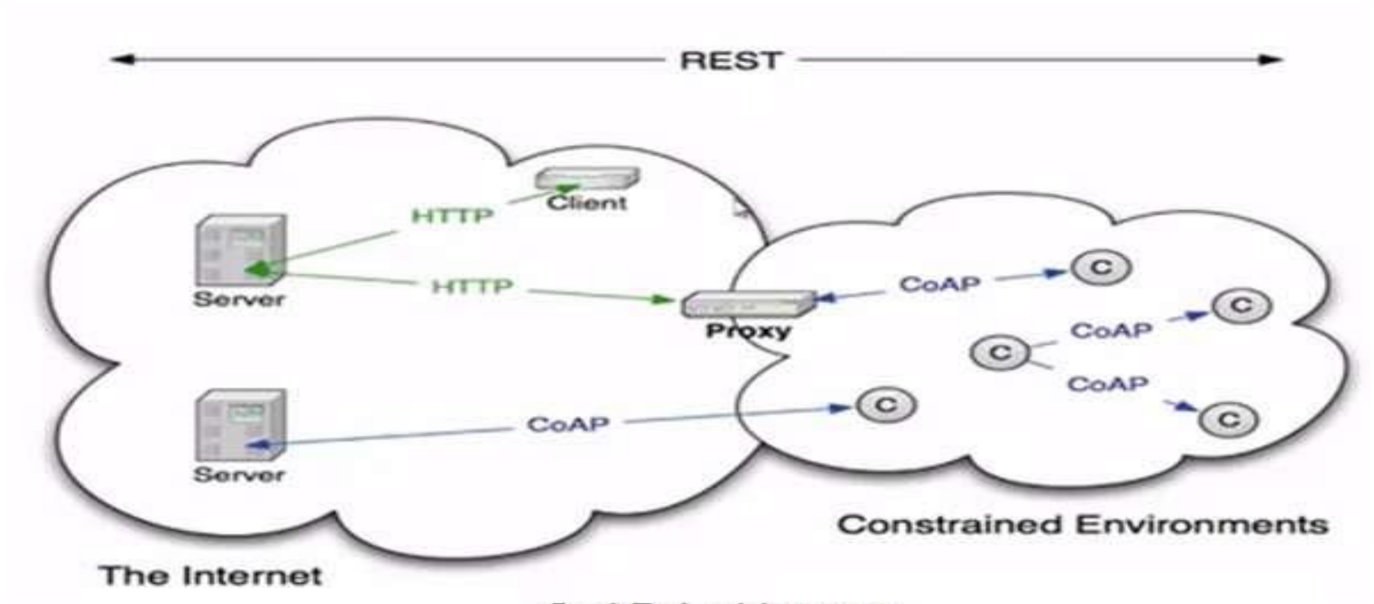
This protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation.

Architecture Of CoAP

- It is specified in RFC 7252.It is open IETF standard
- It is very efficient RESTful protocol.
- Easy to proxy to/from HTTP.
- It is Embedded web transfer protocol
- It uses asynchronous transaction model.
- UDP is binding with reliability and multicast support.
- GET, POST, PUT and DELETE methods are used.
- uses subset of MIME types and HTTP response codes.
- Uses built in discovery mechanism.

Architecture Of CoAP

- ❑URI is supported.
- ❑ It uses small and simple 4 byte header.
- ❑Supports binding to UDP, SMS and TCP.
- ❑ DTLS based PSK, RPK and certificate security is used.



CoAP Architecture

CoAP Structure

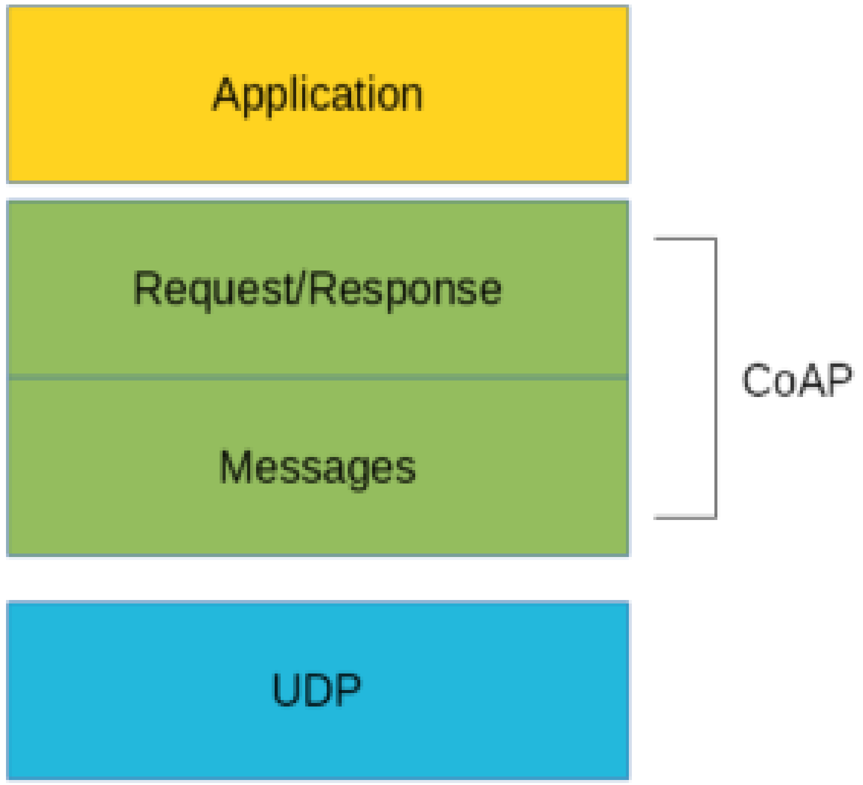
There are two different layers that make CoAP protocol:

1)Messages

2)Request/Response.

The **Messages layer** deals with UDP and with asynchronous messages.

The **Request/Response layer** manages request/response interaction based on request/response messages.



CoAP Structure

Message layer supports 4 types of messages:

1. Confirmable
2. Non-confirmable
3. Acknowledgment
4. Reset

Message layer is meant for Re-transmitting lost packets. Request/Response layer contains methods like GET,PUT,POST and DELETE.

Common Terms in CoAP

Client: The entity that sends a request and the destination of the response

Server: The entity that receives a request from a client and sends back a response to the client

Endpoint: An entity that participates in the CoAP protocol. Usually, an Endpoint is identified with a host

Sender: The entity that sends a message

Recipient: The destination of a message

CoAP Messages Model

This is the lowest layer of CoAP.

This layer deals with UDP exchanging messages between endpoints.

Each CoAP message has a unique ID; this is useful to detect message duplicates.

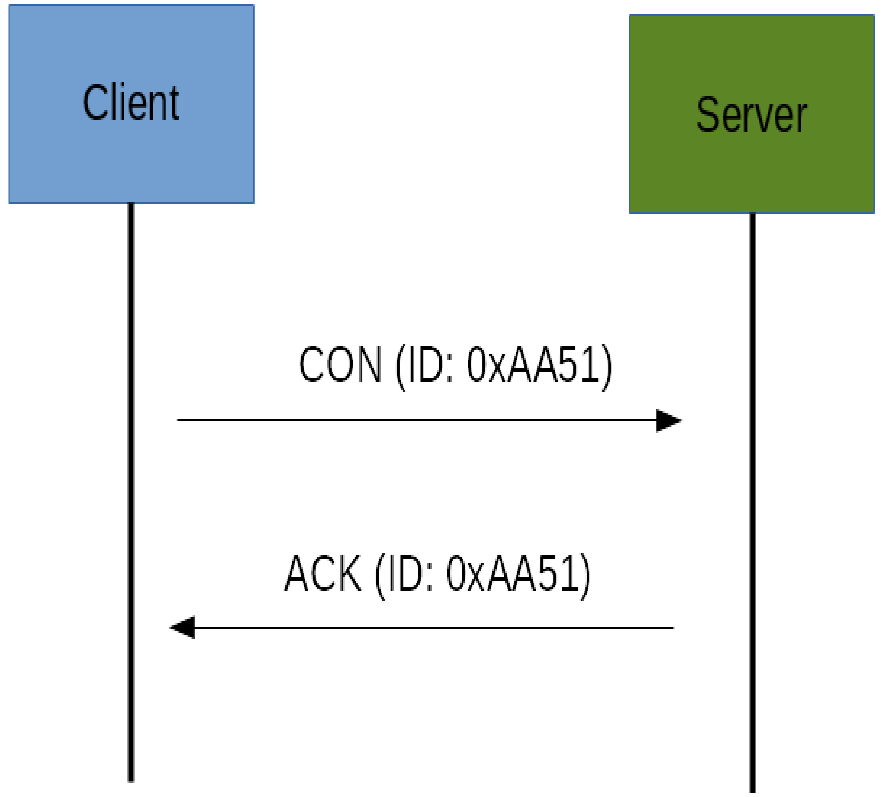
The CoAP protocol uses two kinds of messages:

1. Confirmable message - this is a reliable message.
2. Non-confirmable message-this are unreliable messages.

CoAP Messages Model

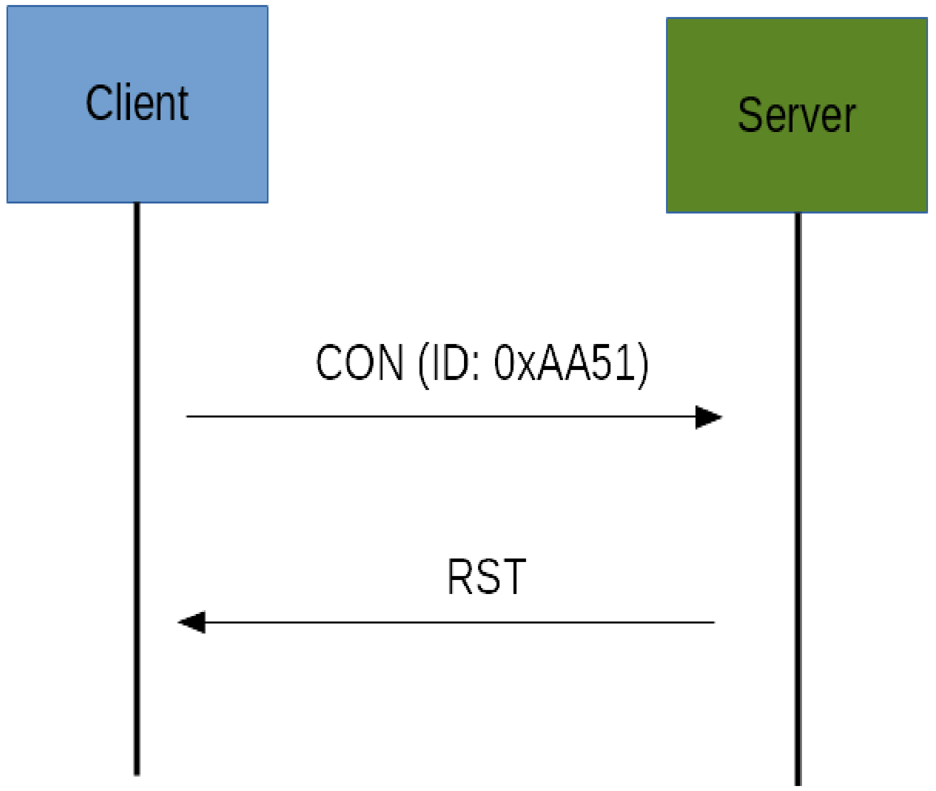
In CoAP, a reliable message is obtained using a **Confirmable message (CON).**

A Confirmable message is sent again and again until the other party sends an **acknowledge message (ACK).** The ACK message contains the same ID of the **confirmable message (CON).**



CoAP Messages Model

if the server has troubles managing the incoming request, it can send back a **Rest message (RST)** instead of the Acknowledge message (ACK)



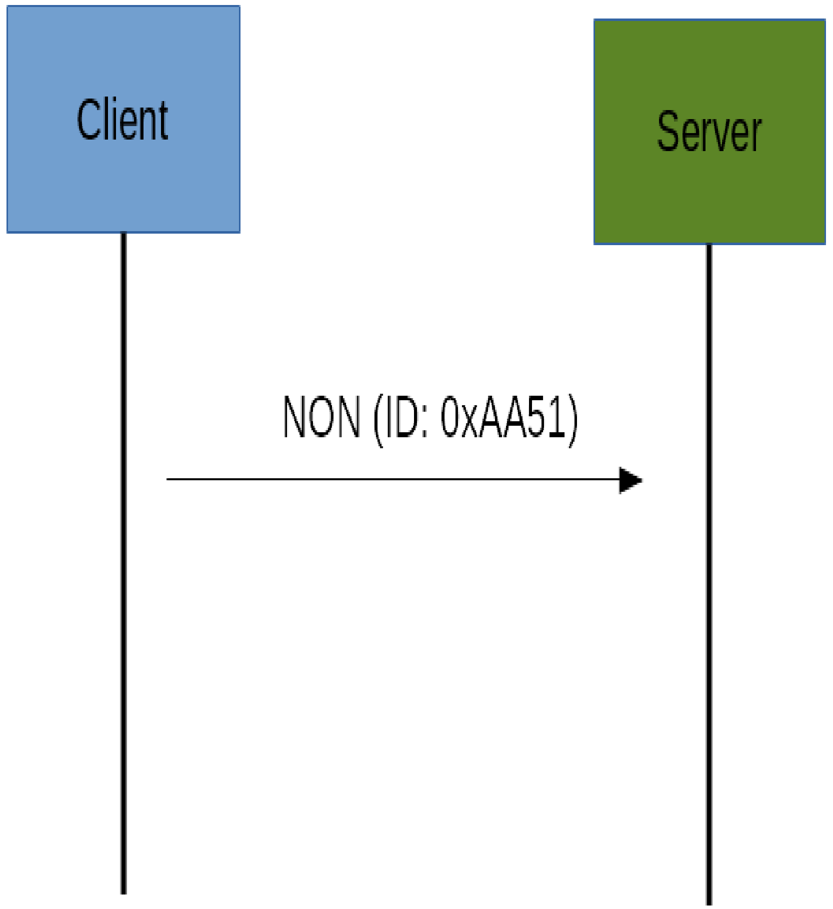
CoAP Messages Model

The other message category is the **Non-confirmable (NON)** messages. These are messages that don't require an Acknowledge by the server.

They are unreliable messages or in other words messages that do not contain critical information that must be delivered to the server.

To this category belongs messages that contain values read from sensors.

Even if these messages are unreliable, they have a unique ID.



CoAP Request/Response Model

The CoAP Request/Response is the second layer in the CoAP abstraction layer.

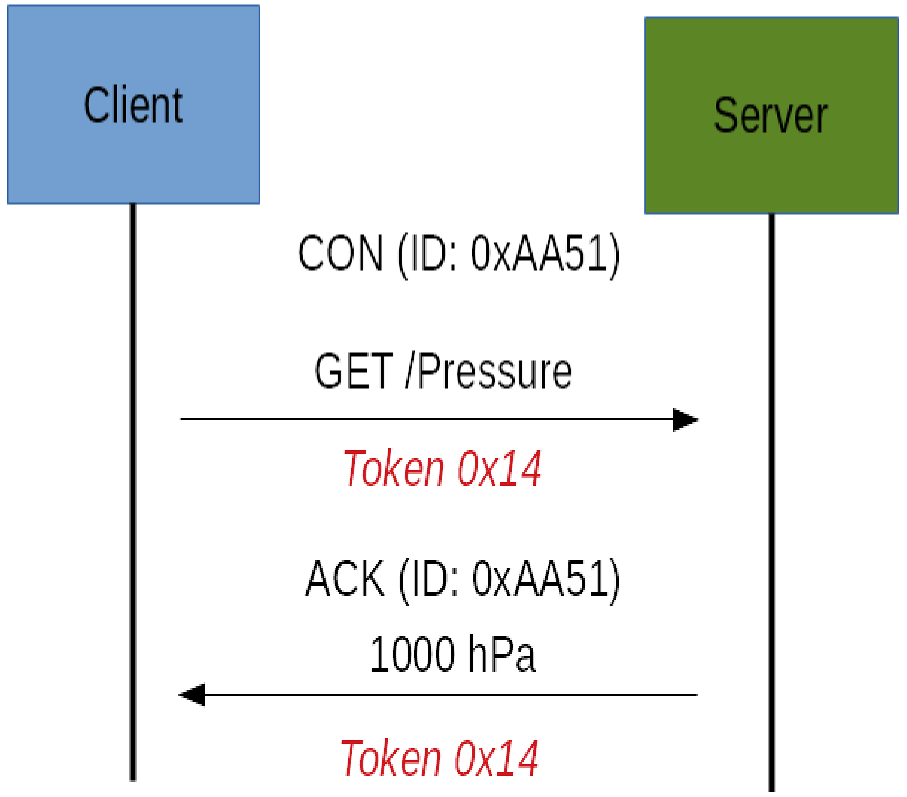
The request is sent using a Confirmable (CON) or Non-Confirmable (NON) message.

There are several scenarios depending on if the server can answer immediately to the client request or the answer if not available.

CoAP Request/Response Model

If the server can answer immediately to the client request, then if the request is carried using a Confirmable message (CON), the server sends back to the client an Acknowledge message containing the response or the error code.

The Token is different from the Message-ID and it is used to match the request and the response.

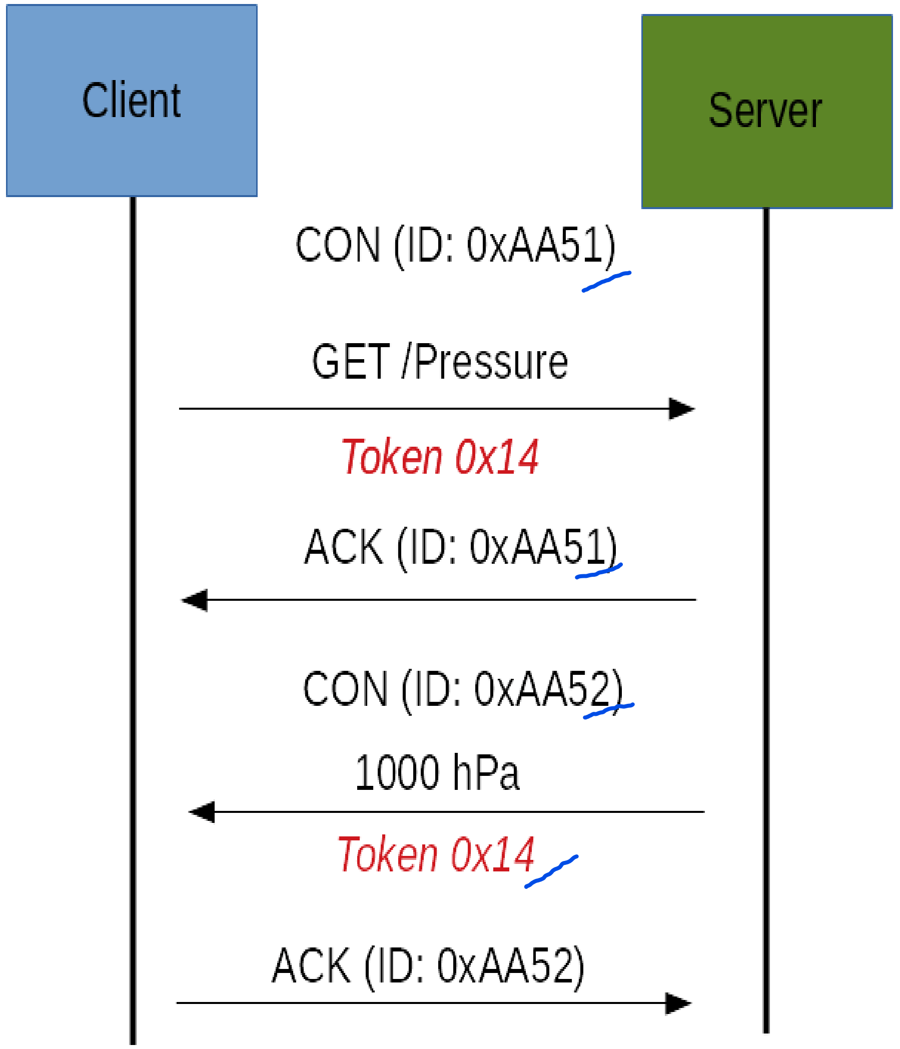


CoAP Request/Response Model

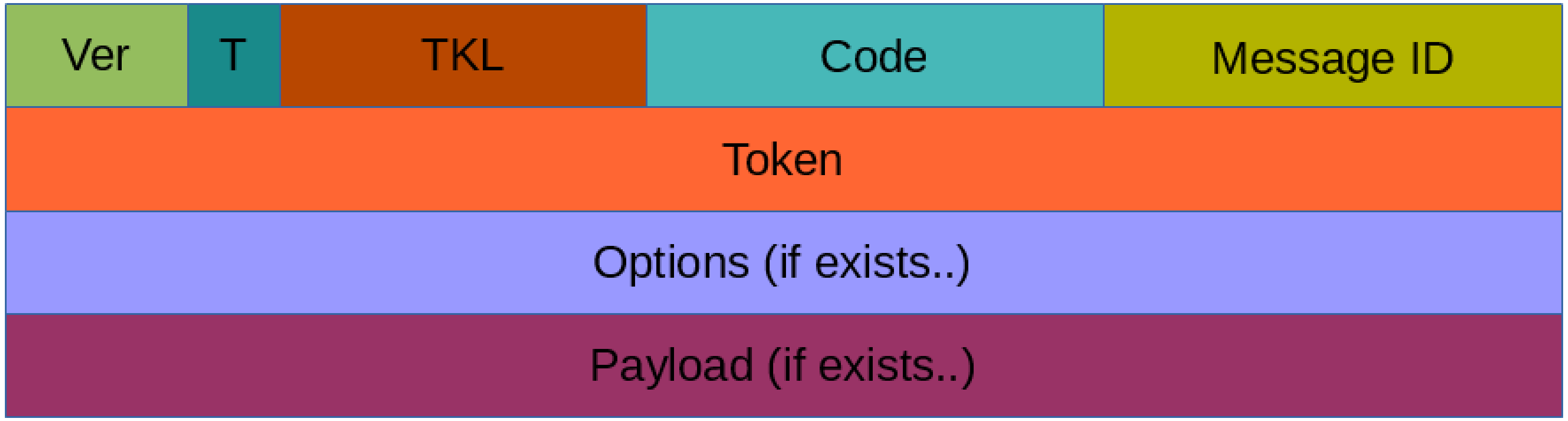
If the server can't answer to the request coming from the client immediately, then it sends an Acknowledge message with an empty response.

As soon as the response is available, then the server sends a new Confirmable message to the client containing the response.

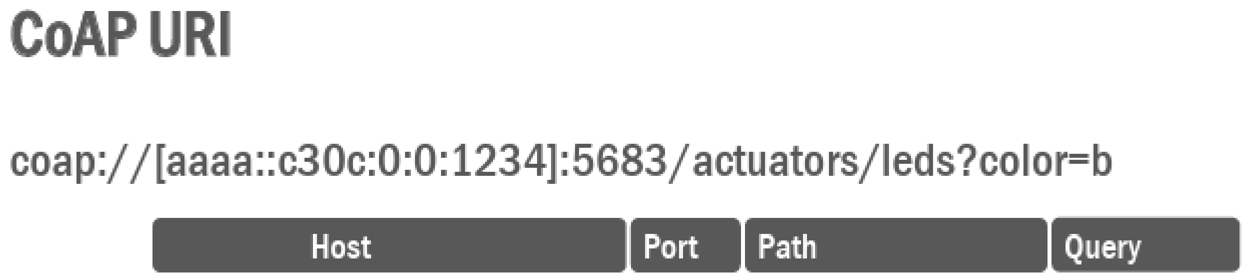
At this point, the client sends back an Acknowledge message:



CoAP Message Format



Where:
Ver: It is a 2 bit unsigned integer indicating the version
T: it is a 2 bit unsigned integer indicating the message type: 0 confirmable, 1 non-confirmable
TKL: Token Length is the token 4 bit length
Code: It is the code response (8 bit length)
Message ID: It is the message ID expressed with 16 bit
And so on.



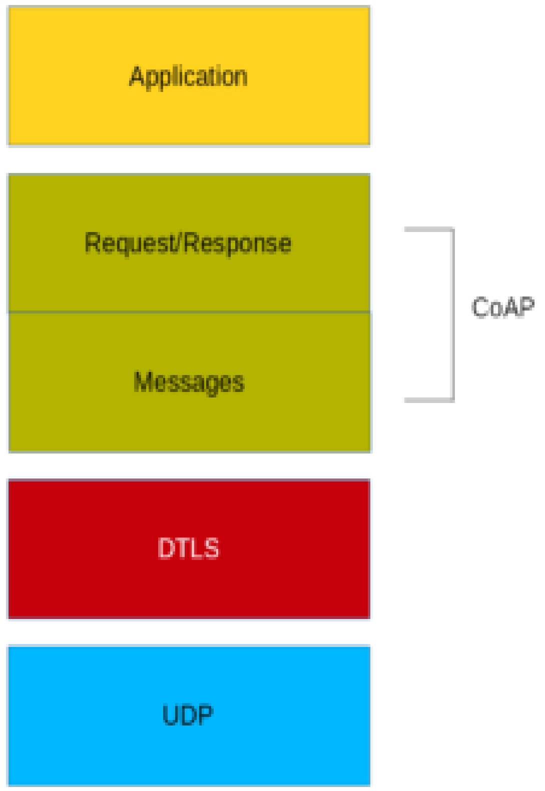
CoAP Security Aspects

CoAP uses UDP to transport information. CoAP relies on UDP security aspects to protect the information.

As HTTP uses TLS over TCP, **CoAP uses Datagram TLS over UDP.** DTLS supports RSA, AES, and so on.

Anyway, we should consider that in some constrained devices some of DTLS cipher suits may not be available.

It is important to notice that some cipher suites introduces some complexity and constrained devices may not have resources enough to manage it.

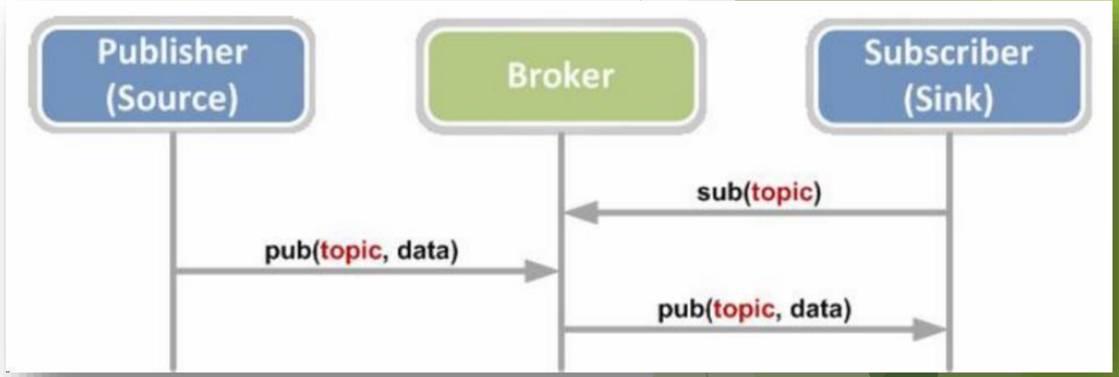


MQTT

Message Queuing Telemetry Transport

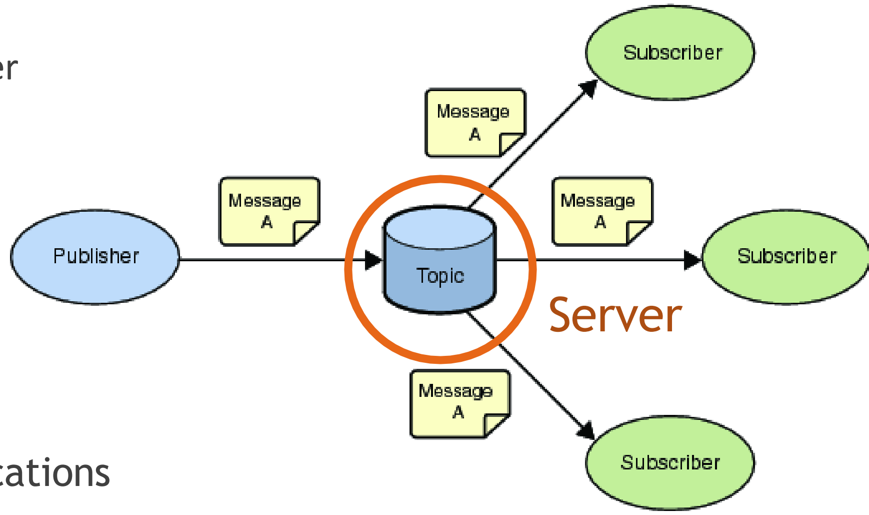
Event based IoT Middleware
- Event pattern of communication (one to many)
- Over IP (TCP)

- MQTT is described on the mqtt.org site as a machine-to-machine (M2M) / IoT connectivity protocol.
- MQTT is an Event based IoT middleware (one to many)
  - publish/subscribe messaging transport protocol
  - Over TCP/IP (or MQTT-S over UDP for LAN)
- Its protocol is lightweight
  - it can be supported by some of the smallest measuring and monitoring devices (ex. Arduino)
  - it can transmit data over far reaching networks
  - It can transmit data over sometimes intermittent networks.



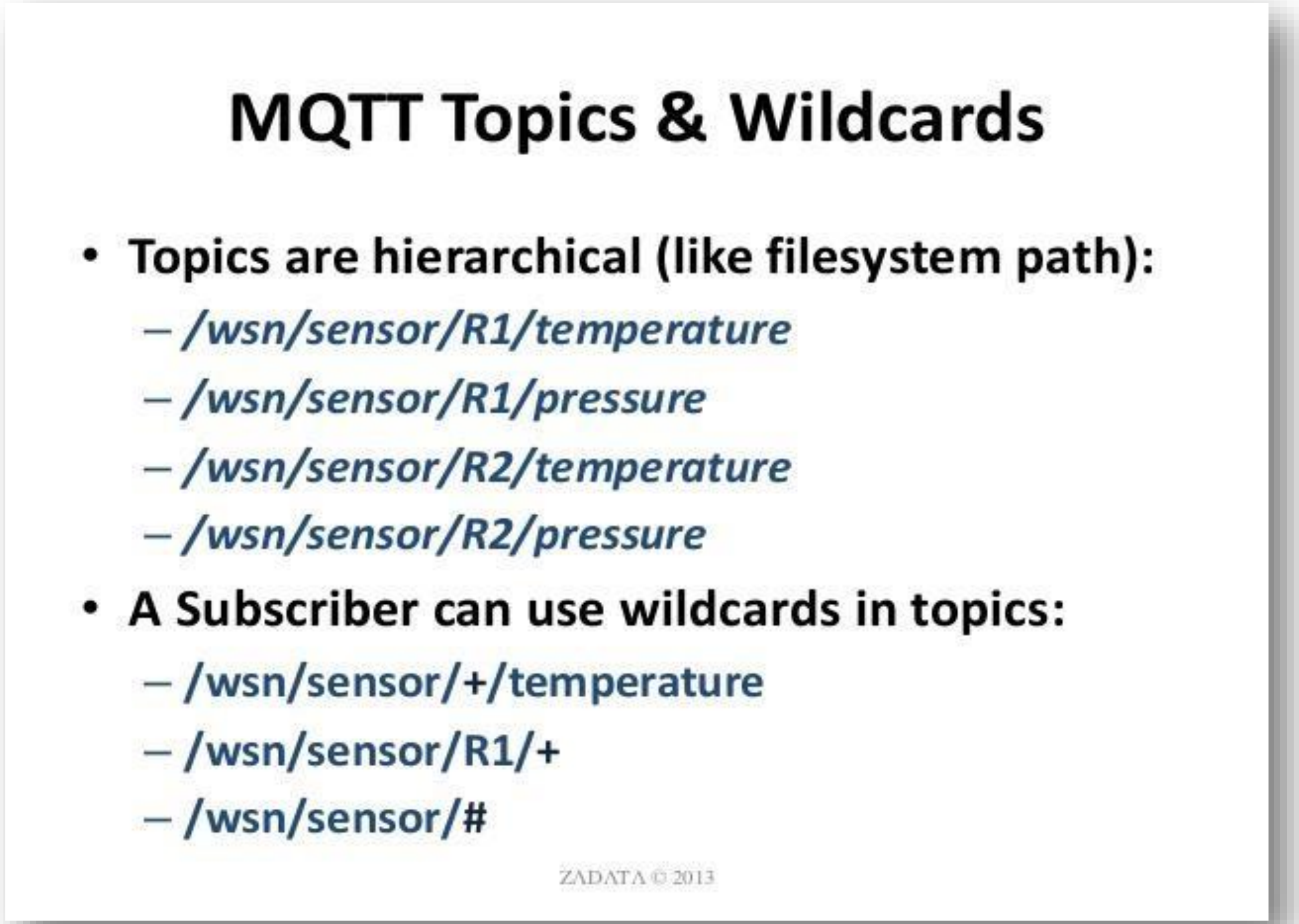
Publish / Subscribe Messaging (One to Many)

- A producer publishes a message (publication) on a topic (subject)
- A consumer subscribes (makes a subscription) for messages on a topic (subject)
- A message server (called BROKER) matches publications to subscriptions
  - If none of them match the message is discarded after modifying the topic
  - If one or more matches the message is delivered to each matching consumer after modifying the topic



- Publish / Subscribe has three important characteristics:
  1. It decouples message senders and receivers, allowing for more flexible applications
  2. It can take a single message and distribute it to many consumers
  3. This collection of consumers can change over time, and vary based on the nature of the message.

MQTT Topic and Wildcards



MQTT Topics & Wildcards

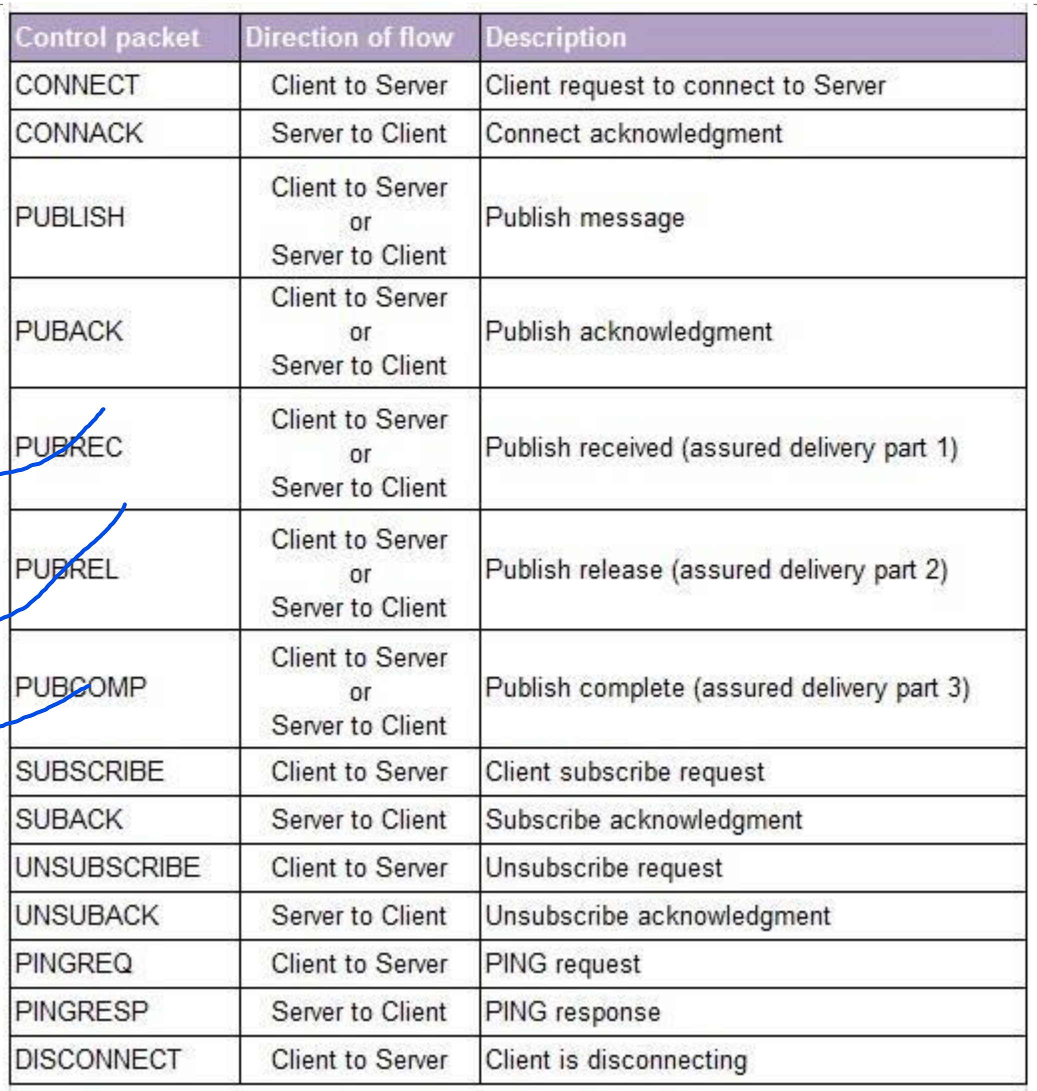
- Topics are hierarchical (like filesystem path):
  - */wsn/sensor/R1/temperature*
  - */wsn/sensor/R1/pressure*
  - */wsn/sensor/R2/temperature*
  - */wsn/sensor/R2/pressure*
- A Subscriber can use wildcards in topics:
  - /wsn/sensor/+/temperature
  - /wsn/sensor/R1/+
  - /wsn/sensor/#

MQTT Topic : Details

- A topic forms the namespace
  - Is hierarchical with each “sub topic” separated by a /
  - An example topic space
    - A house publishes information about itself on:
      - *<country>/<region>/<town>/<postcode>/<house>/energyConsumption*
      - *<country>/<region>/<town>/<postcode>/<house>/solarEnergy*
      - *<country>/<region>/<town>/<postcode>/<house>/alarmState*
      - *<country>/<region>/<town>/<postcode>/<house>/alarmState*
    - *And subscribes for control commands:*
      - *<country>/<region>/<town>/<postcode>/<house>/thermostat/setTemp*
- A subscriber can subscribe to an absolute topic or can use wildcards:
  - Single-level wildcards “+” can appear anywhere in the topic string
  - Multi-level wildcards “#” must appear at the end of the string
  - Wildcards must be next to a separator
  - Cannot be used wildcards when publishing
  - For example
    - *UK/Hants/Hursley/SO212JN/1/energyConsumption*
      - Energy consumption for 1 house in Hursley
    - *UK/Hants/Hursley/+/+/energyConsumption*
      - Energy consumption for all houses in Hursley
    - *UK/Hants/Hursley/SO212JN/#*
      - Details of energy consumption, solar and alarm for all houses in SO212JN

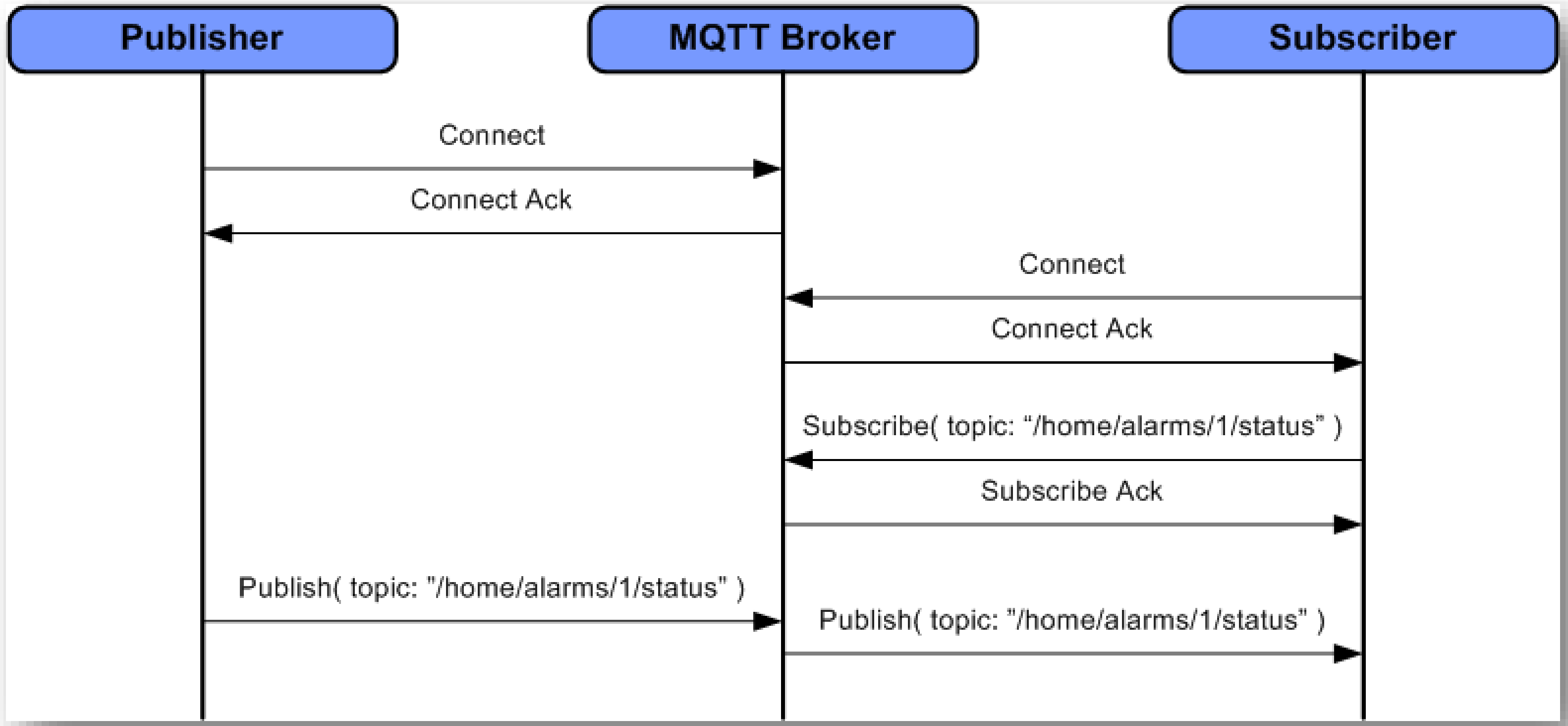
MQTT publish subscribe architecture

- The MQTT messages are delivered asynchronously (“push”) through publish subscribe architecture.
- The MQTT protocol works by exchanging a series of MQTT control packets in a defined way.
- Each control packet has a specific purpose and every bit in the packet is carefully crafted to reduce the data transmitted over the network.
- A MQTT topology has a MQTT server and a MQTT client.
- MQTT client and server communicate through different control packets. Table below briefly describes each of these control packets.



Control packet	Direction of flow	Description
CONNECT	Client to Server	Client request to connect to Server
CONNACK	Server to Client	Connect acknowledgment
PUBLISH	Client to Server or Server to Client	Publish message
PUBACK	Client to Server or Server to Client	Publish acknowledgment
PUBREC	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRIBE	Client to Server	Client subscribe request
SUBACK	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	Client to Server	Unsubscribe request
UNSUBACK	Server to Client	Unsubscribe acknowledgment
PINGREQ	Client to Server	PING request
PINGRESP	Server to Client	PING response
DISCONNECT	Client to Server	Client is disconnecting

Sample of protocol use


Publisher
MQTT Broker
Subscriber
Connect
Connect Ack
Connect
Connect Ack
Subscribe(topic: "/home/alarms/1/status")
Subscribe Ack
Publish(topic: "/home/alarms/1/status")
Publish(topic: "/home/alarms/1/status")

Ideal for constrained networks (low bandwidth, high latency, data limits, and fragile connections)

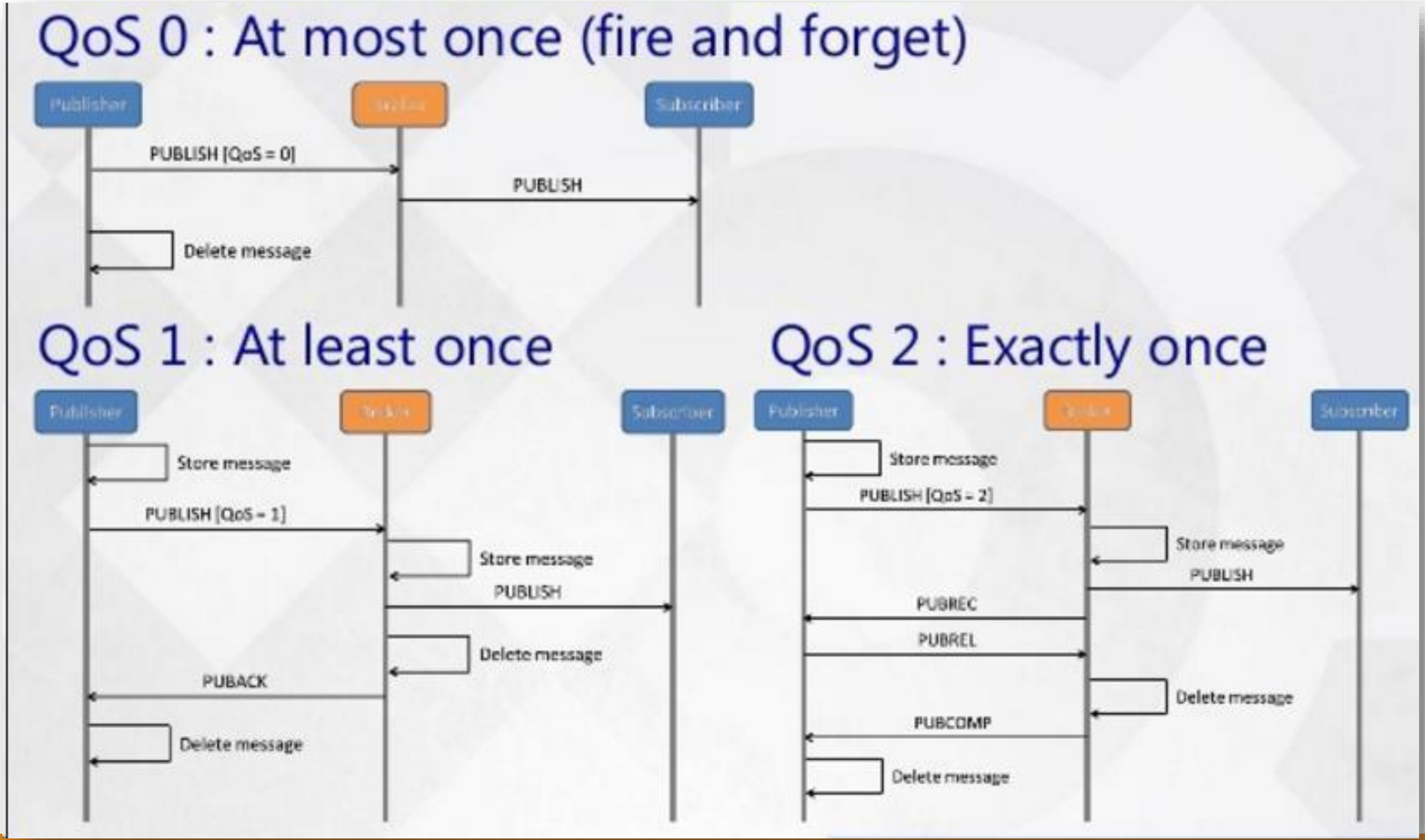
- MQTT control packet headers are kept as small as possible.
- Each MQTT control packet consist of three parts, a fixed header, variable header and payload.
- Each MQTT control packet has a 2 byte Fixed header. Not all the control packet have the variable headers and payload.
- A variable header contains the packet identifier if used by the control packet.
- A payload up to 256 MB could be attached in the packets.
- Having a small header overhead makes this protocol appropriate for IoT by lowering the amount of data transmitted over constrained networks.

Quality of Service (QoS) for MQTT

- Quality of service (QoS) levels determine how each MQTT message is delivered and must be specified for every message sent through MQTT.
- Three QoS for message delivery could be achieved using MQTT:
  - QoS 0 (At most once) - where messages are delivered according to the best efforts of the operating environment. Message loss can occur.
  - QoS 1 (At least once) - where messages are assured to arrive but duplicates can occur.
  - QoS 2 (Exactly once) - where message are assured to arrive exactly once.
- There is a simple rule when considering performance impact of QoS :

“The higher the QoS, the lower the performance".

Quality of Service (QoS) for MQTT



MQTT Clients and APIs

- You can develop an MQTT client application by programming directly to the MQTT protocol specification …… however it is more convenient to use a prebuilt client
- Open Source clients available in Eclipse Paho project
  - C, C++, Java, JavaScript, Lua, Python and Go
- Clients for other languages are available, see mqtt.org/software
  - E.g. Delphi, Erlang, .Net, Objective-C, PERL, PHP, Ruby
  - Not all of the client libraries listed on mqtt.org are current. Some are at an early or experimental stage of development, whilst others are stable and mature.
- Even in shell script like we are seeing in the practical course …

Thanks to

- An Introduction to REST API -Aniruddh Bhilvare
- CoAP-Constrained Application Protocol - SHAIK SABAHAT NOWREEN

Thank you