

Communication Technologies for IoT – Part 2

Application Layer Protocols

Dr. Bibhas Ghoshal

IIIT Allahabad



IoT Communication Protocol requirement

IoT End Network Requirements	Networking Style Impact
Self Healing / Scalable	Mesh capable
Secure	Scalable to no, low, medium, high security without burdening clients
End-node addressability	Device specific addressing scalable to thousands of nodes
Device Requirements	Messaging Protocol Impact
Low power / battery operated	Lightweight connection, preamble, packet
Limited Memory	Small client footprint, persistent state in case of overflow
Low cost	Tiles to memory footprint

Ref : The Internet of Things, Enabling Technologies, Platforms and Use Cases: Pethuru Raj and Anupama C. Rajan, CRC Press

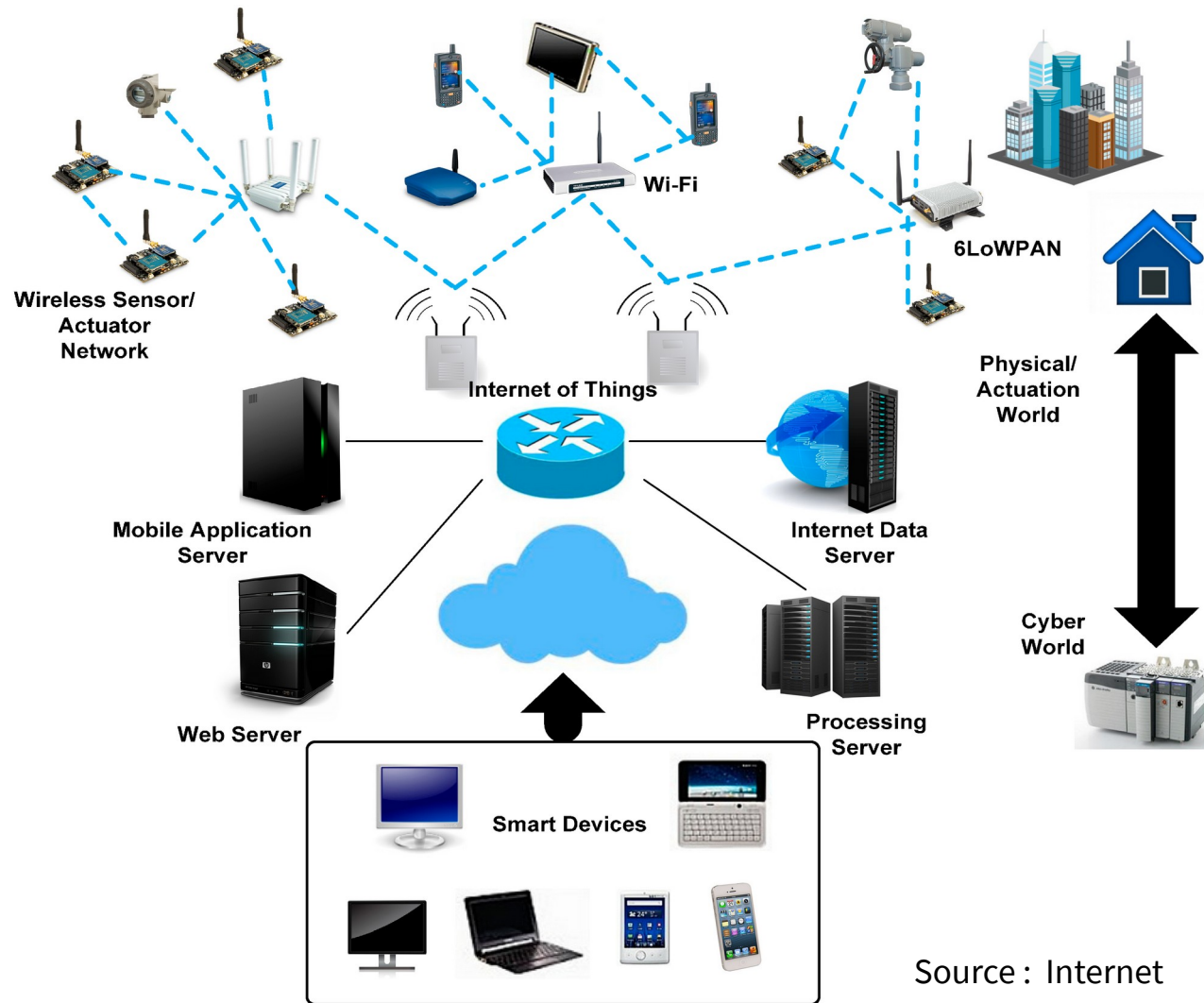
Integration Scenarios in IoT

- Sensor and Actuator Networks
- Device to Device (**D2D**) Integration
- Cloud to Cloud (**C2C**) Integration
- Device and Sensor to Cloud (**D2C**) Integration

Devices talk to each other – **devices northbound and southbound**

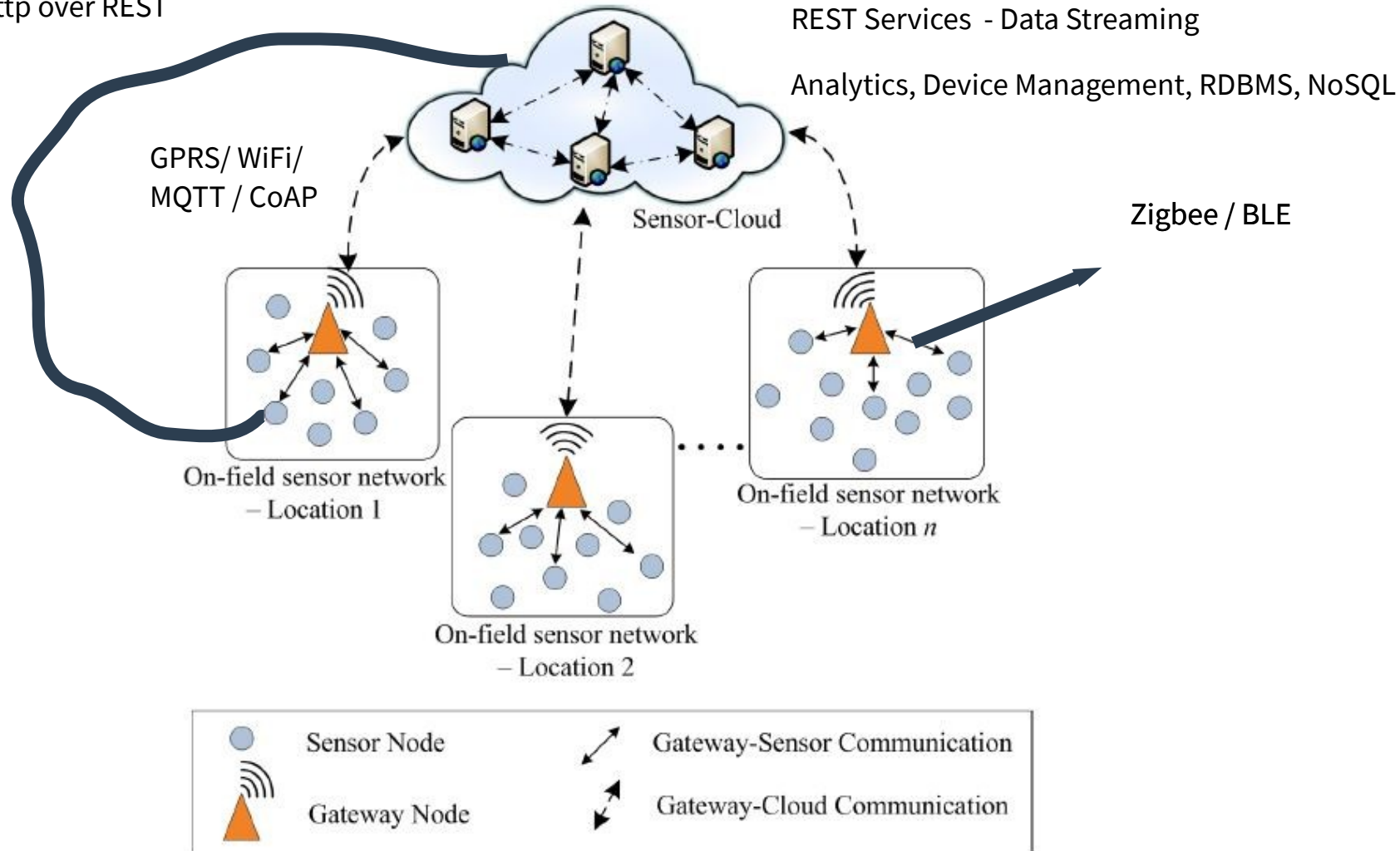
Gateways talk to the **cloud northbound and devices southbound**

Sensor-Actuator Networks



Sensor-Cloud Integration Networks

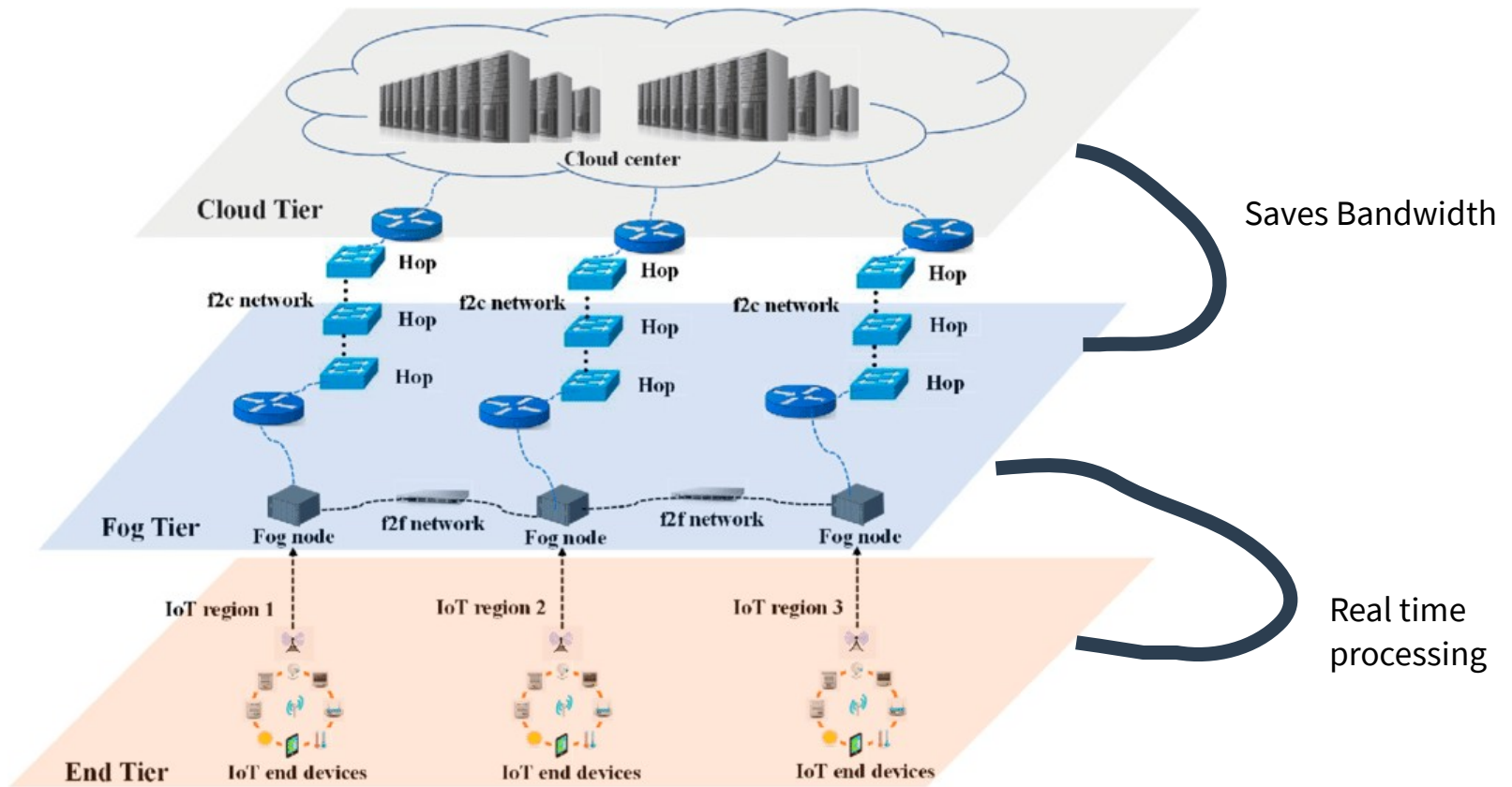
Http over REST



Source : Internet



Fog-Cloud System Architecture



Source : Internet



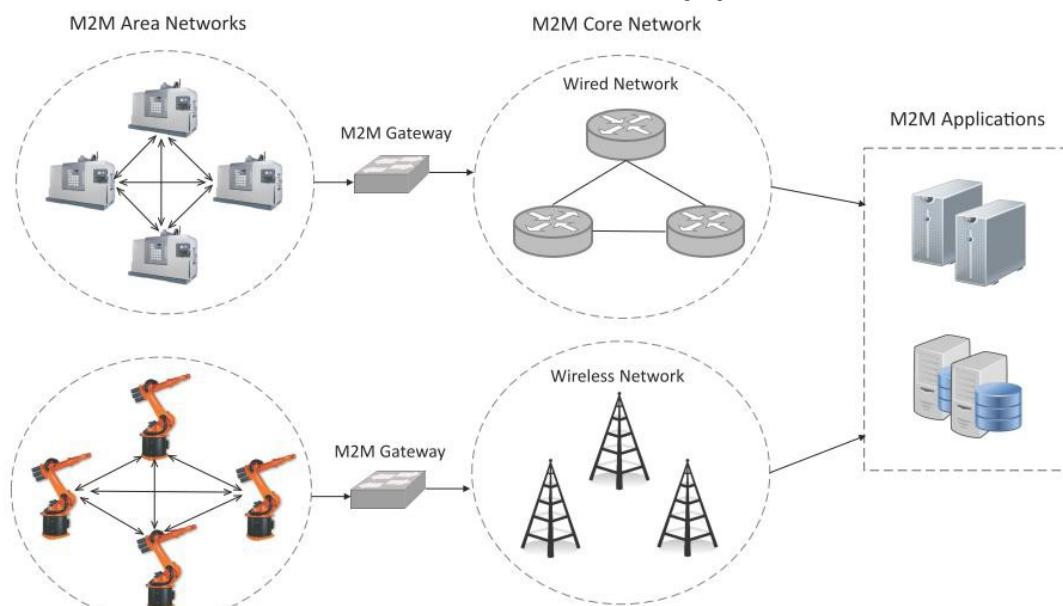
Machine to Machine (M2M) Communication

M2M area network comprises of machines (or M2M nodes) which have embedded hardware modules for sensing, actuation and communication.

Communication technologies used in M2M area network are Zigbee, Bluetooth, 6LoWPAN, IEEE 802.15.4. They use proprietary or non-IP based protocols

Communication network provides connectivity to M2M area network

Communication network uses IP based connectivity protocols



Internet of Things, Bahga and Madisetty. Book website : <http://www.internet-of-things-book.com>

Machine to Machine (M2M) Communication

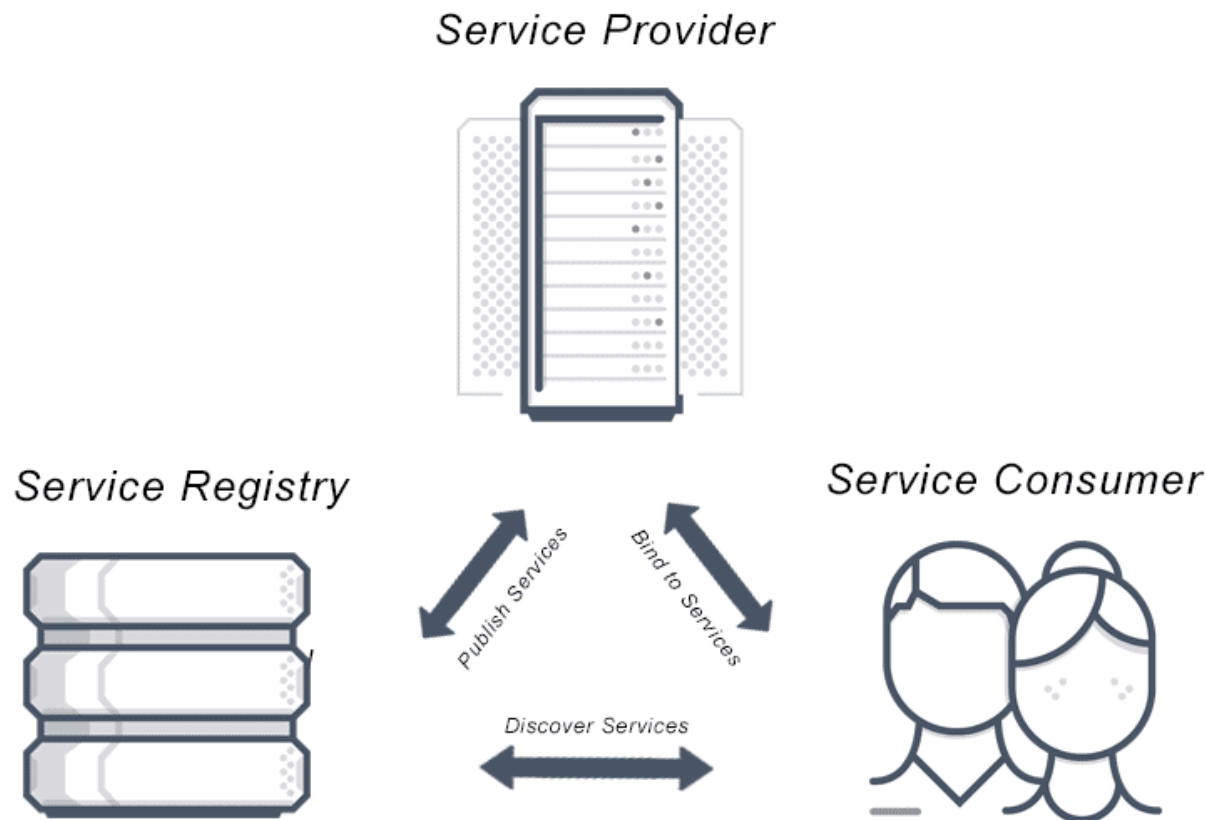
Applications :

1. Security through connected surveillance cameras, alarm systems and access control
2. Object racking and tracing – fleet management, supply chain management, asset tracking, road tolling , traffic optimization
3. Automated payment through integrated Point of Sales (POS), vending machines, gaming consoles
4. Smart health care through continuous monitoring vital signs, telemedicine
5. Remote maintenance of machines vending machines, transport vehicles
6. Advanced metering for power, gas, water, heating etc.
7. Industry automation through production chain monitoring

Service Oriented Architecture

SOA – Splitting apps into set of services each having an interface

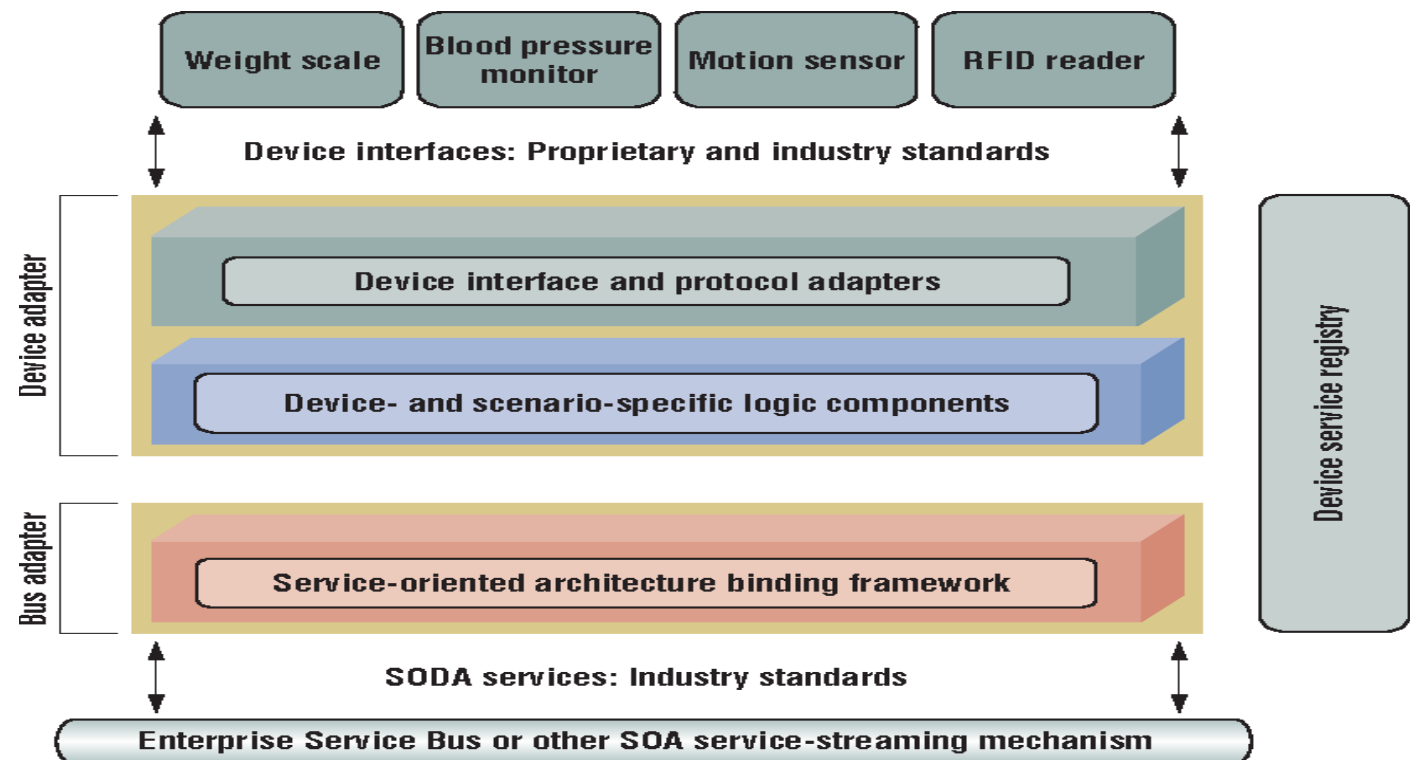
The Service Oriented Architecture Triangle



Service Oriented Device Architecture

SODA – Devices talk to each other through programs using their interfaces

- Device integration allows sharing of services among devices
- Data and event messages are transmitted
- Devices are linked to apps and data sources hosted in web and cloud
- SODA standard - REST



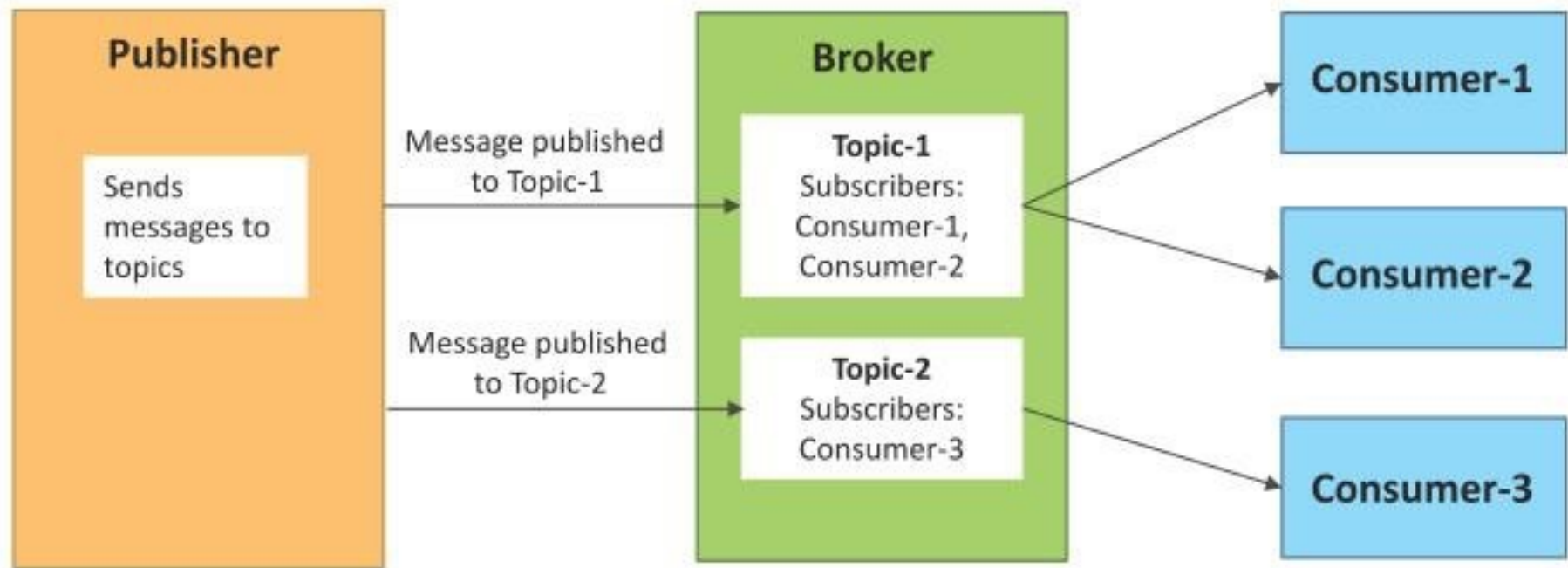
Device Profile for Web Service (DPWS)

Services – standard for M2M communication

- Representational State Transfer (REST) compliant services
- Arbitrary services – arbitrary set of operations, use SOAP messages
- DPWS defines a set of implementation constraints that provide secure and effective mechanism for describing, discovering, messaging, eventing of services for resource constrained devices
- Two type of services :
 1. Hosting service – request a device to participate in discovery and describe other services
 2. Hosted services - presents functionalities of each device
- DPWSim (<http://soureforge.net/projects/dpwsim>) - simulation toolkit to support building device applications that in turn use distinctive services being offered by low as well as high end devices.
- DPWSim allows app developers to prototype, develop and test their IoT apps without the presence of physical devices
- DPWSim can be used for collaboration between manufacturers, developers and designers during the new product development.

Messaging Methods

· Publish Subscribe Model :



Representational State Transfer (REST)

- **Architectural style for providing standards between computer systems on the web making them easier to communicate**
- **Introduced and Collated by Dr. Roy Fielding's thesis :**
Architectural Styles and Design of Network Based Software Architectures
- **Resource Based**
 - **Things Vs. Actions**
 - **Nouns vs. Verbs**
 - **Identified by URIs**
 - **Separate from their representations**
- **REST server provides access to resource and REST clients accesses and modifies resources**
- **Each resource is identified by URIs and uses various representations to represent resources**
- **HTTP methods are used in REST architectures**

Representations

- **How resources get modified**
- **Part of resource state**
 - Transferred between client and server
- **Typically JSON or XML**
- **Example :**
 - Resource : Person
 - Service : Contact Information (*GET* - HTTP Verb)
 - Representation :
 - Name , address, phone number
 - JSON, XML

Stateless Client-Server Model

- **A disconnected system which has separation of concerns**
- **Server contains no client state**
- **Each request contains enough context to process the message**
 - Self descriptive messages
- **Any session state is held on the client**
- **Server responses (responses) are cacheable**
 - Implicitly
 - Explicitly
 - Negotiated

Uniform Interface

- **Defines interface between client and server**
- **Simplifies and decouples the architecture**
- **Fundamentals to RESTful design**
 - **HTTP verbs**
 - *GET* - retrieve a specific resource (by id) or collection of resources
 - *POST* - create a new resource
 - *PUT* - update specific resource (id)
 - *DELETE* – remove a specific resource by id
 - **URIs** (resource name)
 - **HTTP Response** (status , body)

Layered System

- **Client can't assume direct connection to server**
- **Software or hardware intermediaries between client and server**
- **Improves scalability**

Code on Demand

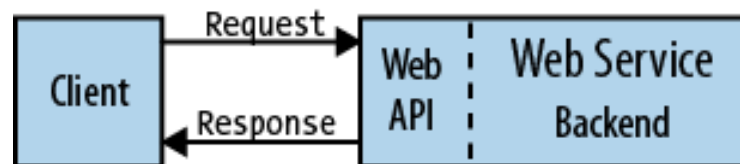
- **Server can temporarily extend client**
- **Transfer logic to client**
- **Client executes logic**
- **For example : Java Applets, Java Scripts**
- **The only optional constraint**

Summary

- **Violating any constraint other than Code on Demand means service is not strictly RESTful**
 - Example : three legged OAUTH2
- **Compliance with REST constraints allows :**
 - Scalability
 - Simplicity
 - Modifiability
 - Visibility
 - Portability
 - Reliability

REST APIs

- Web services are purpose-built web servers that support the needs of a site or any other application.
- Client programs use application programming interfaces (APIs) to communicate with web services.
- API exposes a set of data and functions to facilitate interactions between computer programs and allow them to exchange information
- Web API is the face of a web service, directly listening and responding to client requests.



- A Web API conforming to the REST architectural style is a REST API.
- A REST API makes a web service “RESTful.”
- A REST API consists of an assembly of interlinked resources. This set of resources is known as the REST API’s resource model.

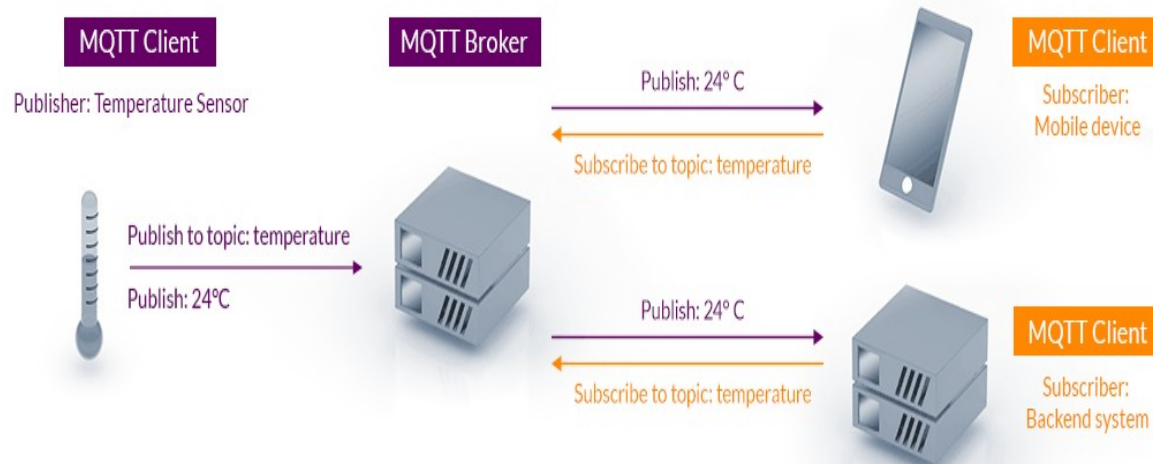
Device Integration Protocols

- **MQTT** : Protocol for collecting device data and communicating it to servers (**D2S**)
- **CoAP** : An optimized protocol
- **XMPP** : protocol for connecting devices to people (people connected to servers)
- **AMQP** : A queuing system designed to connect servers to each other (**S2S**)

MQTT : History and Requirements

- Invented in **1999** by **Andy Stanford-Clark (IBM)** and **Arlen Nipper (Arcom, now Cirrus Link)** : <http://mqtt.org> (Now open source)
- **Protocol for minimal battery loss and minimal bandwidth to connect with oil pipelines via satellite**
- **Core Features of MQTT:**
 - Simple implementation – arbitrary messages upto 256MB
 - Quality of Service data delivery – in order deliver per publisher
 - Lightweight and bandwidth efficient – little client state, TCP/Web sockets
 - Data agnostic
 - Continuous session awareness

MQTT Architecture



- **Space Decoupling** - Pub. And Sub. do not know each other (ip/port)
- **Time Decoupling** – do not have to be actively connected all time
- **Synchronization Decoupling** – sending/ receiving at own speed
- **Scalability**
- **Message Filtering**

MQTT Quality of Service (QoS) Levels

QoS 0 : At most once “ Fire and Forget”, no confirmation, Guaranteed delivery

QoS 1 : At least once , with confirmation required (msgs may delivered more than once)

QoS 2 : Exactly once , 2-phase commit

MQTT supports *Persistent Messages*

Ideal for internet connectivity

Automatic keep alive messages

**QoS 1 and 2 messages are queued for clients which may be offline
But not timed out**

Disconnect, Last Will & Testament and Retained Messages

Clients which disconnect intentionally use Disconnect message

MQTT broker will automatically publish Last Will and Testament message on behalf of clients with unintentionally terminated messages

MQTT supports Retained messages which are automatically delivered when clients subscribes to a topic

MQTT Vs. HTTP

	MQTT	HTTP
Purpose	Messaging	Documents
Protocol Efficiency	High	Average
Power Efficiency	Yes	No
Client Languages	Many	Many

MQTT Vs. Message Queues

- A message queue stores message until they are consumed
- A message is only consumed by one client
- Queues are named and must be created explicitly

Client, Broker and Connection Establishment

• Client :

- any device (from a micro controller up to a full-fledged server) that runs an MQTT library and connects to an MQTT broker over a network. Any device that speaks MQTT over a TCP/IP stack can be called an MQTT client.
- MQTT client libraries are available for a huge variety of programming languages. For example, Android, Arduino, C, C++, C#, Go, iOS, Java, JavaScript, and .NET.

• Broker :

- responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients.
- holds the session data of all clients that have persistent sessions, including subscriptions and missed messages
- authentication and authorization of clients.

Message Filtering by Broker

- **Subject based Filtering**

filtering is based on the subject or topic that is part of each message. The receiving client subscribes to the broker for topics of interest

- topics are strings with a hierarchical structure that allow filtering based on a limited number of expressions

- **Content based Filtering**

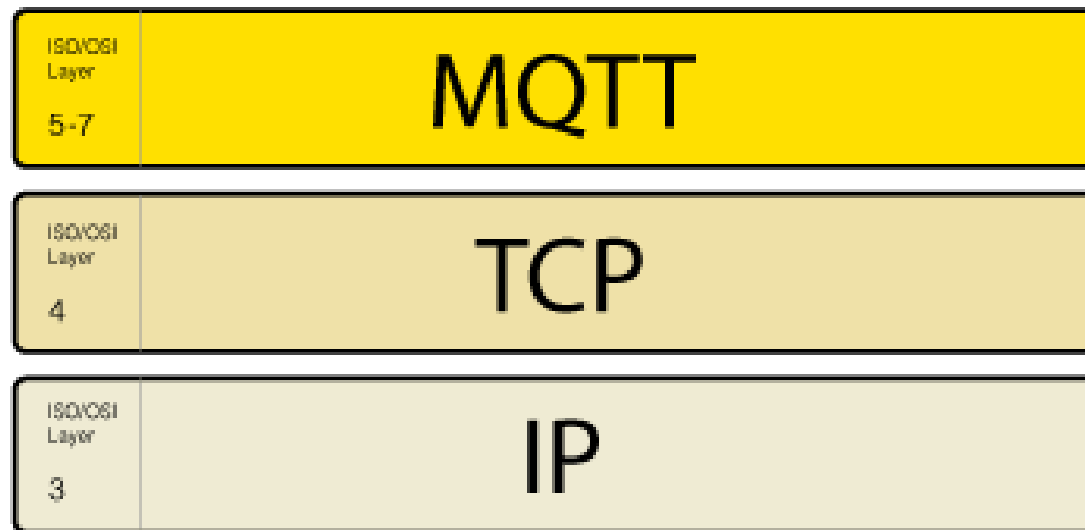
- the broker filters the message based on a specific content filter-language.
- The receiving clients subscribe to filter queries of messages for which they are interested.
- content of the message must be known beforehand and cannot be encrypted or easily changed.

- **Type based Filtering**

- filtering based on the type/class of a message (event) in case Object Oriented languages

MQTT Connection

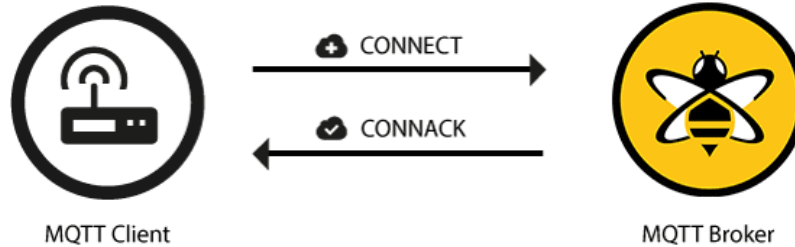
MQTT protocol is based on TCP/IP. Both the client and the broker need to have a TCP/IP stack.



MQTT connection is always between one client and the broker. Clients never connect to each other directly

MQTT Connection

Initiation :



- client sends a **CONNECT** message to the broker.
- broker responds with a **CONNACK** message and a status code.
- Once the connection is established, the broker keeps it open until the client sends a disconnect command or the connection breaks

MQTT-Packet:	
CONNECT	
contains:	Example
clientId	"client-1"
cleanSession	true
username (optional)	"hans"
password (optional)	"letmein"
lastWillTopic (optional)	"/hans/will"
lastWillQos (optional)	2
lastWillMessage (optional)	"unexpected exit"
lastWillRetain (optional)	false
keepAlive	60

MQTT Packet Components

ClientId: identifies each MQTT client that connects to an MQTT broker. The broker uses the ClientId to identify the client and the current state of the client. Therefore, this Id should be unique per client and broker.

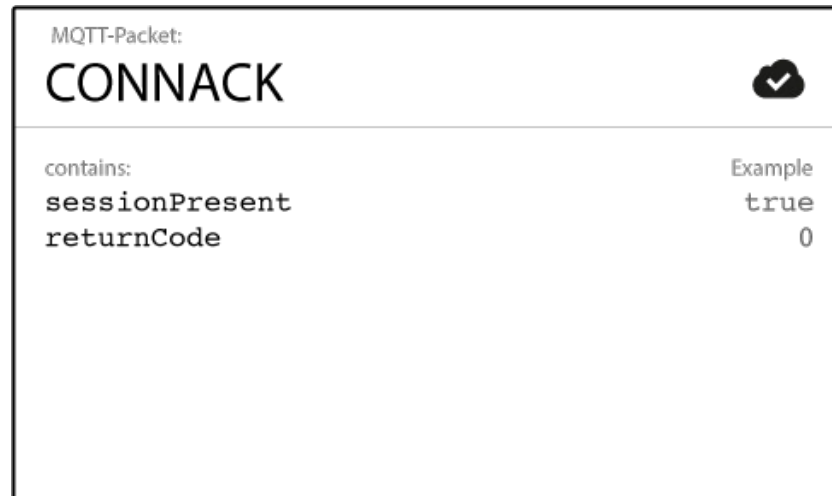
Clean Session : The clean session flag tells the broker whether the client wants to establish a persistent session or not. In a persistent session (**CleanSession = false**), the broker stores all subscriptions for the client and all missed messages for the client

Username/Password : MQTT can send a user name and password for client authentication and authorization.

Will Message : notifies other clients when a client disconnects ungracefully.

KeepAlive : time interval in seconds that the client specifies and communicates to the broker when the connection established. It defines the longest period of time that the broker and client can endure without sending a message.

Broker Response



Session Present flag : The session present flag tells the client whether the broker already has a persistent session available from previous interactions with the client.

Connect return code : tells the client whether the connection attempt was successful or not. (0 = connection accepted)

Publish

MQTT-Packet:	
PUBLISH 	
contains:	Example
packetId (always 0 for qos 0)	4314
topicName	"topic/1"
qos	1
retainFlag	false
payload	"temperature:32.5"
dupFlag	false

MQTT client can publish messages as soon as it connects to a broker.

MQTT utilizes topic-based filtering of the messages on the broker

Each message must contain a topic that the broker can use to forward the message to interested clients. Typically, each message has a payload which contains the data to transmit in byte format.

use case of the client determines how the payload is structured.

The sending client (publisher) decides whether it wants to send binary data, text data, or even full-fledged XML or JSON.

Publish Message Components

Topic Name : simple string that is hierarchically structured with forward slashes as delimiters. For example, “**myhome/livingroom/temperature**”

QoS : indicates the Quality of Service Level (QoS) of the message. There are three levels: 0, 1, and 2. The service level determines what kind of guarantee a message has for reaching the intended recipient (client or broker)

Retain Flag : defines whether the message is saved by the broker as the last known good value for a specified topic. When a new client subscribes to a topic, they receive the last message that is retained on that topic.

Payload : actual content of the message. MQTT is data-agnostic. It is possible to send images, text in any encoding, encrypted data, and virtually every data in binary.

Packet Identifier : uniquely identifies a message as it flows between the client and broker. The packet identifier is only relevant for QoS levels greater than zero. The client library and/or the broker is responsible for setting this internal MQTT identifier.

DUP flag : indicates that the message is a duplicate and was resent because the intended recipient (client or broker) did not acknowledge the original message. This is only relevant for QoS greater than 0.

Subscribe

Packet Identifier : uniquely identifies a message as it flows between the client and broker. The client library and/or the broker is responsible for setting this internal MQTT identifier.

List of Subscriptions : A SUBSCRIBE message can contain multiple subscriptions for a client. Each subscription is made up of a topic and a QoS level. The topic in the subscribe message can contain wildcards that make it possible to subscribe to a topic pattern rather than a specific topic. If there are overlapping subscriptions for one client, the broker delivers the message that has the highest QoS level for that topic.

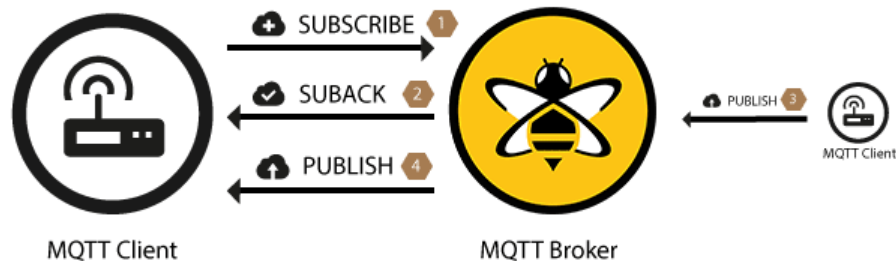
MQTT-Packet:	
SUBSCRIBE	
contains:	Example
packetId	4312
qos1 } (list of topic + qos)	1
topic1 }	"topic/1"
qos2 }	0
topic2 }	"topic/2"
...	...

Subscribe Acknowledgement


MQTT-Packet:	
SUBACK	
contains:	
packetId	Example
returnCode 1 (one returnCode for each topic from SUBSCRIBE, in the same order)	4313
returnCode 2	2
returnCode 3	0
...	...

Packet Identifier : unique identifier used to identify a message, same as in the SUBSCRIBE msg

Return Code : The broker sends one return code for each topic/QoS-pair that it receives in the SUBSCRIBE message.




Unsubscribe

MQTT-Packet:
UNSUBSCRIBE 

contains:

<code>packetId</code>	Example
<code>topic1</code> } (list of topics)	4315
<code>topic2</code>	"topic/1"
...	"topic/2"
	...

MQTT-Packet:
UNSUBACK 

contains:

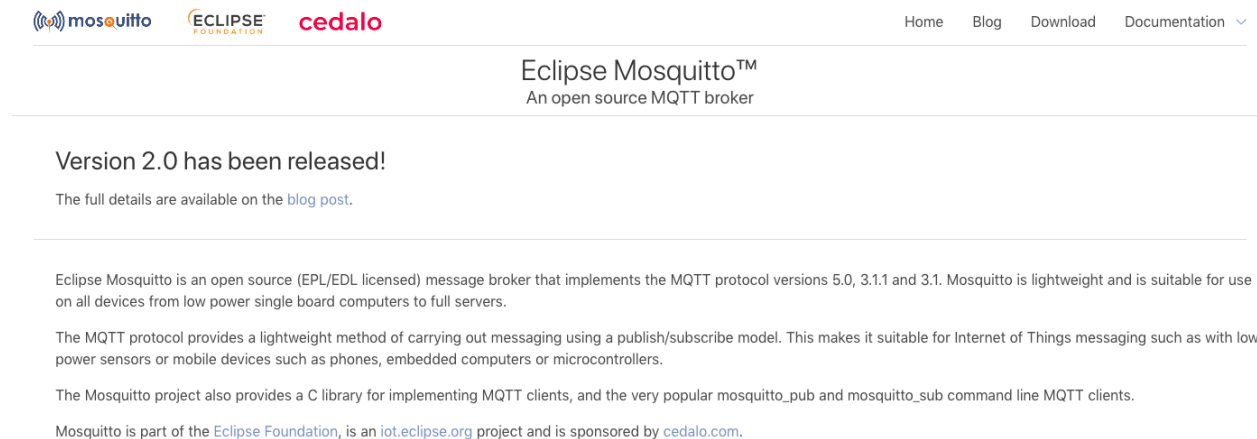
<code>packetId</code>	Example
	4316

Mosquitto MQTT Broker

Mosquitto : lightweight open source message broker

Implements MQTT versions 3.1.0, 3.1.1 and version 5.0

Main website : <https://mosquitto.org/>



The screenshot shows the Eclipse Mosquitto website homepage. At the top, there are logos for Mosquitto, Eclipse Foundation, and cedalo. Navigation links for Home, Blog, Download, and Documentation are visible. The main heading reads "Eclipse Mosquitto™ An open source MQTT broker". A prominent announcement states "Version 2.0 has been released!" with a link to a blog post for full details. Below this, a descriptive paragraph explains that Mosquitto is an open source (EPL/EDL licensed) message broker implementing MQTT versions 5.0, 3.1.1, and 3.1, suitable for various devices. It also mentions the MQTT publish/subscribe model and the availability of C libraries and command-line clients. The footer notes that Mosquitto is part of the Eclipse Foundation and is sponsored by cedalo.com.

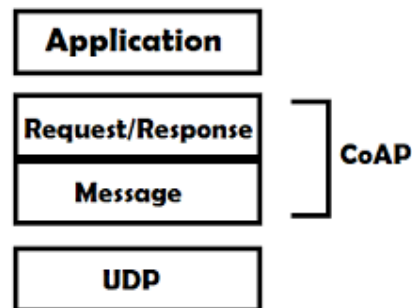
A nice tutorial on Mosquitto is available at :

<https://www.switchdoc.com/2018/02/tutorial-installing-and-testing-mosquitto-mqtt-on-raspberry-pi/>

Constrained Application Protocol (CoAP)

A specialized web transfer protocol for use with constrained nodes and constrained networks in Internet of Things

- CoRE , IETF group
- Uses both request/response and publish/subscribe model
- Proposed Standard : RFC 7252
- Lightweight fast HTTP
- Designed for manipulation of simple resources on constrained node networks
- CoAP is designed to use low power sensors to use RESTful services while meeting power constraints.



CoAP Vs. HTTP

- CoAP is designed for M2M applications over constrained environments

- CoAP offers features that HTTP lacks :

Built-in resource discovery

IP multicast support

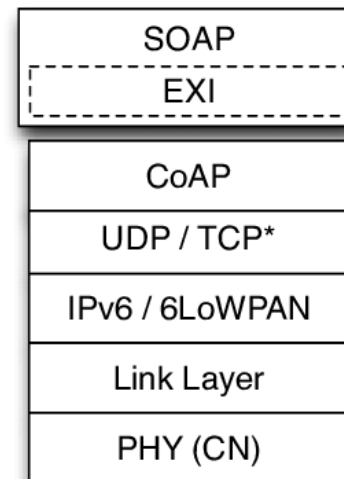
Native push pull model

Asynchronous message exchange

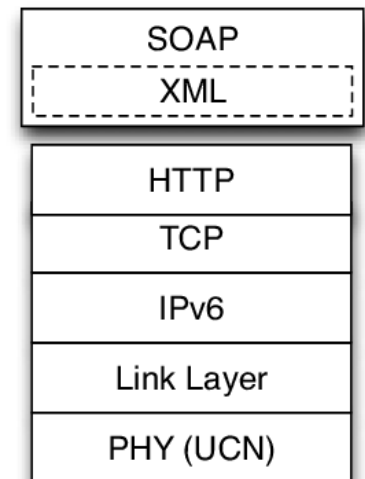
- CoAP Implementations :

Libcoap1 (Open source C implementation)

Sensinode's Nanoservice 2

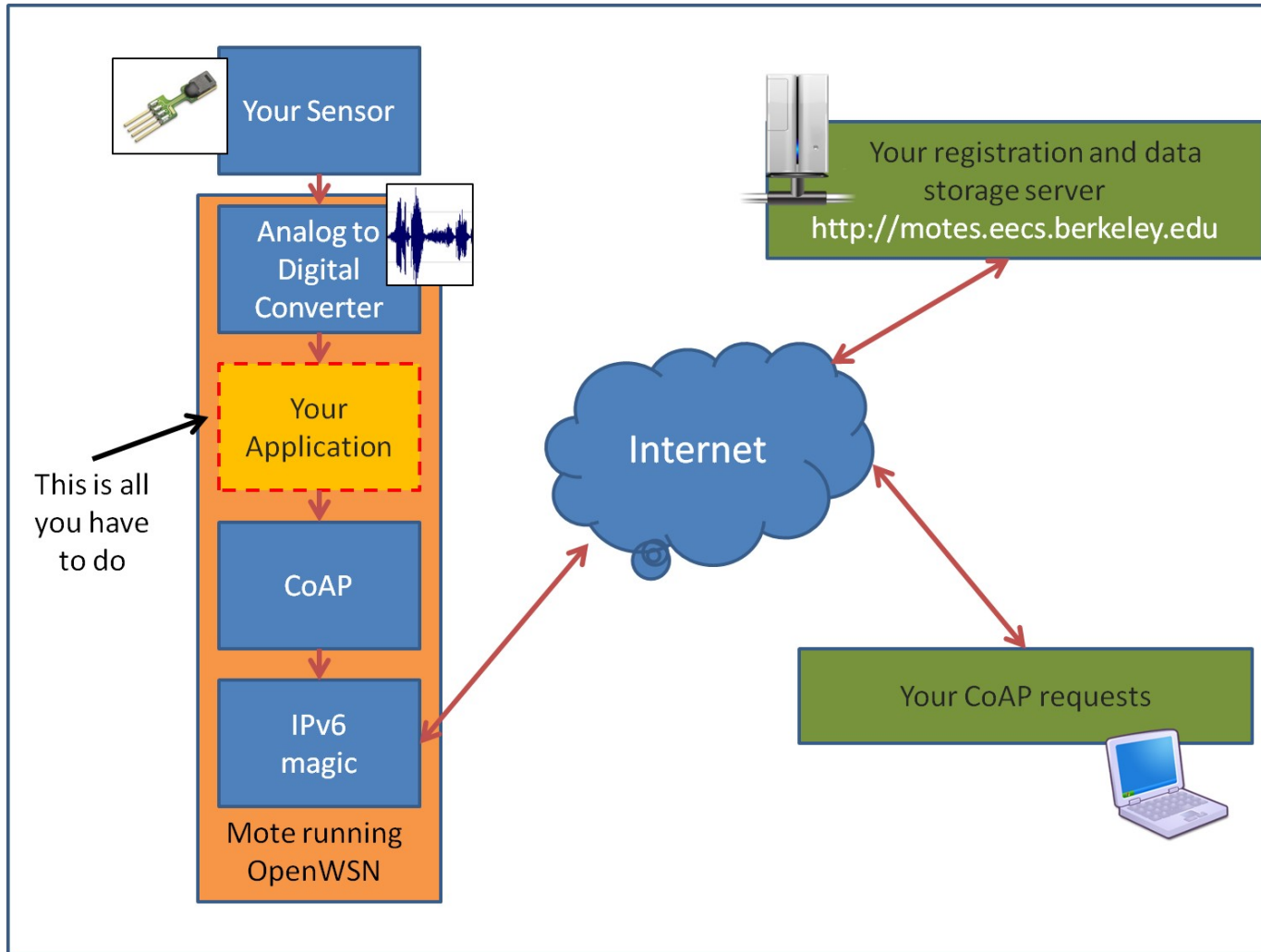


Constrained Protocol Stack



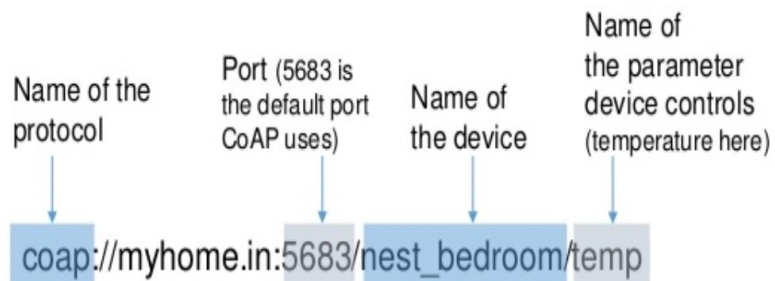
Internet Protocol Stack

CoAP Architecture

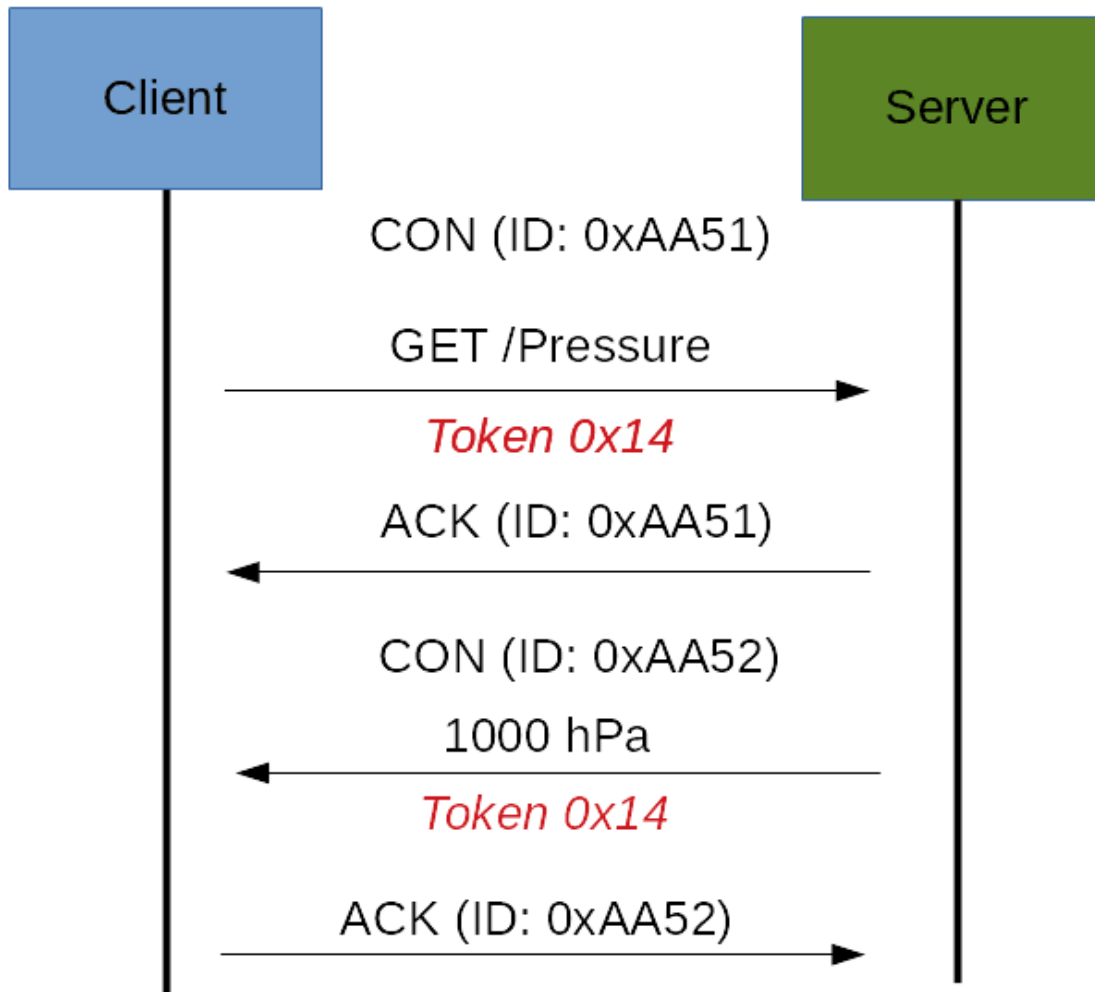


CoAP Messages

CoAP – Request Response



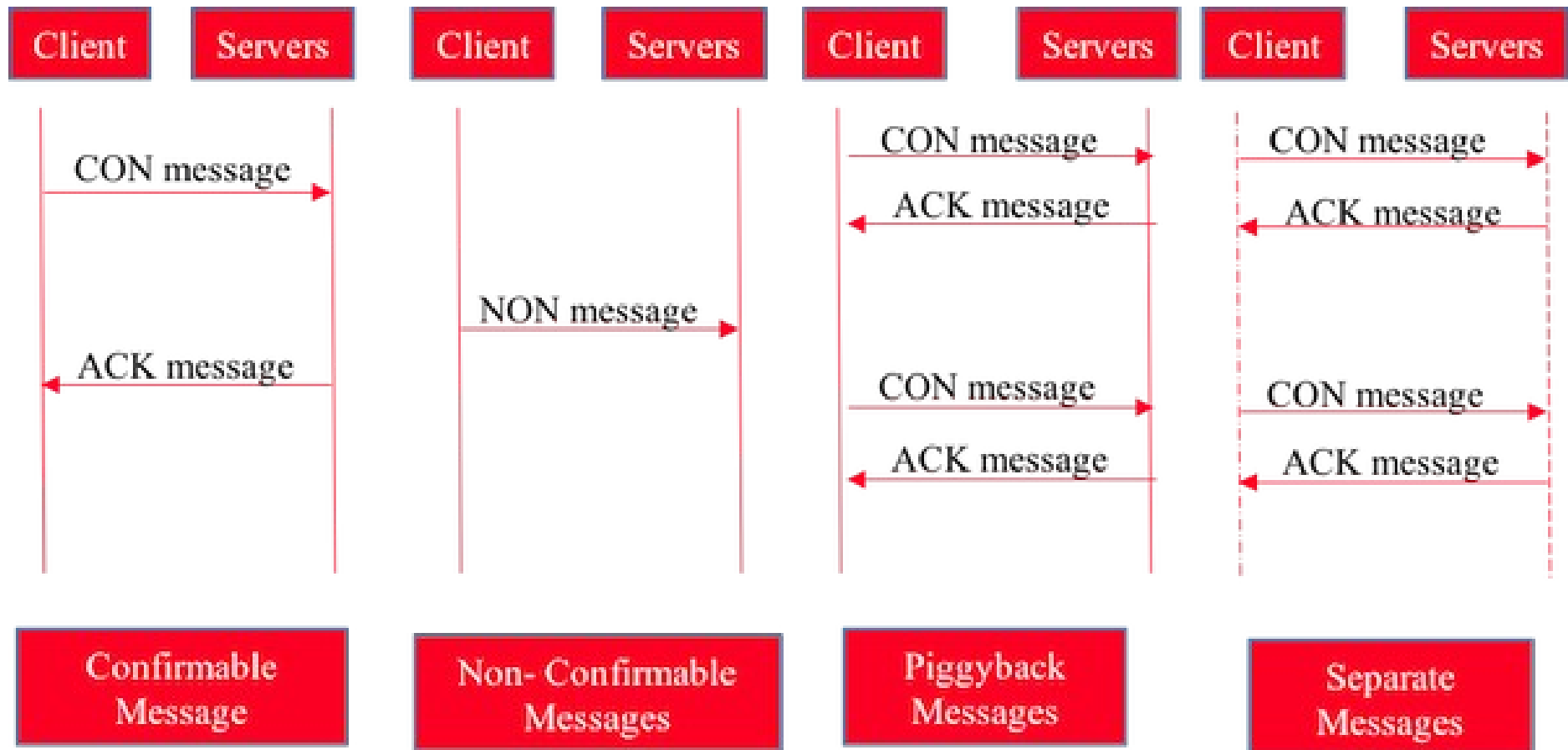
CoAP Message Transfer



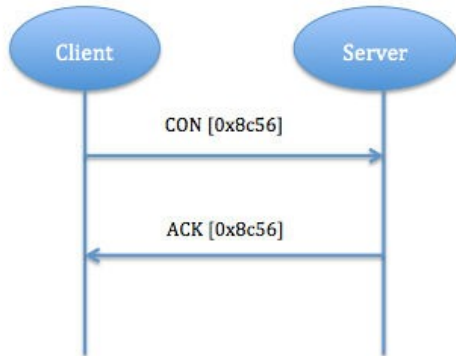
CoAP Messages

- **CoAP Messaging Modes :**
 - Confirmable
 - Non-confirmable
 - Piggyback
 - Separate
- **Confirmable and non-confirmable modes represent reliable and unreliable transmissions**
- **Other modes are used for request/response**
- **Piggyback is used for client/server direct communication where server sends its response directly after receiving message i.e. within acknowledgement message**
- **Separate mode is used when server response comes in a message other than acknowledgement**
- **CoAP uses GET, POST, PUT and DELETE messages**

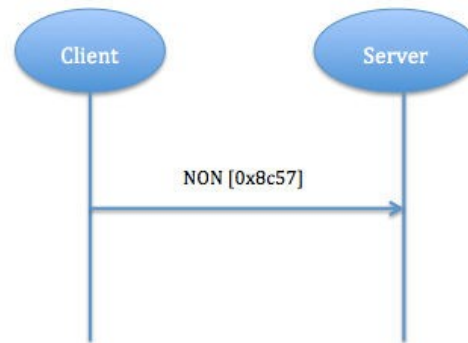
CoAP Message Modes



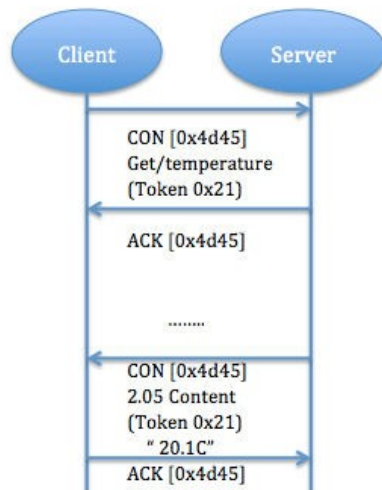
CoAP Message Modes



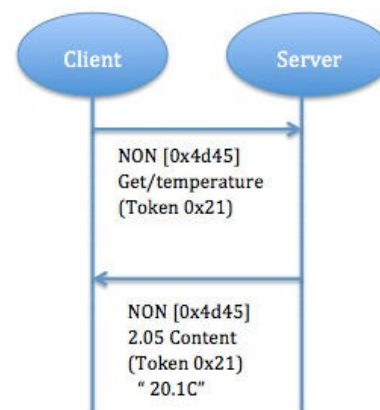
(a) Reliable Transmission



(b) Unreliable Transmission



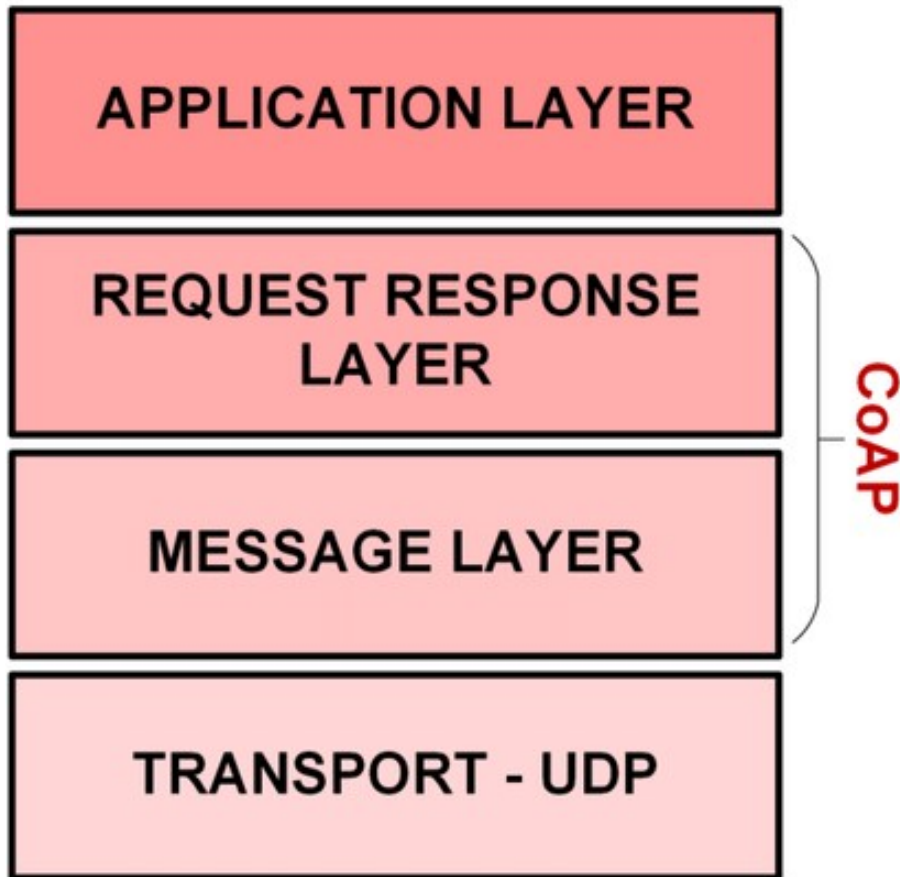
(c) Confirmable Request and Separate Confirmable Response



(d) Non-Confirmable Request and Non-Confirmable Response

CoAP Message Format

Protocol Stack:



CoAP Frame:

