SIOTLAB | SANTA CLARA UNIVERSITY
Internet of Things Research Lab

INTERNET OF THINGS | COEN 243
DEPARTMENT OF COMPUTER ENGINEERING, SANTA CLARA UNIVERSITY, CALIFORNIA
WEI XIAO, INSTRUCTOR (WXIAO@SCU.EDU)
SHIVAKUMAR MATHAPATHI, GUEST INSTRUCTOR (SMATHAPATHI@SCU.EDU)

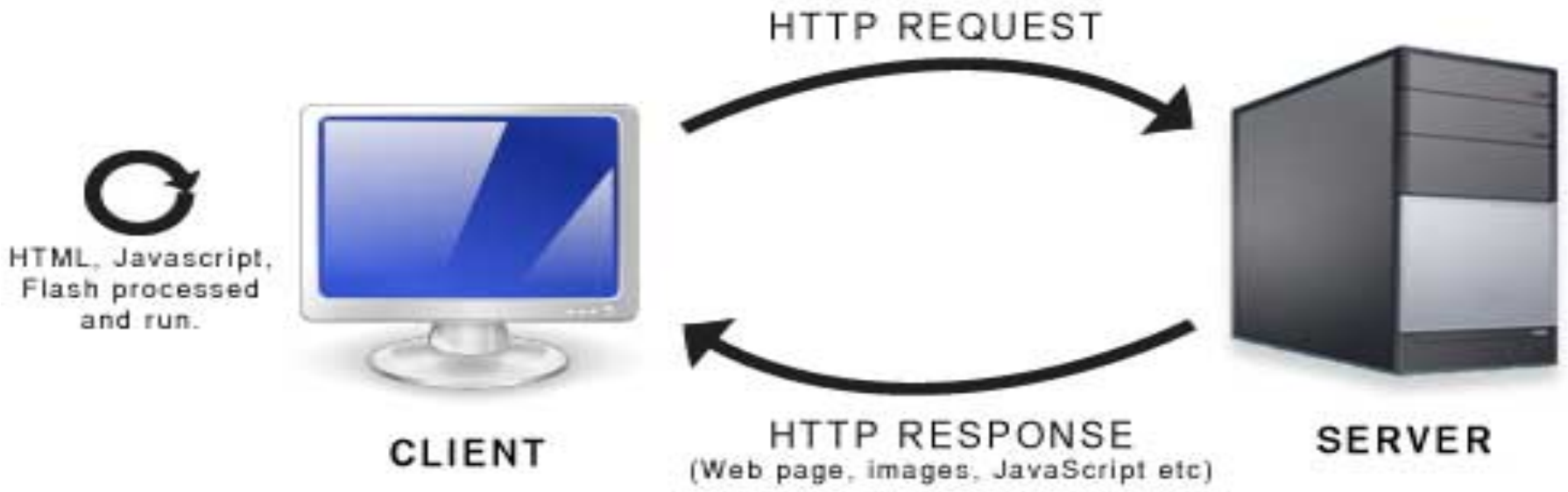
Internet of Things

Application Layer Protocols

Slide Set: 1

Hypertext Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP)



HTTP Request and Response

HTTP Request

An HTTP client sends an HTTP request to a server in the form of a request message which includes following format:

- A Request-line
- Zero or more header (General |Request | Entity) fields followed by CRLF
- An empty line indicating the end of the header fields
- Optionally a message-body

HTTP Response

After receiving and interpreting a request message, a server responds with an HTTP response message:

- A Status-line
- Zero or more header (General | Response | Entity) fields followed by CRLF
- An empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields
- Optionally a message-body

HTTP Request Methods and Response Status

HTTP Request Methods

S.N.	Method
1	GET
2	HEAD
3	POST
4	PUT
5	DELETE
6	CONNECT
7	OPTIONS
8	TRACE

HTTP Response Status Codes

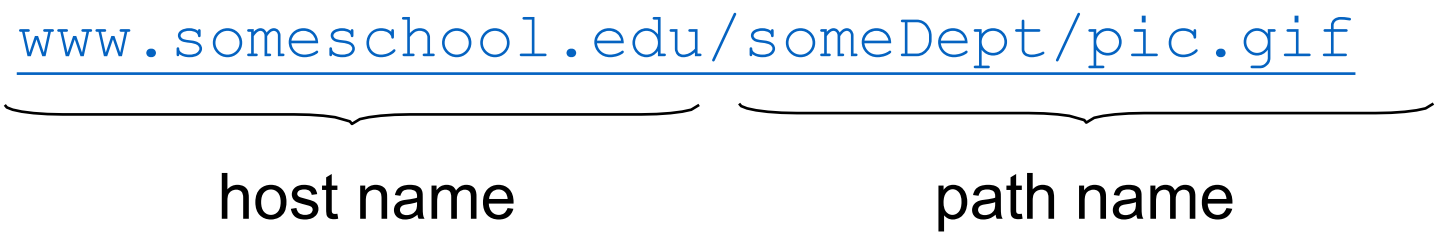
S.N.	Code
1	1xx: Informational
2	2xx: Success
3	3xx: Redirection
4	4xx: Client Error
5	5xx: Server Error

Hypertext Transfer Protocol (HTTP)

- HTTP: is implemented in client and server, allows the client and server to talk to each other

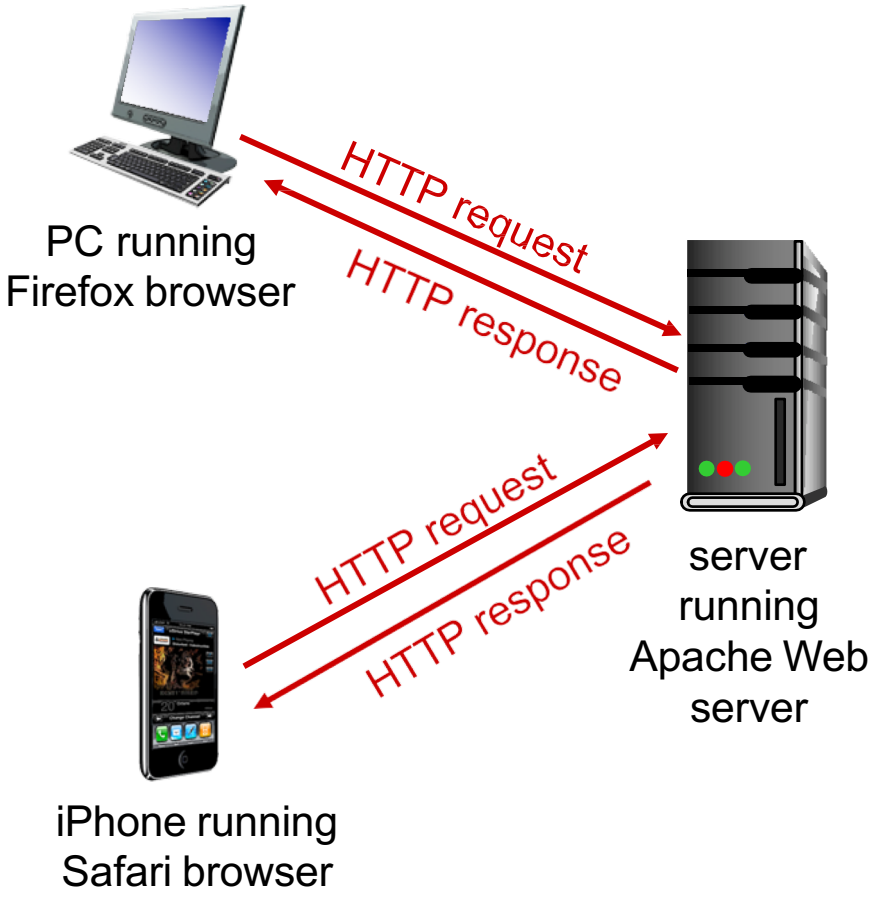
A review…

- A web page consists of **objects**
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web page consists of base HTML file which includes several referenced objects
- Each object is addressable by a URL



Hypertext Transfer Protocol (HTTP)

- **Web's application layer protocol**
- **Client**: browser that requests, receives, (using HTTP protocol) and "displays" Web objects.
  - Example: Firefox, Chrome, Safari
- **Server**: Web server sends (using HTTP protocol) objects in response to requests
  - Example: Apache, Microsoft Internet Information Server



Hypertext Transfer Protocol (HTTP)

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

- **HTTP is "stateless"**
  - **Server maintains no information about past client requests**
  - For example, it is not important for the server if the client has asked for the same object several times in the past second

Hypertext Transfer Protocol (HTTP)

REST

- HTTP is based on **Representational State Transfer (REST),** an architectural style that makes information available as resources identified by URIs

- Principles of REST
  - **Resources**: Objects identified by their Uniform Resource Identifier (URI)
  - **Representations:** Transfer JSON, XML, HTML, or some other defined format, to represent data objects and attributes
  - **Messages:** For example, GET, POST, PUT, and DELETE
  - **Stateless:** The client–server communication is constrained by no client context being stored on the server between requests
    - **Each request from any client contains all the information necessary to service the request, and session state is held in the client**
  - Web service APIs that adhere to the REST architectural constraints are called **RESTful APIs**

Hypertext Transfer Protocol (HTTP)

Should each request/response pair be sent over a separate TCP connection, or should all of the requests and their corresponding responses be sent over the same TCP connection?

Non-persistent HTTP

- At most one object sent over TCP connection, and then the connection is closed
- Downloading multiple objects required multiple connections

Persistent HTTP

- Multiple objects can be sent over a single TCP connection between client and server

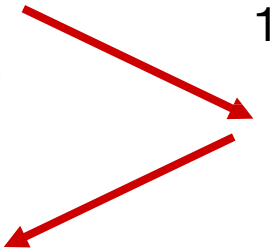
Hypertext Transfer Protocol (HTTP)

Non-persistent HTTP

qSuppose user enters URL:
www.someSchool.edu/someDepartment/home.index

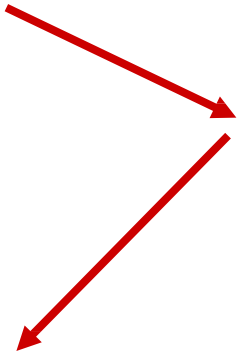
(contains text, references to 10 jpeg images)

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



1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80; "**accepts**" connection, notifying client

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



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

Hypertext Transfer Protocol (HTTP)

Non-persistent HTTP

4. **HTTP server closes TCP connection**. In fact, TCP does not close the connection until it knows for sure that the client has received the response

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

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

Hypertext Transfer Protocol (HTTP)

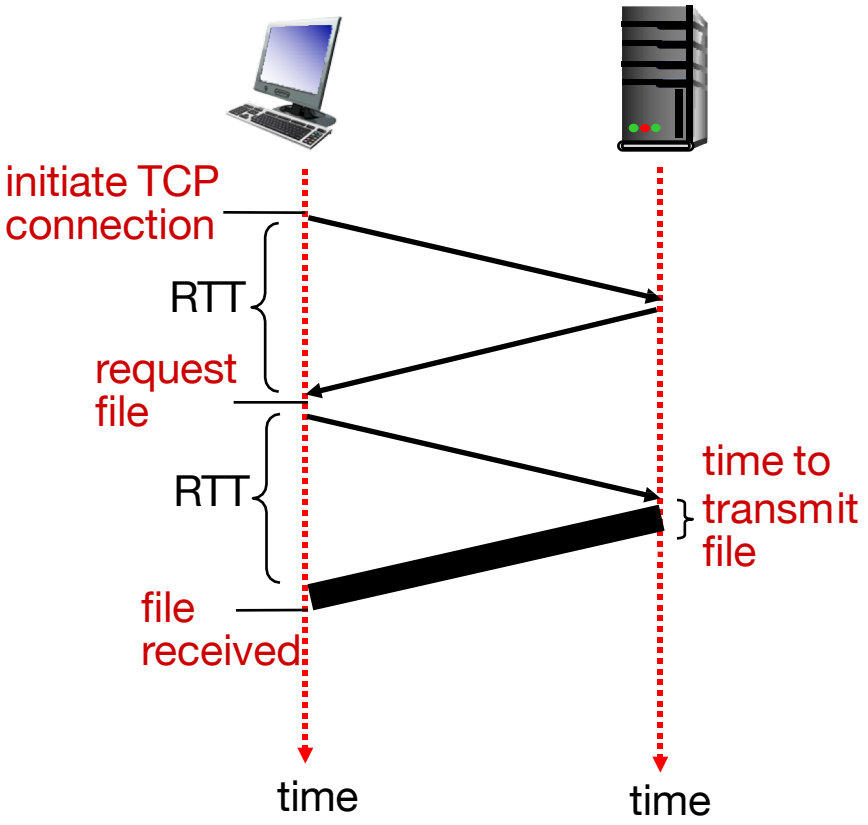
Non-persistent HTTP

Round Trip Time (RTT): Time for a **small** packet to travel from client to server and back

RTT= prop delays + queuing delays + processing delays

Non-persistent HTTP response time:

= 2RTT+ file transmission time



Hypertext Transfer Protocol (HTTP)

Non-persistent HTTP

Non-persistent HTTP issues:

- Requires 2 RTTs per object
- OS overhead for **each** TCP connection:
  - In fact, for each connection, TCP buffers must be allocated and TCP variables must be kept in both client and server

Persistent HTTP

- Server leaves connection open after sending response
- Subsequent HTTP messages between same client/server sent over open connection
- Client sends requests as soon as it encounters a referenced object
  - In fact, the client does not need to wait for a reply before making another request (pipelining)
- As little as one RTT for all the referenced objects
- HTTP server closes the connection when it is not used for a certain time
- **The default mode of HTTP used persistent connections with pipelining**

Hypertext Transfer Protocol (HTTP)

Persistent HTTP

- In HTTP 1.1 and HTTP/2
  - **All connections are considered persistent unless declared otherwise**
  - The HTTP persistent connections **do not use separate keep-alive messages**, they just allow multiple requests to use a single connection
  - However, the default connection timeout of Apache httpd 1.3 and 2.0 is as little as 15 seconds, and just 5 seconds for Apache httpd 2.2 and above
  - The advantage of a short timeout is the ability to deliver multiple components of a web page quickly while not consuming resources to run multiple server processes or threads for too long

Hypertext Transfer Protocol (HTTP)

Messages

- **Two types of HTTP messages: Request, Response**
  - ASCII (human-readable format)

- HTTP requests and HTTP responses use a generic message format of RFC 822 for transferring the required data
- This generic message format consists of the following four items.
  - A Start-line
  - Zero or more header lines followed by CRLF (`\r\n`)
  - An empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields
  - Optionally a message-body

Messages

Message Start-Line

- A start-line will have the following generic syntax:

```
start-line = Request-Line | Status-Line
```

sent by client — sent by server

qExample:

- **Request-Line sent by client**

```
GET  /hello.htm  HTTP/1.1
```

- **Status-Line sent by server**

```
HTTP/1.1  200  OK
```

Messages

Header Fields

- HTTP header fields provide required information about the request or response, or about the object sent in the message body
- Four types of HTTP message headers:
  - **General-header:** These header fields have general applicability for both request and response messages
  - **Request-header:** For request messages only
  - **Response-header:** For response messages only
  - **Entity-header:** These header fields define meta information about the entity-body or, if no body is present, about the resource identified by the request

Hypertext Transfer Protocol (HTTP)

Request Message

- **Request-Line**
- The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF
- The elements are separated by space (SP) characters

```
Request-Line = Method SP Request-URI SP
                                    HTTP-Version CRLF
```

- **Request Method**
- The request **method** indicates the method to be performed on the resource identified by the given **Request-URI**
- The method is case-sensitive and should always be mentioned in uppercase

Hypertext Transfer Protocol (HTTP)

Request Message

Method and Description
GET: The GET method is used to retrieve information from the given server using a given URI; Requests using GET should only retrieve data and should have no other effect on the data
HEAD: Same as GET, but it transfers the status line and the header section only
POST: A POST request is used to send data to the server, for example, customer information, file upload, etc., using HTML forms
PUT: Replaces all the current representations of the target resource with the uploaded content
DELETE: Removes all the current representations of the target resource given by URI
CONNECT: Establishes a tunnel to the server identified by a given URI
OPTIONS: Describe the communication options for the target resource
TRACE: Performs a message loop back test along with the path to the target resource

Hypertext Transfer Protocol (HTTP)

Response Message

Code and Description
1xx: Informational: It means the request was received and the process is continuing
2xx: Success: It means the action was successfully received, understood, and accepted
3xx: Redirection: It means further action must be taken in order to complete the request
4xx: Client Error: It means the request contains incorrect syntax or cannot be fulfilled
5xx: Server Error: It means the server failed to fulfill an apparently valid request

Hypertext Transfer Protocol (HTTP)

Response Message

Status line

Header lines

Message Body

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Hypertext Transfer Protocol (HTTP)

- The Web consists of three technologies: HTML, HTTP/REST, and URI
- **Only HTTP/REST and URI are useful for M2M communication**

- For M2M, data formats are required to replace HTML
  - **Java Script Object Notation (JSON)**
  - **Efficient XML Interchange (EXI)** (based on XML): A very compact representation for XML; intended to simultaneously optimize performance and the utilization of computational resources

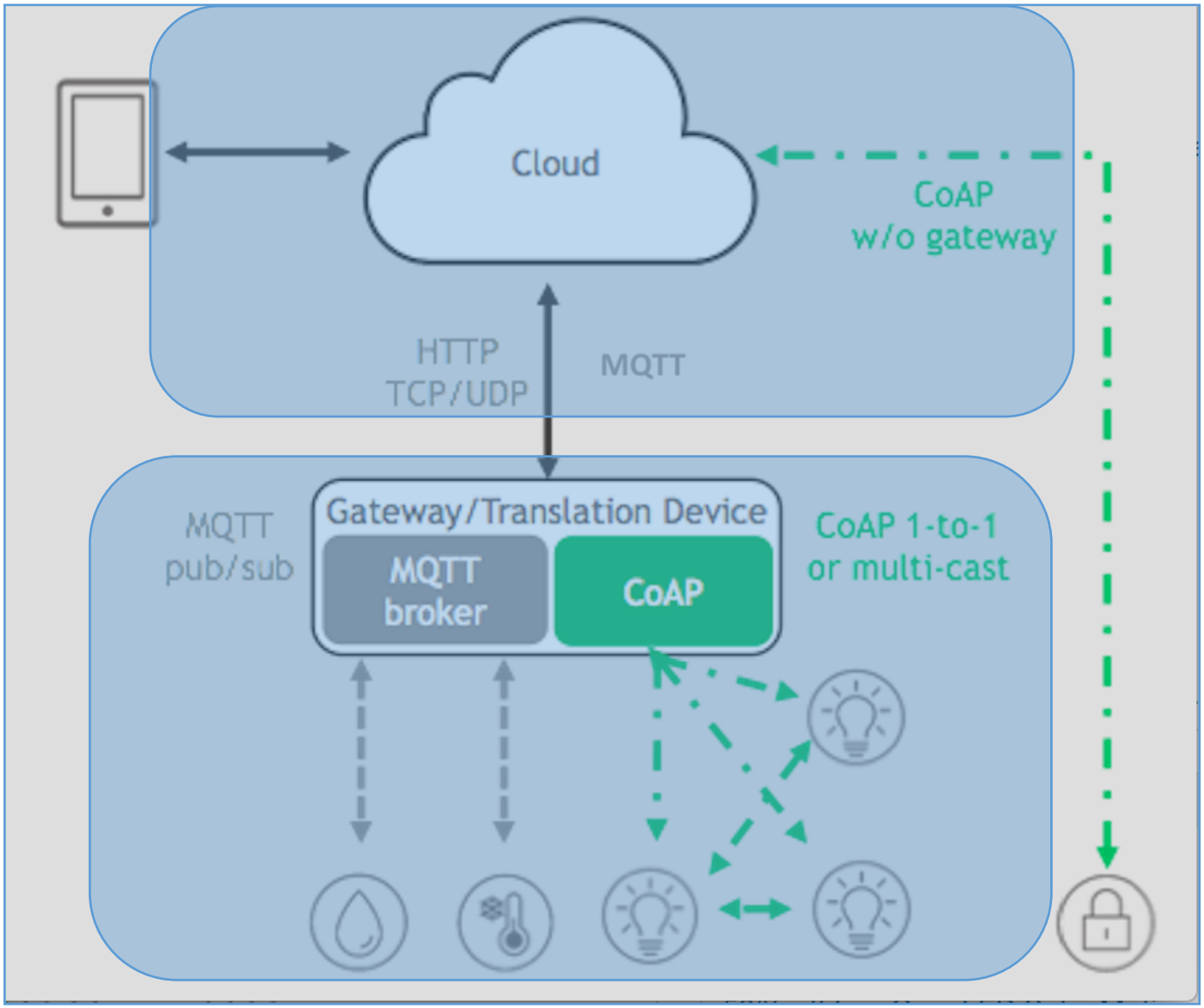
- **HTTP is an expensive protocol in terms of implementation and resource usage**

- Request headers today vary in size from **~200 bytes to over 2KB**
- As applications use more cookies and user agents expand features, typical header sizes of **700-800 bytes** is common

Machine to Machine Protocols

Machine to Machine Protocol

Where M2M is being used?

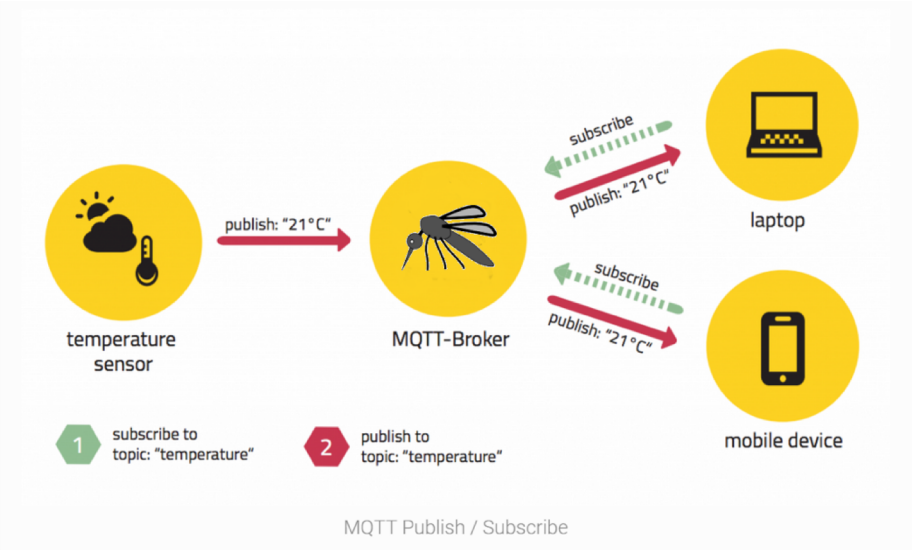


Message Queue Telemetry Transport (MQTT)

Message Queue Telemetry Transport (MQTT)

Overview

- MQTT history
- Light-weight messaging protocol, rides on TCP
- Broker / Clients architecture
- Publication / Subscription messaging model
- No pre-defined format for payload



MQTT Publish / Subscribe

Message Queue Telemetry Transport (MQTT)

Publish & Subscribe Pattern

- The **publish/subscribe** message pattern to provide **one-to-many message distribution** and decoupling of applications
- Publish/subscribe is **event-driven** and enables messages to be pushed to clients
- The central communication point is the MQTT **broker**, it is in charge of dispatching all messages between the senders and the rightful receiver
- Each client that publishes a message to the broker, includes a **topic** into the message. **The topic is the routing information for the broker**
- Each client that wants to receive messages subscribes to a certain topic and the broker delivers all messages with the matching topic to the client
- Therefore, the clients don't have to know each other
- This architecture enables highly scalable solutions without dependencies between the data producers and the data consumers

Message Queue Telemetry Transport (MQTT)

Difference between MQTT and traditional message Queue

- A Message Queue stores message until they are consumed. In MQTT, it is possible the message is not processed by any client.
- In Message Queue, a message will only be consumed by one client
- MQTT has more flexibility compared to a message queue

Message Queue Telemetry Transport (MQTT)

MQTT Client

- MQTT client includes publisher or subscriber
- In general, a MQTT client can be both a publisher & subscriber at the same time
- A MQTT client can run on any device from a micro controller up to a server. MQTT C client code only takes 30KB, Java code is about 100KB.
- MQTT client libraries are available for a huge variety of programming languages, e.g, C/C++, Arduino, Java, JavaScript, Android, iOS, C#, .NET

  https://github.com/mqtt/mqtt.github.io/wiki/libraries
- MQTT client: Eclipse Paho 

  MQTT.fx(available for Win/MacOS/Linux) etc.

Message Queue Telemetry Transport (MQTT)

MQTT Broker

- MQTT Broker is responsible for receiving all messages, filtering them, and sending the messages to all subscribed clients.
- It holds the session of all persistent clients including subscriptions and missed messages
- Authentication and authorization of clients.
- Self Hosted MQTT brokers:

  Eclipse Mosquitto 

  HiveMQ(licensed) 

- Cloud based MQTT brokers:

  AWS 

  IBM Bluemix 

  HiveMQ (broker.hivemq.com) 

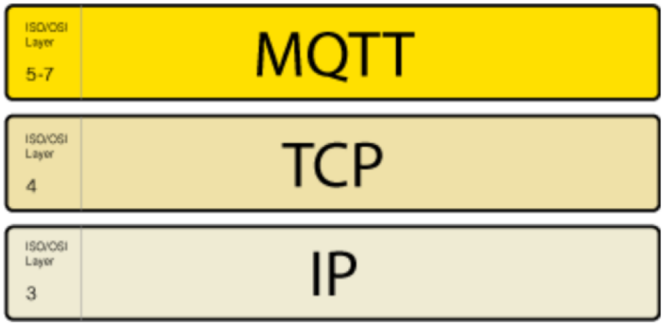
  Microsoft Azure 

  Eclipse Mosquitto (test.mosquitto.org) 

Message Queue Telemetry Transport (MQTT)

Connection

- MQTT protocol is based on top of TCP/IP, both client and broker need to have a TCP/IP stack.

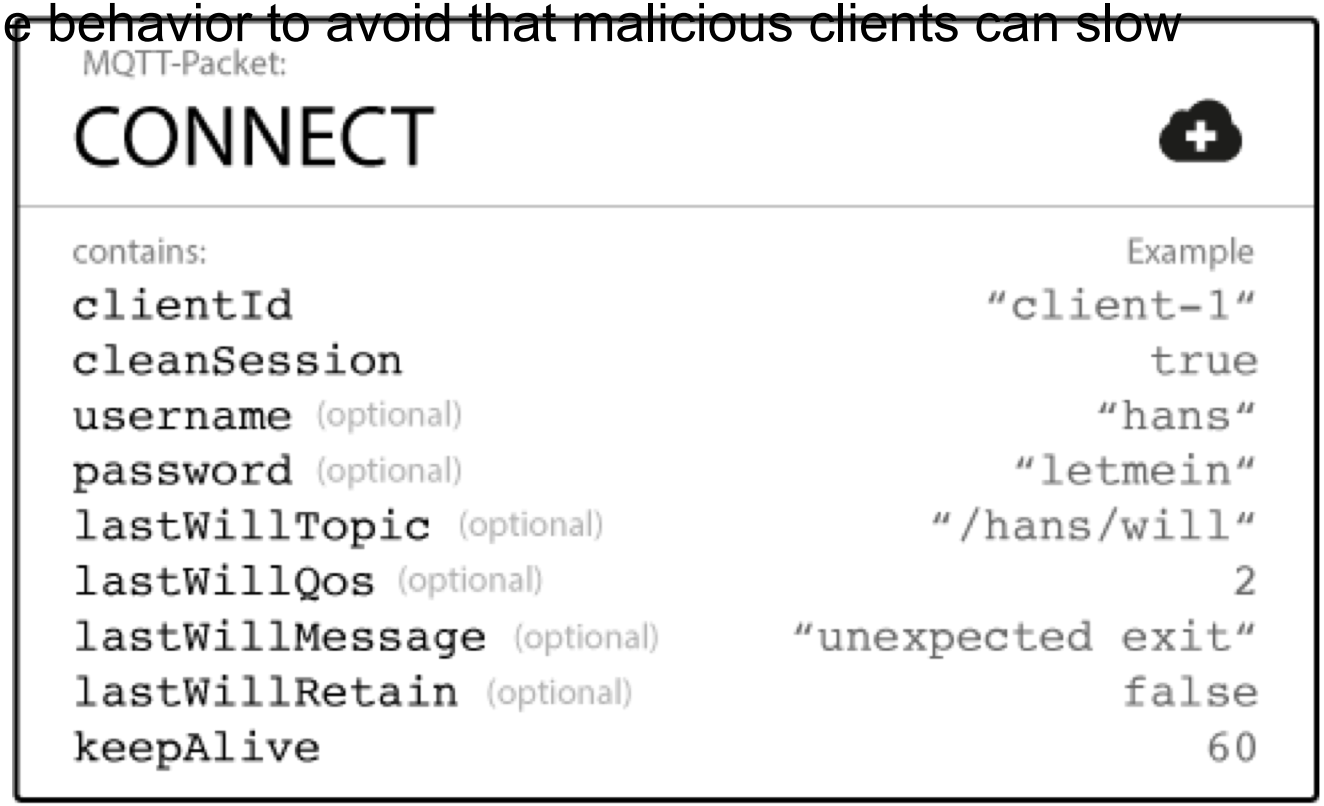


- The MQTT connection itself is always between one client and the broker, no client is connected to another client directly
- The connection is initiated through **a client sending a CONNECT message** to the broker
- The **broker response with a CONNACK and a status code**
- Once the connection is established, the broker will keep it open as long as the client doesn't send a disconnect command or it looses the connection

Message Queue Telemetry Transport (MQTT)

CONNECT Message

- This is sent from the client to the broker to initiate a connection
- If the CONNECT message is malformed (according to the MQTT specifications) or it takes too long from opening a network socket to sending it, the broker will close the connection
- This is a reasonable behavior to avoid that malicious clients can slow down the broker



Message Queue Telemetry Transport (MQTT)

CONNECT Message

ClientId: Unique client identifier, can be empty

CleanSession: Indicates whether the client wants to establish a persistent session or not

- A persistent session (CleanSession is false) means, that the broker will store all subscriptions for the client and also all missed messages, when subscribing with Quality of Service (QoS) 1 or 2
- If clean session is set to true, the broker won't store anything for the client and will also purge all information from a previous persistent session

Username/Password: Can be none. Otherwise, Username/password are sent in plaintext. Application must encrypt the password, or relies on TLS underneath.

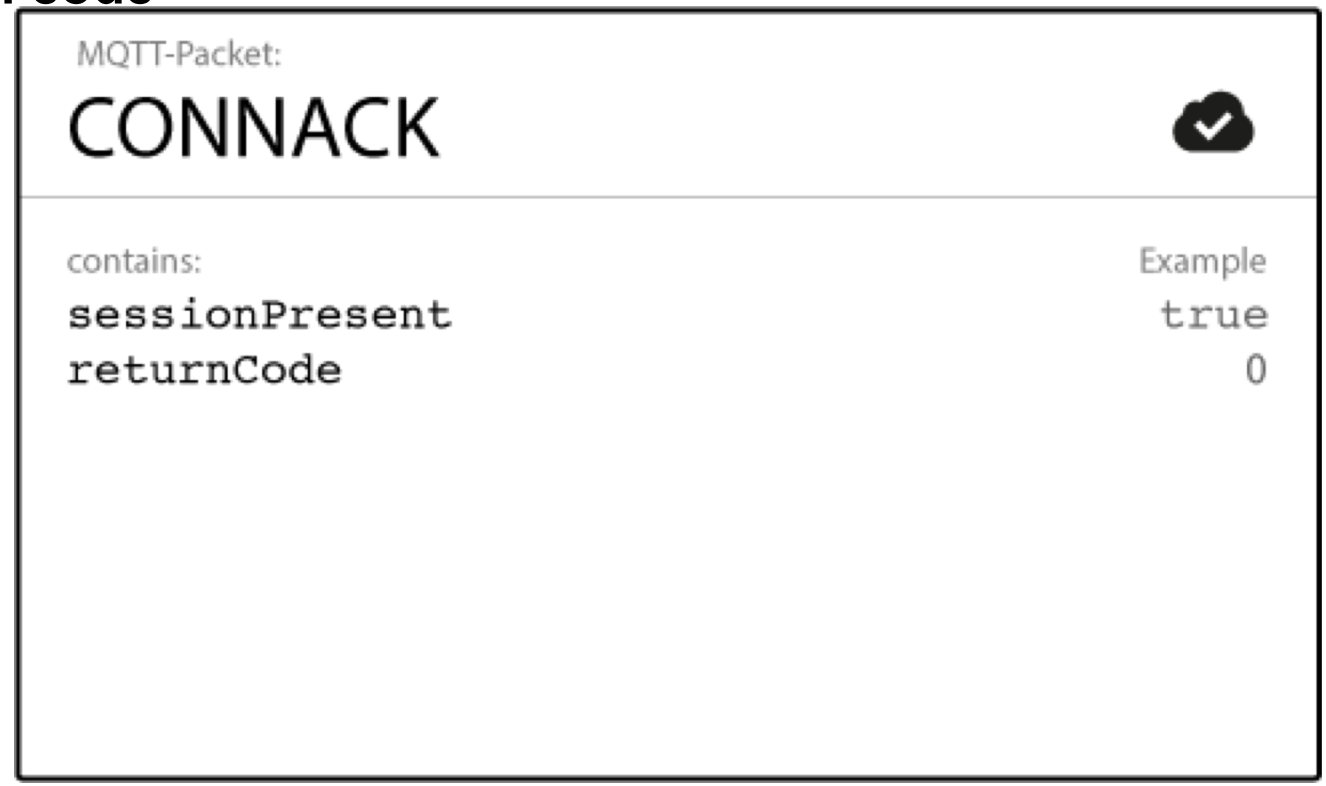
Will Message: Its purpose is to notify other clients when a client disconnects ungracefully. The broker sends this message on behalf of the client.

Keep Alive: A time interval the client is committed to send a PING to the broker, so each other know if the other end is alive and reachable.

Message Queue Telemetry Transport (MQTT)

CONNACK Message

- When a broker obtains a CONNECT message, it responds with a CONNACK message
- The CONNACK contains only two data entries: **session present flag**, **connect return code**



CONNACK Message

SessionPresent flag

§ The session present flag indicate, whether the broker already has a persistent session of the client from previous interactions

- If a client connects and has set CleanSession to true, this flag is always false, because there is no session available
- If the client has set CleanSession to false, the flag is depending on, if there are session information available for the ClientId
- If stored session information exist, then the flag is true and otherwise it is false
- This flag was added newly in MQTT 3.1.1 and **helps the client to determine, whether it has to subscribe to topics or if these are still stored in his session**

Message Queue Telemetry Transport (MQTT)

CONNACK Message

Return Code

Return Code	Return Code Response
0	Connection Accepted
1	Connection Refused, unacceptable protocol version
2	Connection Refused, identifier rejected
3	Connection Refused, Server unavailable
4	Connection Refused, bad user name or password
5	Connection Refused, not authorized

Message Queue Telemetry Transport (MQTT)

Topics

- The central concept in MQTT to dispatch messages are **topics**
- **A topic is a simple string** that can have **more hierarchy levels**, which are separated by a **slash.**

❑ Example: Topic for sending temperature data of the living room
`house/living-room/temperature`

- A topic is case sensitive
- UTF-8

Topics

- Wild cards

❑ Single level wild card: `house/+/temperature`

- `house/living-room/temperature`
- `house/kitchen/temperature`

The **plus sign is a single level wild card** and only allows arbitrary values for one hierarchy

❑ Multiple level wild card(only at the end): `house/living-room/#`

- `house/living-room/temperature`
- `house/living-room/humidity`

Message Queue Telemetry Transport (MQTT)

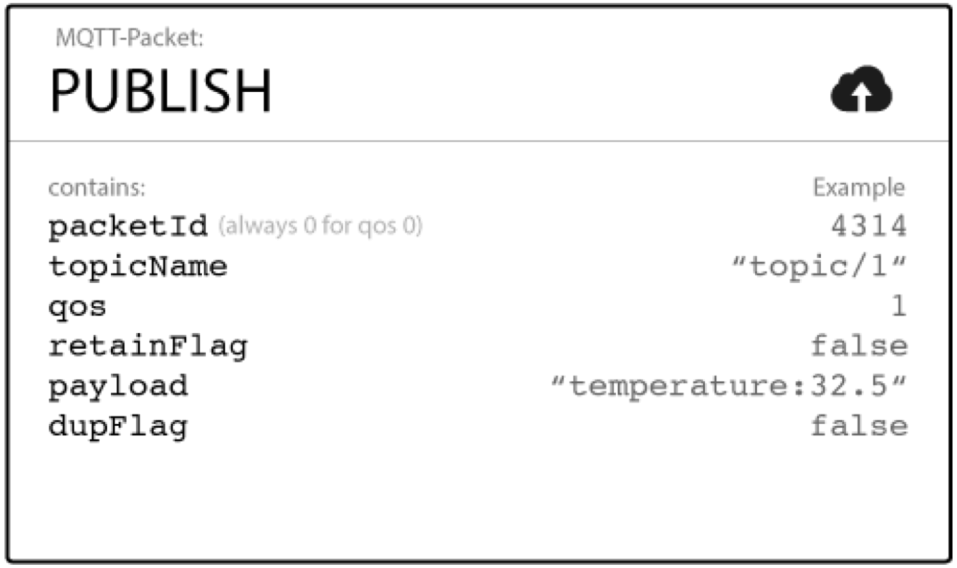
Payload

- **MQTT is agnostic to the content of the payload**
- The format of the payload being sent in MQTT is unspecified
- The message broker does not know (or care) anything about the format of the data and **it is up to the system designer to specify an overall format of the data**
  - § JSON is usually used

Message Queue Telemetry Transport (MQTT)

Publish/Subscribe Process

Publish



- **Packet Identifier:** The packet identifier is a unique identifier between client and broker to identify a message in a message flow, but only relevant for QoS greater than zero. The MQTT identifier is set by the client and/or the broker
- **Topic Name:** A simple string, which is hierarchically structured with forward slashes as delimiters

Publish/Subscribe Process

Publish

- **QoS:** A Quality of Service Level (QoS) for this message. Possible values are 0, 1 or 2.
- **Retain-Flag:** Determines if the message will be saved by the broker for the specified topic as last known good value. New clients that subscribe to that topic will receive the last retained message on that topic instantly after subscribing
- **Payload:** This is the actual content of the message. In binary format.
- **DUP flag**: Indicates that this message is a duplicate and is resent **because the other end didn't acknowledge the original message. Only relevant for QoS greater than 0.** This resend/duplicate mechanism is typically handled by the MQTT client library or the broker as an implementation detail

Publish/Subscribe Process

Publish

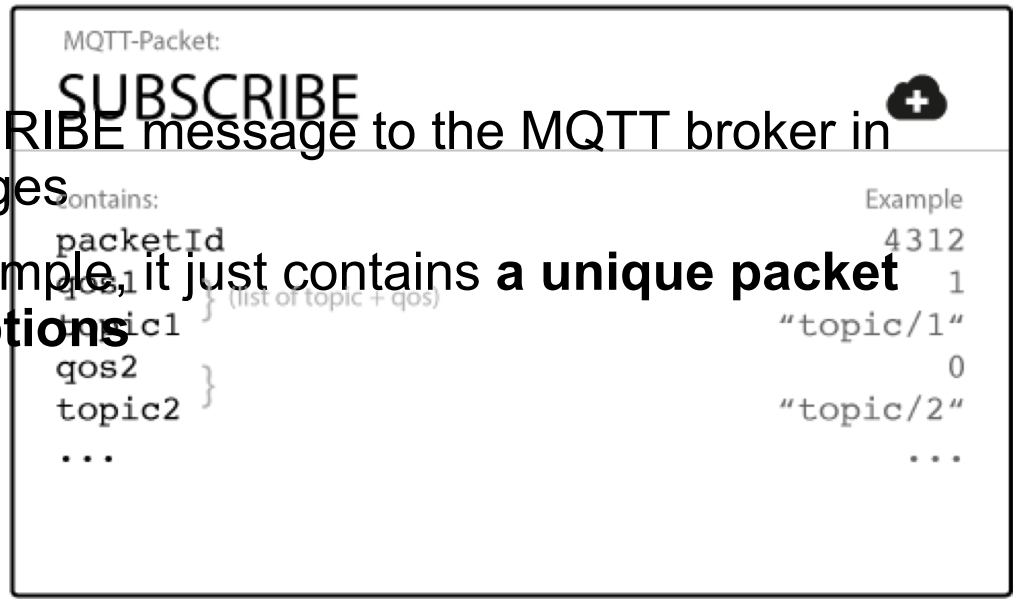
- When a client sends a publish to a MQTT broker, the broker will read the publish, acknowledge the publish if needed (according to the QoS Level) and then process it
- **Processing includes determining which clients have subscribed on that topic and then sending out the message to the selected clients which subscribe to that topic**

Message Queue Telemetry Transport (MQTT)

Publish/Subscribe Process

Subscribe

- A client needs to send a SUBSCRIBE message to the MQTT broker in order to receive relevant messages
- A subscribe message is pretty simple, it just contains **a unique packet identifier** and **a list of subscriptions**



Message Queue Telemetry Transport (MQTT)

Publish/Subscribe Process

Subscribe

- **Packet Identifier**
  - The packet identifier is a unique identifier between client and broker to identify a message in a message flow
  - **This is only relevant for QoS greater than zero**
  - Setting this MQTT internal identifier is the responsibility of the client library and/or the broker

- **List of Subscriptions**
  - A SUBSCRIBE message can contain an arbitrary number of subscriptions for a client
  - **Each subscription is a pair of a topic and QoS level**
  - The topic in the subscribe message can also contain wildcards, which makes it possible to subscribe to certain topic patterns
  - If there are overlapping subscriptions for one client, the highest QoS level for that topic wins and will be used by the broker for delivering the message

Message Queue Telemetry Transport (MQTT)

QoS

- QoS is a major feature of MQTT, it **makes communication in unreliable networks a lot easier**
- MQTT handles retransmission and guarantees the delivery of the message, regardless how unreliable the underlying transport is
- Three QoS modes for message delivery:
  - **Mode 0: At most once**: No acks from the receiver, or stored and redelivered by the send.
  - **Mode 1: At least once:** Messages are assured to arrive but duplicates may occur
  - **Mode 2: Exactly once:** Message are assured to arrive exactly once. Highest level QoS, but the slowest.
- QoS is set by the client, depending on its network reliability and application logic. The broker will honor the QoS set by clients on each end. Therefore, QoS can be downgraded.

Message Queue Telemetry Transport (MQTT)

Persistent Session and Queue Management

- A persistent session saves all information relevant for the client on the broker. The session is identified by the clientId.
- The followings are stored in a persistent session:
  - ❑ Existence of a session, even when there are no subscriptions
  - ❑ All subscriptions
  - ❑ All messages with a QoS 1 or 2, which are not confirmed by the client
  - ❑ All new QoS 1 or 2 messages, which the client missed while offline
  - ❑ All received QoS 2 messages, which are not yet confirmed to the client

Message Queue Telemetry Transport (MQTT)

Security

- MQTT supports TLS(Transport security Layer) SSL
- TLS and SSL are cryptographic protocols which use a handshake mechanism to negotiate various parameters to create a secure connection between the client and the server
- After the handshake is completed, an encrypted communication between client and server is established and no attacker can eavesdrop any part of the communication
- Servers provide a X509 certificate, typically issued by a trusted authority, which clients use to verify the identity of the server
- For Authentication, MQTT specification leaves it to individual brokers to implement.

Message Queue Telemetry Transport (MQTT)

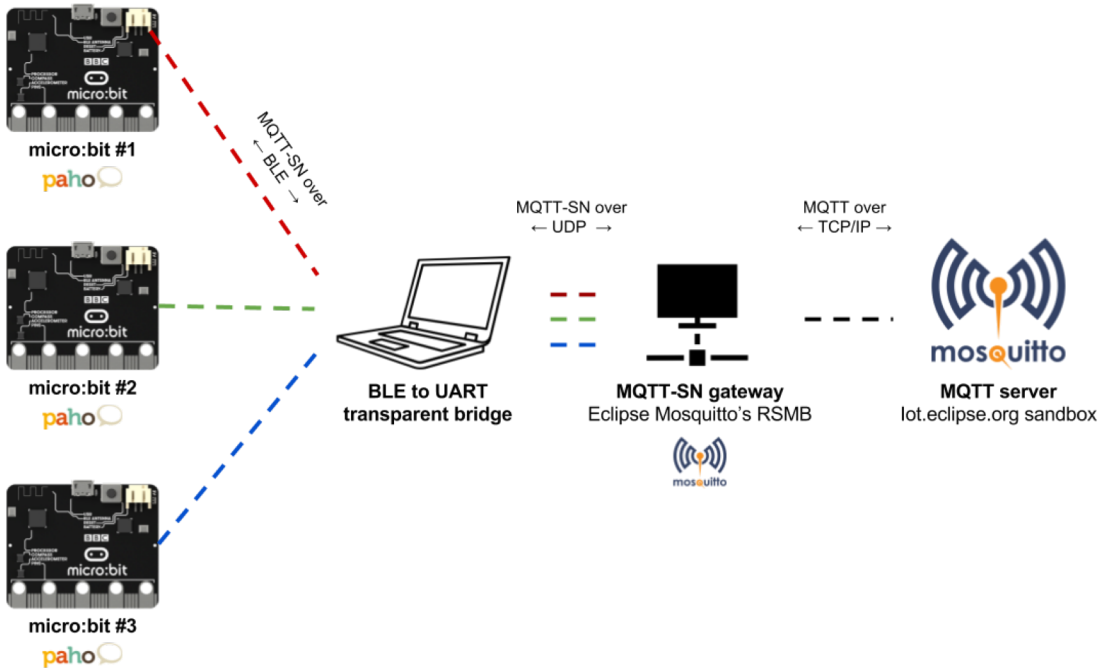
What are the limitations?

- Topics can get very long.
- Runs on TCP, requires TCP/IP stack.
- How about non TCP-IP connections?

Message Queue Telemetry Transport (MQTT)

MQTT-SN (MQTT Sensors Network)

- MQTT-SN supports **topic ID** instead of using a topic name. Saves media bandwidth and device memory
- Topic ID to topic name can be preconfigured in MQTT-SN gateway
- MQTT-SN does not require TCP/IP stack, can be used over UDP. Supports protocols like ZigBee, Z-Wave, BLE.



Message Queue Telemetry Transport (MQTT)

Configuring MQTT on a Raspberry Pi

```
sudo apt-get update
```

//Install Mosquitto broker and clients

```
sudo apt-get install mosquitto mosquitto-clients
```

//Install Python Language Binding

```
sudo apt-get install python-mosquitto
```

//Launch the broker

```
mosquitto –v
```
//Launch Mosquitto with verbose message

```
mosquitto –d
```
//Launch Mosquitto in background

//Pub & Sub messages

```
mosquitto_sub –h 10.0.1.10 –p 1883 –t coen243
mosquitto_pub –h 10.0.1.10 –p 1883 –t coen243 –m “hello from COEN243”
```

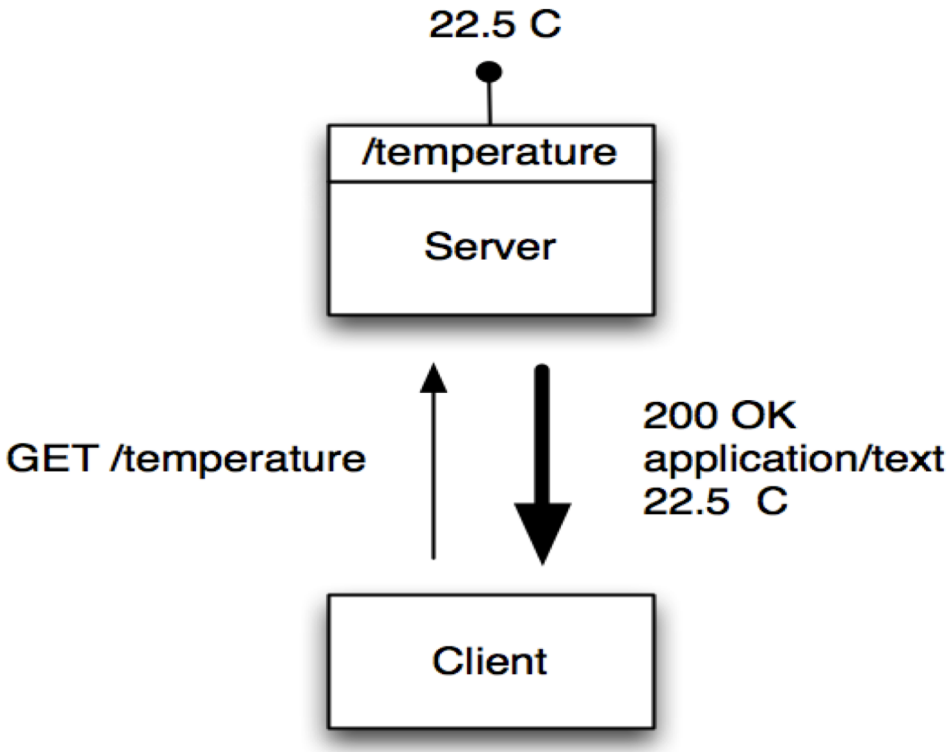
Constrained Application Protocol (CoAP)

Constrained Application Protocol (CoAP)

- The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained networks in the Internet of Things.

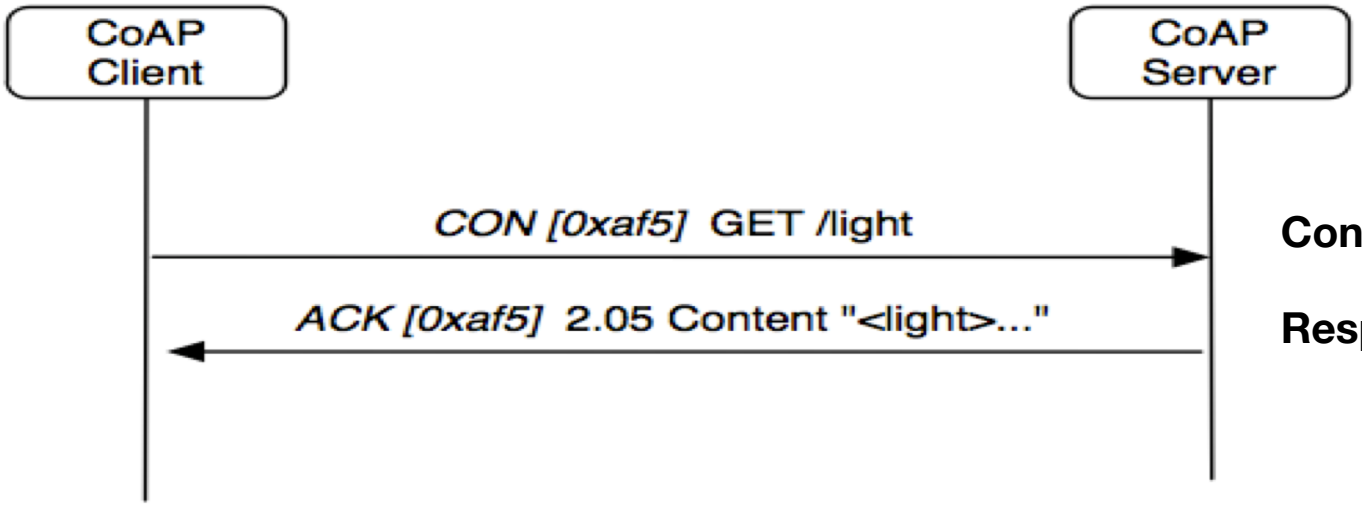
- The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation.

CoAP is REST for Constrained devices



- Makes each device a lightweight server that exposes a REST API.
- A CoAP device can be both client and server.
- Roles can be reversed and the sensor, as a client, can update a REST API at another node, device or server.

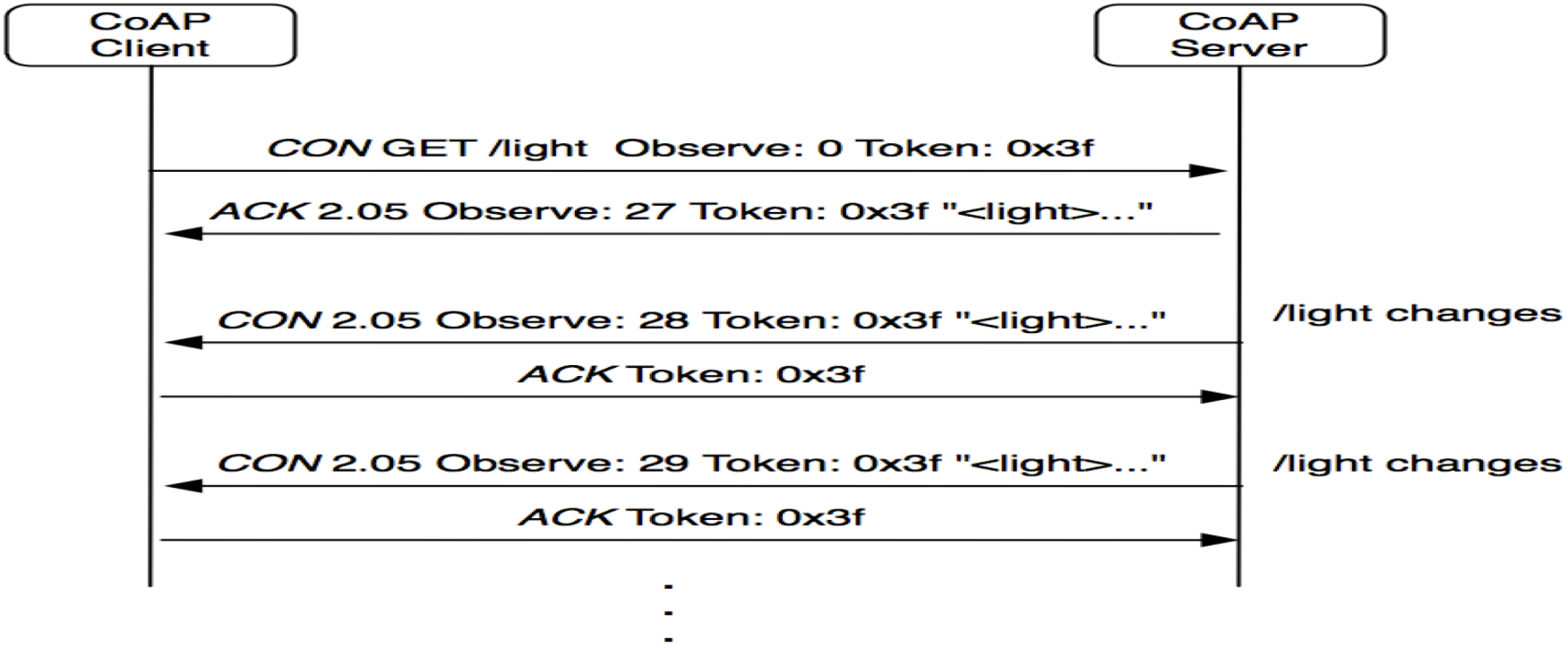
CoAP Request - Response



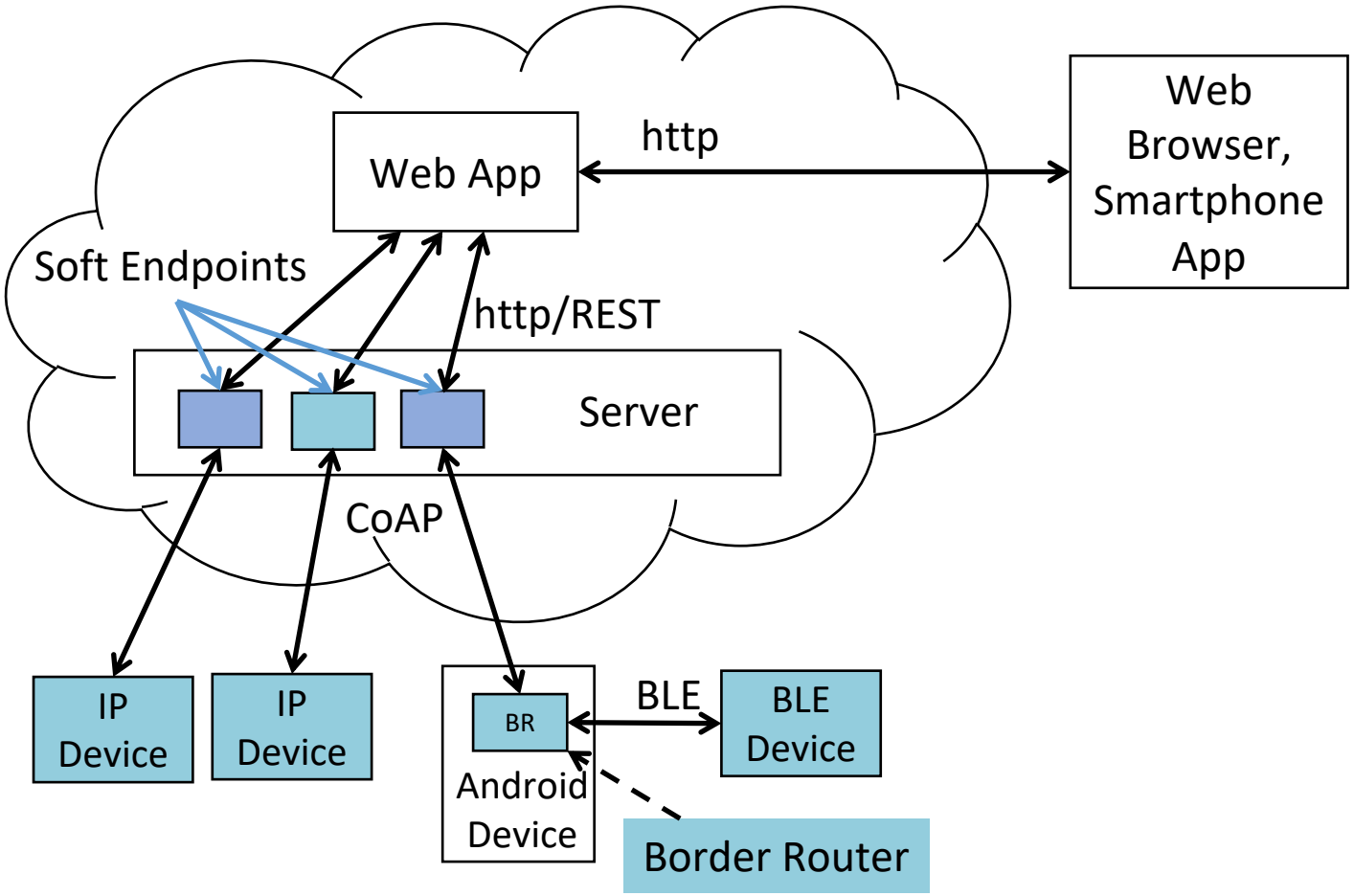
Confirmable Request

Response with Data Payload

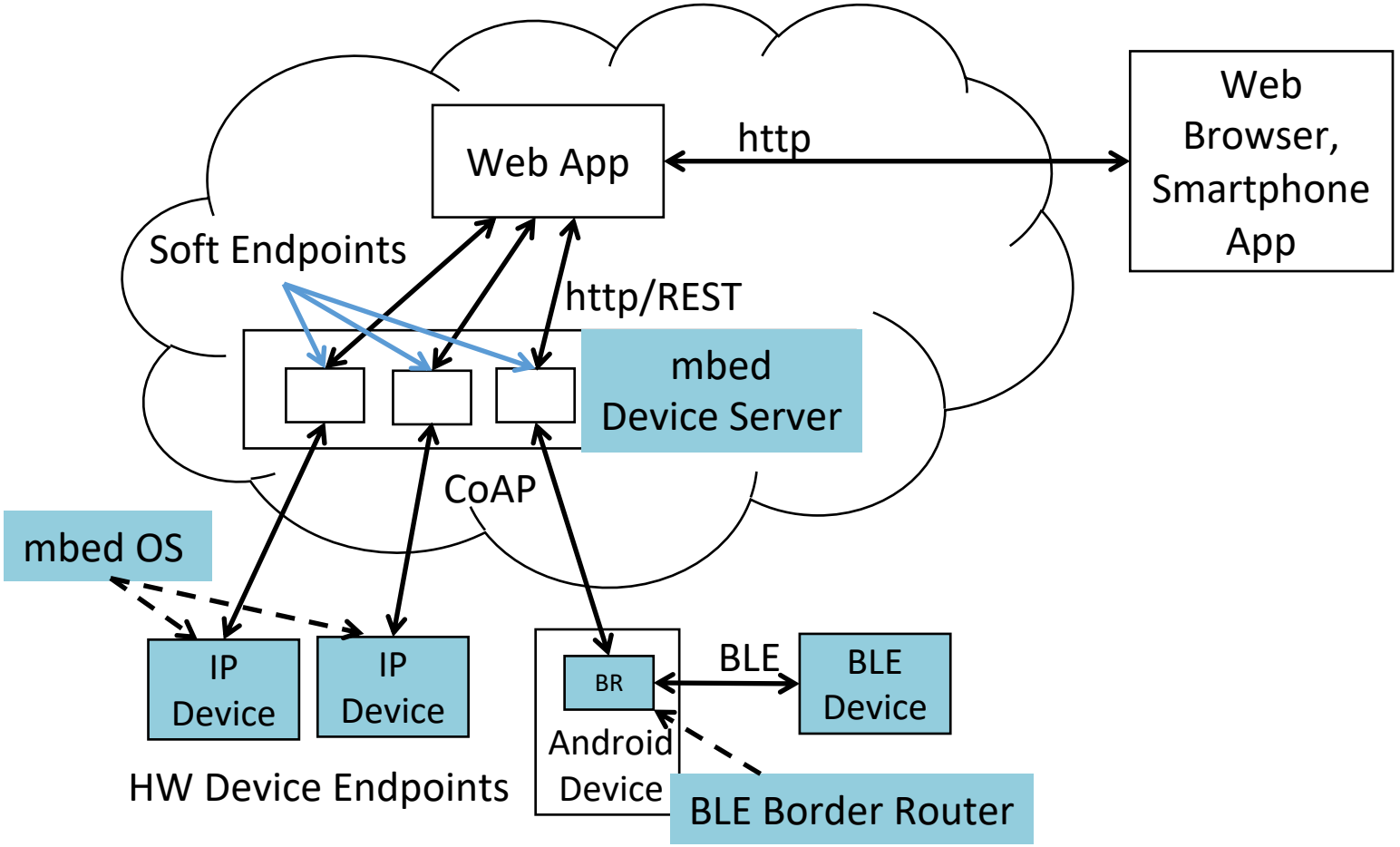
CoAP Observe – Asynchronous Notification



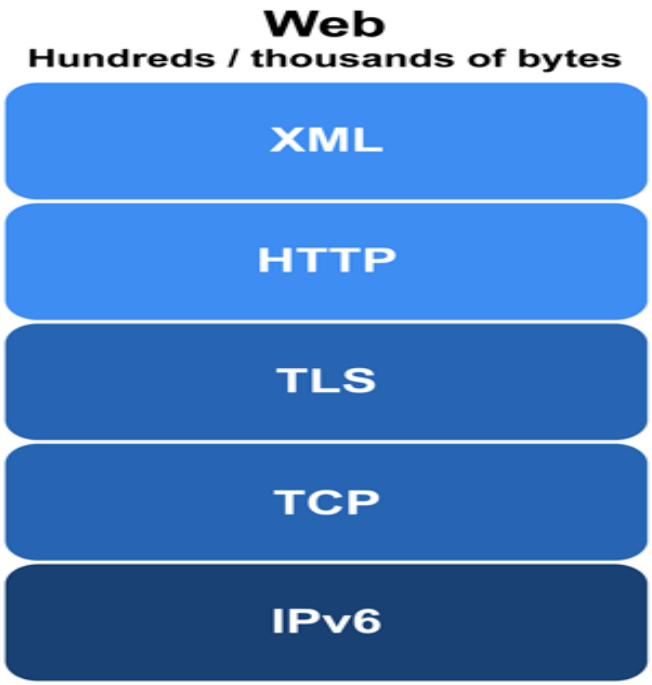
Reference Architecture



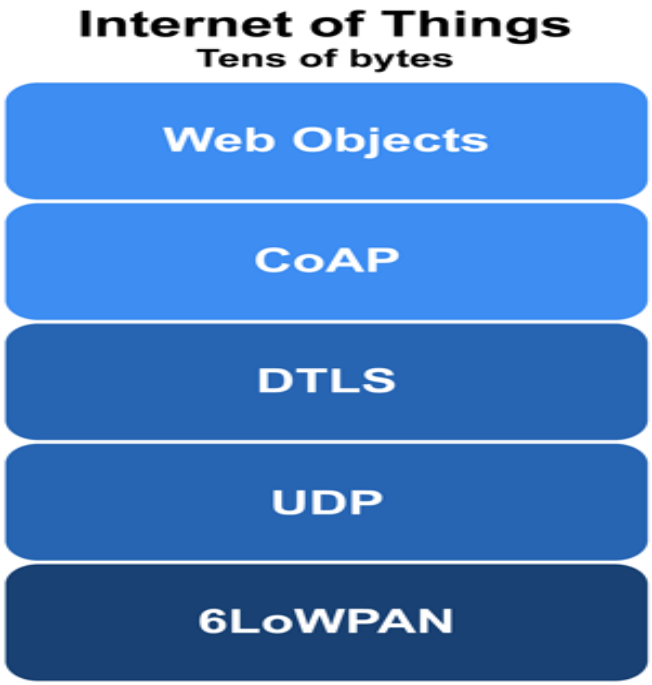
Mapping Reference Architecture



Comparison of Protocols



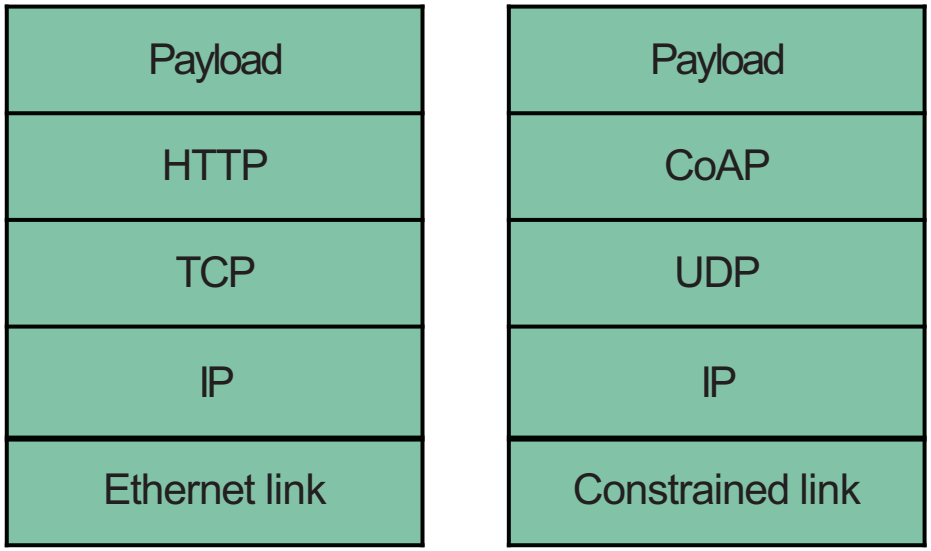
- Inefficient content encoding
- Huge overhead, difficult parsing
- Requires full Internet devices



- Efficient objects
- Efficient Web
- Optimized IP access

Constrained Application Protocol (CoAP)

- **Constrained Application Protocol (CoAP):** A simple, low overhead protocol designed for environments like low-end microcontrollers and constrained networks with critical bandwidth and a high error rate such as 6LowPANs
- The IETF Constrained RESTful Environments (CoRE) working group aims to make the REST paradigm available for devices and networks that might be too constrained to use the typical approaches around the HTTP protocol
- IETF **Constrained RESTful Environments (CoRE)** working group created **CoAP**



Constrained Application Protocol (CoAP)

- **CoAP resembles HTTP in terms of the REST model** with `GET`, `POST`, `PUT` and `DELETE` methods, URIs, response codes, MIME types, etc.
- CoAP can easily interface with HTTP using proxy components, where HTTP clients can talk to CoAP servers and vice versa
- This enables better Web integration and the ability to meet IoT needs

- One of the key design goals of CoAP is to avoid fragmentation at underlying layers, especially at the link layer
- The whole CoAP packet should fit into a single datagram compatible with a single frame at the Ethernet or IEEE 802.15.4 layer

Constrained Application Protocol (CoAP)

Observer

- When using HTTP:
  - Transactions are always client-initiated
  - The client must perform GET operations frequently to update its status about the resources
- This pull model is expensive when:
  - The power is limited
  - The network resourced are limited
  - Nodes sleep most of the time

- **CoAP supports an asynchronous approach for pushing information from servers to clients**
- When a client sends a GET request, it can specify the "**observe**" option to specify its interest in future updates they allow for a client to receive updates when a subject's state change
- They allow for a client to receive updates when a subject's state changes

Constrained Application Protocol (CoAP)

Observer

qExample:

- A light that reacts to a sensor measuring the ambient light level
- The light adjusts its brightness and color accordingly
- There are two peers, where both have URLs, so the light can **link** to the sensor
- We are using CoAP, so the light can **observe** the sensor
- The light (client) becomes an Observer by setting the Observer option within a request
- The light receives updates until one of the following conditions are met:
  - A Confirmable message is unacknowledged
  - The client sends an explicit removal message
  - The Max-Age of the last notification is exceeded

Constrained Application Protocol (CoAP)

Discovery

- Devices must be able to discover each other and their resources
- CoRE deals with autonomous devices and embedded systems
  - The importance of uniform, interoperable resource discovery is much greater than on the current Web
- **To ensure interoperability between CoAP endpoints, the protocol includes a technique for discovering and advertising resource descriptions**
- The description format is standardized because the descriptors are machine interpreted
- A **Discover** operation is added with a **GET** request on the predefined path `/.well-known/core`
- This returns a list of available resources along with applicable paths in the response

Constrained Application Protocol (CoAP)

Discovery

- A ping operation is implemented by sending an empty message to which the peer **responds with an empty reset (RST) message** indicating the '**liveness**' of the endpoint

Message Encoding

- Most **response codes** are similar to HTTP
- For instance, 2.xx indicates success, 4.xx indicates client error and 5.xx indicates server error
- Common response codes:

2.01	Created
2.02	Deleted
2.04	Changed
2.05	Content
4.04	Not found (resource)
4.05	Method not allowed

Constrained Application Protocol (CoAP)

Message Encoding

- **CoAP URIs:** CoAP URIs consist of:
  - Hostname - specified by option field Uri-Host
  - Port number - specified by option field Uri-Port
  - Path - specified by option field Uri-Path
  - Query string - specified by option field Uri-Query
- Uri-Host and Uri-Port are implicit as they are part of underlying layers

qExample, if we request a resource with URI

`coap://hostname:port/leds/red?q=state&on`

the following options are generated:

```
Option#1 Uri-Path leds,
Option#2 Uri-Path red,
Option#3 Uri-Query q=state
Option#4 Uri-Query on
```

Constrained Application Protocol (CoAP)

Methods

- Similar to HTTP, CoAP utilizes methods such as GET, PUT, POST and DELETE to achieve Create, Retrieve, Update and Delete (CRUD) operations
- These methods can be used to create, update, query and delete the resources on the server representing events of an IoT application
- For example, a server getting sensor readings directly can define suitable resources and provide data when the GET method is requested by clients, or sensor readings can be updated by using the POST method by a client holding sensor values
- Besides, resources can be updated using the GET method with a query string
- Similarly, clients can use the POST method to update resources which may represent actuators - the PUT method is preferred to POST for idempotent operations

Constrained Application Protocol (CoAP)

Methods

- As mentioned earlier, The **Observe** method is defined, where the notification is sent periodically or on an event basis, for a single request
- If the request is a CON message, each response is of the CON type and will be acknowledged by the client
- **Observe can be initiated with an extended GET request, with the added Observe option set to zero**
- It can be cancelled by sending a **reset message** for one of the notifications or by sending an explicit **GET request** with the *Observe* option set to value 1
- Notifications use the Observe option with sequential values for reordering of delayed responses and the *Max-Age* option to keep freshness in the cache
- The combination of resource update, Observe method is analogous to publish-subscribe mechanisms in other protocols like MQTT
- CoAP also offers block wise transfers, which enable a large amount of data exceeding datagram limits, and eliminates fragmentation at underlying layers with better reliability

Constrained Application Protocol (CoAP)

- CoAP can be used in several different ways:
  - You can talk CoAP directly from a mobile device or browser to an IoT device
  - You can utilize servers and proxies along the way that perform caching and protocol translation as well as handle security duties such as authentication and access control

OPC-UA

OPC-UA

- OPC-UA is a standard for communication and information modeling in Automation industries.
- Client / Server architecture. Server exposes a set of standard services to navigate through the available objects, read and write data, or subscribe for data changes or events.
- The main goal is to provide interoperability between applications from different vendors by defining the meta model.

LWM2M (Light Weight Machine To Machine)

LWM2M

- LWM2M is a device lifecycle management specification
- It is lightweight.
- Provides a specification for functions like:

❑ firmware upgrade

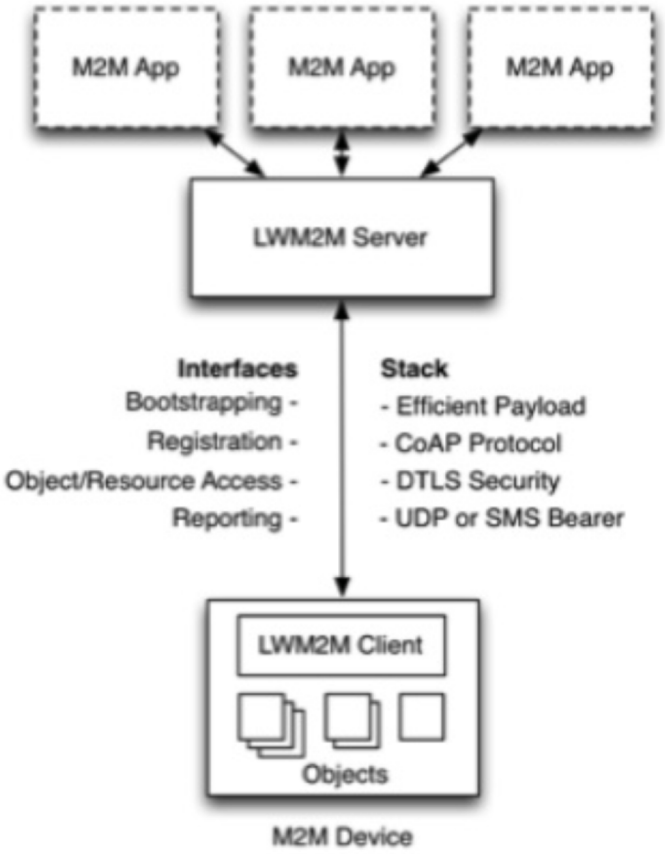
❑ provisioning of certificates

❑ access control policies

❑ connectivity monitoring

❑ etc.

Architecture Overview



LWM2M server

- Persistent endpoint through which devices and apps interact
- Deployable on gateways and in the cloud

LWM2M client

- Hosts resources(objects) that represent a physical device
- Based on CoAP

Eclipse Open Source Solutions

- MQTT broker: Eclipse Mosquitto

  https://mosquitto.org/

- MQTT clients: Eclipse Paho (C, C++, Java, Android, JS, Python, Go)

  https://www.eclipse.org/paho/

- CoAP: Eclipse Californium (Java)

  https://www.eclipse.org/californium/

- OPC-UA: Eclipse Milo (Java)

  https://projects.eclipse.org/proposals/milo

- LWM2M: Eclipse Wakaama(C) / Eclipse Leshan(Java)

  https://www.eclipse.org/wakaama/

  https://www.eclipse.org/leshan/