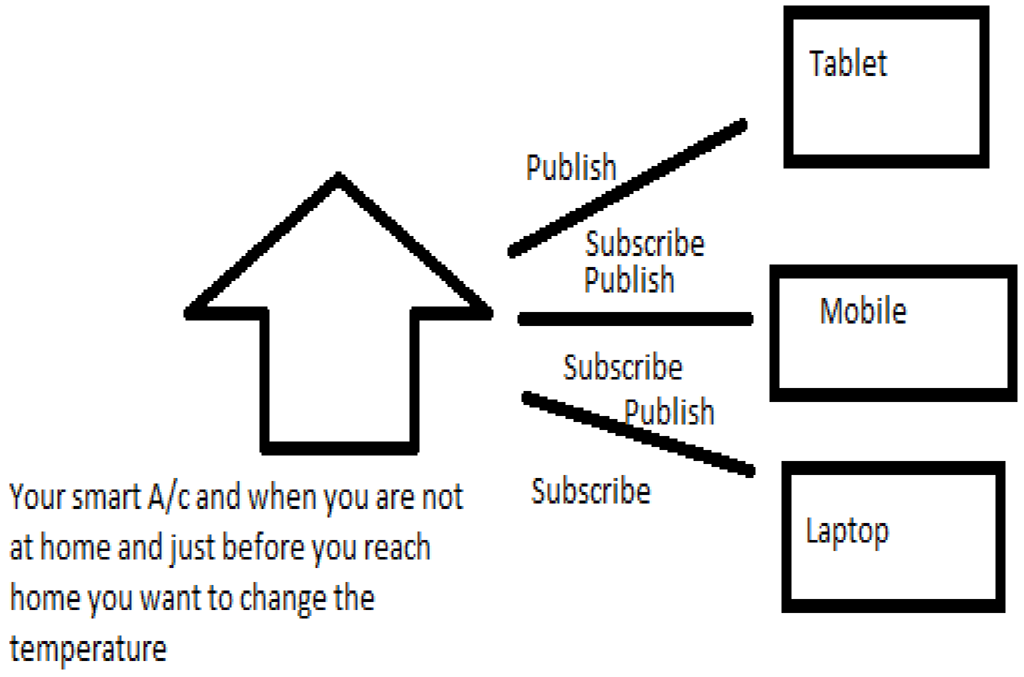
Understanding MQTT and CoAP Protcols

-Tarun Bareja

In today's modern world IOT (Internet of things) i.e. smart devices like your smart TV, smart camera, and smart air conditioners have become essential part of our life. Have you ever thought how do these devices communicate? To get an answer to this we need to understand MQTT protocol or Message Queuing Telemetry Transport Protocol. Let's dig more on this.

Message Queuing Telemetry Transport or MQTT is a simple **messaging protocol**, designed for constrained devices with low-bandwidth. So, it's the perfect solution for **Internet of Things applications**. MQTT allows you to send commands to control outputs, read and publish data from sensor nodes. The MQTT protocol uses a **publish/subscribe** architecture in contrast to **HTTP** with its **request/response** paradigm. **Publish/Subscribe** is event-driven and enables messages to be pushed to clients.


Tablet
Publish
Subscribe
Publish
Mobile
Subscribe
Publish
Subscribe
Laptop
Your smart A/c and when you are not at home and just before you reach home you want to change the temperature

History of MQTT

Andy Stanford-Clark (IBM) and Arlen Nipper (Cirrus Link, then Eurotech) authored the first version of the protocol in 1999. It was used to **monitor an oil pipeline** through the desert. The goal was to have a protocol that is **bandwidth-efficient, lightweight and uses little battery power**, because the devices were connected via satellite link and at this time it was extremely expensive.

In 2013, IBM submitted **MQTT v3.1** to the **OASIS specification body** with a charter that ensured only minor changes to the specification could be accepted. MQTT-SN is a variation of the main protocol aimed at embedded devices on non-TCP/IP networks, such as Zigbee. Historically, the "MQ" in "MQTT" came from the IBM MQ (then 'MQSeries') MQ product line. However, the protocol provides publish-and-subscribe messaging (no queues, in spite of the name) and was specifically designed for resource-constrained devices and low bandwidth, high latency networks such as dial up lines and satellite links, for example.

Overview

Now, that we have some understanding of MQTT let's dig a little deeper into the technical aspects of MQTT. First, the protocol runs on top of the **TCP/IP networking stack**. When clients connect and **publish/subscribe**, MQTT has different message types that help with the handshaking of that process. The MQTT header is **two bytes** and **first byte is constant.**

In the first byte, you specify the type of message being sent as well as the QoS level retain, and DUP (duplication) flags. The second byte is the remaining length field. The MQTT protocol defines two types of network entities: a message broker and a number of clients.

An **MQTT broker** is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients. An **MQTT client** is 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. Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers in turn do not have to be configured with any data about the publishers.

If a broker receives a topic for which there are no current subscribers, it will discard the topic unless the publisher indicates that the topic is to be retained. This allows new subscribers to a topic to receive the most current value rather than waiting for the next update from a publisher. When a publishing client first connects to the broker, it can set up a default message to be sent to subscribers if the broker detects that the publishing client has unexpectedly disconnected from the broker.

Let’s understand some terminologies commonly used of MQTT.

Client – Any publisher or subscriber that connects to the centralized broker over a network is considered to be the client. It’s important to note that there are servers and clients in MQTT. Both publishers and subscribers are called as clients since they connect to the centralized service.

Broker – The broker is the software that receives all the messages from the publishing clients and sends them to the subscribing clients. It holds the connection to persistent clients. Since the broker can become the bottleneck or result in a single point of failure, it is often clustered for scalability and reliability. Some of the commercial implementations of MQTT brokers include HiveMQ, Xively, AWS IoT, and Loop.

Topic – A topic in MQTT is an endpoint to that the clients connect. It acts as the central distribution hub for publishing and subscribing messages. In MQTT, a topic is a well-known location for the publisher and subscriber. It’s created on the fly when either of the clients establishes the connection with the broker

Connection – MQTT can be utilized by clients based on TCP/IP. The standard port exposed by brokers is 1883, which is not a secure port. Those brokers who support TLS/SSL typically use port 8883. For secure communication, the clients and the broker rely on digital certificates. AWS IoT is one of the secure implementations of MQTT, which requires the clients to use the X.509 certificates.

Connect: Waits for a connection to be established with the server and creates a link between the nodes.

Disconnect: Waits for the MQTT client to finish any work it must do, and for the TCP/IP session to disconnect.

Publish: Returns immediately to the application thread after passing the request to the MQTT client.

Quality of service (QoS): Each connection to the broker can specify a quality of service measure. These are classified in increasing order of overhead:

- **At most once** - the message is sent only once and the client and broker take no additional steps to acknowledge delivery (fire and forget).
- **At least once** - the message is re-tried by the sender multiple times until acknowledgement is received (acknowledged delivery).
- **Exactly once** - the sender and receiver engage in a two-level handshake to ensure only one copy of the message is received (assured delivery).

Practical

Prerequisites: I am using windows 10 as a host machine.

You would need the following:

VMware Workstation: You can download it from here. Or Virtual box

Kali Linux: You can download if from here.

Raspberry pi: You can download it from here.

Once you have them downloaded. Open VMware Workstation and install Kali Linux and Raspberry pi.

Follow the commands in Raspberry pi.

- sudo apt-get update
- sudo apt-get install mosquitto
- sudo apt-get install mosquitto-clients
- sudo service mosquitto stop
- sudo service mosquitto start
- mosquitto_pub -t test -h 192.168.174.146 -m hello

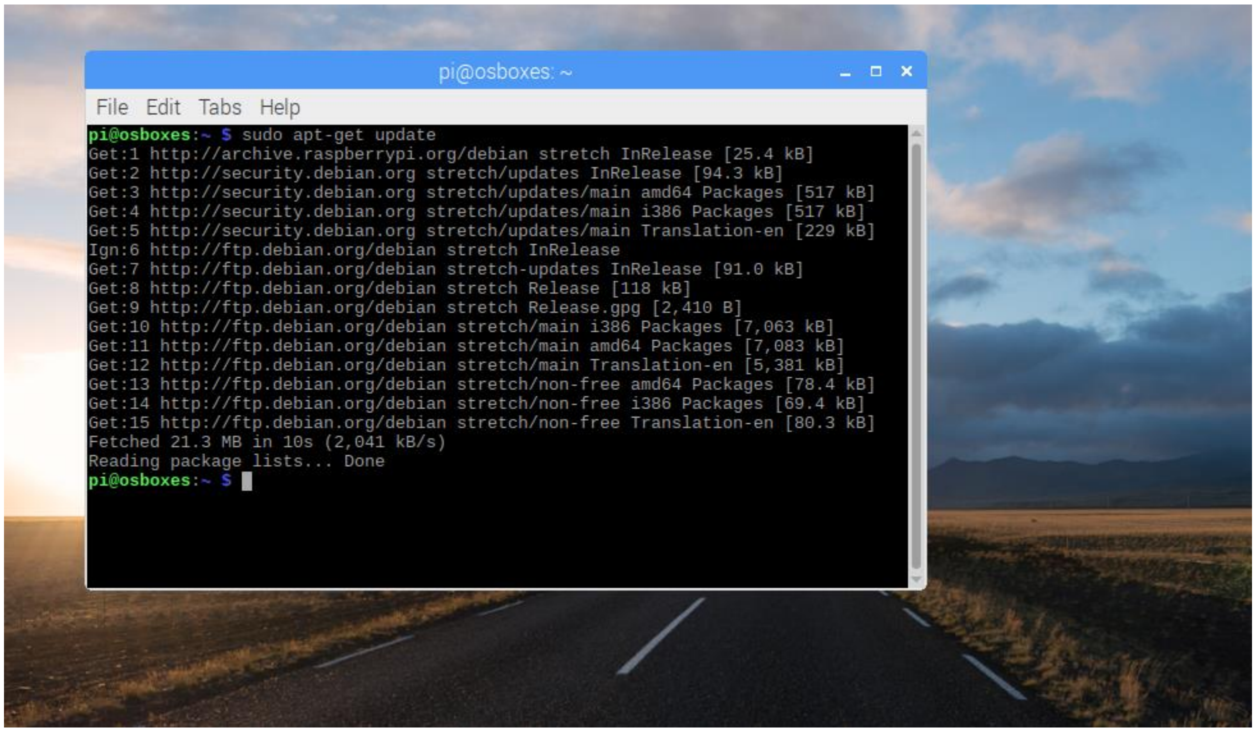
Now, open Kali linux and follow the following commands.

- sudo apt-get update
- sudo apt-get install mosquitto
- sudo apt-get install mosquitto-clients
- mosquitto_sub -h 192.168.174.146 -t test

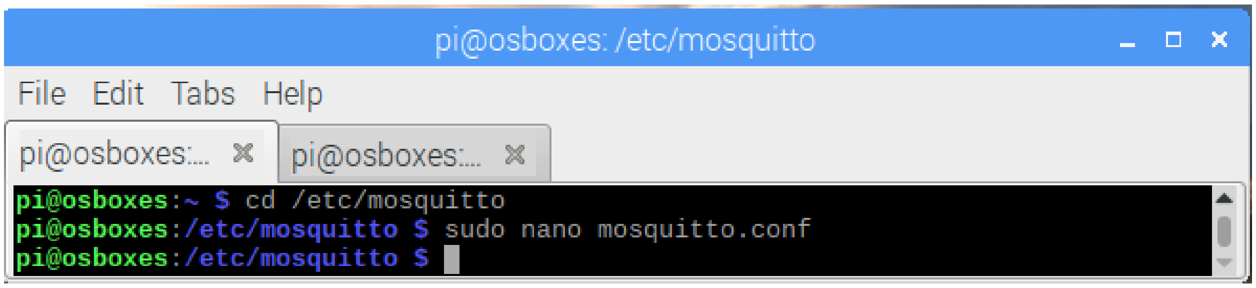
  Note: when dup flag is 0 it means this is the first attempt at sending this published packet, if the flag is 1 it indicates a possible re-attempt at sending the message

POC

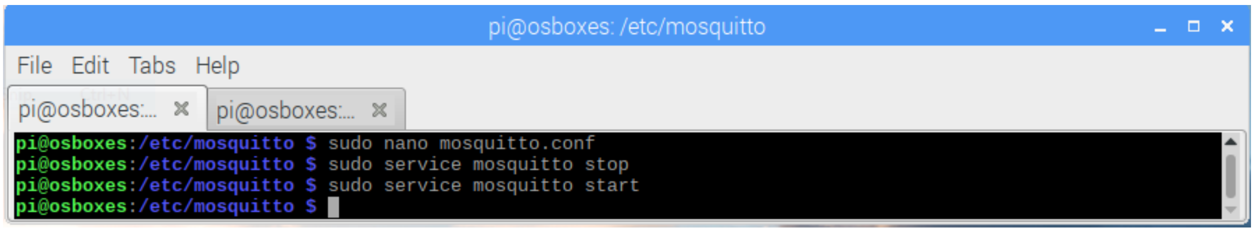
Open Raspberry Pi.



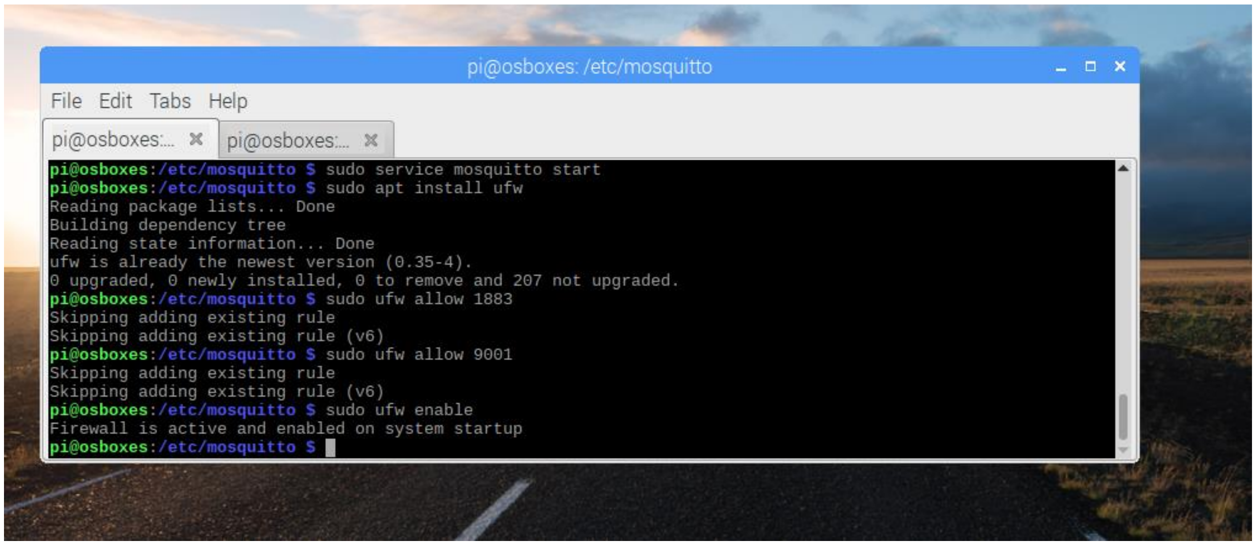
```
pi@osboxes:~ $ sudo apt-get update
Get:1 http://archive.raspberrypi.org/debian stretch InRelease [25.4 kB]
Get:2 http://security.debian.org stretch/updates InRelease [94.3 kB]
Get:3 http://security.debian.org stretch/updates/main amd64 Packages [517 kB]
Get:4 http://security.debian.org stretch/updates/main i386 Packages [517 kB]
Get:5 http://security.debian.org stretch/updates/main Translation-en [229 kB]
Ign:6 http://ftp.debian.org/debian stretch InRelease
Get:7 http://ftp.debian.org/debian stretch-updates InRelease [91.0 kB]
Get:8 http://ftp.debian.org/debian stretch Release [118 kB]
Get:9 http://ftp.debian.org/debian stretch Release.gpg [2,410 B]
Get:10 http://ftp.debian.org/debian stretch/main i386 Packages [7,063 kB]
Get:11 http://ftp.debian.org/debian stretch/main amd64 Packages [7,083 kB]
Get:12 http://ftp.debian.org/debian stretch/main Translation-en [5,381 kB]
Get:13 http://ftp.debian.org/debian stretch/non-free amd64 Packages [78.4 kB]
Get:14 http://ftp.debian.org/debian stretch/non-free i386 Packages [69.4 kB]
Get:15 http://ftp.debian.org/debian stretch/non-free Translation-en [80.3 kB]
Fetched 21.3 MB in 10s (2,041 kB/s)
Reading package lists... Done
pi@osboxes:~ $
```



```
pi@osboxes:~ $ cd /etc/mosquitto
pi@osboxes:/etc/mosquitto $ sudo nano mosquitto.conf
pi@osboxes:/etc/mosquitto $
```

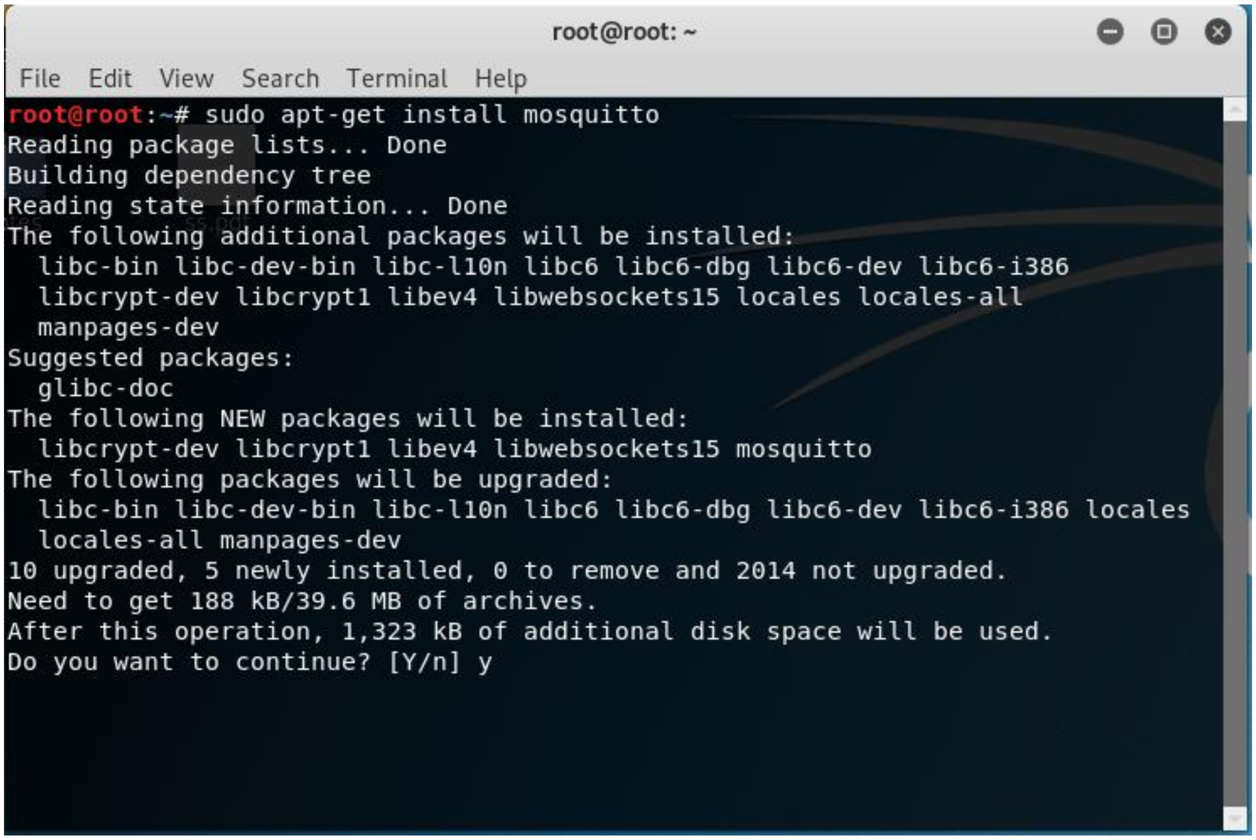


```
pi@osboxes:/etc/mosquitto $ sudo nano mosquitto.conf
pi@osboxes:/etc/mosquitto $ sudo service mosquitto stop
pi@osboxes:/etc/mosquitto $ sudo service mosquitto start
pi@osboxes:/etc/mosquitto $
```

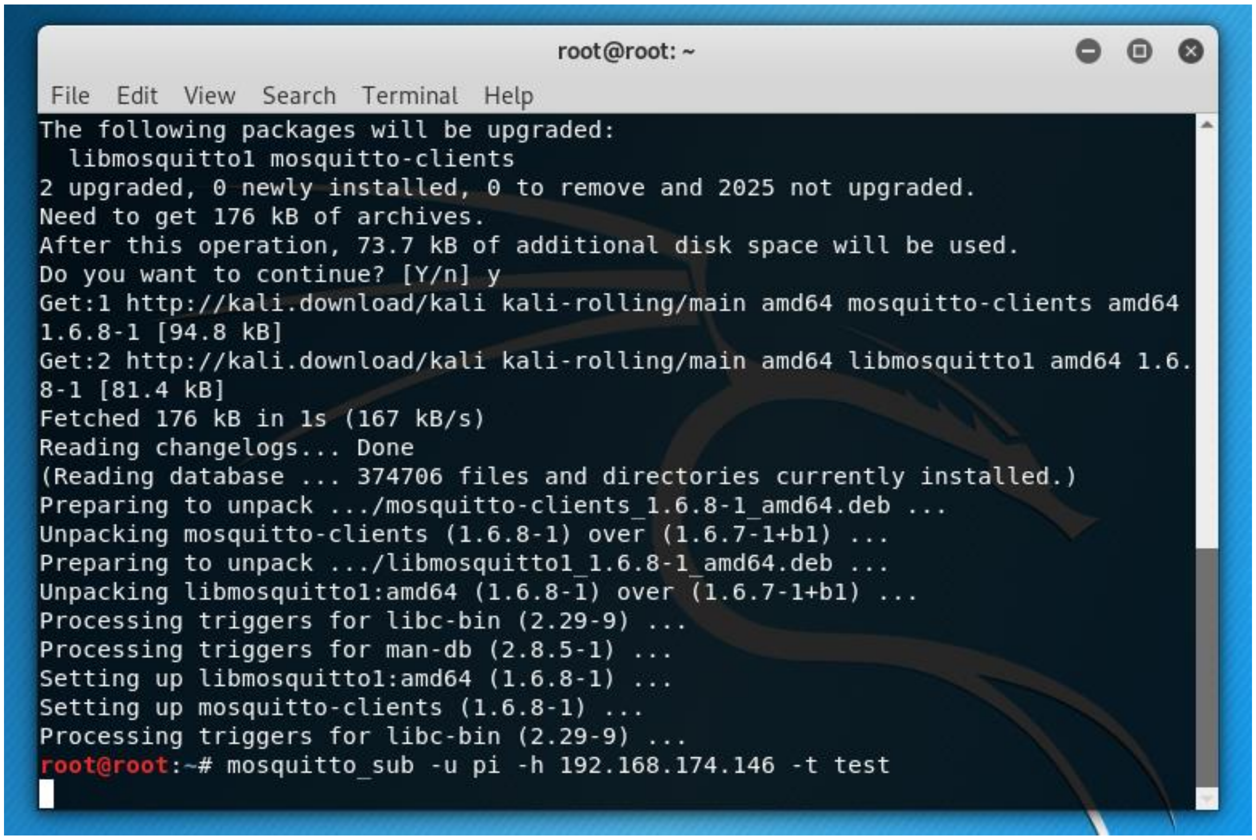


```
pi@osboxes:/etc/mosquitto $ sudo service mosquitto start
pi@osboxes:/etc/mosquitto $ sudo apt install ufw
Reading package lists... Done
Building dependency tree
Reading state information... Done
ufw is already the newest version (0.35-4).
0 upgraded, 0 newly installed, 0 to remove and 207 not upgraded.
pi@osboxes:/etc/mosquitto $ sudo ufw allow 1883
Skipping adding existing rule
Skipping adding existing rule (v6)
pi@osboxes:/etc/mosquitto $ sudo ufw allow 9001
Skipping adding existing rule
Skipping adding existing rule (v6)
pi@osboxes:/etc/mosquitto $ sudo ufw enable
Firewall is active and enabled on system startup
pi@osboxes:/etc/mosquitto $
```

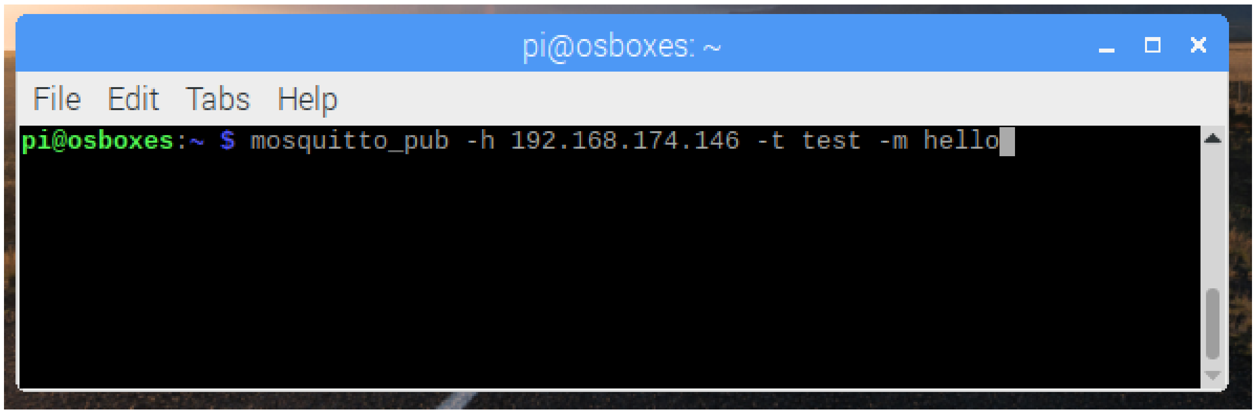
Open Kali Linux now



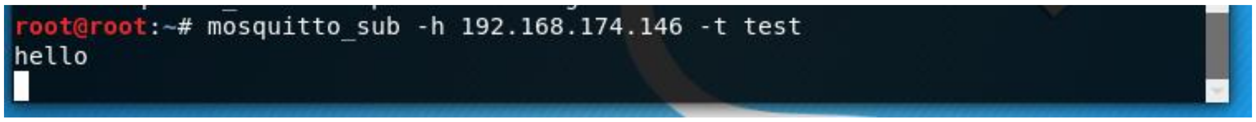
```
root@root:~# sudo apt-get install mosquitto
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libc-bin libc-dev-bin libc-l10n libc6 libc6-dbg libc6-dev libc6-i386
  libcrypt-dev libcrypt1 libev4 libwebsockets15 locales locales-all
  manpages-dev
Suggested packages:
  glibc-doc
The following NEW packages will be installed:
  libcrypt-dev libcrypt1 libev4 libwebsockets15 mosquitto
The following packages will be upgraded:
  libc-bin libc-dev-bin libc-l10n libc6 libc6-dbg libc6-dev libc6-i386 locales
  locales-all manpages-dev
10 upgraded, 5 newly installed, 0 to remove and 2014 not upgraded.
Need to get 188 kB/39.6 MB of archives.
After this operation, 1,323 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
```



```
The following packages will be upgraded:
  libmosquitto1 mosquitto-clients
2 upgraded, 0 newly installed, 0 to remove and 2025 not upgraded.
Need to get 176 kB of archives.
After this operation, 73.7 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://kali.download/kali kali-rolling/main amd64 mosquitto-clients amd64
1.6.8-1 [94.8 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 libmosquitto1 amd64 1.6.
8-1 [81.4 kB]
Fetched 176 kB in 1s (167 kB/s)
Reading changelogs... Done
(Reading database ... 374706 files and directories currently installed.)
Preparing to unpack .../mosquitto-clients_1.6.8-1_amd64.deb ...
Unpacking mosquitto-clients (1.6.8-1) over (1.6.7-1+b1) ...
Preparing to unpack .../libmosquitto1_1.6.8-1_amd64.deb ...
Unpacking libmosquitto1:amd64 (1.6.8-1) over (1.6.7-1+b1) ...
Processing triggers for libc-bin (2.29-9) ...
Processing triggers for man-db (2.8.5-1) ...
Setting up libmosquitto1:amd64 (1.6.8-1) ...
Setting up mosquitto-clients (1.6.8-1) ...
Processing triggers for libc-bin (2.29-9) ...
root@root:~# mosquitto_sub -u pi -h 192.168.174.146 -t test
```



```
pi@osboxes:~ $ mosquitto_pub -h 192.168.174.146 -t test -m hello
```



```
root@root:~# mosquitto_sub -h 192.168.174.146 -t test
hello
```

Constrained Application Protocol (CoAP)

Constrained Application Protocol (CoAP) is a specialized Internet Application Protocol for constrained devices, as defined in RFC 7252. It enables those constrained devices called "nodes" to communicate with the wider Internet using similar protocols. CoAP is designed for use between devices on the same constrained network (e.g., low-powered networks), between devices and general nodes on the Internet, and between devices on different constrained networks both joined by an internet. CoAP is also being used via other mechanisms, such as **SMS** on mobile communication networks.

CoAP is a **service layer protocol** that is intended for use in resource-constrained internet devices, such as wireless sensor network nodes. CoAP is designed to easily translate to HTTP for simplified integration with the web, while also meeting specialized requirements such as multicast support, very low overhead, and simplicity. Multicast, low overhead, and simplicity are extremely important for Internet of Things (IoT) and Machine-to-Machine (M2M) devices, which tend to be deeply embedded and have much less memory and power supply than traditional internet devices have. Therefore, efficiency is very important. CoAP can run on most devices that support UDP or a UDP analogue. There is one more protocol that works similar to CoAP protocol that is MQTT protocol. I'll make one more article for MQTT Protocol. Let's see how MQTT and CoAP are similar. MQTT is **publish/subscribe messaging protocol** designed for lightweight M2M communications. It was originally developed by IBM and is now an open standard. Both MQTT and CoAP:

1. Are open standards
2. Are better suited to constrained environments than HTTP
3. Provide mechanisms for asynchronous communication
4. Run on IP
5. Have a range of implementations.

MQTT and CoAP are both useful as IoT protocols, but have fundamental differences.

1. MQTT is a **many-to-many communication protocol** for passing messages between multiple clients through a central broker. It decouples producer and consumer by letting clients publish and having the broker decide where to route and copy messages. While MQTT has some **support for persistence**, it does best as a communications bus for live data.
2. CoAP is a **one-to-one protocol** for transferring state information between client and server. While it has support for observing resources, CoAP is best suited to a **state transfer model, not purely event based.**

3. MQTT clients make a **long-lived outgoing TCP connection** to a broker. This usually presents no problem for devices behind NAT. CoAP clients and servers both send and receive UDP packets.

4. MQTT provides no support for **labeling messages** with types or other metadata to help clients understand it. MQTT messages can be used for any purpose, but all clients must know the message formats up-front to allow communication. CoAP, conversely, provides inbuilt support for content negotiation and discovery allowing devices to probe each other to find ways of exchanging data.

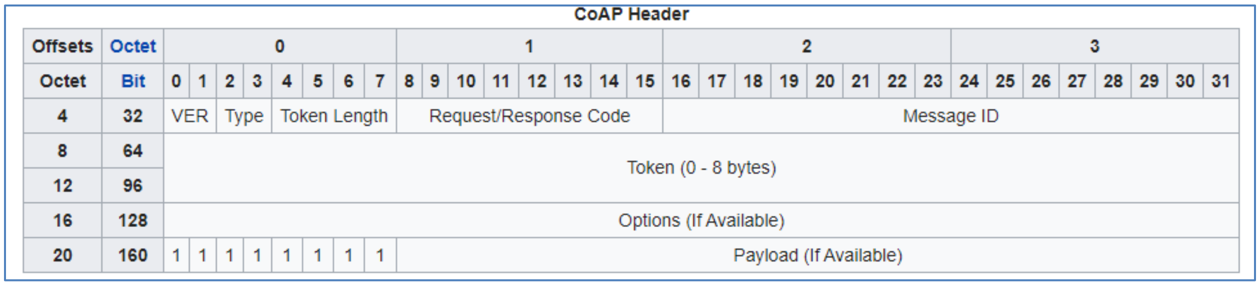
Architecture of CoAP

Like HTTP, CoAP is a document transfer protocol. Unlike HTTP, CoAP is designed for the needs of constrained devices. CoAP packets are much smaller than HTTP TCP flows. Bit fields and mappings from strings to integers are used extensively to save space. Packets are simple to generate and can be parsed in place without consuming extra RAM in constrained devices. CoAP runs over UDP, not TCP. Clients and servers communicate through connectionless datagrams. Retries and reordering are implemented in the application stack. Removing the need for TCP may allow full IP networking in small microcontrollers. CoAP allows UDP broadcast and multicast to be used for addressing. CoAP follows a client/server model. Clients make requests to servers, servers send back responses. Clients may GET, PUT, POST and DELETE resources. CoAP is designed to interoperate with HTTP and the RESTful web at large through simple proxies. Because CoAP is datagram based, it may be used on top of SMS and other packet based communications protocols.

Message formats

The smallest CoAP message is 4 bytes in length, if omitting Token, Options and Payload. CoAP makes use of two message types, requests and responses, using a simple, binary, base header format. The base header may be followed by options in an optimized Type-Length-Value format. CoAP is by default bound to UDP and optionally to DTLS, providing a high level of communications security.

Any bytes after the headers in the packet are considered the message body. The length of the message body is implied by the datagram length. When bound to UDP, the entire message MUST fit within a single datagram.



CoAP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
4	32	VER		Type		Token Length				Request/Response Code								Message ID															
8	64	Token (0 - 8 bytes)																															
12	96																																
16	128	Options (If Available)																															
20	160	1	1	1	1	1	1	1	1	Payload (If Available)																							

The first 4 bytes are mandatory in all CoAP datagrams.

Version (VER) (2 bits)

Indicates the CoAP version number.

Type (2 bits)

This describes the datagram's message type for the two message type context of Request and Response.

Request

0: Confirmable: This message expects a corresponding Acknowledgement message.

1: Non-confirmable: This message does not expect a confirmation message.

Response

2: Acknowledgement: This message is a response that acknowledge a confirmable message

3: Reset: This message indicates that it had received a message but could not process it.

Token Length (4 bits)

Indicates the length of the variable-length Token field, which may be 0-8 bytes in length.

Request/Response Code (8 bits)

0	1	2	3	4	5	6	7
Class				Code			

The three most significant bits form a number known as the "class", which is analogous to the class of HTTP status codes. The five least significant bits form a code that communicates further detail about the request or response. The entire code is typically communicated in the form class.code.

Message ID (16 bits)

Used to detect message duplication and to match messages of type Acknowledgement/Reset to messages of type Confirmable/Non-confirmable.:Response messages will have the same Message ID as request.

Token

Optional field whose size is indicated by the Token Length field, whose values is generated by the client. The server must echo every token value without any modification back to the client. It is intended for use as a client-local identifier to provide extra context for certain concurrent transactions.

Option Delta:

- 0 to 12: For delta between 0 to 12 : Small delta between the last option id and the desired option id
- 13: For delta from 13 to 268 : Option Delta Extended is 8bit that is the Option Delta value minus 13
- 14: For delta from 269 to 65,804 : Option Delta Extended is 16bit that is the Option Delta value minus 269
- 15: Reserved for Payload Marker, where the Options Delta/length are set together as 0xFF.

Option Length:

- 0 to 12: For Option Length between 0 to 12 : Small Option Length between the last option id and the desired option id
- 13: For Option Length from 13 to 268 : Option Length Extended is 8bit that is the Option Length value minus 13
- 14: For Option Length from 269 to 65,804 : Option Length Extended is 16bit that is the Option Length value minus 269
- 15: Reserved for future use. Is an error if Option Delta field is set to 0xFF.

Option Value:

- Size of Option Value field is defined by Option Length value in bytes.
- Semantic and format this field depends on the respective option.

Payload:

The payload is the part of transmitted data that is the actual intended message. Headers and metadata are sent only to enable payload delivery.

Security issues

CoAP uses **DTLS (Datagram Transport Layer Security)** as a secure protocol and **UDP (User Datagram Protocol)** is used as a transfer protocols. Therefore, the attacks on UDP or DTLS could actually cause CoAP attack. Most of the DTLS attacks can be carried out in a single session and strong authenticated encryption algorithm is needed. **MITM (Man in the middle)** is one the CoAP attacks, phishing and sniffing could be considered as CoAP attack.

CoAP was designed as a lightweight **machine-to-machine (M2M) protocol** that can run on smart devices where memory and computing resources are scarce. In a very simplistic explanation, CoAP is very similar to HTTP, but instead of working on top of TCP packets, it works on top of UDP, a lighter data transfer format created as a TCP alternative.

Just like **HTTP** is used to transport data and commands (GET, POST, CONNECT) between a client and a server, CoAP also allows the same multicast and command transmission features, but without needing the same amount of resources, making it ideal for today's rising wave of Internet of Things (IoT) devices. But just like any other UDP-based protocol, CoAP is inherently susceptible to IP address spoofing and packet amplification, the two major factors that enable the amplification of a DDoS attack. An attacker can send a small UDP packet to a CoAP client (an IoT device), and the client would respond with a much larger packet. In the world of DDoS attacks, the size of this packet response is known as an amplification factor, and for CoAP, this can range from 10 to 50, depending on the initial packet and the resulting response.

References

- https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html
- "MQTT 3.1.1 specification". OASIS. December 10, 2015. Retrieved April 25, 2017.
- https://en.wikipedia.org/wiki/MQTT
- "IBM MQ". IBM. Retrieved November 18, 2013.
- http://mqtt.org/
- https://en.wikipedia.org/wiki/Constrained_Application_Protocol
- https://coap.technology/
- https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php