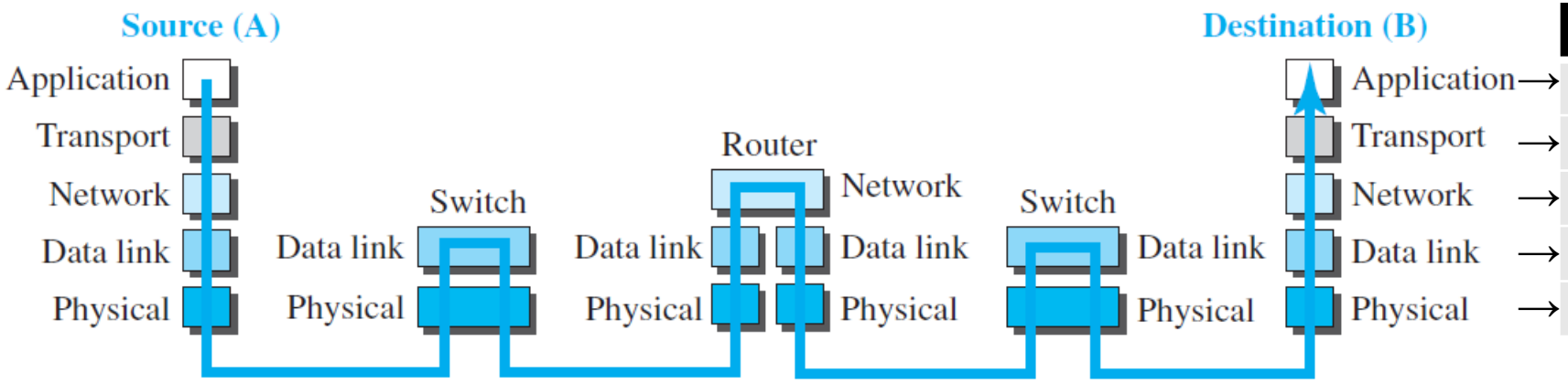
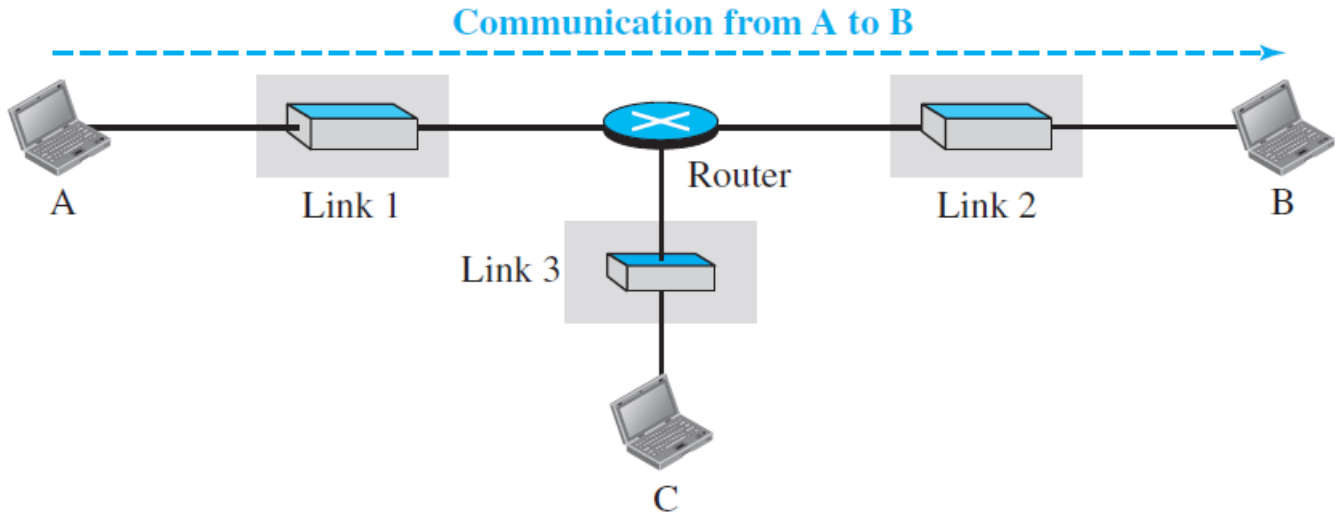
Lecture 4

Messaging Protocols

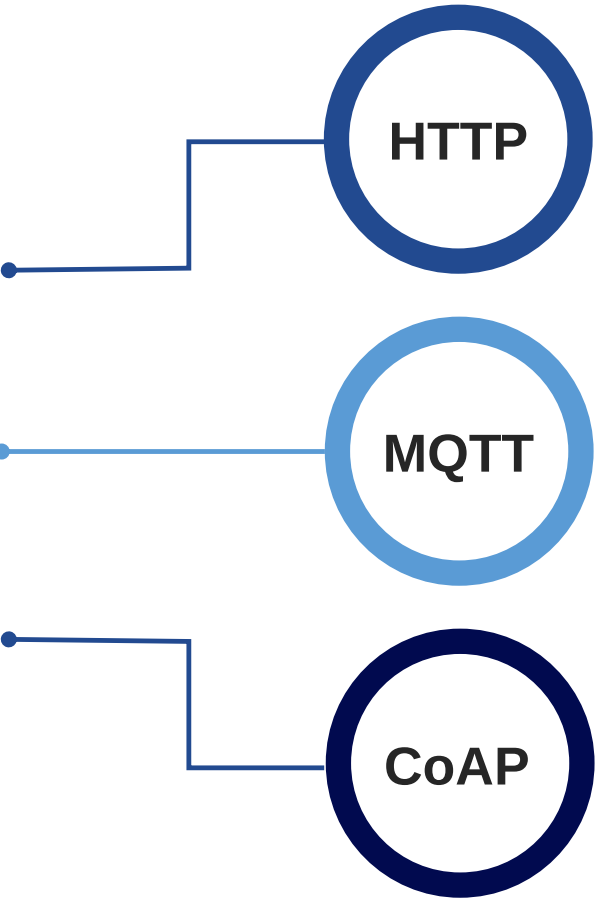
TCP/IP Protocol Suite



Protocol	Data Unit	Addressing
HTTP, FTP, SNMP, SMTP	Message	n/A
TCP/UDP	Segment	Port #'s
IP	Datagram	IP address
Ethernet, Wi-Fi	Frames	MAC Address
10 Base T, 802.11	Bits	N/A



Application Protocols



Hypertext Transfer Protocol

is an application layer protocol in the Internet protocol suite model for distributed, collaborative, hypermedia information systems.

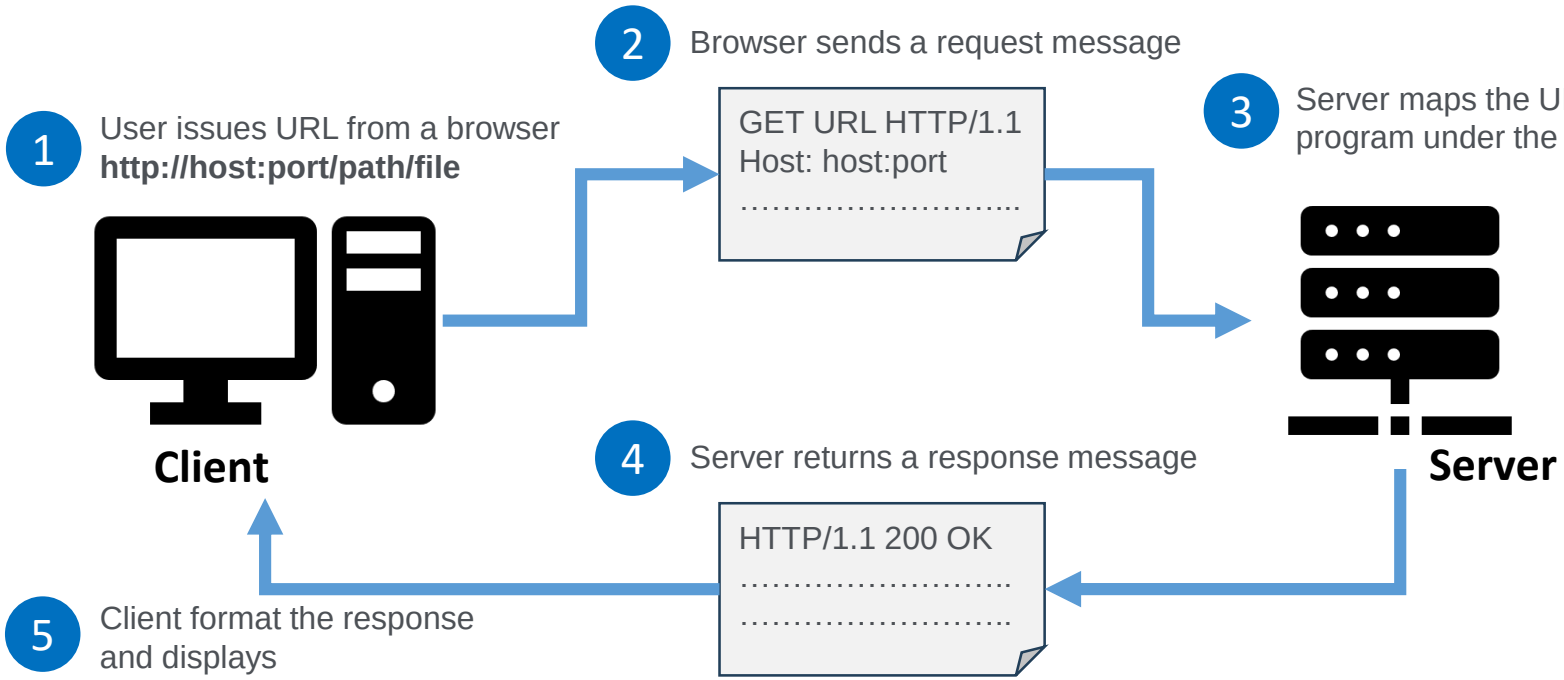
Message Queue Telemetry Transport Protocol

is a lightweight, publish-subscribe, machine to machine network protocol designed for connections with remote locations that have devices with resource constraints or limited network bandwidth.

Constrained Application Protocol

is a specialized Internet application protocol for constrained devices,

HTTP



❑ **Connectionless**
After receiving the response, the client disconnects the connection.

❑ **Media independent**
Any type of data can be sent by HTTP as long as both the client and the server know how to handle the data content.

❑ **Stateless**
The server and client are aware of each other only during a current request.

HTTP Request

GET

Retrieve data from server

GET /spot-gold.html HTTP/1.1
Host: goldprice.org

POST

Add data to an existing resource

POST /spot-gold.html HTTP/1.1
Host: goldprice.org

PUT

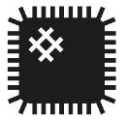
Update an existing resource on the server

PUT /spot-gold.html HTTP/1.1
Host: goldprice.org

DELETE

Delete data from server

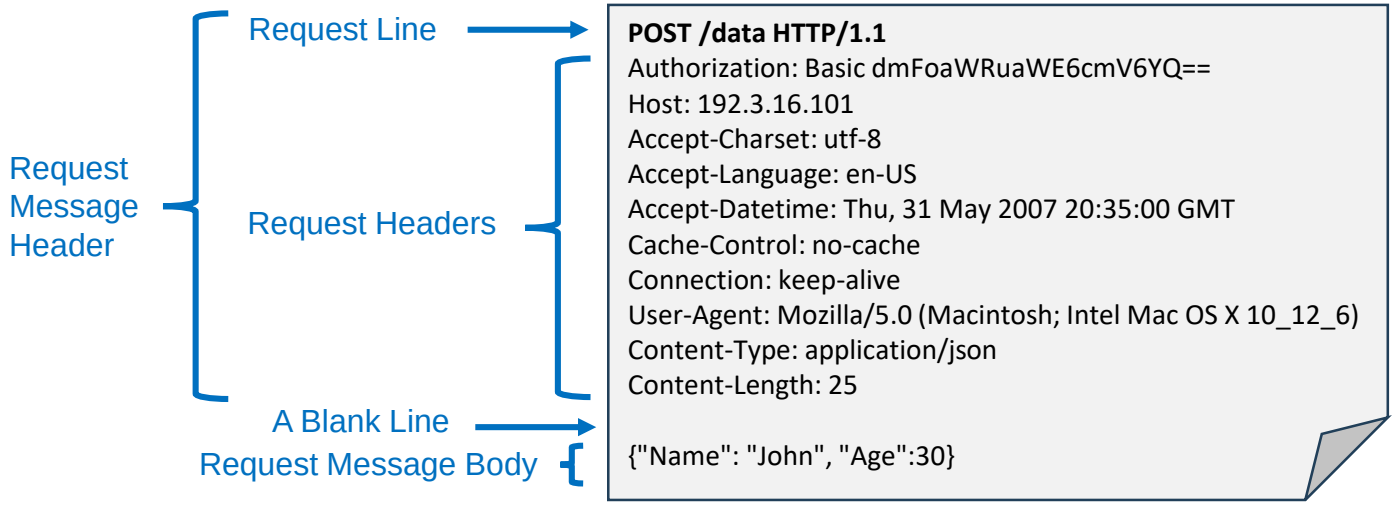
DELETE /spot-gold.html HTTP/1.1
Host: goldprice.org



PATCH

Partially update a resource

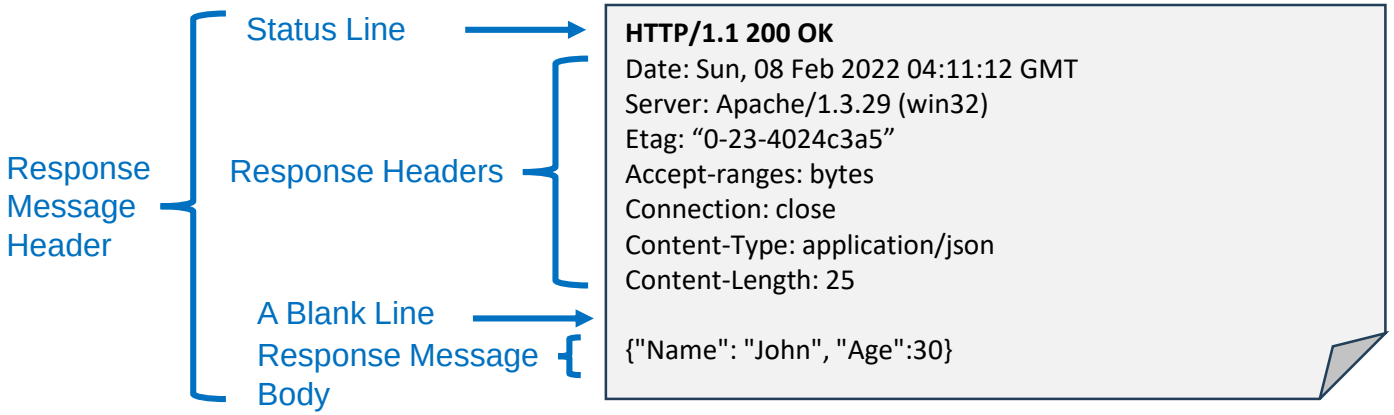
PATCH /spot-gold.html HTTP/1.1
Host: goldprice.org



Request line: *Request-method* *Request-URL* *HTTP-version*

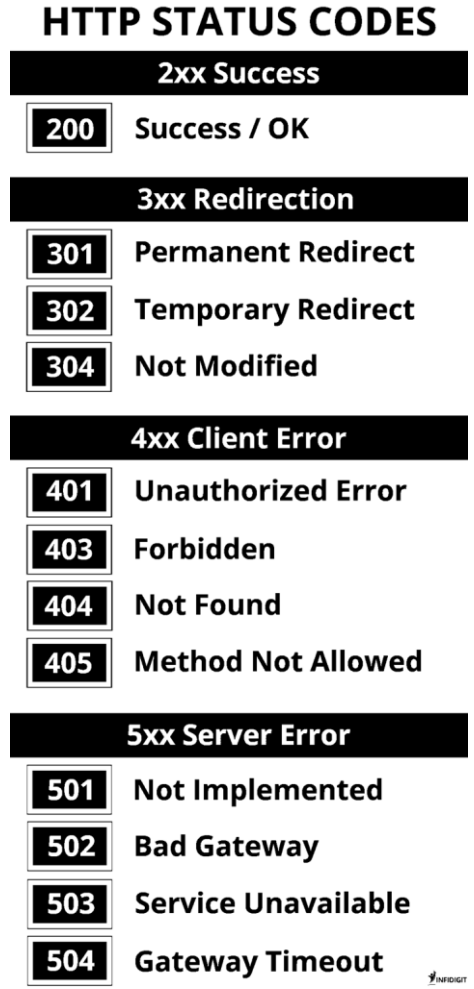
Request Headers: *Key: Value* pairs

HTTP Response

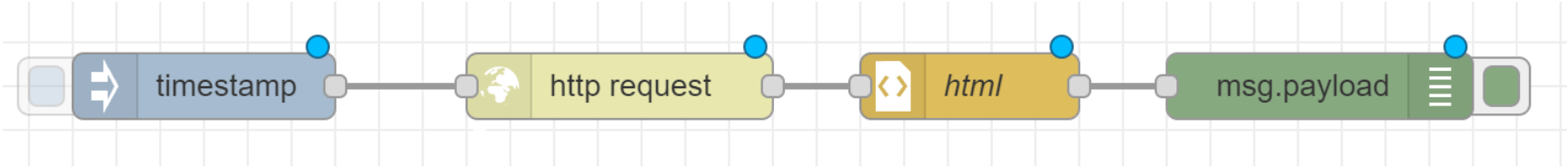


Status line: *HTTP-version* *Status-code* *Reason-phrase*

Response Headers: *Key: Value* pairs



HTTP Request

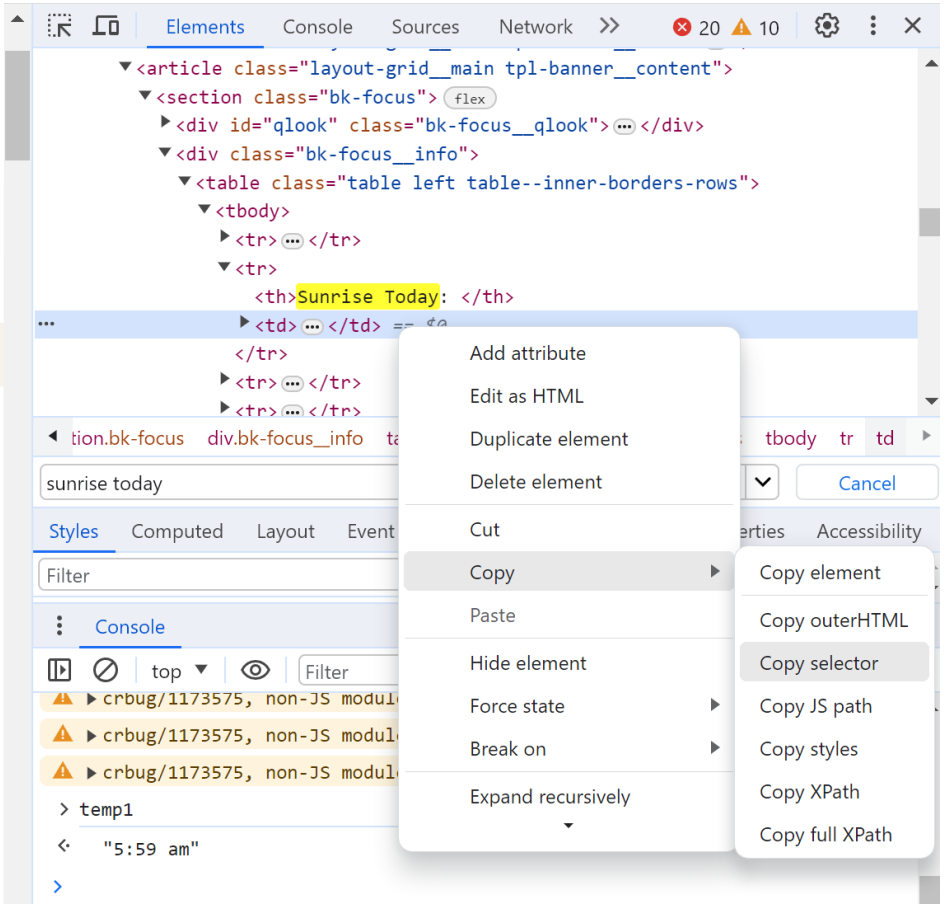


Time/General | Weather | Time Zone | DST Changes | Sun & Moon

Sun & Moon Today | Sunrise & Sunset | Moonrise & Moonset | Moon Phases | Eclipses | Night Sky

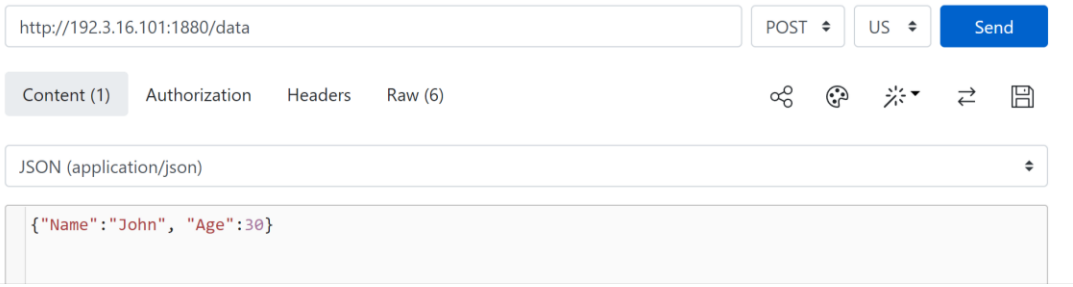
Moon: 91.0%
Waning Gibbous

Current Time:	Oct 2, 2023 at 3:33:50 am
Sunrise Today:	5:59 am → 94° East
Sunset Today:	5:47 pm ← 266° West
Moonrise Today:	7:35 pm ↗ 64° Northeast
Moonset Today:	9:11 am ↖ 293° Northwest
Daylight Hours:	11 hours, 48 minutes (-2m 13s)

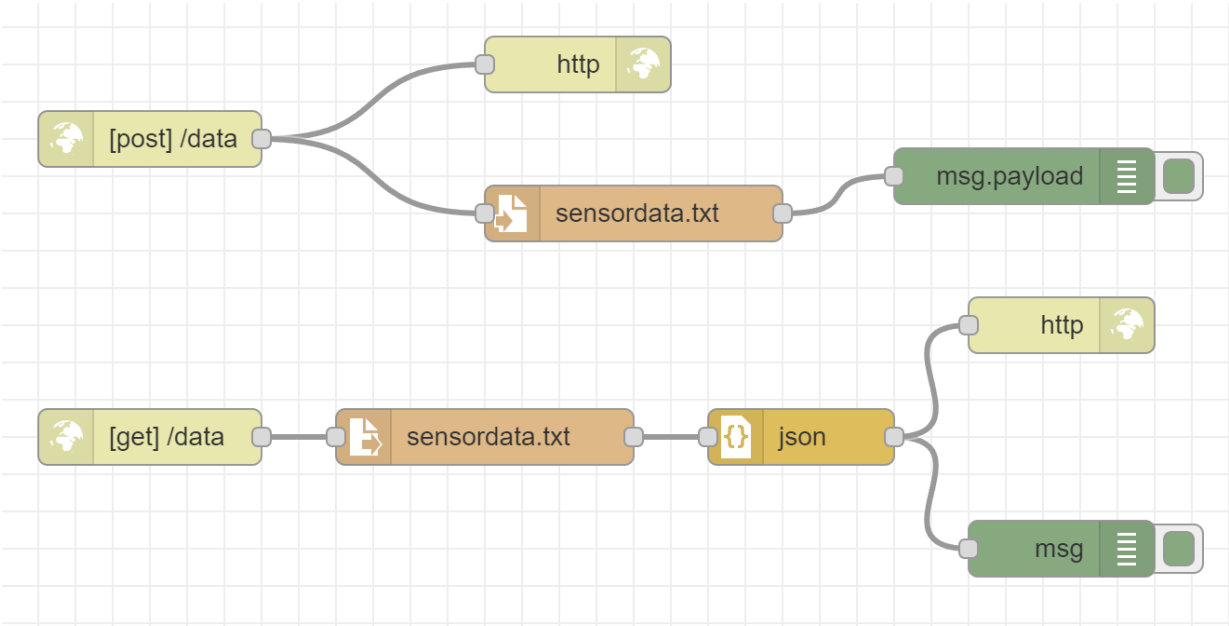


HTTP Client-Server

HTTP Client –reqbin.com



HTTP Server



HTTP Client-Server

HTTP Client – Smartphone sensor

Google Play

Games Apps Movies & TV Books

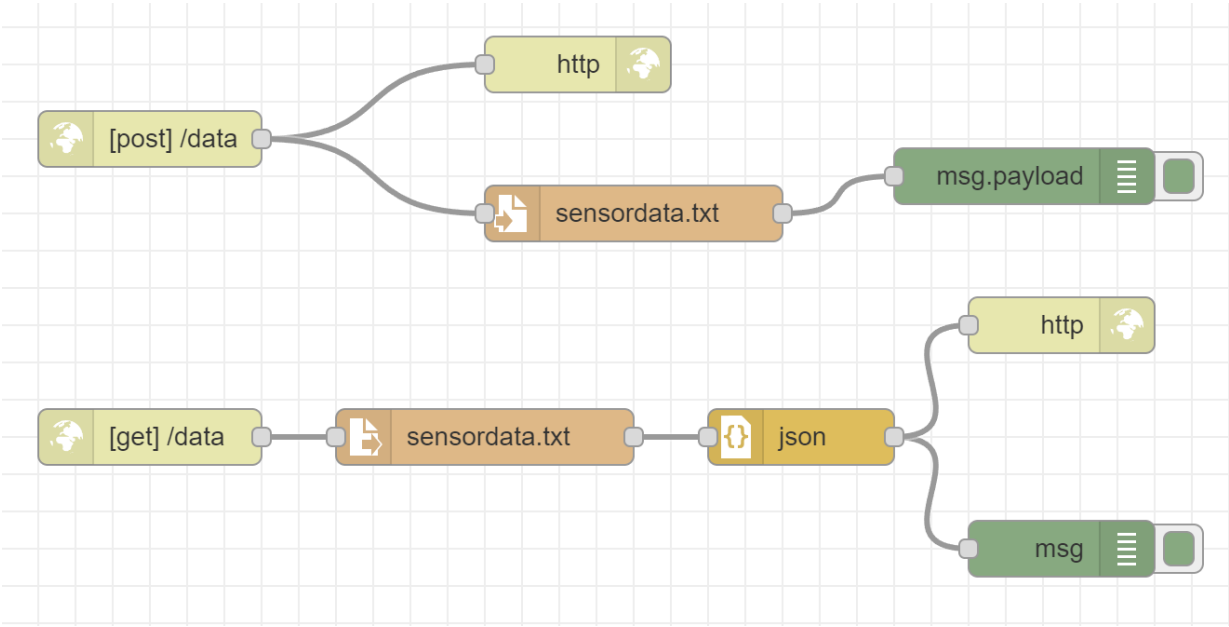
Sensor Logger

Kelvin Tsz Hei Choi

In-app purchases

4.4★ 216 reviews | 10K+ Downloads | Everyone

HTTP Server

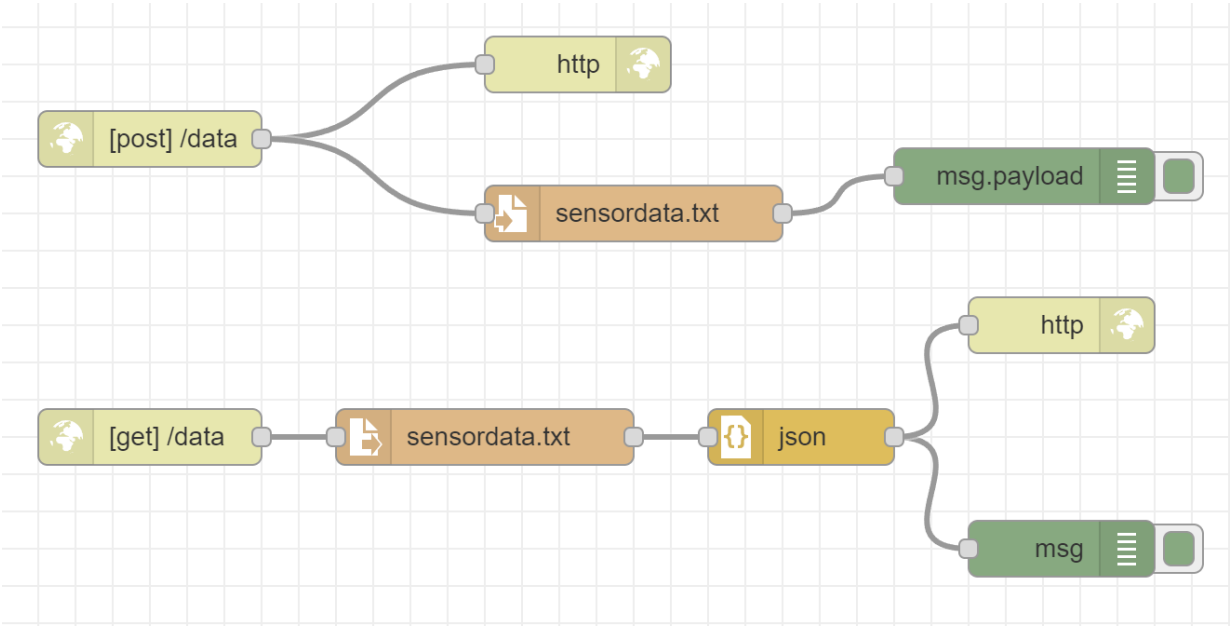


HTTP Client-Server

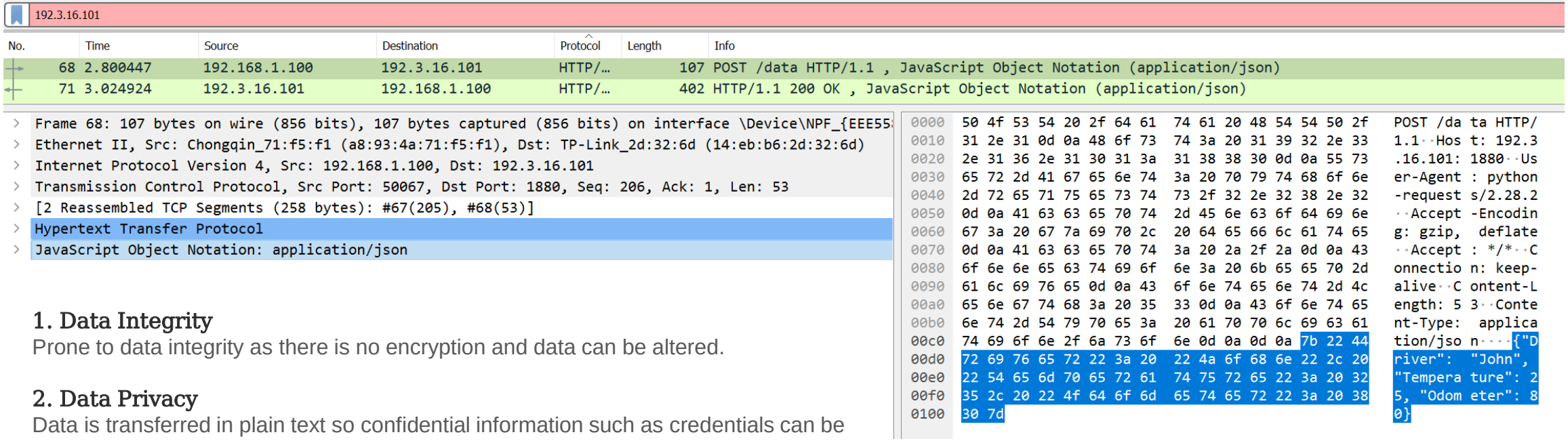
HTTP Client - Post Using Python

```
import requests

Temp=25
Speed=80
url = 'http://192.3.16.101:1880/data'
data = {'Driver': 'John', 'Temperature': 25, 'Odometer':
80}
response = requests.post(url, data)
print(response.status_code)
print(response.text)
```

HTTP Server



HTTP Drawbacks



1. Data Integrity
Prone to data integrity as there is no encryption and data can be altered.

2. Data Privacy
Data is transferred in plain text so confidential information such as credentials can be viewed.

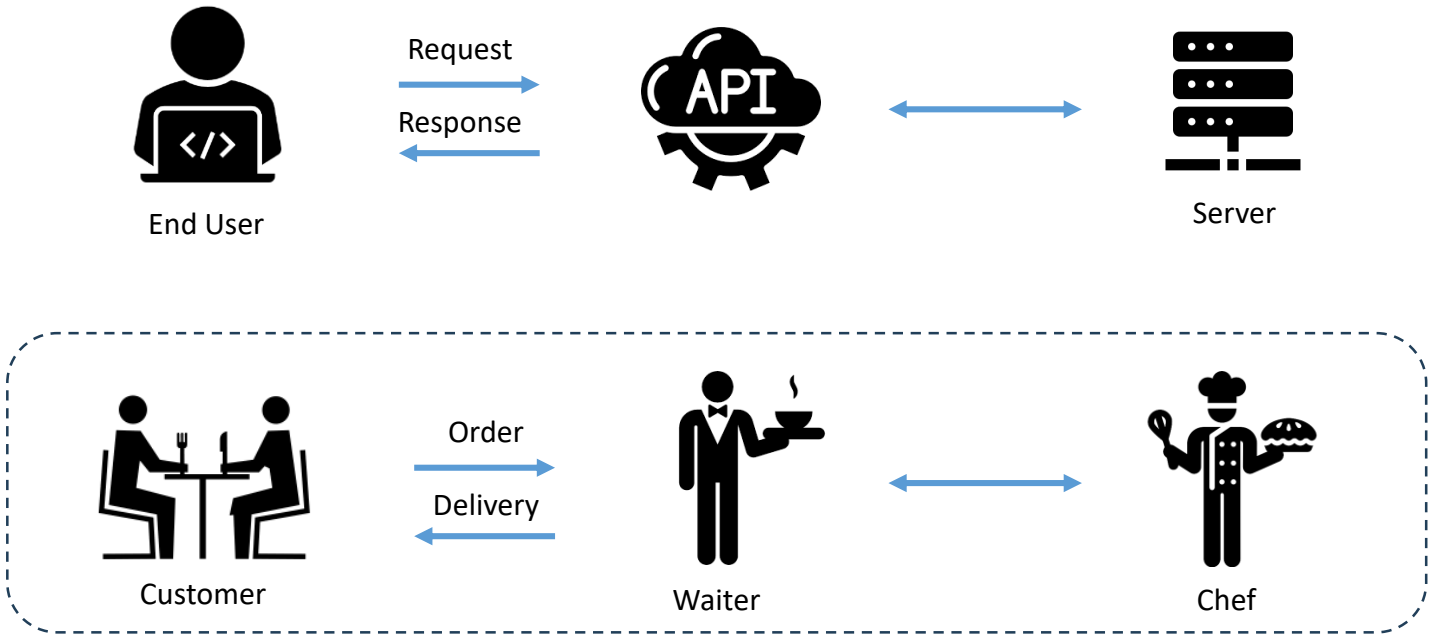
3. Server Availability
Clients cannot terminate the connection even if they received all data they need.
Therefore, during this time period, server will not be present.

4. Administrative Overhead
Heavy protocol with many headers.

5. IoT Device Support
More power consumption so not suitable for IoT devices.

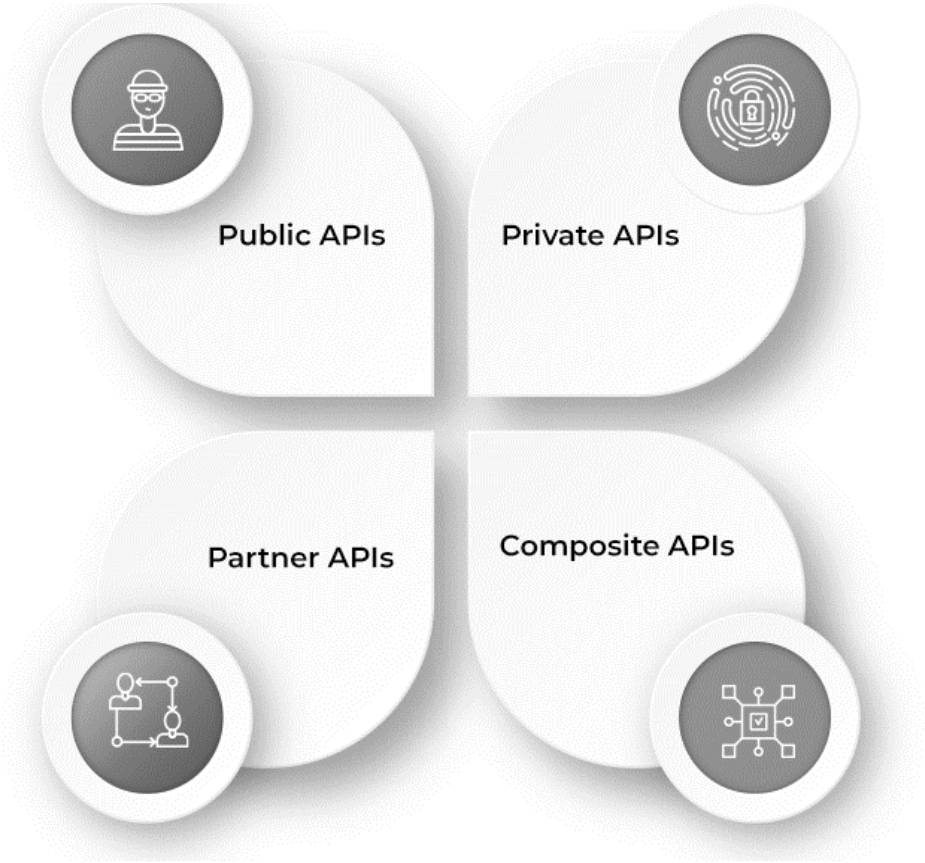
Application Programming Interface

- An **API** is a communication interface that allows two separate software components to share data.
- An API operates as a bridge between internal and external software operations, allowing for a fluid interchange of data that frequently passes undetected by the end-user.
- Through APIs, accessing structured data in JSON or XML format is facilitated while also guaranteeing security and accessibility to resources.



Types of APIs

Public APIs
- Open source and disseminated for general use.
- Can be accessed using the HTTP protocol.
- Allow users to request information from any enterprise that provides the interface (e.g., Google Maps API)



Private APIs
- Exist within a software vendor's system framework.
- Aim to bolster communication and boost productivity.
- Used to privately transmit data among internal business applications such as ERP, financial systems, or CRM.

Partner APIs
- Allow two different companies to enter into an exclusive data-sharing agreement.
- Vendors gain access to the data streams of partner companies. In return, the company granting access to its data receives added services or system features.
- Developers can normally access these partner interfaces in self-service mode using an open API dev portal.

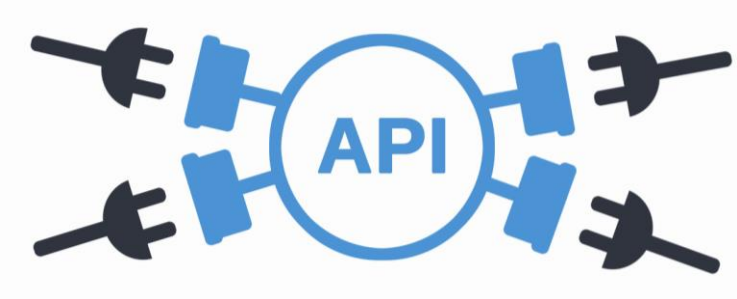
Composite APIs
- Combine different service or data APIs.
- Enables dev teams to access multiple endpoints by raising a single call.
- Often seen in microservices architectures, where data from more than one source is frequently needed to complete a given task.
- Compile multiple calls sequentially and create a single API request.

API Protocols

Representational State Transfer (REST)

More flexible than SOAP as it transmits data in JSON, HTML, media files and plain text. It Can only use HTTP and HTTPS to communicate.

- Implementation is flexible, scalable, and lightweight.
- Stateless communication: Forbids the storage of client data between GET requests.
- Caching: Enables web browsers to store the request response received locally and access it periodically to enhance efficiency.



Google Remote Procedure Calls (gRPC)

An open-source RPC architecture that can operate in numerous environments.

- The transport layer primarily relies on HTTP.
- Developers can specify custom functions that allow for flexible inter-service communication.
- Offers extra features such as timeouts, authentication, and flow control.
- Data is transmitted in protocol buffers, a platform and language-agnostic mechanism that allows for data to be structured intuitively.

Graph query language (GraphQL)

Is a database query language, and a server-side runtime for APIs developed at Facebook. By design, this protocol prioritizes giving users the exact data requested–no less, no more.

- Developer-friendly and supports the creation of fast and flexible APIs, including composite APIs.
- Can be used as an alternative for REST.

Simple Object Access Protocol (SOAP)

Uses XML files to transmit data over an HTTP, SMTP, TCP, and UDP. Since XML is difficult to debug, SOAP is less popular for modern-day API requirements.

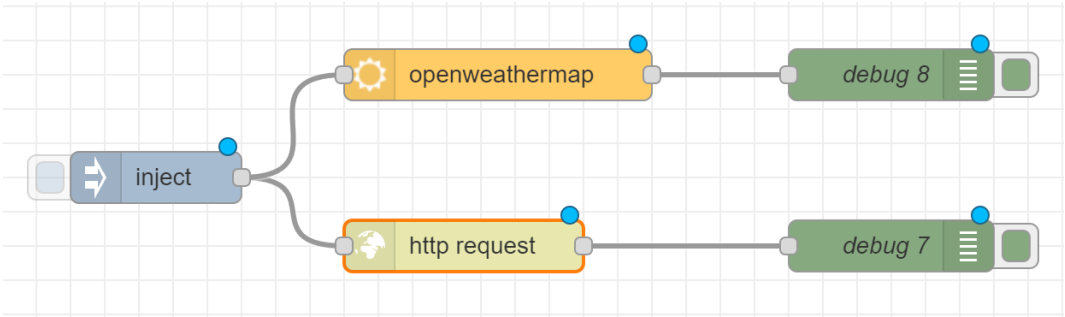
REST API

OpenWeather

API call

```
https://api.openweathermap.org/data/3.0/onecall?lat=
{lat}&lon={lon}&exclude={part}&appid={API key}
```

Key	Name	Status	Actions
44f33cb49f03b290c74bef5a2894cd4a	MyAPI	Active	




```
{
   "lat":33.44,
   "lon":-94.04,
   "timezone":"America/Chicago",
   "timezone_offset":-18000,
   "current":{
      "dt":1684929490,
      "sunrise":1684926645,
      "sunset":1684977332,
      "temp":292.55,
      "feels_like":292.87,
      "pressure":1014,
      "humidity":89,
      "dew_point":290.69,
      "uvi":0.16,
      "clouds":53,
      "visibility":10000,
      "wind_speed":3.13,
      "wind_deg":93,
      "wind_gust":6.71,
      "weather":[
         {
            "id":803,
            "main":"Clouds",
            "description":"broken clouds",
            "icon":"04d"
         }
```

Mini Project

Smart E-bike

1. The e-bike sends its current location (latitude & longitude) and speed to the IoT platform.

2. OpenWeather API is called to get the temperature chance of precipitation and wind speed and etc. of the location.

3. If the chance of precipitation higher than 30% and wind speed is greater than 20 km/h an alarm is sent to the cyclist.

4. Also, if it is too hot, the maximum power of the motor is limited according to the temperature.

Mini Project

Rule-based outfit recommendation system

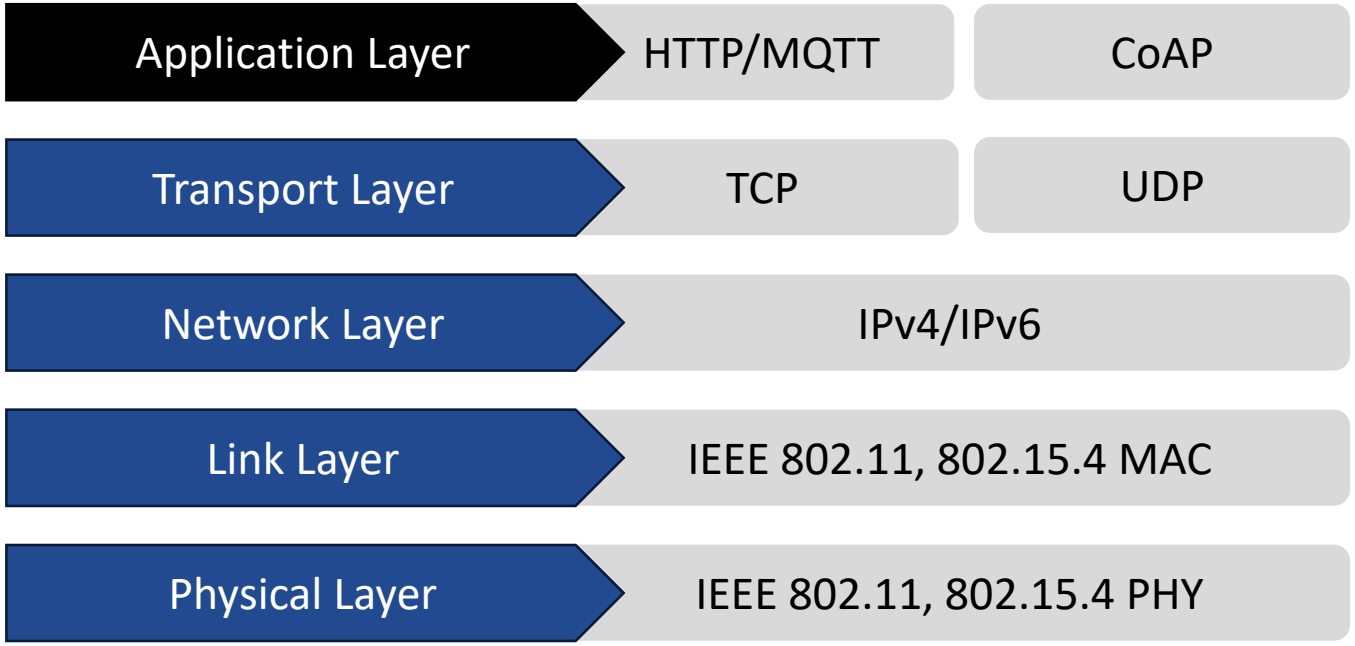
1. IoT device sends the latitude & longitude of the current location.

2. OpenWeather API is called to get the UV index, Temperature, chance of precipitation, wind speed and etc. of the location.

3. The recommendation are sent to the IoT device to pick the suitable outwear, sunglasses, umbrella, etc.

IoT Protocol Stack

To fully address constrained devices and networks, optimized IoT protocols are required.



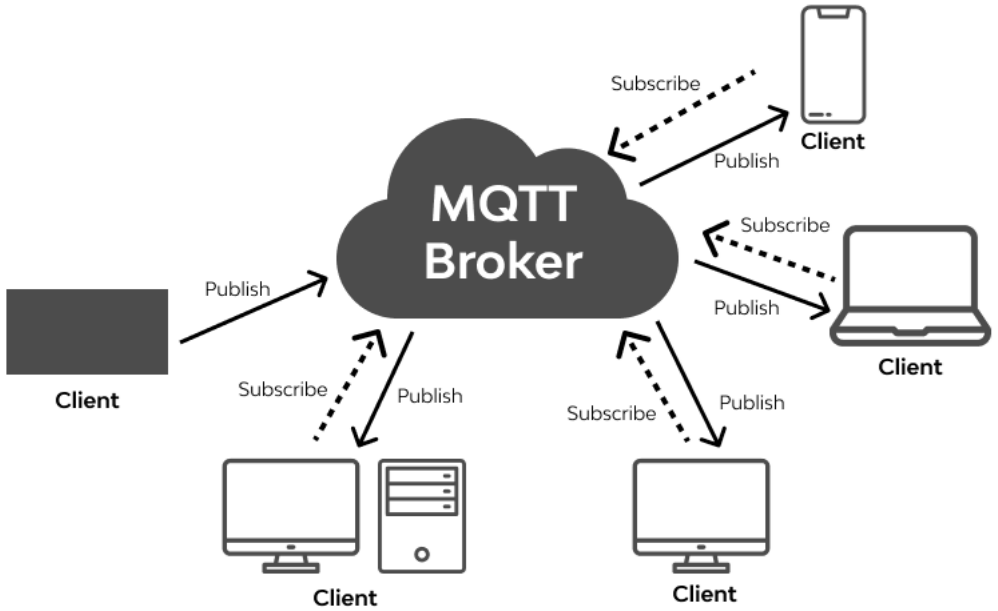
MQTT

MQTT is a lightweight and scalable publish/subscribe IoT messaging protocol that can be easily implemented to relay data through a central server called the broker.

❑ **Space decoupling**
Publisher and subscriber do not need to know each other (for example, no exchange of IP address and port).

❑ **Time decoupling**
Publisher and subscriber do not need to run at the same time.

❑ **Synchronization decoupling**
Operations on both components do not need to be interrupted during publishing or receiving.



MQTT vs. HTTP

Faster
Less overhead
Less power consumption

Slower
More overhead
More power consumption

In MQTT a client does not have to pull the information it needs, but if there is new data to be sent, the server (broker) pushes the information to the client.

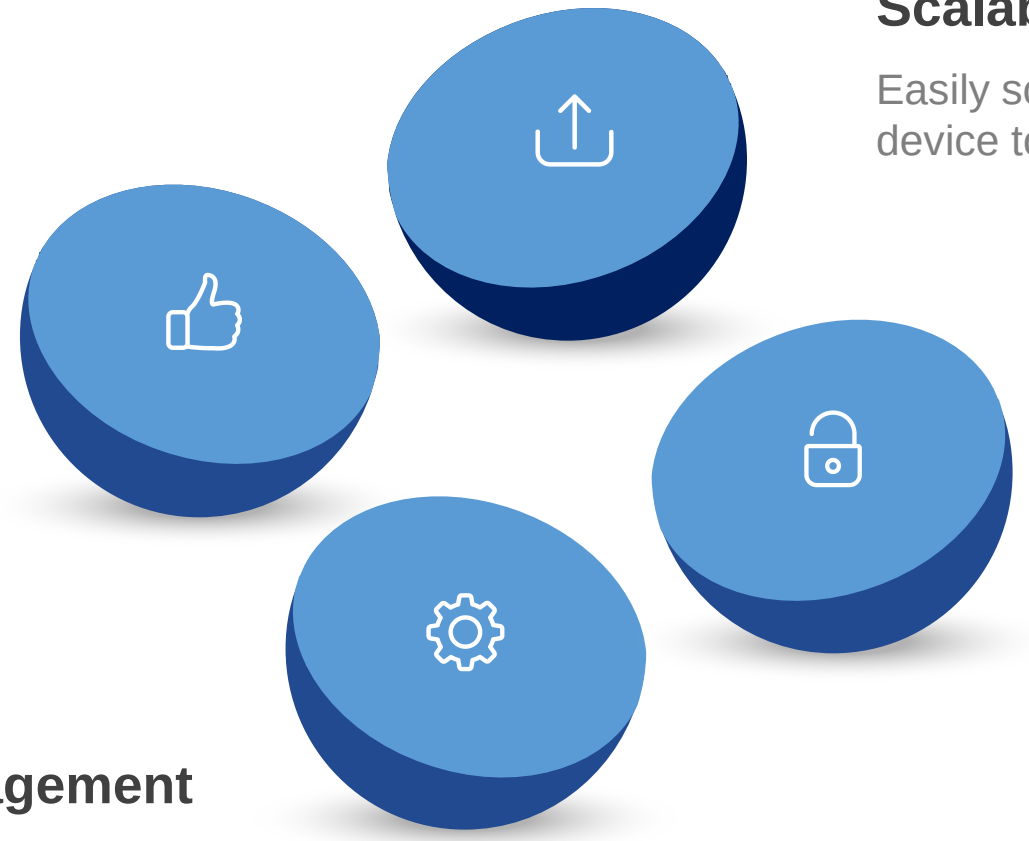
MQTT Advantages

Scalability

Easily scales from a single device to thousands

Optimization

Reduces network strain on tenuous networks such as cellular or satellite



Security

Eliminates vulnerable and insecure connections

Management

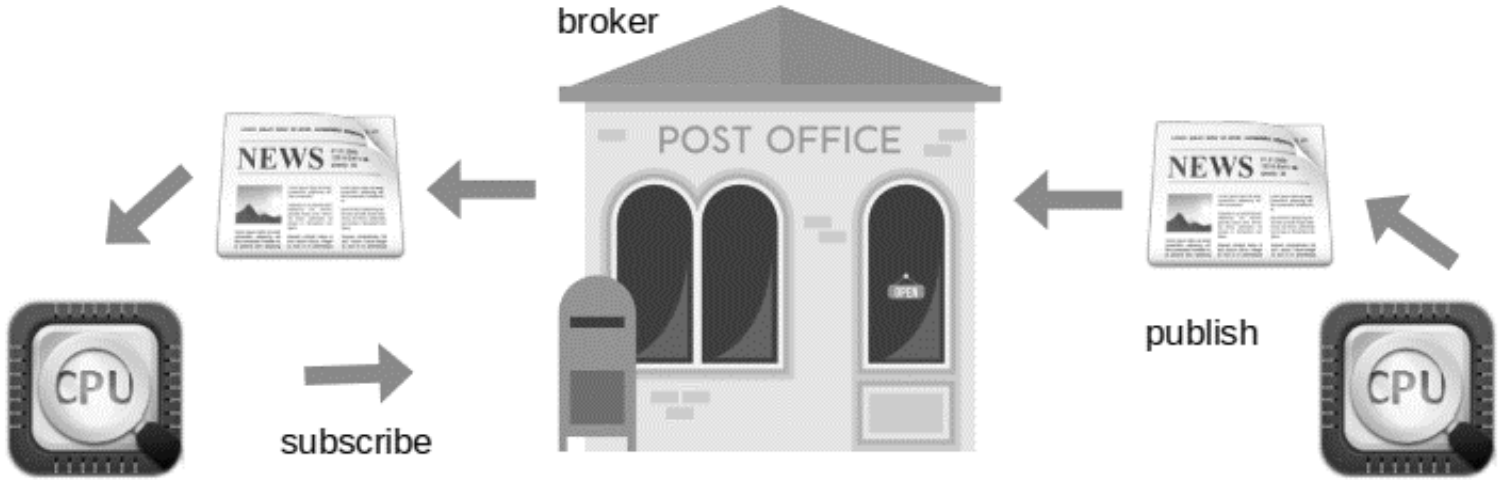
Manages client connect states including credentials, security, and certificates

MQTT Pub/Sub

Topic

- MQTT uses subject line called “topic” instead of addresses.
- Everyone who needs a copy of that message subscribes to that topic.
- The broker handles all devices by using topics to manage the messages.
- A topic is a simple string that can have more hierarchy levels, which are separated by a slash.
- Topics are case sensitive.

Home/office/lamp ≠ home/office/LAMP



MQTT Pub/Sub

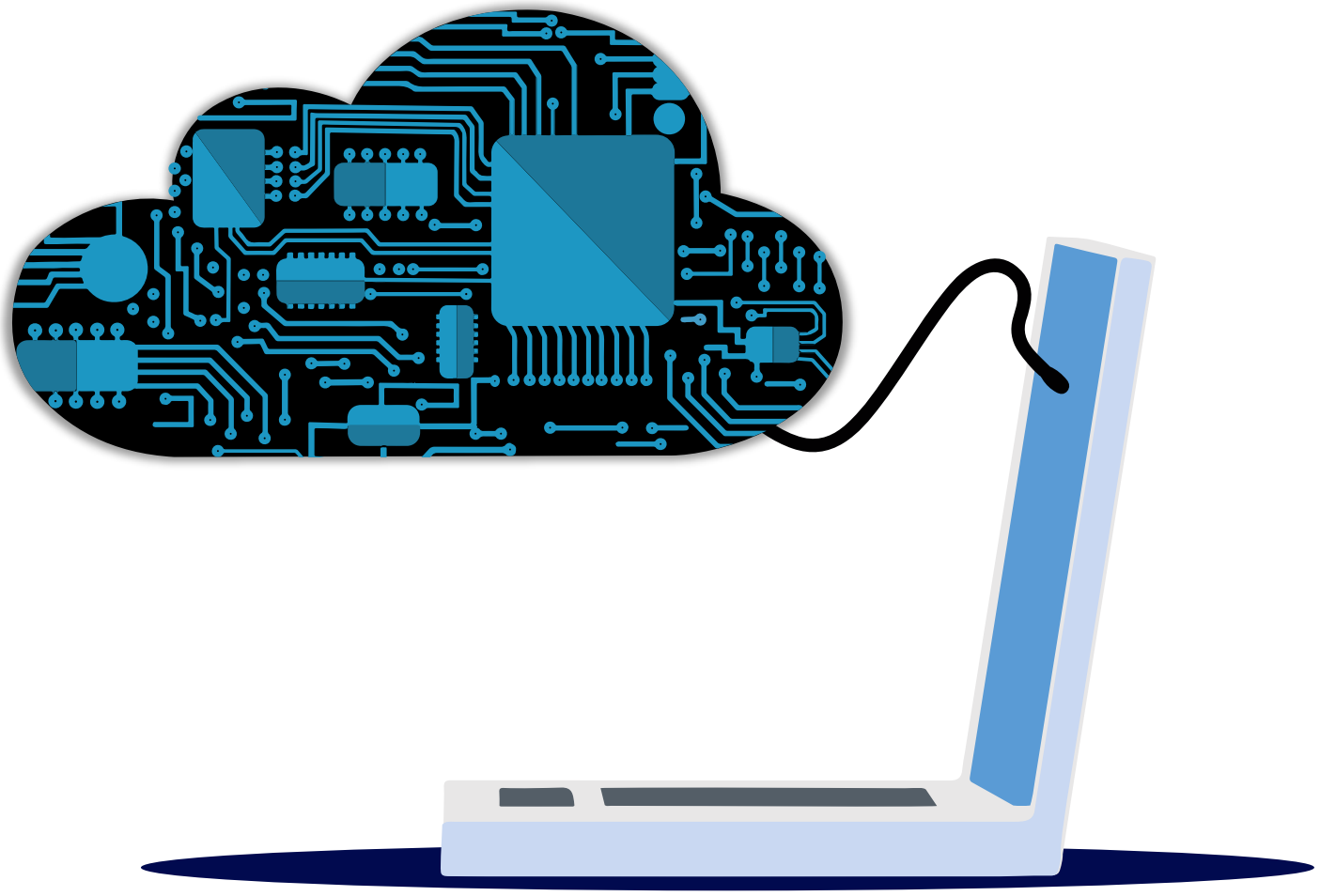
Features

- New devices can be easily added without touching the existing infrastructure
- Multiple publishers and multiple subscribers.
- Since the new devices only communicate with the broker, they do not need to be compatible with the other devices.
- A many-to-many model multiple devices can be publishing the same topic and have one or multiple subscribers.

Bidirectional Communication

Each client can both produce and consume data by publishing messages and subscribing to topics.

The IoT devices can send sensor data and at the same time receive configuration information and control commands.

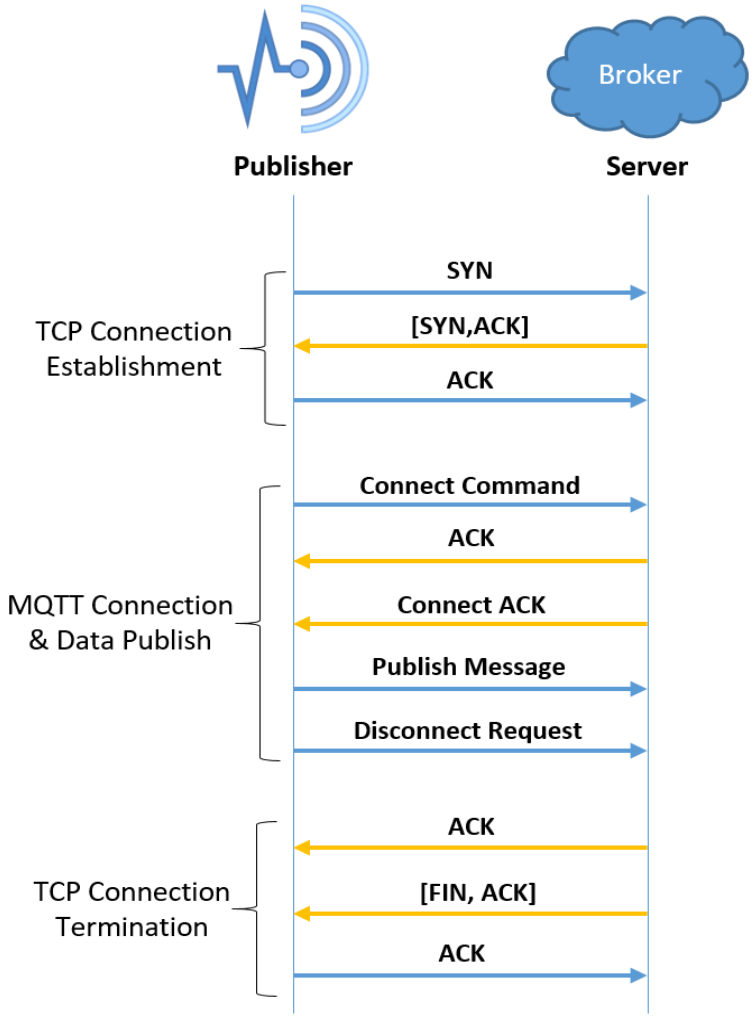


MQTT Connection

Publisher-broker handshake

MQTT is a TCP/IP-based protocol where a TCP connection needs to be established prior to transferring the data.

- TCP Connection Establishment
- MQTT Connection Establishment and Data publish
- TCP Connection Termination



MQTT Broker

MQTT brokers help implement the publish-subscribe communication model between devices and applications. The MQTT broker also helps implement rules and filters that help make the communications efficient and secure.

Currently, an MQTT broker is available in the following variants:

Open Source

- HiveMQ CE
- Mosquitto
- VerneMQ

Commercial

- HiveMQ Professional and Enterprise
- EMQ
- VerneMQ

Cloud (Managed)

- HiveMQ Cloud
- CloudMQTT
- AWS IoT Core
- Azure IoT Hub

General-purpose

- Solace PubSub+
- IBM MQ
- RabbitMQ
- ActiveMQ

mosquitto

MOSCA

EMQ

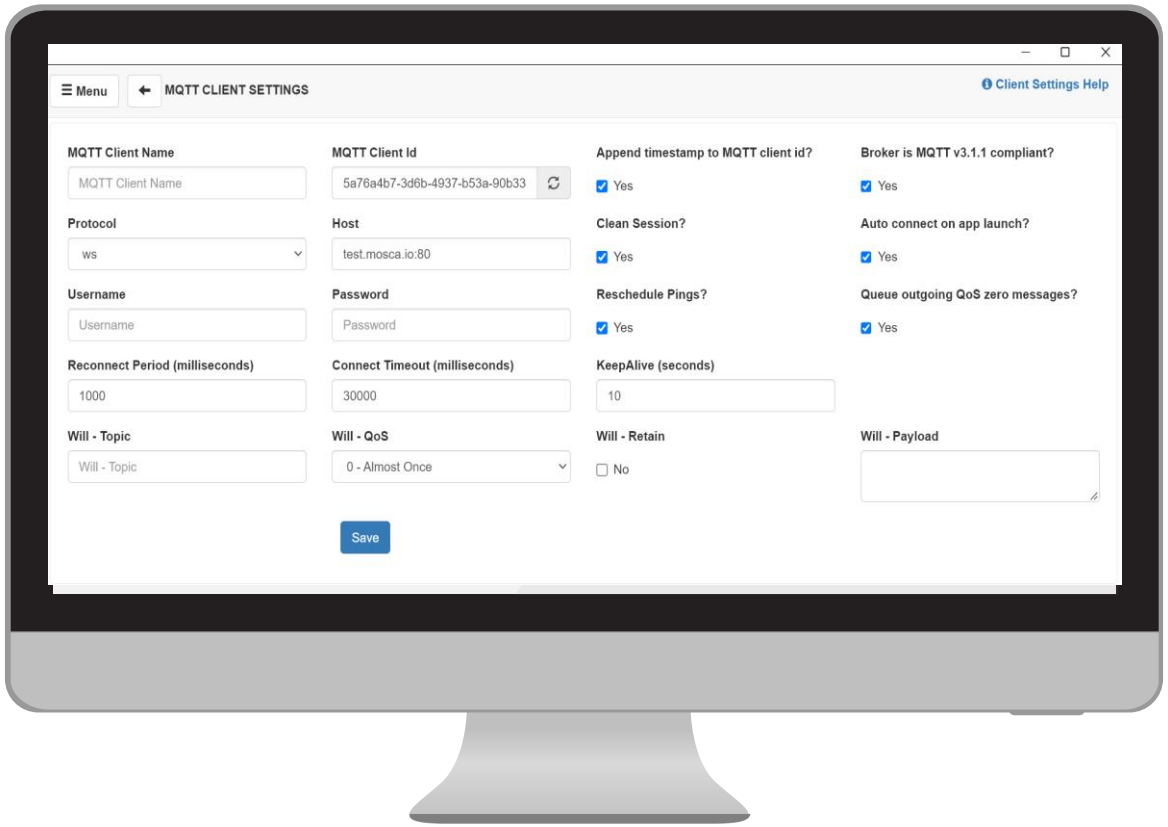
HIVEMQ

IBM MQ

MQTT Broker

Download **MQTTBox** app
from Microsoft Store

OR **MQTT.fx** app from Softblade

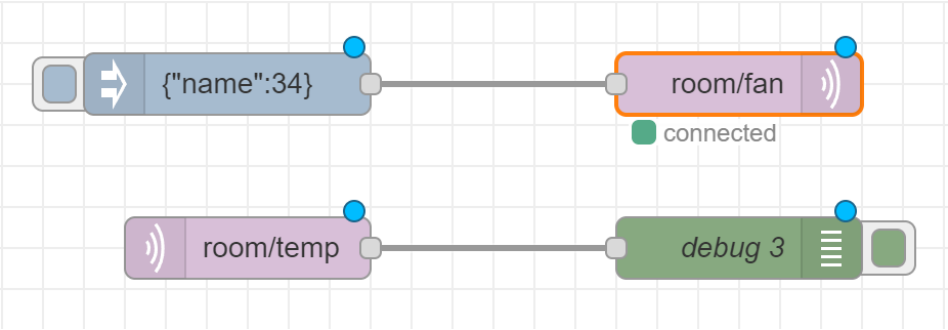


Connect to public MQTT broker:
broker.hivemq.com:1883

MQTT Broker

Connect to public MQTT broker:

broker.hivemq.com:1883

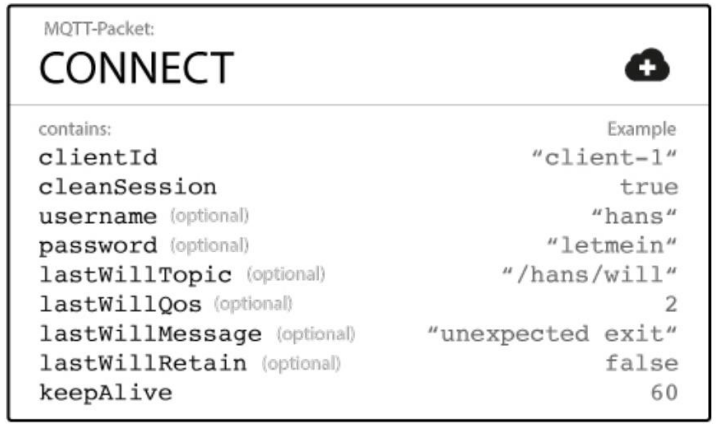



```python
import random
import json
import time
from paho.mqtt import client as mqtt_client

broker = 'broker.hivemq.com'
port = 1883
topic = "room/temp"
# Generate a Client ID with the publish prefix.
client_id = f'publish-{random.randint(0, 1000)}'

def connect_mqtt():
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT Broker!")
        else:
            print("Failed to connect, return code %d\n", rc)

    client = mqtt_client.Client(client_id)
    # client.username_pw_set(username, password)
    client.on_connect = on_connect
    client.on_message = on_message  # attach function to callback
    #client.will_set("room/lights", payload="Off", qos=0, retain=True)
    client.connect(broker, port, keepalive=3)
    return client

def on_message(client, userdata, message):
    print("Message received: " + message.payload.decode('utf-8'))

def publish(client):
    msg_count = 1
    while True:
        time.sleep(2)
        Temp= round(random.uniform(0, 50), 2);
        Humi= random.randint(0,100);
        msg=json.dumps({"temp": Temp,"humidity":Humi})
        result = client.publish(topic, msg, qos=1, retain=True)
        # result: [0, 1]
        status = result[0]
        if status == 0:
            print(f"Send `{msg}` to topic `{topic}`")
        else:
            print(f"Failed to send message to topic {topic}")
        msg_count += 1
        if msg_count > 20:
            break

def run():
    client = connect_mqtt()
    client.loop_start()
    client.subscribe("room/fan")
    publish(client)
    client.loop_stop()


if __name__ == '__main__':
    run()
```

MQTT Connect

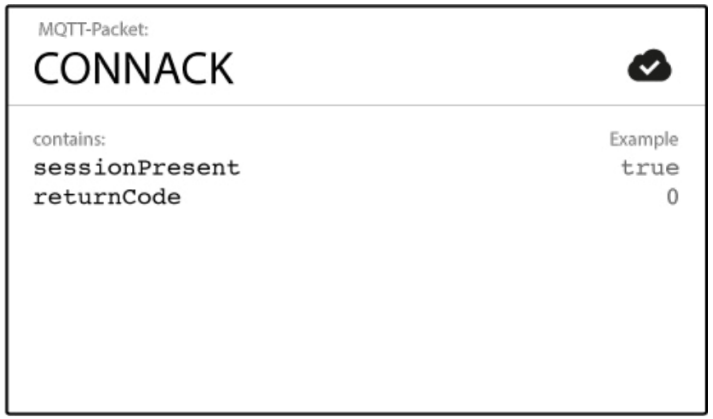
CONNECT

To initiate a connection, the client sends a command message to the broker.



CONNACK

The broker responds with a CONNACK message.



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

MQTT Pub/Sub Attributes

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-Packet:

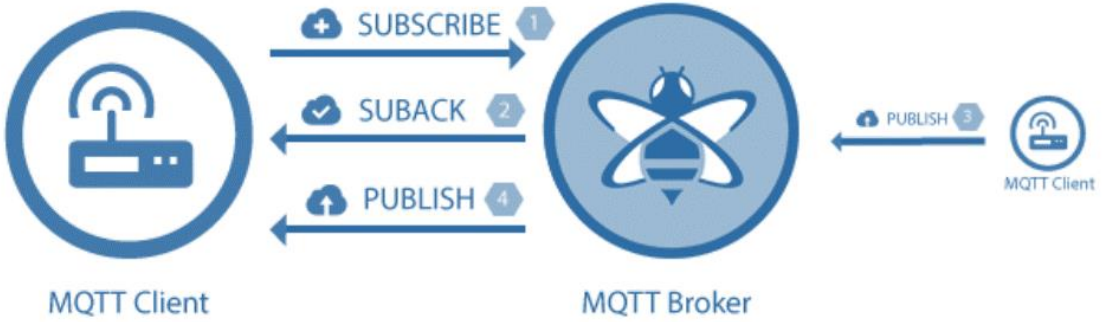
SUBSCRIBE

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

MQTT-Packet:

SUBACK

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



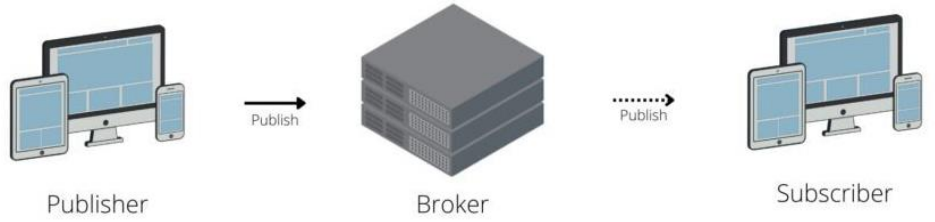
Return Code	Return Code Response
0	Success - Maximum QoS 0
1	Success - Maximum QoS 1
2	Success - Maximum QoS 2
128	Failure

MQTT QoS

0

Fire & forget - Unreliable

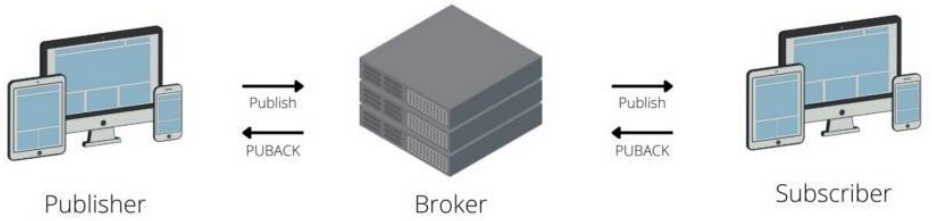
Message is delivered without duplication; no acknowledgment is required



1

At least once - Reliable

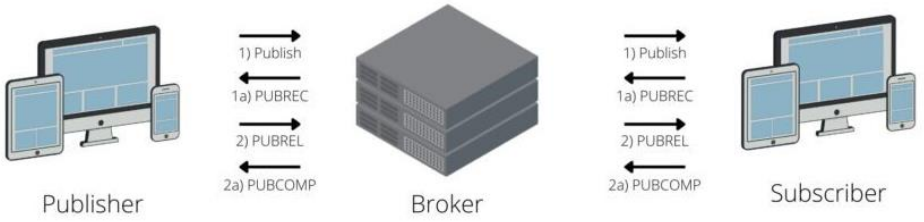
Message is delivered at least once with possible duplications; acknowledgment is required.



2

Exactly once – Reliable without duplication

The message is delivered without duplications



MQTT QoS

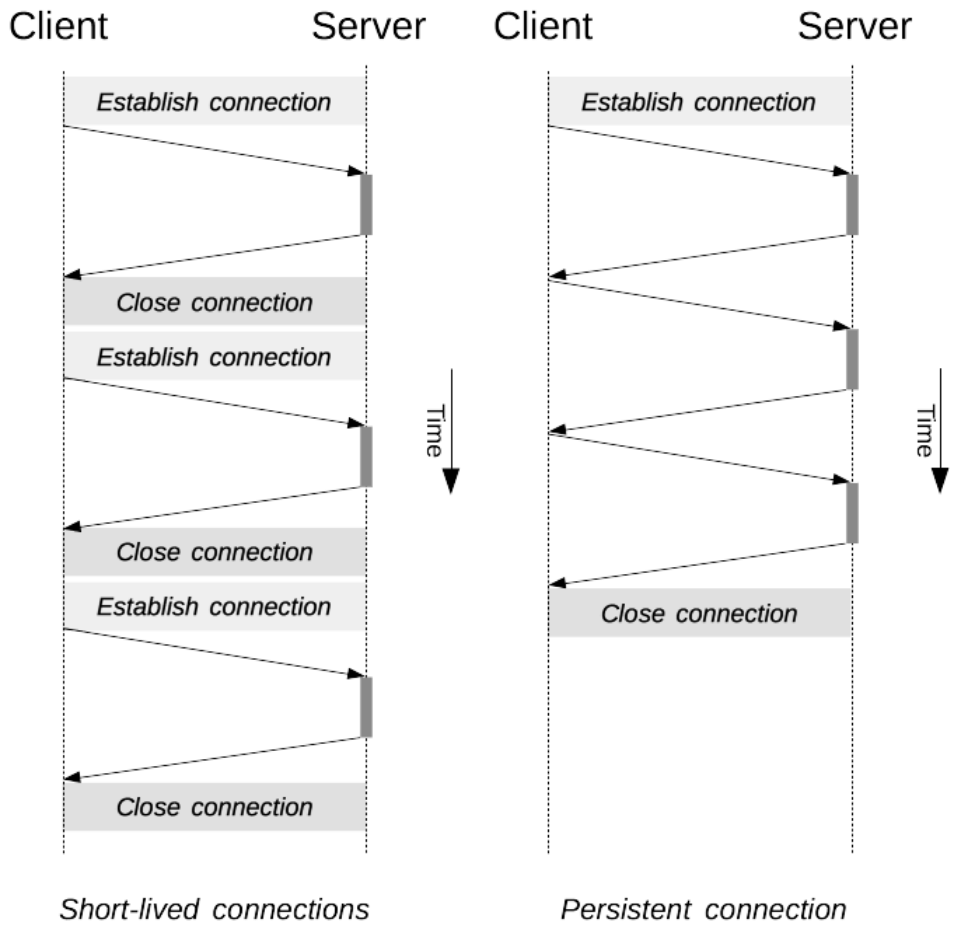
QoS publishing	QoS subscribing	QoS Publisher >> Broker	QoS Broker >> Subscriber
0	1 or 2	0	0
1 or 2	0	1 or 2	0
2	1	2	1
1	1	1	1

MQTT Session

Although MQTT is not a message queue by definition, it can queue messages for clients

Clean Session = False → Persistent session
The broker stores all subscriptions for the client and all missed messages for the client that subscribed with a QoS level 1 or 2.

Clean Session = True → Not persistent session
The broker does not store anything for the client and purges all information from any previous persistent session.



MQTT Session

Best Practices

Clean Session = False → Persistent session

- The client must get all messages from a certain topic, even if it is offline. You want the broker to queue the messages for the client and deliver them as soon as the client is back online.
- The client has limited resources. You want the broker to store the subscription information of the client and restore the interrupted communication quickly.
- The client needs to resume all QoS 1 and 2 publish messages after a reconnect.

Clean Session = True → Not persistent session

- The client needs only to publish messages to topics, the client does not need to subscribe to topics. You don't want the broker to store session information or retry transmission of QoS 1 and 2 messages.
- The client does not need to get messages that it misses offline.

MQTT Session

How long does the broker store messages?

- The broker stores the session until the clients comes back online and receives the message.
- The memory limit of the operating system is the primary constraint on message storage.

What's stored in a persistent session?

- Existence of a session (even if there are no subscriptions).
- All the subscriptions of the client.
- All messages in a QoS 1 or 2 flow that the client has not yet confirmed.
- All new QoS 1 or 2 messages that the client missed while offline.
- All QoS 2 messages received from the client that are not yet completely acknowledged.

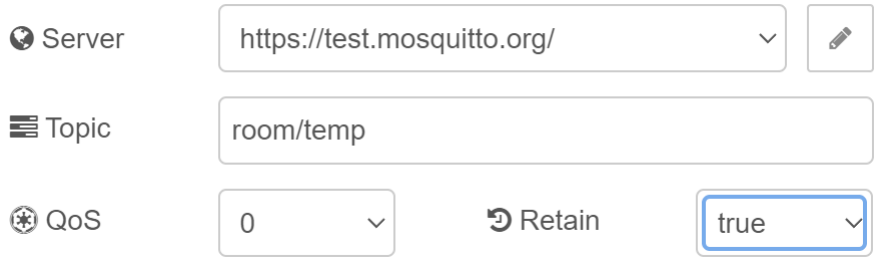
Retained Messages

- The broker stores the last retained message and the corresponding QoS for that topic.
- Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe.
- The broker stores only one retained message per topic.

Retained Messages

Publish a retained message

```
client.publish(topic, msg, qos=1,
retain=True)
```



Delete a retained message

- Send a retained message with a zero-byte payload on the topic where you want to delete the previous retained message.
- The broker deletes the retained message and new subscribers no longer get a retained message for that topic.
- Often, it is not even necessary to delete, because each new retained message overwrites the previous one

MQTT Certificates

The MQTT brokers can be aware of the state of their clients and communicate that state to other clients.

It acts as the device's initial handshake, letting the network know that the device is now online and operational.

It signals the unavailability or offline status of the device.

If the client disconnects ungracefully, the broker sends the LWT message on behalf of the client.

Sparkplug

LWT

- Each client can specify its last will message when it connects to a broker.
- The last will message is a normal MQTT message with a topic, retained message flag, QoS, and payload.
- In response to the ungraceful disconnect, the broker sends the last-will message to all subscribed clients of the last-will message topic.
- If the client disconnects gracefully with a correct DISCONNECT message, the broker discards the stored LWT message.

MQTT-Packet:

CONNECT

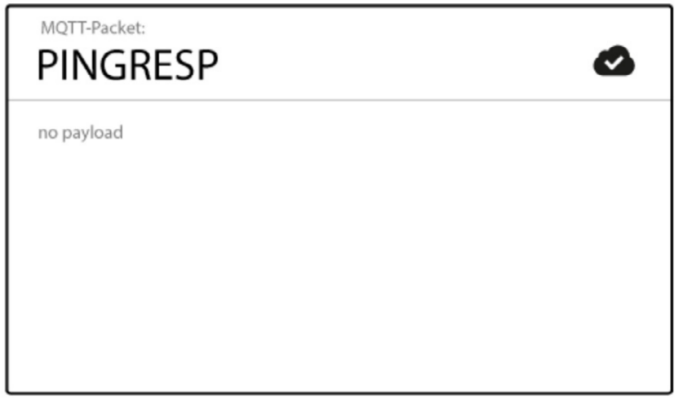
contains:	Example
clientId	"client-1"
cleanSession	true
username (optional)	"hans"
password (optional)	"letmein"
lastWillTopic (optional)	"/hans/will"
lastWillQos (optional)	2
lastWillMessage (optional)	"unexpected exit"
lastWillRetain (optional)	false
keepAlive	60

```
client.will_set("room/lights", payload="Off", qos=0,
retain=True)
```

KeepAlive Timer

Keep Alive Timer

- If the client does not send a messages during the keep-alive period, it must send a PINGREQ packet to the broker to confirm that it is available and to make sure that the broker is also still available.
- The broker must disconnect a client that does not send a a message or a PINGREQ packet in one and a half times the keep alive interval.
- Likewise, the client is expected to close the connection if it does not receive a response from the broker in a reasonable amount of time.
- The maximum keep alive is 18h 12min 15 sec.
- If the keep alive interval is 0, the keep alive mechanism is deactivated



MQTT Wildcards

Wildcards are a special type of topic that can only be used for subscription. Clients can subscribe to a wildcard topic to receive messages from multiple matching topics, eliminating the need to subscribe to each topic individually and reducing overhead.

Single Level: +

By subscribing to a topic with a single-level wildcard, any topic that contains an arbitrary string in place of the wildcard will be matched.

Home/groundfloor/+/temperature

Multi Level:

It covers multiple topic levels. It must be placed as the last character in the topic, preceded by a forward slash.

Home/groundfloor/#

MQTT Broker Monitoring

The $-symbol topics are reserved for internal statistics of the MQTT broker.
There is no official standardization for such topics.

test.mosquitto.org

$SYS/broker/clients/connected → Number of clients currently connected to the MQTT broker.

$SYS/broker/clients/disconnected → Number of clients that have disconnected from the MQTT broker.

$SYS/broker/clients/total → Total count of clients.

$SYS/broker/messages/sent → The count of messages sent by the MQTT broker.

$SYS/broker/uptime → Duration the MQTT broker has been running.

$SYS/# → All info

https://github.com/mqtt/mqtt.org/wiki/SYS-Topics

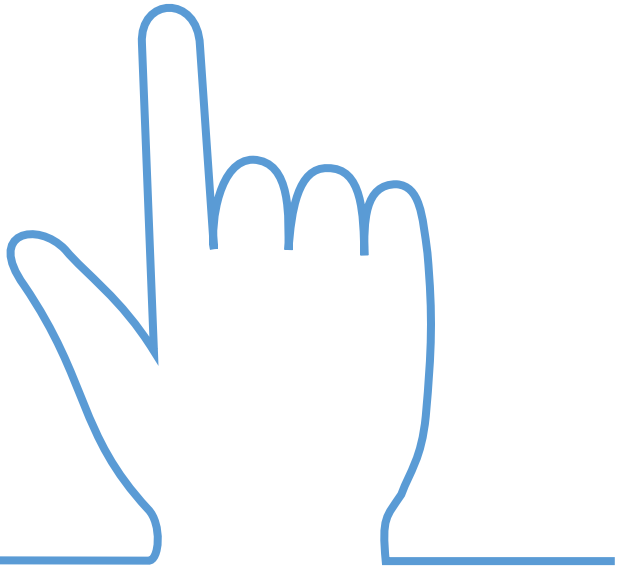
MQTT Security

- The central broker in the MQTT architecture increases the security by decoupling the devices and applications.

- All devices have to authenticate against a central security location (the broker) and only one port is required to be open to the broker (the secure port 8883).

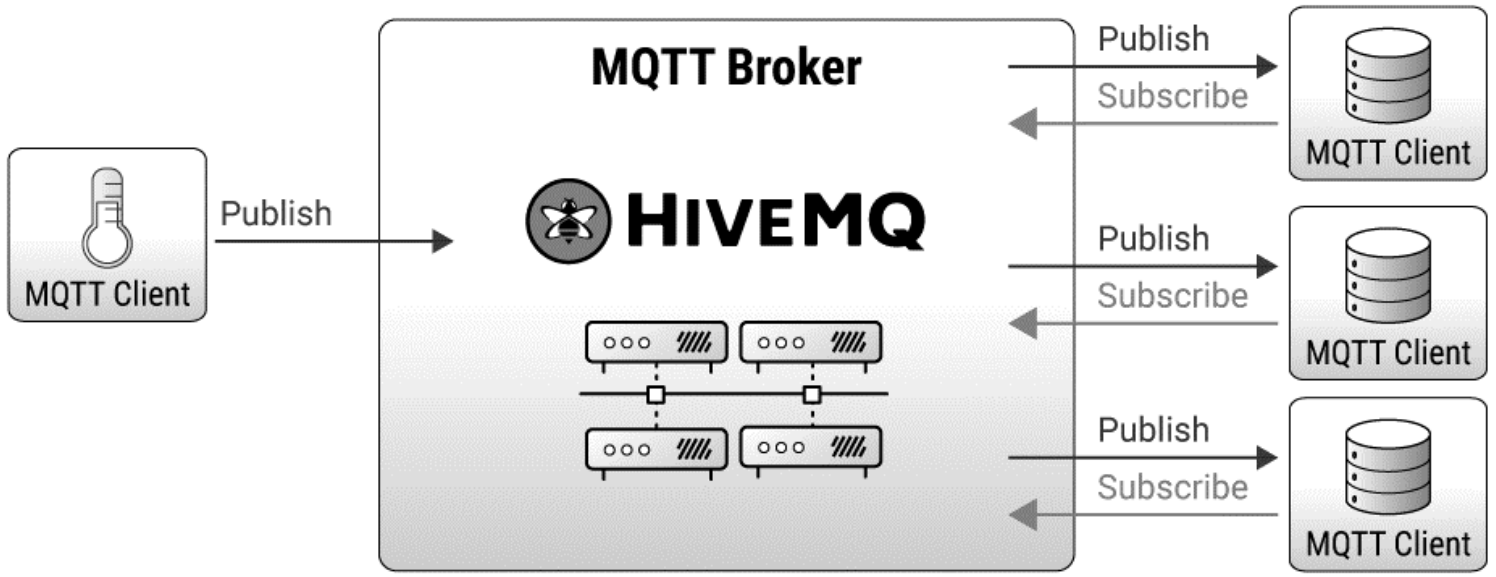
- MQTT provides this security using:
  - Transport Layer Security (TLS) encryption
  - Username/password protected connections.

- Each client is unaware of IP addresses and domains of other devices.



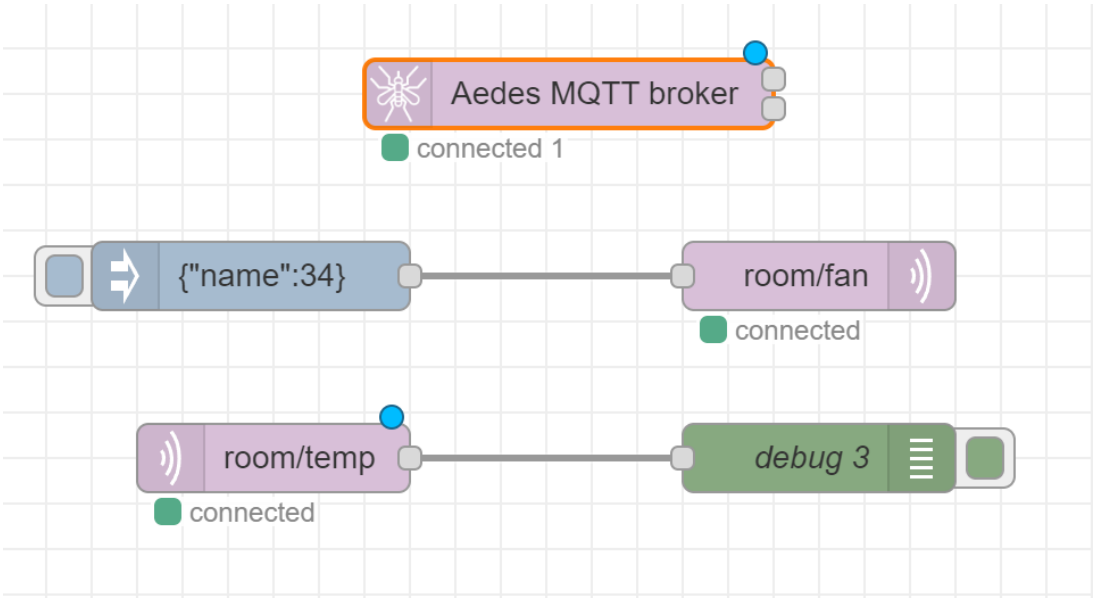
MQTT Broker

Securely connect to a private cloud-based MQTT broker:
www.hivemq.com



MQTT Broker

Connect to the secured installed MQTT broker:



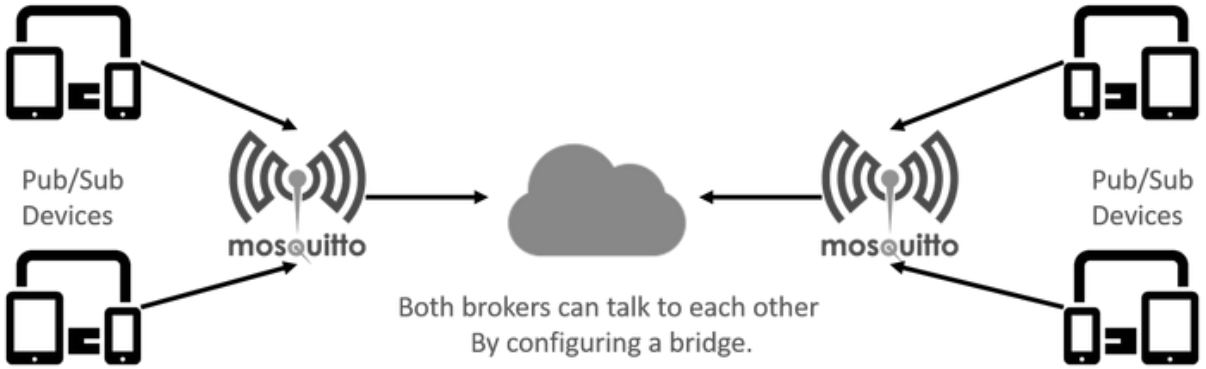

```python
import random
import json
import time
from paho.mqtt import client as mqtt_client

broker = '192.3.16.101'
port = 1883
topic = "room/temp"
# Generate a Client ID with the publish prefix.
client_id = f'publish-{random.randint(0, 1000)}'

def connect_mqtt():
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT Broker!")
        else:
            print("Failed to connect, return code %d\n", rc)

    client = mqtt_client.Client(client_id)
    # client.username_pw_set(username, password)
    client.on_connect = on_connect
    client.on_message = on_message  # attach function to callback
    #client.will_set("room/lights", payload="Off", qos=0, retain=True)
    client.connect(broker, port, keepalive=3)
    return client

def on_message(client, userdata, message):
    print("Message received: " + message.payload.decode('utf-8'))

def publish(client):
    msg_count = 1
    while True:
        time.sleep(2)
        Temp= round(random.uniform(0, 50), 2);
        Humi= random.randint(0,100);
        msg=json.dumps({"temp": Temp,"humidity":Humi})
        result = client.publish(topic, msg, qos=1, retain=True)
        # result: [0, 1]
        status = result[0]
        if status == 0:
            print(f"Send `{msg}` to topic `{topic}`")
        else:
            print(f"Failed to send message to topic {topic}")
        msg_count += 1
        if msg_count > 20:
            break

def run():
    client = connect_mqtt()
    client.loop_start()
    client.subscribe("room/fan")
    publish(client)
    client.loop_stop()


if __name__ == '__main__':
    run()
```

MQTT Reliability

The broker is the central point of communication between all devices, which raises concern about having a single source of failure.

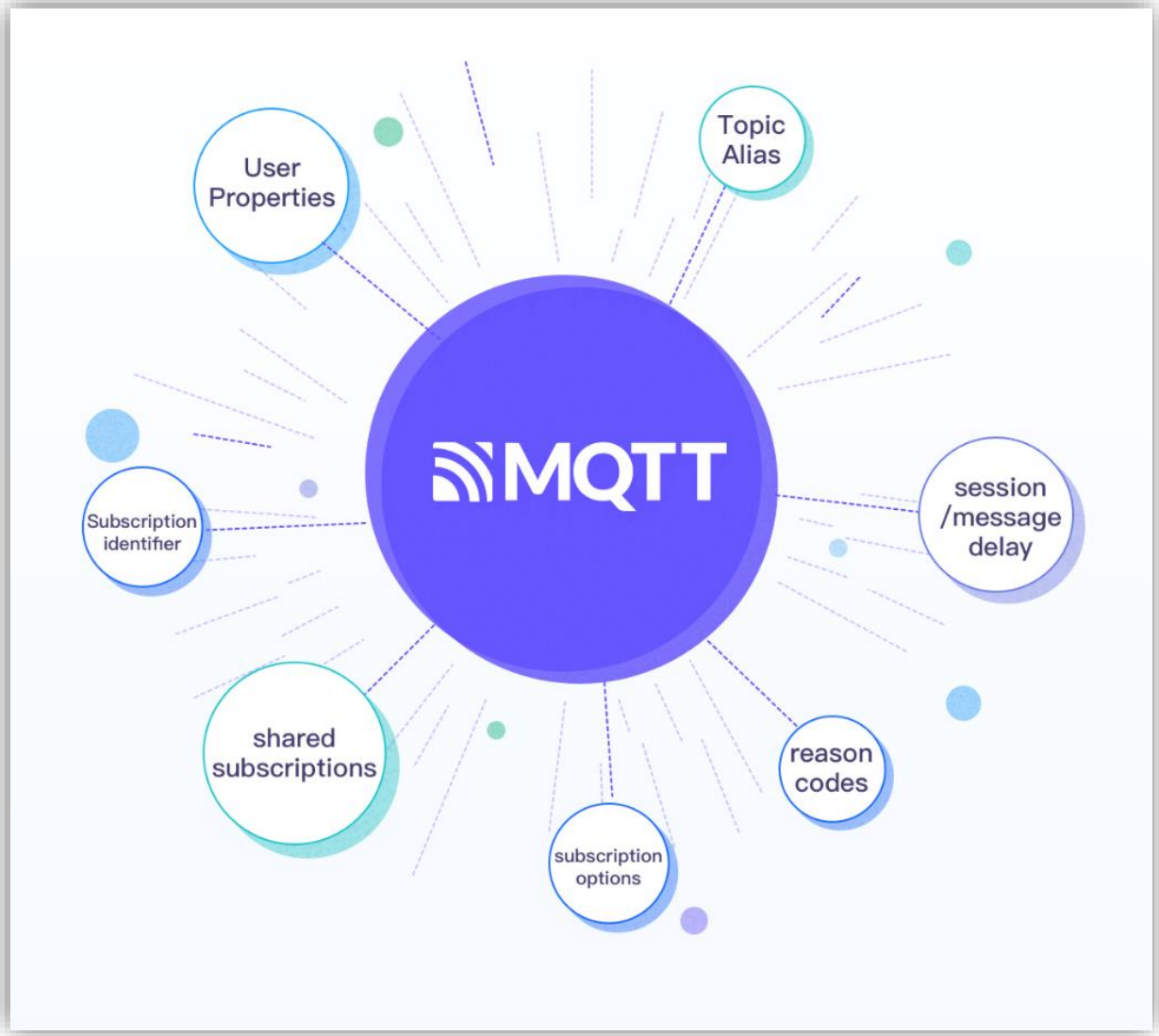
Most broker software and clients support automatic handover to a backup/redundant broker if the primary server fails.

The software can also be set up to share the load of clients across multiple servers on cite, in the cloud, or a combination of both.



MQTT 5.0

MQTT 5.0 adds features that improve the performance, stability, and scalability of large systems.



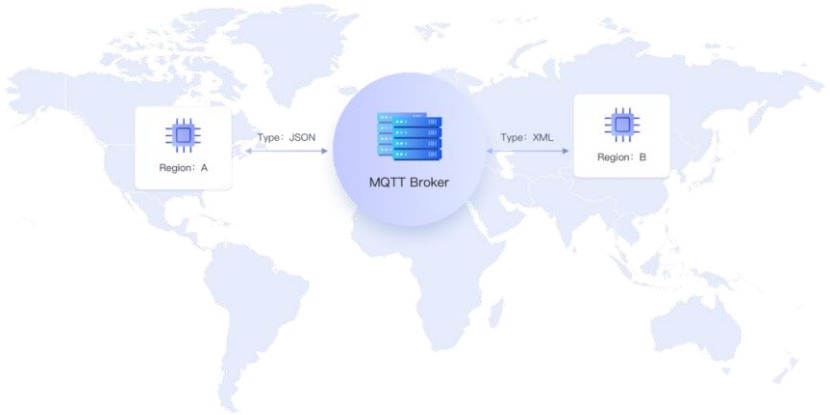
MQTT 5.0

User Properties

User-defined properties that allow users to add their metadata to MQTT messages and transmit additional user-defined information to expand more application scenarios (similar to HTTP headers).

Resource Analysis

```
{
  "region": "A",
  "type": "JSON"
}
```

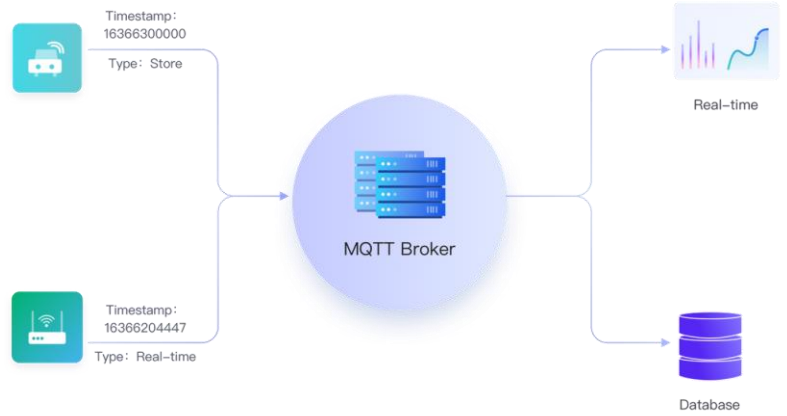


File Transfer

```
{
  "filename": "test.txt",
  "content": "xxxx"
}
```

Message Routing

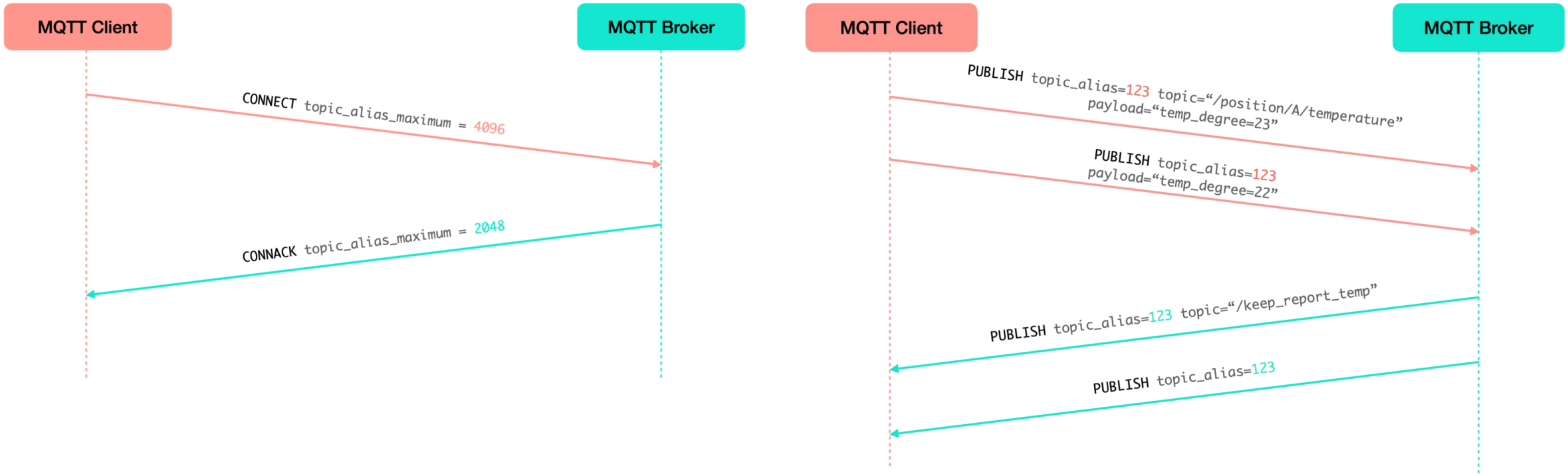
```
{
  "type": "real-time",
  "timestamp": 1636620444
}
```



MQTT 5.0

Topic Alias

Allows users to reduce the possibly long and repeatedly used topic name to a 2-byte integer, so as to reduce the bandwidth consumption when publishing messages.



MQTT 5.0

Flow Control

- The sender has one initial sending quota. Ever time it sends a PUBLISH packet with QoS 1 or 2, the sending quota is reduced by one. Whenever it receives a response packet (PUBACK and PUBCOMP or PUBREC), the sending quota is increased by one.

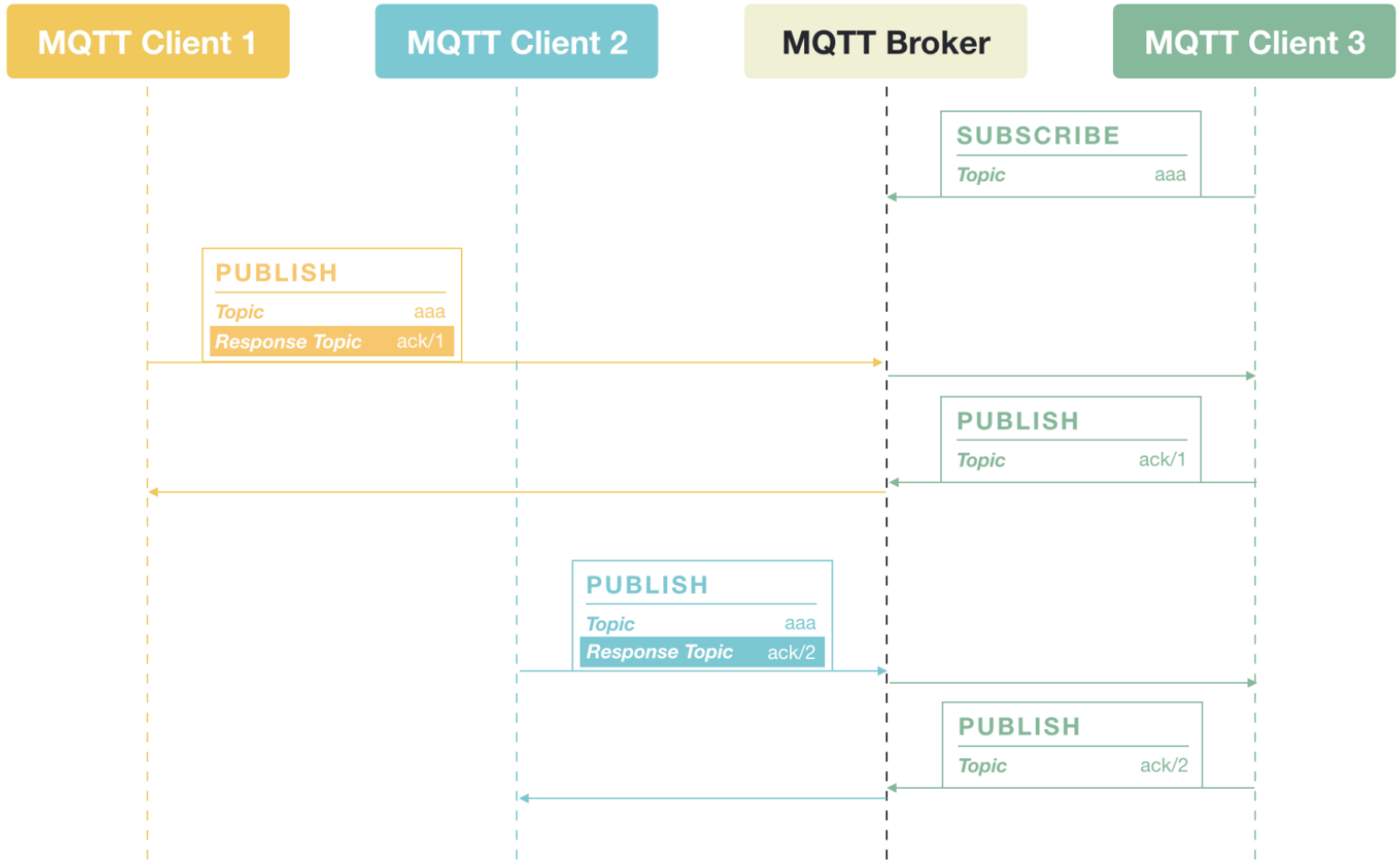
Receive Maximum attribute

- The Receive Maximum attribute exists in the CONNECT packet and CONNACK packet and indicates the largest number of PUBLISH packet with the QoS which is 1 and 2 that the client and server willing to process simultaneously, that is the maximum sending quota that the opposite can use.

MQTT 5.0

Request Response

- Some business scenarios may need the subscriber to trigger some actions and return results, or need to request some information from the subscriber.
- The two communication parties need to negotiate a request topic and a response topic in advance.



MQTT 5.0

Session Expiry Interval

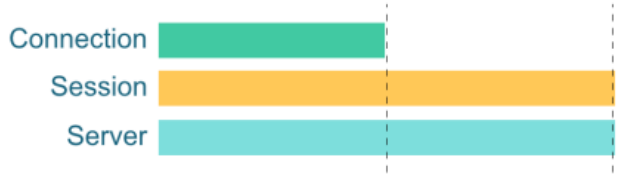
- The duration the broker retains the client's session information.
- If this interval is set to zero, or if the CONNECT packet does not specify an expiry value, the session data is promptly deleted from the broker as soon as the client's network connection terminates.
- The maximum session expiry interval is UINT_MAX (4,294,967,295), enabling an offline session to persist for an extended duration of just over 136 years following client disconnection.

MQTT-Packet:

CONNECT

contains:	Example
clientId	"client-1"
sessionExpiryInterval	**120**
username (optional)	""
passwort (optional)	""
lastWillTopic (optional)	"status/offline"
lastWillQos (optional)	2
lastWillMessage (optional)	"unexpected exit"
lastWillRetain (optional)	false
keepAlive	60

Session Expiry Interval = 0xFFFFFFFF



Session Expiry Interval = 0

Session Expiry Interval > 0

MQTT 5.0

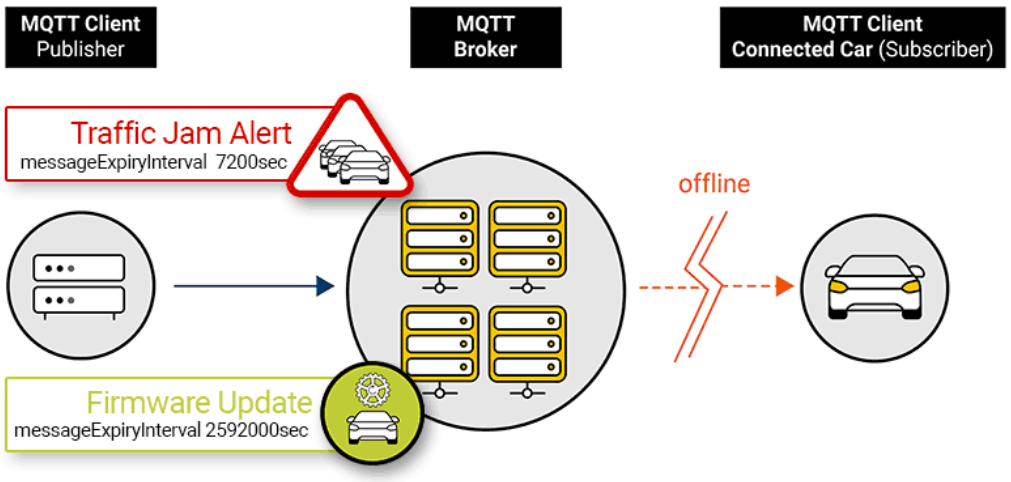
Message Expiry Interval

- Clients can set a unique Message Expiry Interval in seconds for each PUBLISH message.
- This interval establishes the duration the broker preserves the PUBLISH message for subscribers that match the topic but are currently offline.
- If the interval isn't defined, the broker must indefinitely hold the message for matching subscribers, yet disconnected subscribers.
- If the 'retained=true' option is selected during the PUBLISH message, the interval also dictates the length of time a message is retained on a particular topic.

MQTT-Packet:

PUBLISH

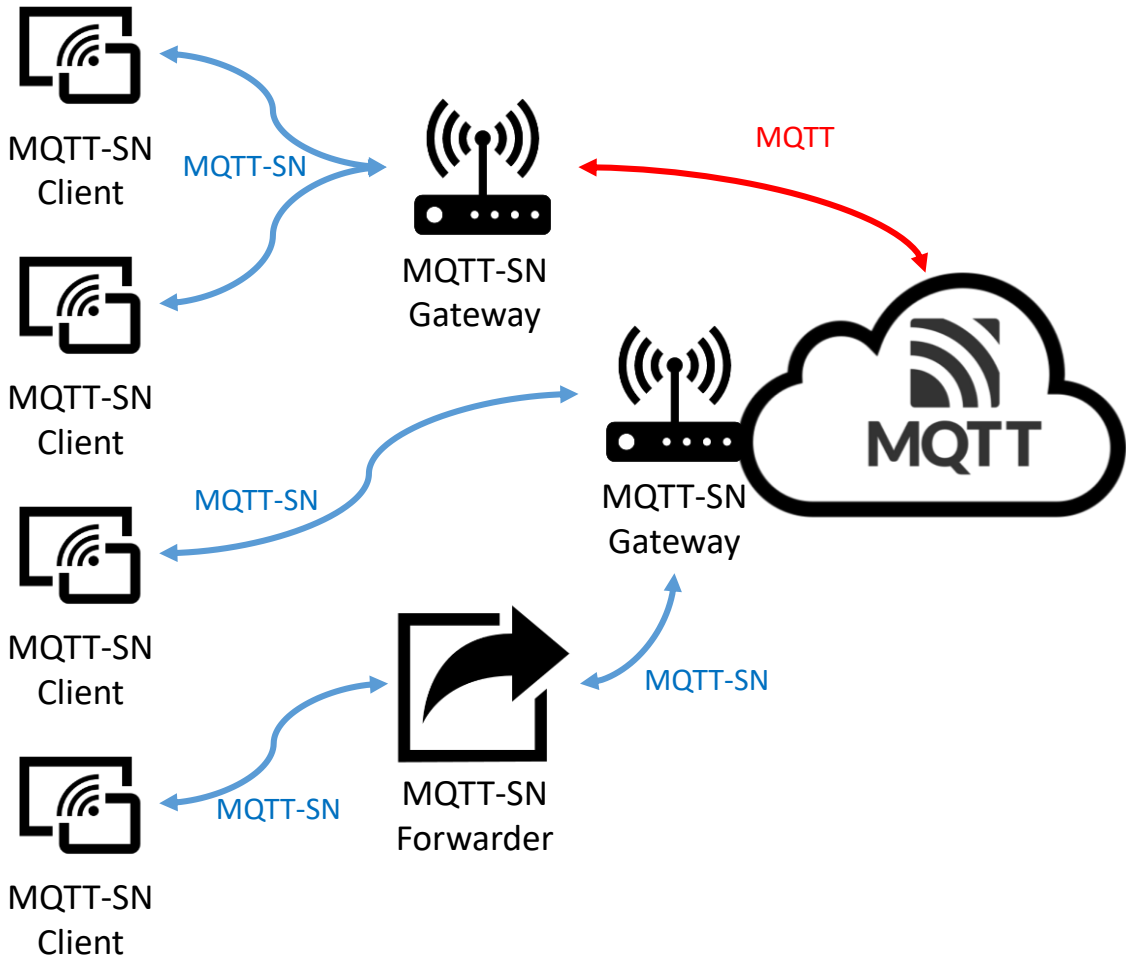
contains:	Example
packetId	4314
topicName	"topic/1"
qos	1
messageExpiryInterval	120
retainFlag	true
payload	"temperature:22:5"
dupFlag	false



MQTT-SN

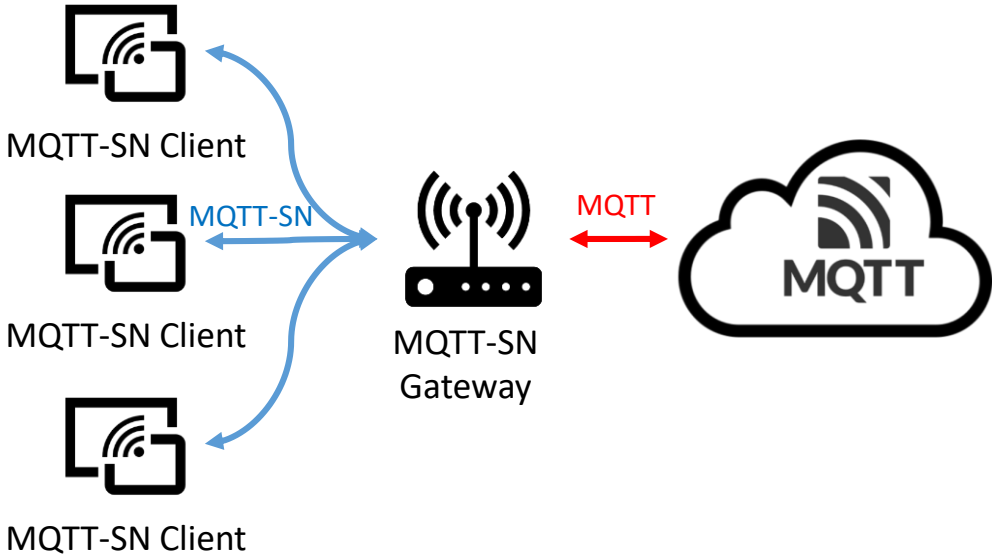
MQTT for Sensor Networks

- MQTT-SN can run over simplified medium and UDP.
- Connect message split into three messages two are optional and are used for the will topic and message.
- Topic id's used in place of topic names.
- Short Topic names
- Pre-defined topics.
- Discovery process to let clients discover the Gateway
- Will topic and messages can be changed during the session.
- Off line keep alive procedure for sleeping clients.
- Payload size of 60 bytes (for MQTT it is 256 MB).

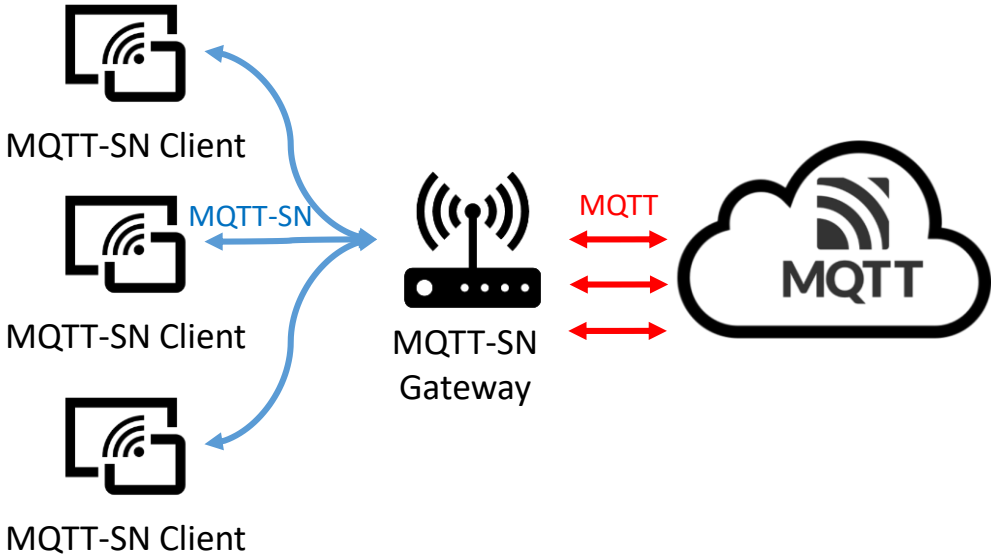


MQTT-SN

Gateway Types

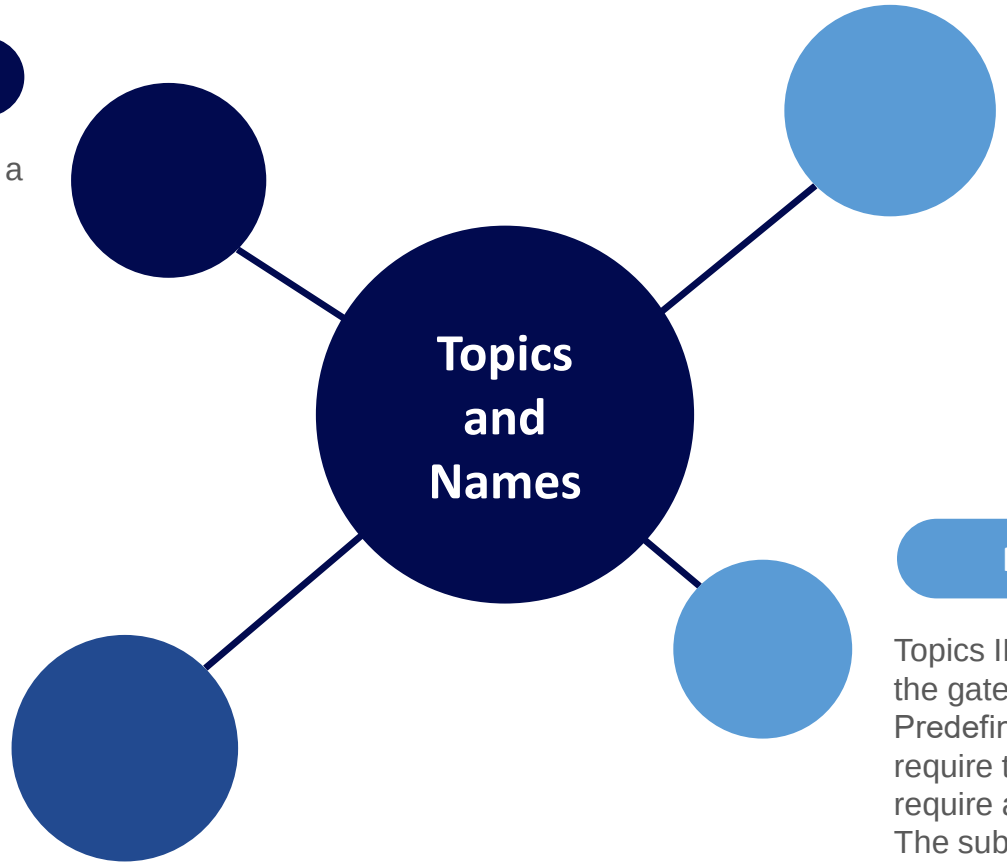


Aggregated Gateway
The MQTT-SN connections share a single MQTT.



Transparent Gateway
Each MQTT-SN connection has an MQTT connection to the broker.

MQTT-SN Topics



Topic Names

When you subscribe to a topic using a topic name (same as MQTT) the subscribe function returns a topic ID which can be used to publish messages on this topic.
Topic names cannot be used to publish messages!

Short Topic Names

A two character name e.g. “AA”. You can also publish on this topic without requiring topic registration.
Can be used for both publishing and subscribing

Topic IDs

A two byte field which is obtained from a broker or client using the register mechanism and also from a broker using a subscribe message.

Pre-defined Topic IDs

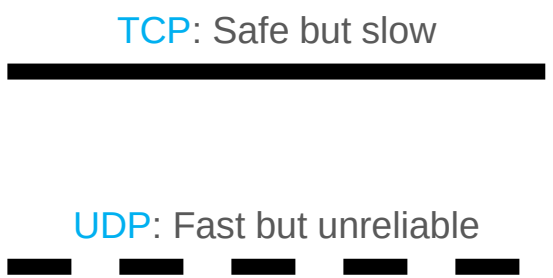
Topics IDs can be predefined on both the gateway and client.
Predefined topic names also don’t require topic registration, but they do require a topic list.
The subscribe message contains a flag to differentiate between a short topic name and a predefined topic ID.

MQTT-SN QoS

MQTT-SN supports QOS 0,1,2 as per MQTT, but it also supports a special publish QOS of 3 or -1.

It is known as QOS -1 but the QOS flag in the message is set to 11 or decimal 3.

Publishing messages with a QOS of -1 or 3 doesn't require an initial connection to have been set up, and requires the use of short topic names or pre-defined topic ids.



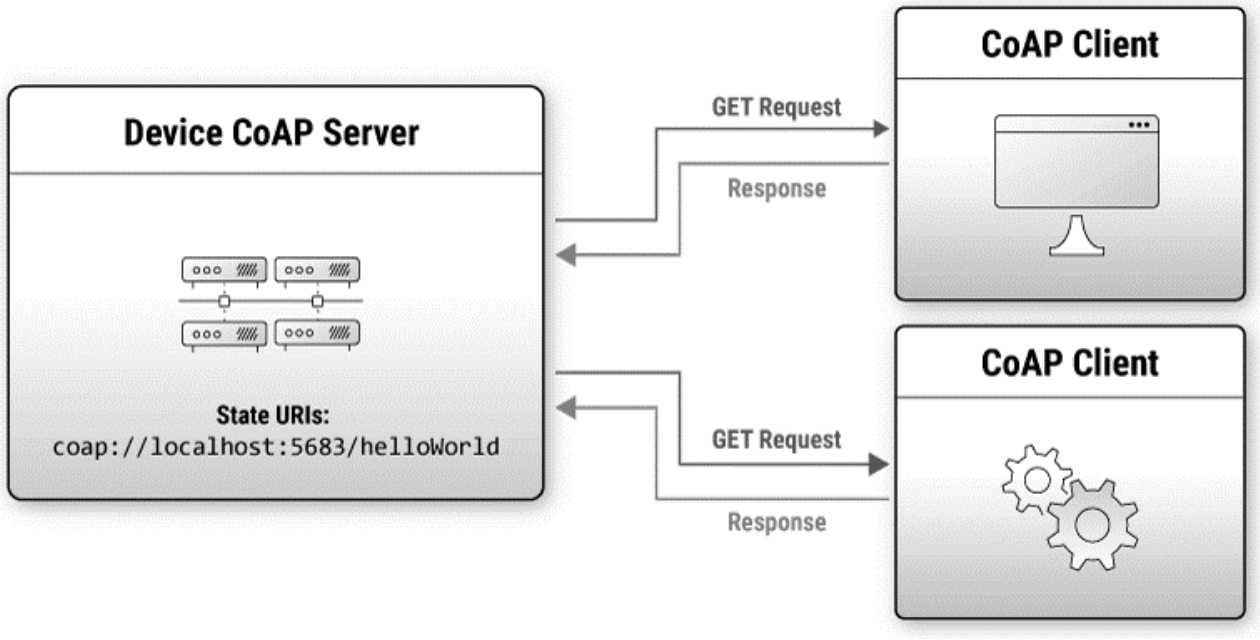
CoAP

CoAP is designed to enable simple, constrained devices to join the IoT even through constrained networks with low bandwidth and low availability.

CoAP uses UDP as the underlying network protocol.

CoAP is a client-server IoT protocol where the client makes a request and the server sends back a response as it happens in HTTP.

CoAP uses the resource model mapped to the Universal Resource Identifier (URI) instead of MQTT topics.



CoAP vs. MQTT

Payload of 256 bytes.

The MQTT messages are sent at QoS 0 and the CoAP messages without confirmation.

	MQTT (TCP) Bytes	CoAP (UDP) Bytes
Establish connection	166	None
Subscribe	159	None
For each message published	388	312+75 (request + response)
Sum for 1 message	1101	387
Sum for 10 messages	8085	3870
Sum for 100 messages	77925	38700

CoAP Message

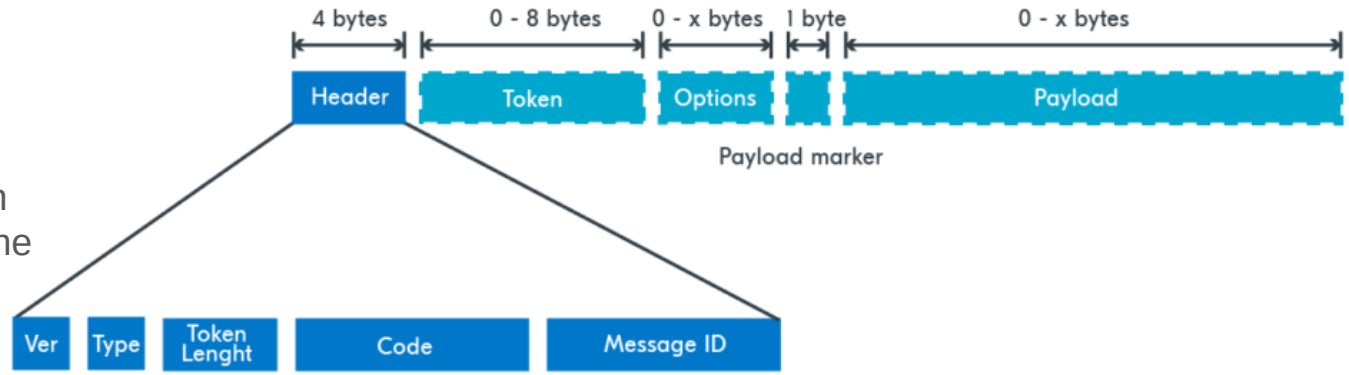
CoAP version
This is set to 1, other values are reserved for future versions.

Type
Indicates if the message is confirmable, non-confirmable, an acknowledgment or reset.

Token length
Indicates the length of the variable-length token field. A Token is used to match responses to requests independently from the underlying messages

Code
Split into two parts, class (0-7) and detail (0-31), where class indicates request, success response or error response and detail gives additional information to the class (e.g., GET, POST,PUT DELETE).

Message ID
Unique ID in network byte order, used to detect duplicates and optionally for reliability.

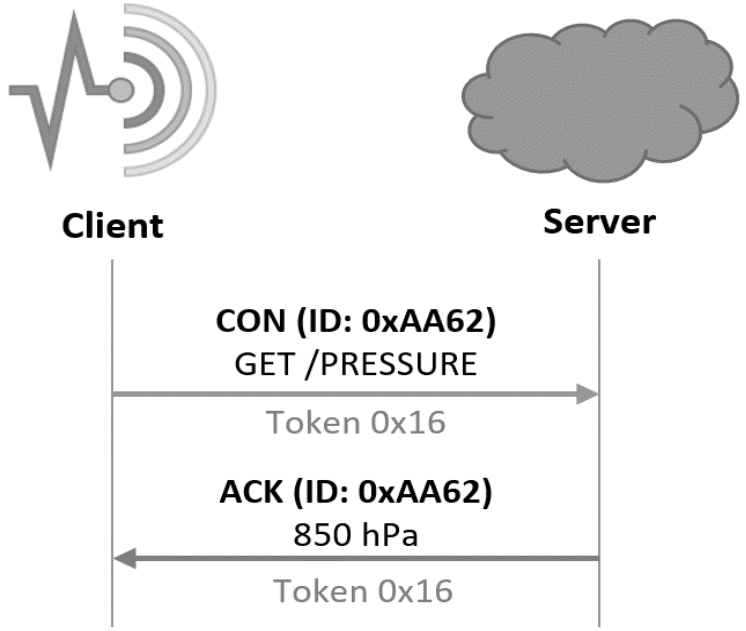


CoAP Operation

The token is generated by the client and should be generated so they are unique and random.

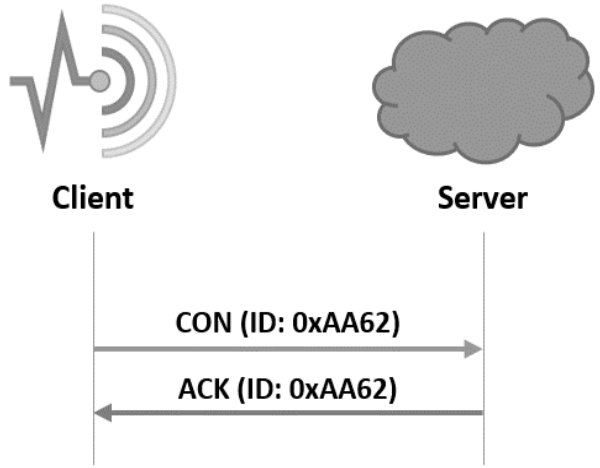
In contrast to the message ID, the token is a nontrivial, randomized string to guard against spoofing and can also be used as a client-local identifier.

Message IDs are sequentially assigned, and therefore easily guessable.

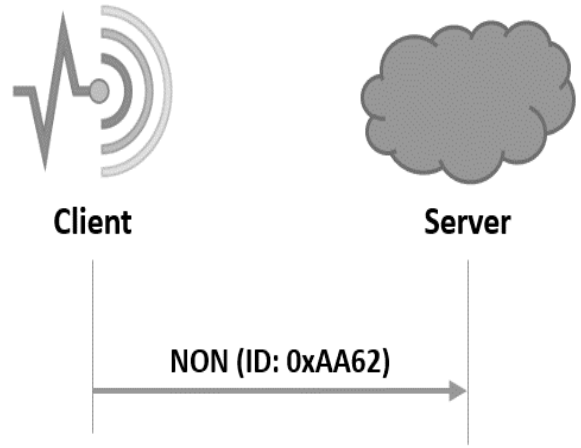


CoAP Reliability

Confirmable messages (CON)
is a reliable message where the client keeps sending the message using a default timeout and exponential back-off between retransmissions until an ACK with the same ID is received from the server.

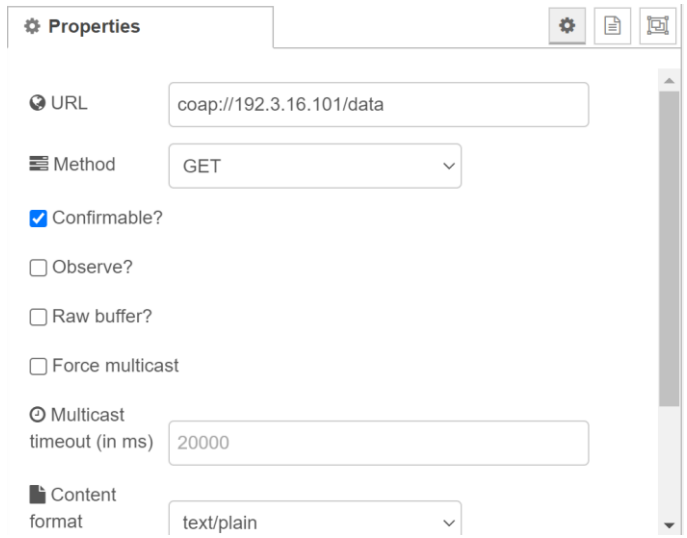


Non-confirmable messages (NON)
Although the non-confirmable messages are not acknowledged, they still have a message ID for duplicate detection.

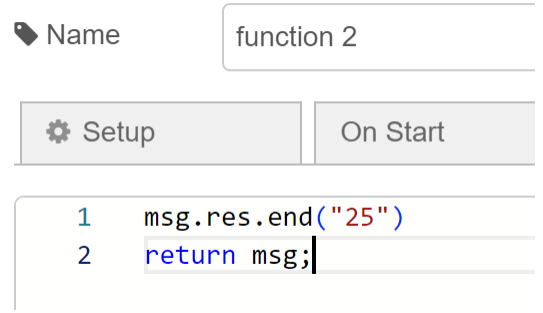


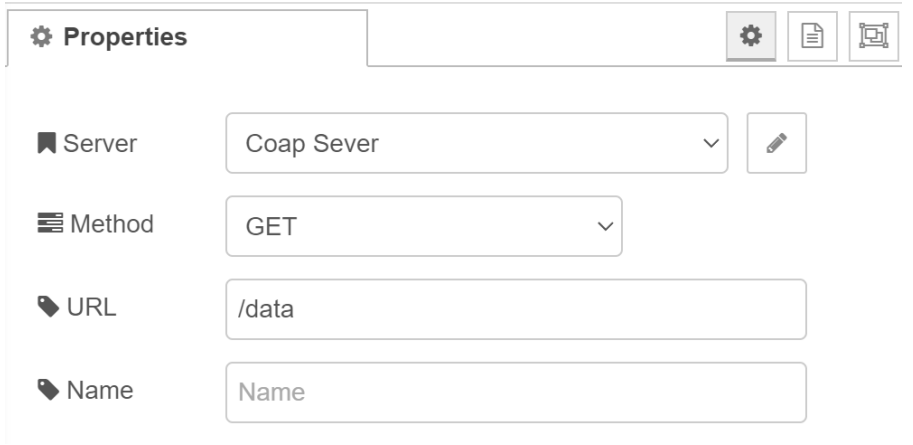
CoAP

Client



Server



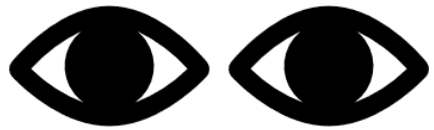


CoAP Observer

Observers reduce the need for constant polling by allowing a client to receive updates based on the state of the subject.

1. Request for Observation:
The client sends a CoAP request to the server, indicating its interest in observing a specific resource. This request includes an "Observe" option with a value of 0.

2. Server Response:
The server responds with the current representation of the resource and includes an "Observe" option with a value of 1. This indicates that the resource is being observed.



3. Observation Updates:
As long as the client remains interested in the resource, the server sends notifications (CoAP responses) to the client whenever the resource changes. These notifications contain the updated representation of the resource and increment the "Observe" value.

4. Termination:
If the client no longer wishes to observe the resource, it can send a new CoAP request with an "Observe" value of 1 to indicate that it wants to cancel the observation.

CoAP Security

Similar to HTTP, CoAP is a plaintext protocol by default which means that to secure the communication, an additional encrypted protocol as a wrapper is required.

In CoAP the data encryption is done by Datagram Transport Layer Security (DTLS) over UDP to secure and protect the information.


Thank You