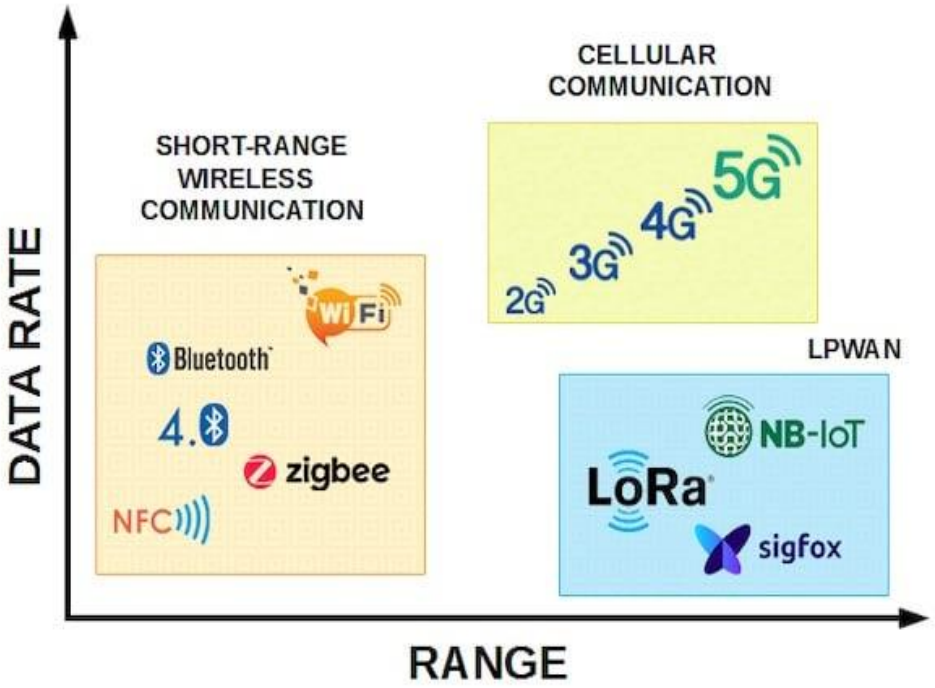
Unit 6.5 (Extra)
MQTT Advanced

Tecnológico de Monterrey

IoT Connectivity

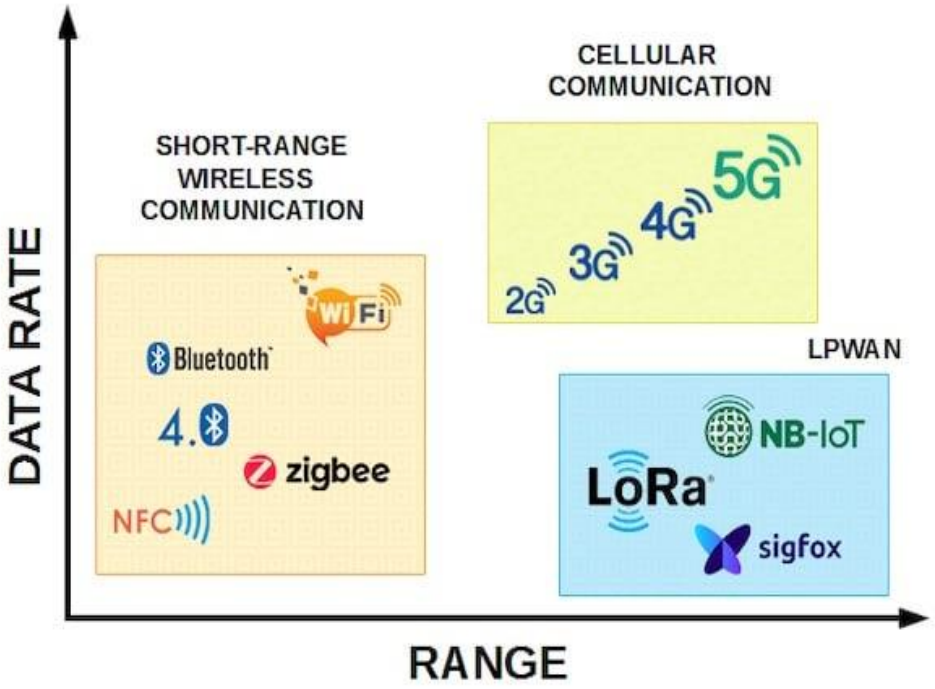
IoT Connectivity

- **Speed or Data Rate:** the amount of information to be transmitted within a time duration.
- **Range:** the maximum distance between two intercommunicating nodes. It mainly depends upon the transmitting power, the frequency band used, and the type of modulation. It can be also affected by the meteorological conditions or the physical placement of the nodes.
- **Interoperability:** the capability to exchange information between nodes, even if they are of different types.
- **Cost:** the price of installing and maintaining a specific technology. Power consumption, maintenance, and scalability have a big impact on the network cost.

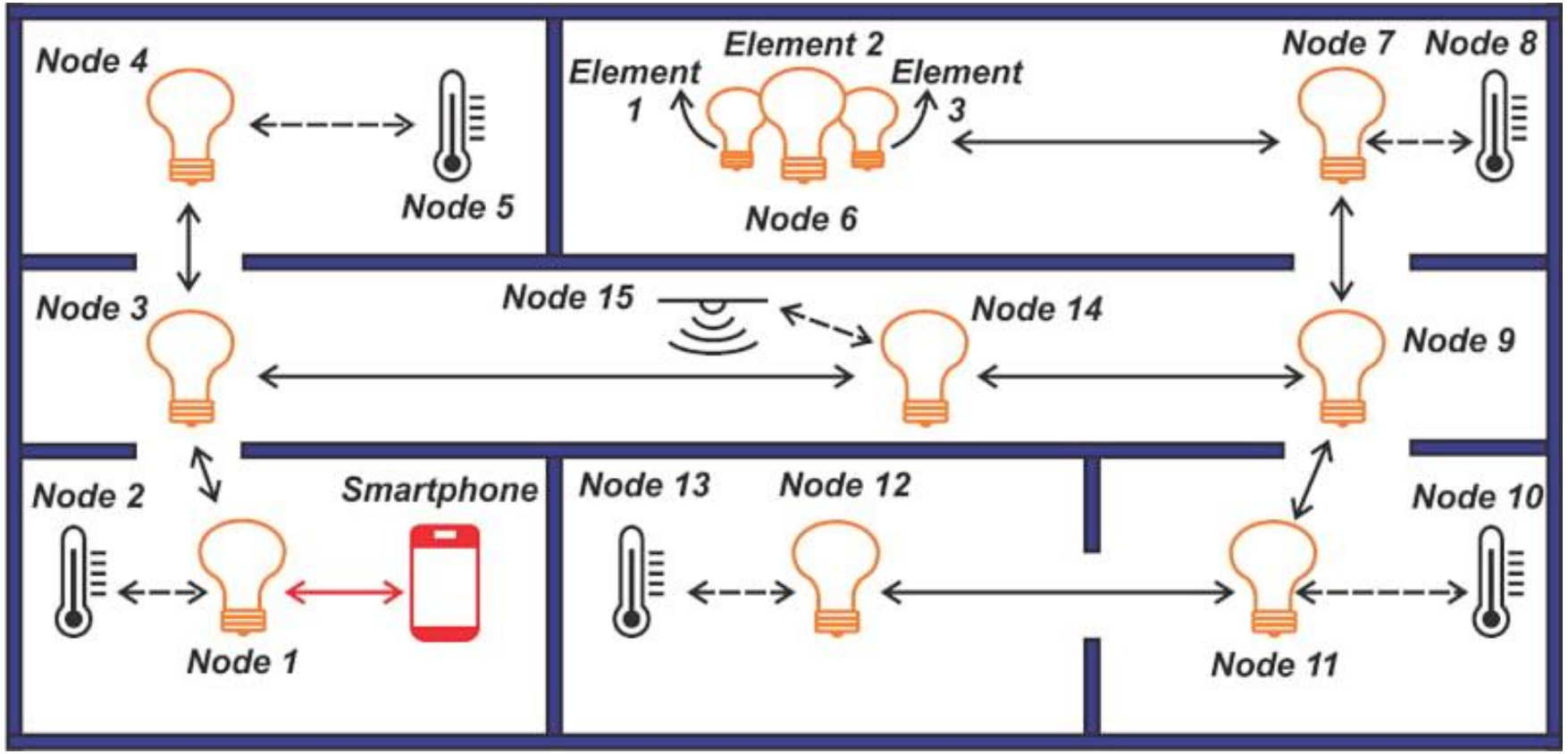


IoT Connectivity

- **Power Consumption**: the amount of energy that a node needs to work within its lifetime. This parameter defines the need for permanent power or the use of a battery.
- **Scalability:** the challenge of deploying a higher number of nodes, increasing the number of end-users, as well as the amount of data to store and process without the need of migrating the technology.
- **Network Topology:** the way nodes communicate with each other. Topologies can be the same as those used in traditional networks. Star, mesh, point-to-point, and point-to-multipoint.
- **Security:** the way to protect data being sent and received.



IoT Connectivity



MQTT Foundations

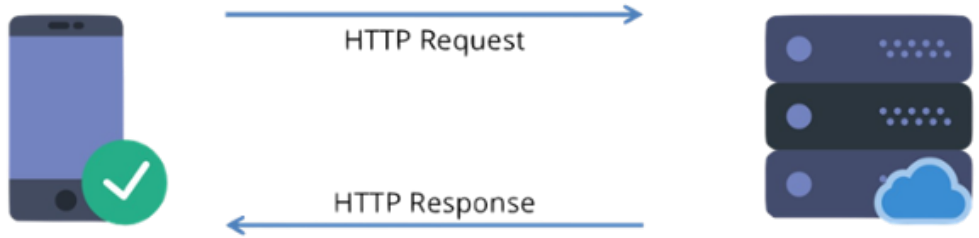
MQTT

- MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for **high-latency**, **low-bandwidth**, and **unreliable networks**.

- It **consumes very little power** on the device it's running on.

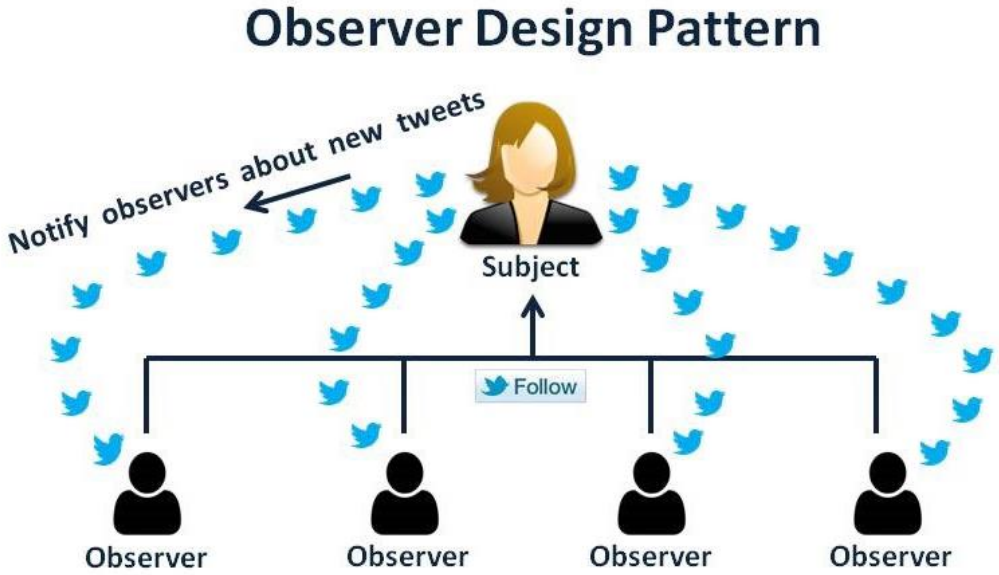
Request-Response

- Two actors: **Client and Server**.
- The server is **all the time listening to requests** that may arrive from its clients
- Server only establishes a **temporary connection**.



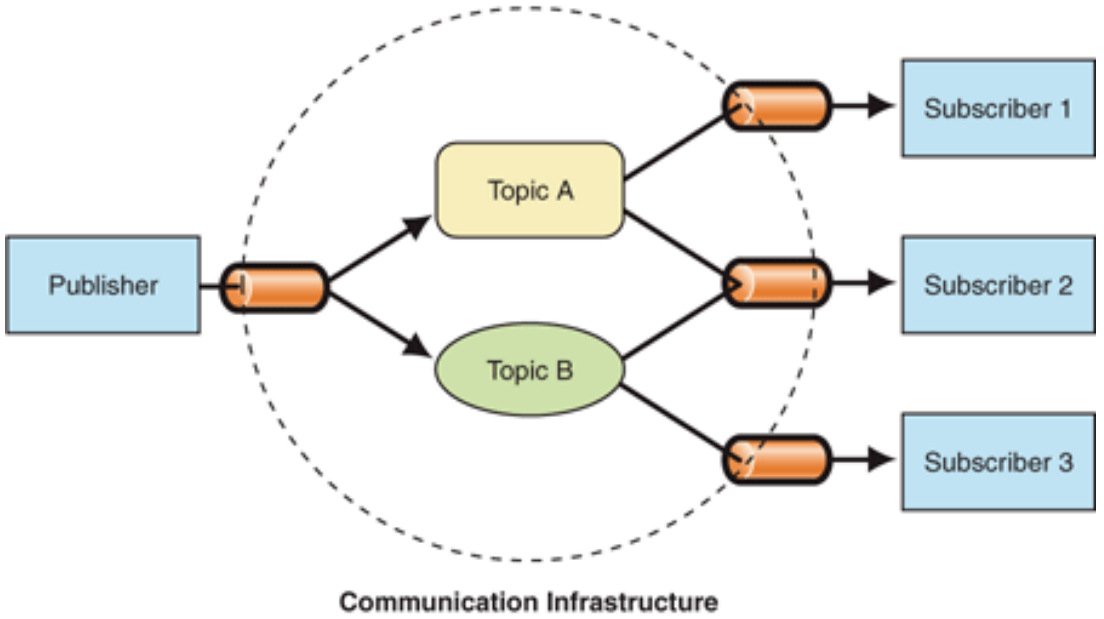
Observer

- Observer we have two main actors: the **observers** and the **subject**.
- **Observers** will make a request to subscribe to the subject and thus be notified when there is any state change.
- **Subject** will have a list of its observers so that it knows to whom to send notifications when there is a change of state.
- The registration can be undone.



Publish-Subscribe

- Like the Observer pattern.
- There is a **Broker role**, which is responsible for filtering messages and knowing exactly who to send them to.
- The **publisher and subscriber do not need to know each other** directly and only need to know the Broker.

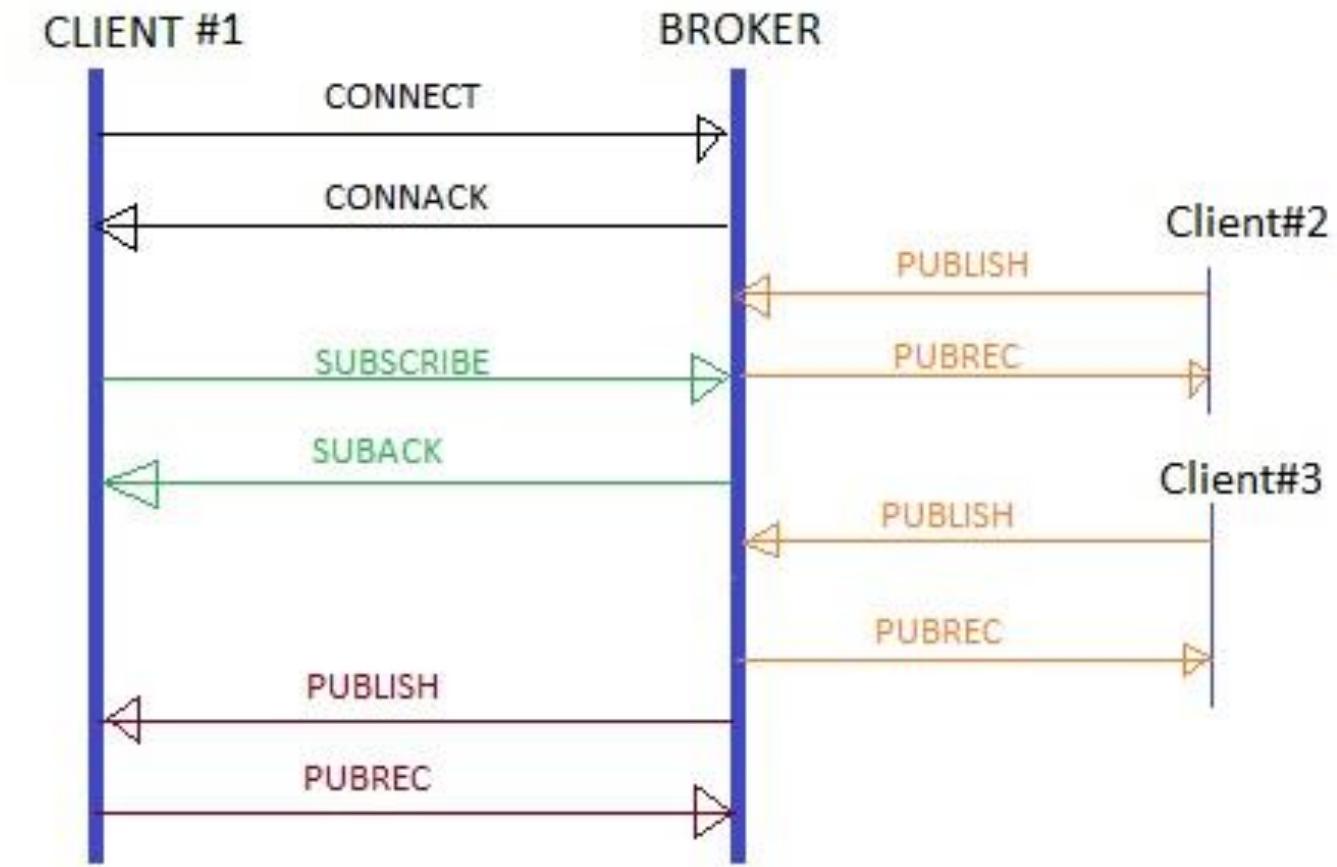


MQTT - Publish-Subscribe Pattern

MQTT Messages

- **Connect:** Attempts to create a connection with the Broker and waits until the connection is established, starting to listen for published messages.
- **Disconnect:** It waits until the client finishes some action that has been performed and ends the TCP/IP connection, thus stopping listening to the messages that will be published.
- **Publish:** returns the information that was sent by the MQTT client.

MQTT Basic Flow



CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment

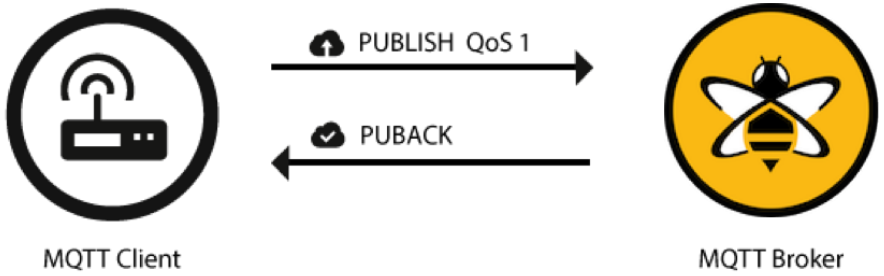
Quality of Service

A Quality of Service (QoS) level is simply **an agreement between the sender and receiver** of a message **that guarantees the deliverability** of said message.

QoS 0: (fire and forget), in this QoS the message is sent only once and there will be no further steps, so the message will not be stored, nor will there be feedback to know if it reached the recipient.

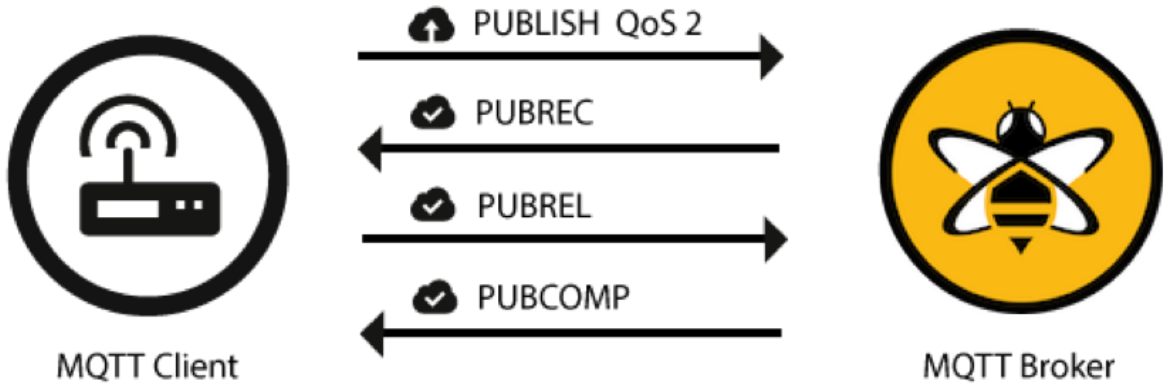
PUBLISH	3	Client to Server or Server to Client	Publish message

QoS 1: the message is delivered at least once, with a wait for receiving feedback on the delivery of the message, called PUBACK. Not receiving the PUBACK, the message will continue to be sent until there is feedback. In this QoS, it can happen that the **message is sent several times and processed several times**.



PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment

QoS 2: In this transfer mode, the message is delivered exactly once, requiring the message to be stored locally at the sender and receiver until it is processed.

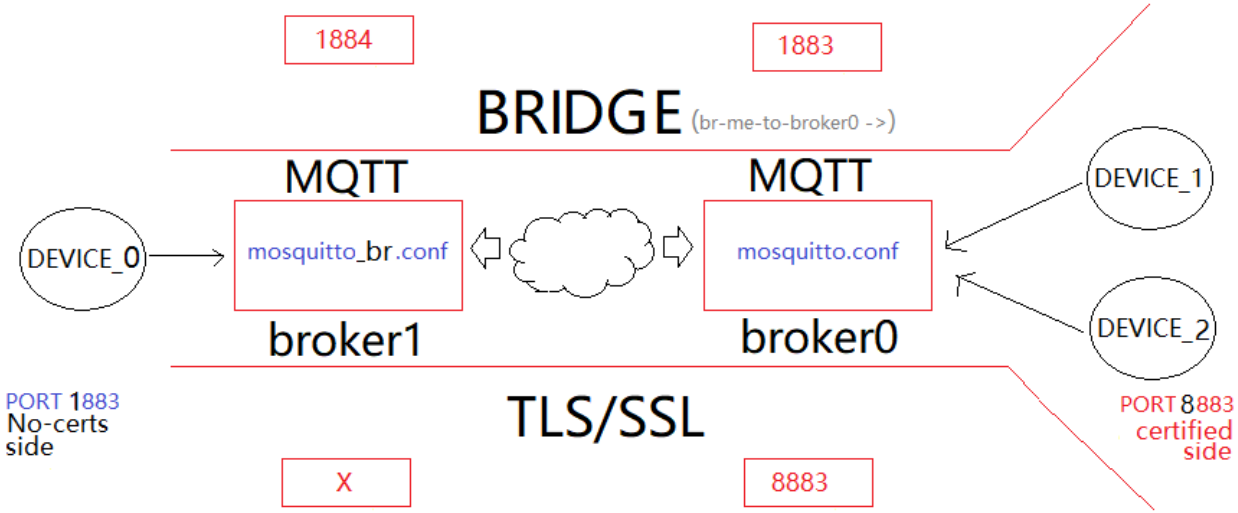


PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (assured delivery part 3)

MQTT Advanced Concepts

Bridge Mode

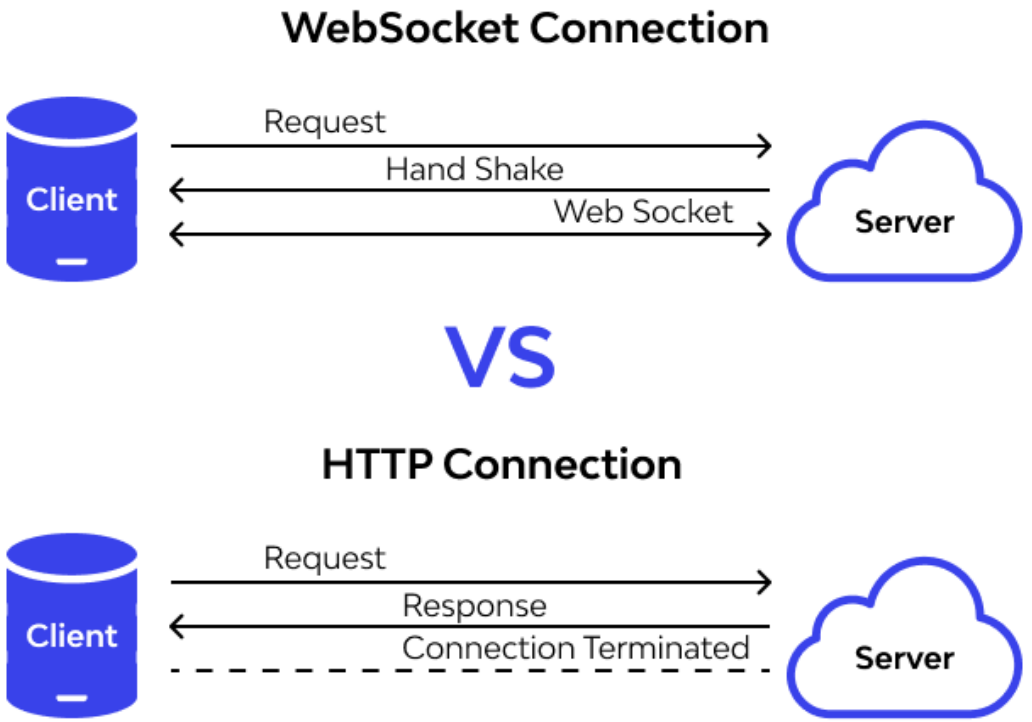
- There are situations where the publishers and subscribers are not directly connected in the same Broker.
- It happens because of scalability, interoperability, or security reasons.
- In this situation, we use a Bridge-Broker.



WebSocket

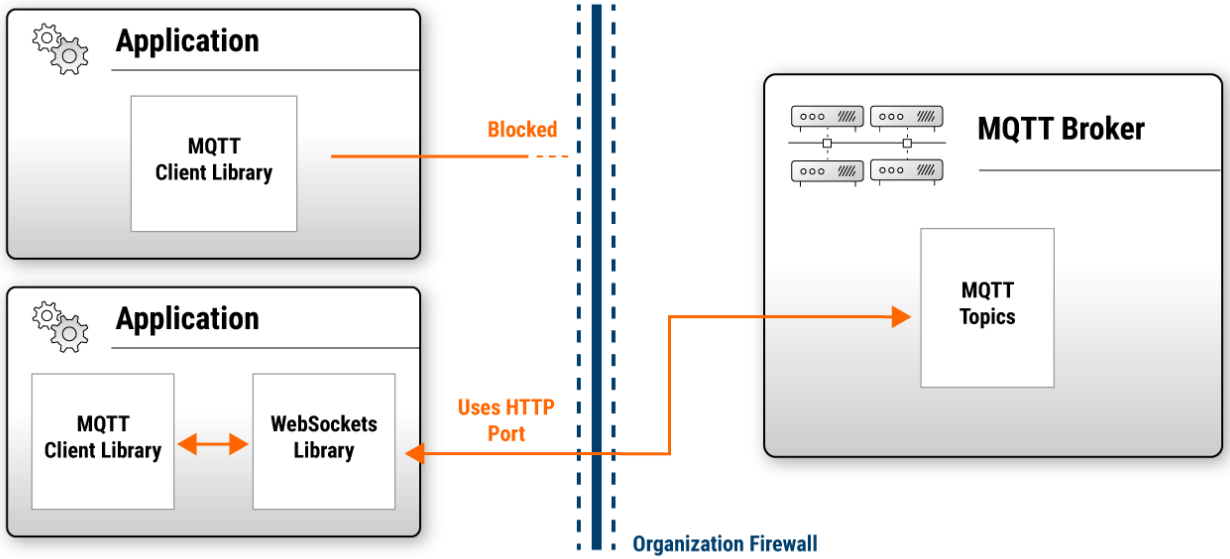
- **WebSocket Protocol (RFC 6455):** was created to allow web applications to conduct two-way communications with web servers, as HTTP cannot do that without awkward workarounds.

- A web browser that has WebSocket support allows web applications to communicate with a server like TCP does for regular applications - **sending and receiving data asynchronously in both directions**.



WebSocket

- WebSocket is an alternative when the client does not support sending the messages using a direct TCP stack (for example, Java Script).
- This approach includes overhead in the communication process.

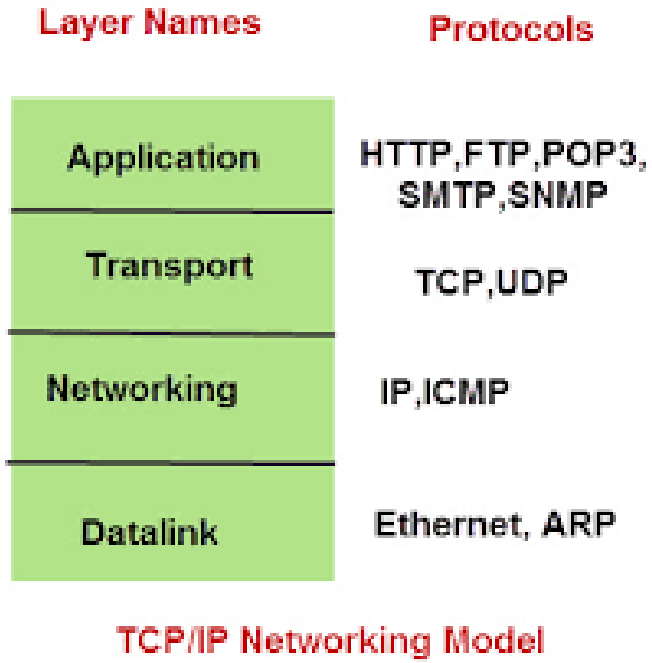


Security Challenges in IoT

- Security is seen as a **trade-off between the level of protection and the degree of usability**.
  - IoT devices have limited computing power and memory capacity. Many **cryptographic algorithms require more resources** than tiny IoT devices possess.
  - Another security challenge arises from the **need to update devices** in the field. Critical security issues that require updates to be rolled out to all devices simultaneously are hampered by unreliable networks on which many IoT devices run.
  - Because user acceptance depends more than ever on easy installation and maintenance, security must be **intuitive for the user**.

Approaches to security in MQTT

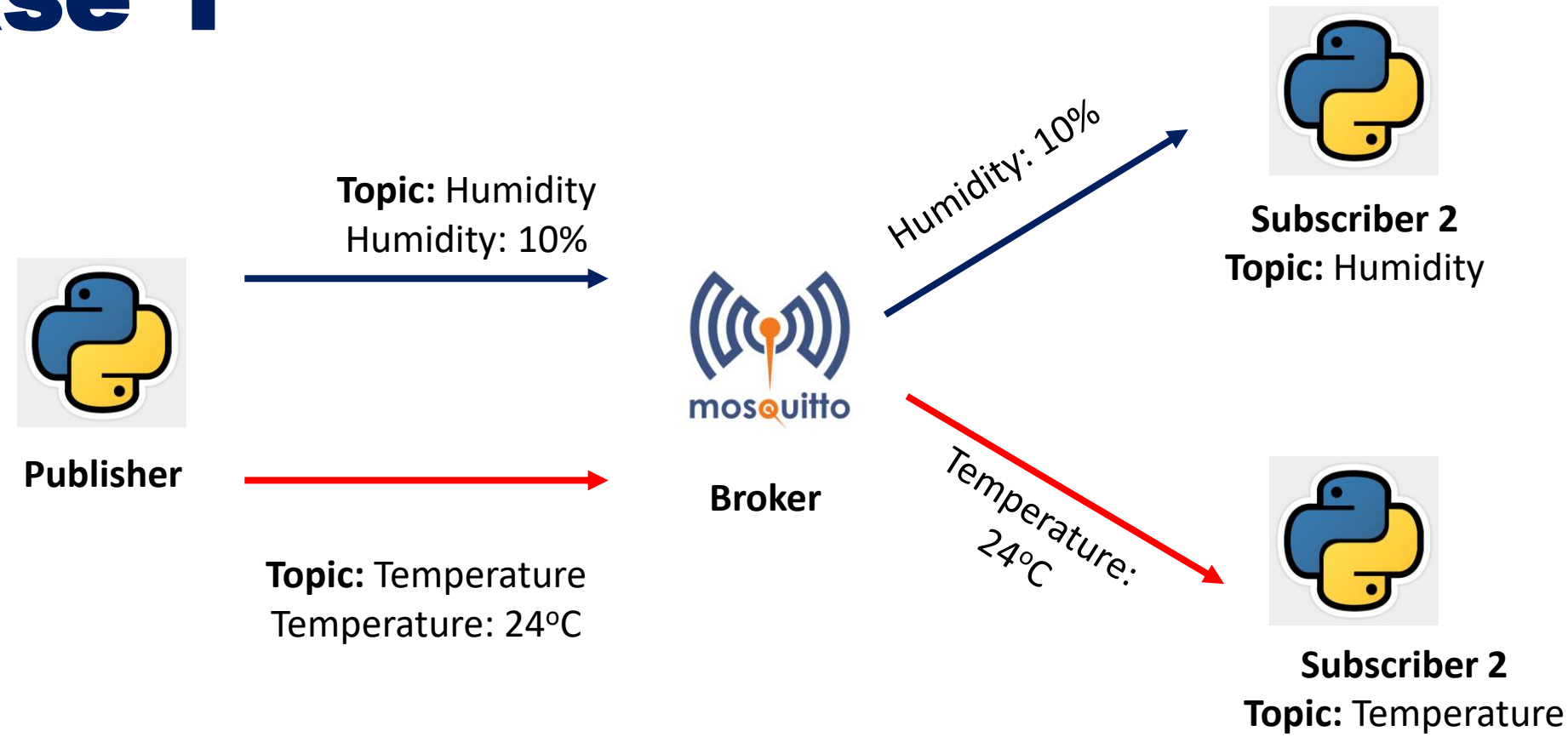
- **Network level:** One way to provide a secure and trustworthy connection is to use a physically secure network or VPN for all communication between clients and brokers.
- **Transport level:** When confidentiality is the primary goal, TLS/SSL is commonly used for transport encryption. This method is a secure and proven way to make sure that data can't be read during transmission and provides client-certificate authentication to verify the identity of both sides.
- **Application Level:** On the transport level, communication is encrypted, and identities are authenticated. The MQTT protocol provides a client identifier and username/password credentials to authenticate devices on the application level.



Security in MQTT is divided into **multiple layers**.

Basic Scenario Implementation

Case 1


Topic: Humidity
Humidity: 10%
Publisher
Topic: Temperature
Temperature: 24ºC
mosquitto
Broker
Humidity: 10%
Temperature: 24ºC
Subscriber 2
Topic: Humidity
Subscriber 2
Topic: Temperature

Case 1

/etc/mosquitto/mosquitto.conf

```
allow_anonymous true
port 1883
persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

log_dest stdout

include_dir /etc/mosquitto/conf.d
~
```

no authentication

Listen port 1883

Print log in console

```
root@ubuntu-dev:/etc/mosquitto# mosquitto -c mosquitto.conf -v
```

Case 1

mqqt_publish.py

```python
import random
import time

from paho.mqtt import client as mqtt_client

broker = 'localhost'
port = 1883
top_temp = "srv/temperature"
top_hum = "srv/humidity"

client_id = f'python-mqtt-{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.connect(broker, port)
    return client
```

params

Connect function

Client(client_id="", clean_session=True, userdata=None, protocol=MQTTv311, transport="tcp")

Case 1

mqqt_publish.py

```python
def publish(client,topic, msg):
    result = client.publish(topic, msg)
    # 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}")


def run():
    client = connect_mqtt()
    msg_count = 0
    client.loop_start()

    while (True):
        time.sleep(1)
        temperature = 20 + (random.randint(0, 100) * 4)
        msg = f"temperature: {temperature}"
        publish(client, top_temp, msg)

        humidity = (random.randint(0, 100))
        msg = f"humidity: {humidity}"
        publish(client, top_hum, msg)

        msg_count += 1

run()
```

LOOP_START() / LOOP_STOP(): These functions implement a threaded interface to the network loop. Calling loop_start() once, before or after connect*(), runs a thread in the background to call loop() automatically. This frees up the main thread for other work that may be blocked. This call also handles reconnecting to the broker. Call loop_stop() to stop the background thread. The force argument is currently ignored.

Case 1

mqqt_sub_temp.py

```python
import random

from paho.mqtt import client as mqtt_client

broker = 'localhost'
port = 1883
top_temp = "srv/temperature"
client_id = f'python-mqtt-{random.randint(0, 1000)}'

def connect_mqtt() -> mqtt_client:
    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.connect(broker, port)
    return client
```

Called when a message has been received on a topic that the client subscribes to, and the message does not match an existing topic filter callback. Use message_callback_add() to define a callback that will be called for specific topic filters. on_message will serve as fallback when none matched.

```python
def subscribe(client: mqtt_client):
    def on_message(client, userdata, msg):
        print(f"Received `{msg.payload.decode()}` from `{msg.topic}` topic")
    client.subscribe(top_temp)
    client.on_message = on_message
```

Case 1

mqqt_sub_temp.py

```
def run():
    client = connect_mqtt()
    subscribe(client)
    client.loop_forever()

run()
```

This is a blocking form of the network loop and will not return until the client calls disconnect(). It automatically handles reconnecting.

Case 1

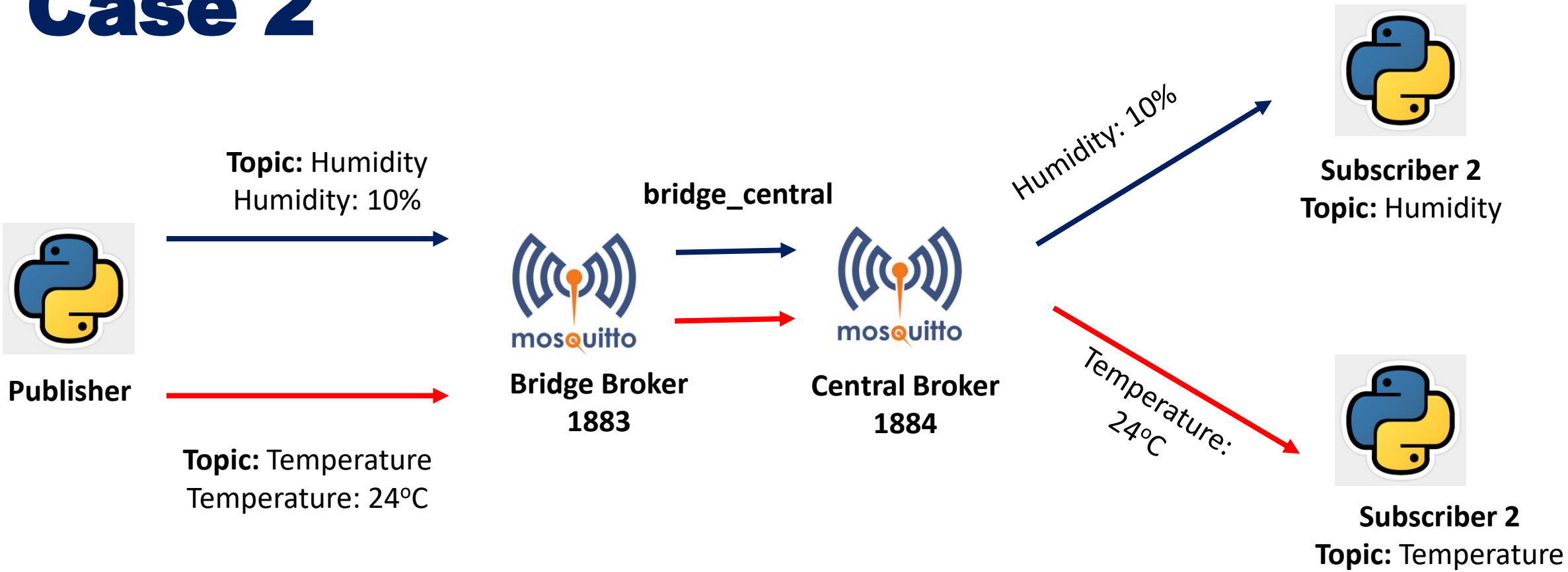
mqqt_sub_humid.py

```
def run():
    client = connect_mqtt()
    subscribe(client)
    client.loop_forever()

run()
```

This is a blocking form of the network loop and will not return until the client calls disconnect(). It automatically handles reconnecting.

Case 2


Topic: Humidity
Humidity: 10%
bridge_central
Humidity: 10%
Subscriber 2
Topic: Humidity
Publisher
Bridge Broker
1883
Central Broker
1884
Topic: Temperature
Temperature: 24°C
Temperature: 24°C
Subscriber 2
Topic: Temperature

Case 2

Central Broker
1884

```
allow_anonymous true

port 1884

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

log_dest stdout

include_dir /etc/mosquitto/conf.d
```

Case 2

Bridge Broker
1883

```
allow_anonymous true
port 1883

#connection name string
connection bridge_central

# type of bridge
topic srv/# out

#central broker address
address localhost:1884

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

log_dest stdout

include_dir /etc/mosquitto/conf.d
```

Connection name: used to log

Address of the central broker

Case 2

Bridge Broker
1883

```
allow_anonymous true
port 1883

#connection name string
connection bridge_central

# type of bridge
topic srv/# out

#central broker address
address localhost:1884

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

log_dest stdout
```

out = publish from the broker
in = receive from the remote broker
both = publish and receive

The general format is

topic topic pattern direction QOS local prefix/remote prefix.

Case 2

Check that port is correct!

```python
from paho.mqtt import client as mqtt_client

broker = 'localhost'
port = 1883
top_temp = "srv/temperature"
top_hum = "srv/humidity"
```

Publisher

Change the QOS of publish

```python
def publish(client,topic, msg):
    result = client.publish(topic, msg, qos=2)
    # 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}")
```

Case 2

Check that port is correct!

Subscriber 2
Topic: Humidity

```
broker = 'localhost'
port = 1884
top_temp = "srv/humidity"
client_id = f'python-mqtt-{random.randint(0, 1000)}'
```

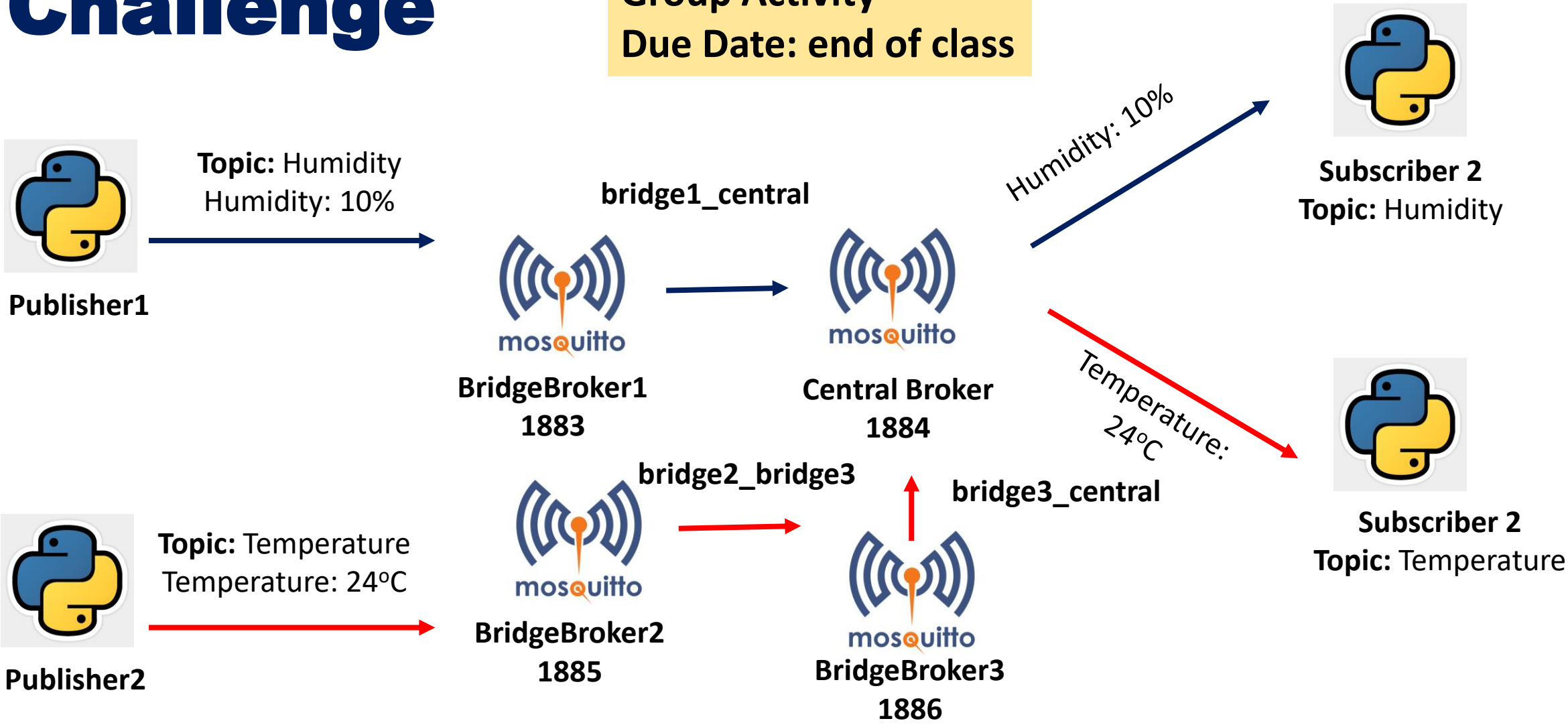
Change the QOS of SUBSCRIPTION

Subscriber 2
Topic: Temperature

```
def subscribe(client: mqtt_client):
    def on_message(client, userdata, msg):
        print(f"Received `{msg.payload.decode()}` from `{msg.topic}` topic")
    client.subscribe(top_temp, qos=2)
    client.on_message = on_message
```

Challenge

Group Activity
Due Date: end of class



Unit 6.5 (Extra)
MQTT Advanced

Tecnológico de Monterrey

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved