

Production Supporting Systems in Factories

ระบบสนับสนุนการผลิตในโรงงานอุตสาหกรรม

MQTT

MQTT

- MQTT is a Client Server publish/subscribe messaging transport protocol.
- It is light weight, open, simple, and designed so as to be easy to implement.
- Ideal for use in many situations
 - Machine to Machine (M2M)
 - **Internet of Things (IoT)**

Publish/subscribe pattern

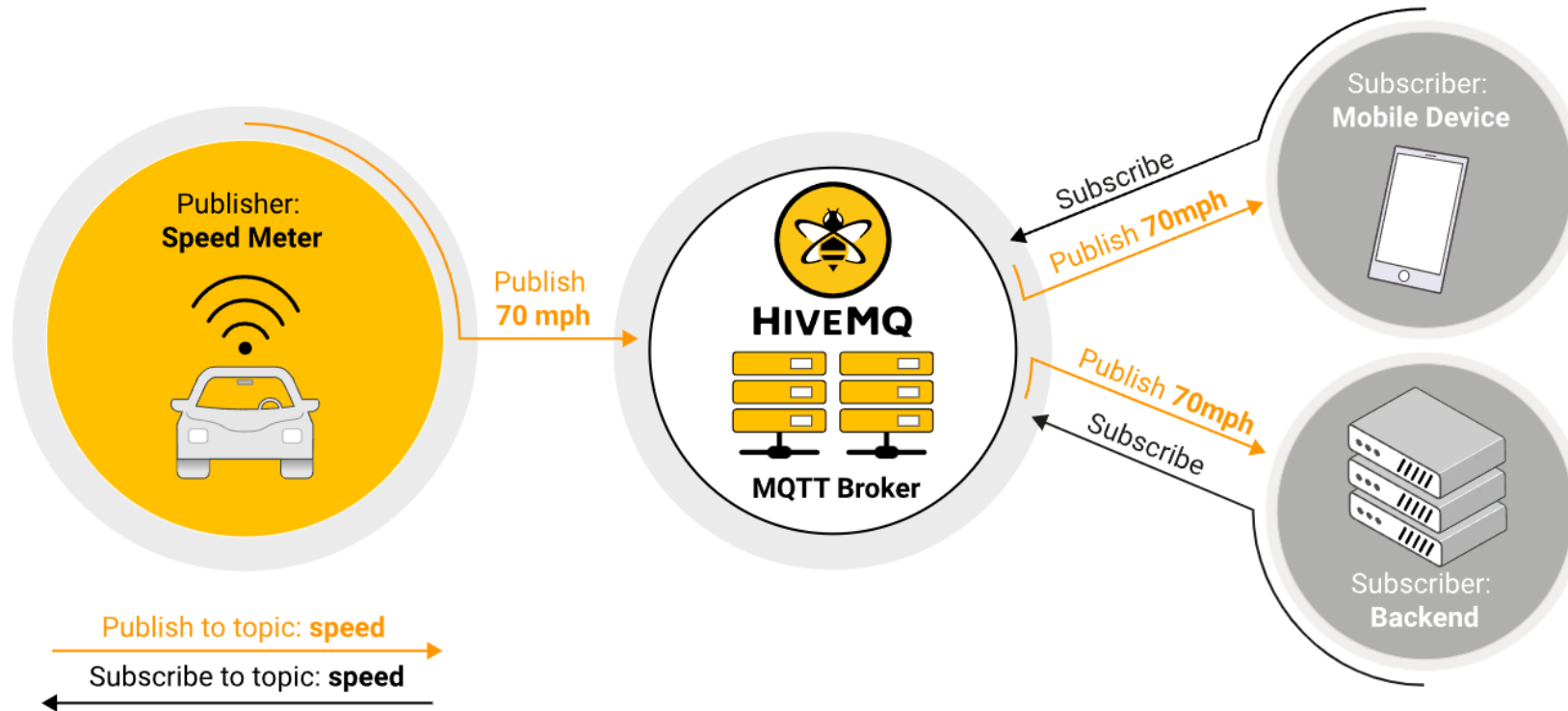
- The publish/subscribe pattern (also known as `pub/sub`) provides an **alternative** to traditional client-server architecture.
 - In client-server architecture, A client communicates directly with an endpoint.

Publishers and subscribers

- The `pub/sub` model decouples
 - a client that sends a message (the `publisher`) from
 - a client or clients that receive the messages (the `subscribers`).
- The `publishers` and `subscribers` never contact each other directly.
 - In fact, they are not even **aware that the other exists**.

Broker

- The connection between `publishers` and `subscribers` is handled by a third component (the `broker`).
- The job of the broker is to filter all incoming messages and distribute them correctly to subscribers.



Aspects of pub/sub architecture

- **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.

Topic

- `Topic` refers to an UTF-8 string that the broker uses to filter messages for each connected client.
- The topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator).

topic level
separator
↓
myhome / groundfloor / livingroom / temperature
└───┘ └───┘
topic level topic level

Topic examples

- Each topic must contain at least 1 character.
 - Actually, `/` is a topic.
- Topic string permits empty spaces.
 - `USA/California/San Francisco/Silicon Valley` is a valid topic.
- Topics are case-sensitive.
 - `myhome/temperature` and `MyHome/Temperature` are two different topics.

- Wildcards

single-level
wildcard
↓
myhome / groundfloor / + / temperature
|
only one level

- ✓ myhome / groundfloor / livingroom / temperature
- ✓ myhome / groundfloor / kitchen / temperature
- ✗ myhome / groundfloor / kitchen / brightness
- ✗ myhome / firstfloor / kitchen / temperature
- ✗ myhome / groundfloor / kitchen / fridge / temperature

multi-level
wildcard
↓
myhome / groundfloor / #

only at the end
multiple topic levels

- ✓ myhome / groundfloor / livingroom / temperature
- ✓ myhome / groundfloor / kitchen / temperature
- ✓ myhome / groundfloor / kitchen / brightness
- ✗ myhome / firstfloor / kitchen / temperature

Enough talk. Let's try it.

MQTT Terminal (Android)

- Broker URL/IP : 35.240.248.118
- Client ID : <<Your ID>>
- Port : 1883
- Publish Topic : test/main
- Subscribe Topic : test/main

8:41 87%

← Edit Server

Connection Name
prodsup

Client ID
my-id

Borker URL / IP
35.240.248.118

Port
1883

☐ SSL ☐ Web Socket

Enable user authentication

Publish Topic
test/main

☒ QoS 0 ☐ QoS 1 ☐ QoS 2

☐ Retained


Subscribe Topic
test/main

☒ QoS 0 ☐ QoS 1 ☐ QoS 2

Cancel Change

MQTTool (iOS)

- Host : 35.240.248.118
- Port : 1883
- Client ID : Leave blank

Host: 35.240.248.118 

Port: 1883 Clean Session: ☒

Client Id: MQTTTool-2059422473

Leave blank for unauthenticated access

Username:

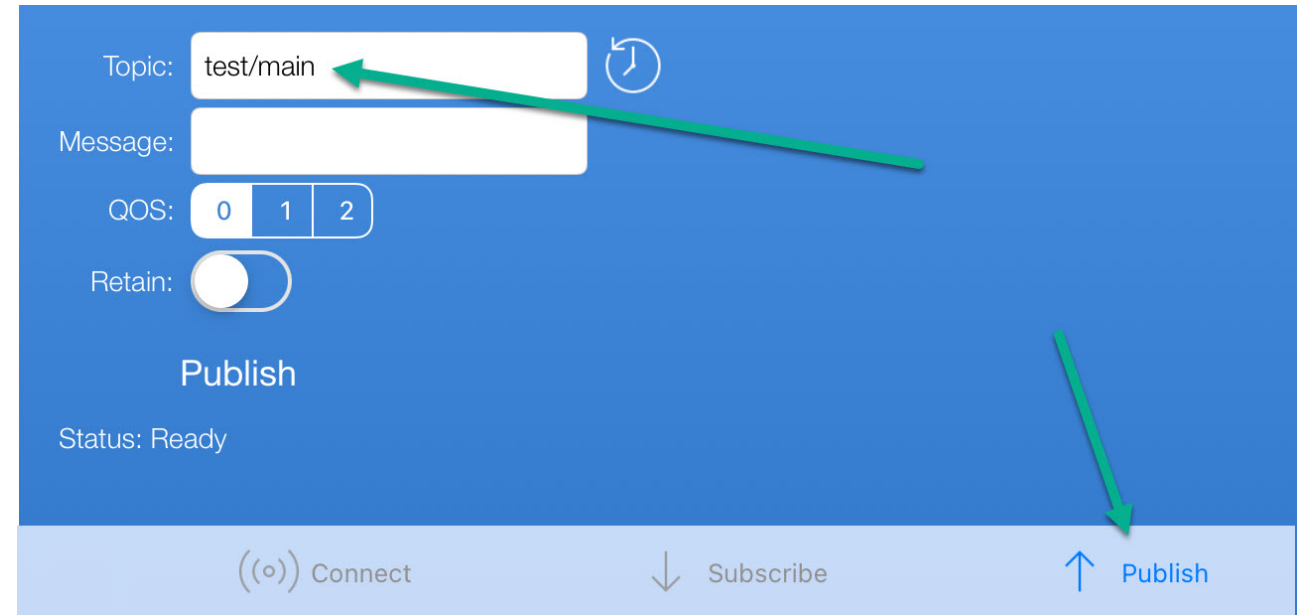
Password:

Disconnect Save Password ☐

Status: Connected to 35.240.248.118:1883

Click ₁₅

- `topic: test/main`



The image shows a screenshot of an MQTT client interface with a blue background. At the top, there is a 'Topic:' label followed by a text input field containing 'test/main'. A green arrow points from the right side of the interface to this input field. Below the topic field is a 'Message:' label followed by an empty text input field. Underneath the message field are three buttons for QoS: '0', '1', and '2', with '0' being the selected option. Below the QoS buttons is a 'Retain:' label followed by a toggle switch that is currently turned off. A 'Publish' button is located below the retain toggle. At the bottom of the interface, there is a light blue bar containing three buttons: '((o)) Connect', '↓ Subscribe', and '↑ Publish'. A green arrow points from the right side of the interface to the 'Publish' button. The status 'Status: Ready' is displayed above the bottom bar.

Topic: test/main

Message:

QOS: 0 1 2

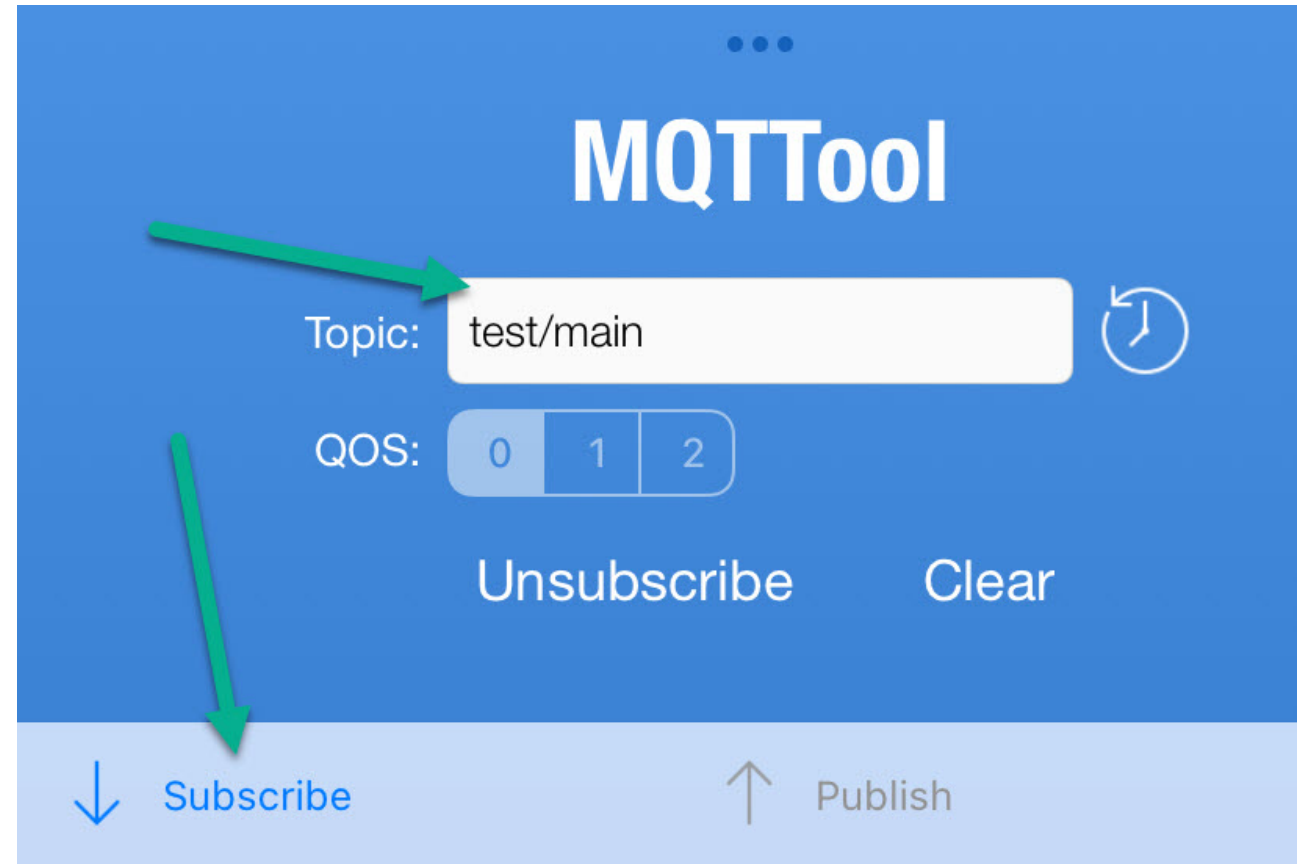
Retain: ☐

Publish

Status: Ready

Connect Subscribe Publish

- `topic: test/main`



Quality of service

- The Quality of Service (QoS) level is the guarantee of delivery for a specific message.

Levels of QoS

- **0 : At most once**
 - No guarantee of delivery (fire and forget)
 - Fastest
- **1 : At least once**
 - Guarantees that a message is *delivered at least one time* to the receiver.
 - Multiple delivery can occur.
- **2 : Exactly once**
 - Each message is *received only once* by the intended recipients.
 - Slowest

Setting QoS level

There are the two sides of message delivery:

- publishing client → broker
- broker → subscribing client

publishing client → broker

- The publishing client defines the qos level of the message.

broker → subscribing client

- The broker transmits the message to subscribing clients using the QoS level that each subscribing client defines during the subscription process.
- If the subscribing client defines a lower QoS than the publishing client
 - the broker transmits the message with the lower quality of service.

General use for QoS level

- 0: You don't mind if a few messages are lost occasionally.
- 1: You need to get every message and your use case can handle duplicates.
 - *Generally recommended.*
- 2: It is critical to your application to receive all messages exactly once.

Retained message

- A retained message is a normal MQTT message with the `retained` flag set to `true`.
- 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 message demo

- [Websocket Client](#)
- host : 35.240.248.118
- port : 9001 (Different)

Connection

● connected

⌵

Host

35.240.248.118

Port

9001

ClientID

clientId-sDP16mLSKq

Disconnect

Username

Password

Keep Alive

60

SSL

☐

Clean Session

☒

Last-Will Topic

Last-Will QoS

0

Last-Will Retain

☐

Last-Will Message

- Topc : test/temperature

Publish

Topic

test/temperature

QoS

1

Retain

☒

Publish

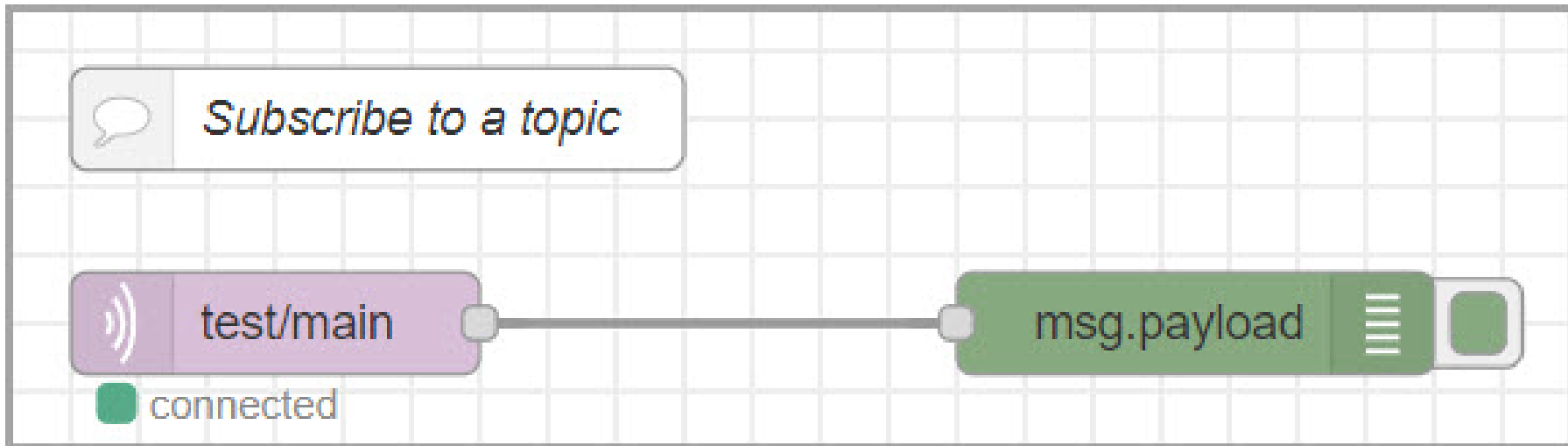
Message

30

Module 3-1: MQTT in Node-Red

Subscribe to a topic

- Flow
 - mqtt in, debug



- mqtt in node
 - Choose Add new mqtt-broker...
 - Click edit (Next page)
 - Topic : test/main
 - QoS : 1

Edit mqtt in node

Delete Cancel Done

Properties

Server ProdSup

Action ProdSup
Add new mqtt-broker...

Topic test/main

QoS 1

Output auto-detect (string or buffer)

Name Name

- Server : 35.240.248.118
- Port : 1883

Edit mqtt in node > Add new mqtt-broker config node

Cancel Add

⚙ Properties

🔖 Name ProdSup2

Connection Security Messages

🌐 Server 35.240.248.118 Port 1883

☒ Connect automatically

☐ Use TLS

⚙ Protocol MQTT V3.1.1

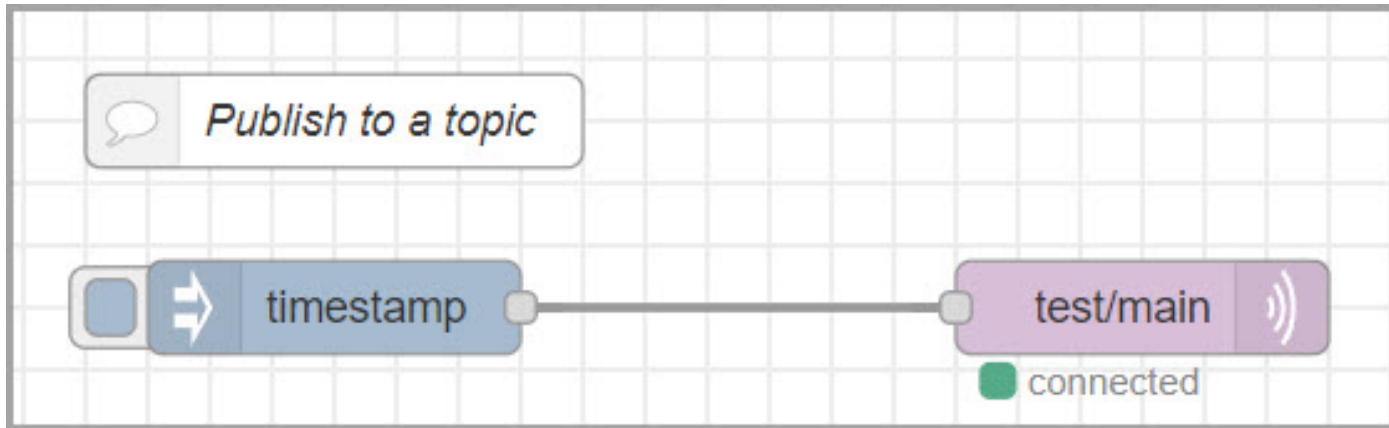
🔖 Client ID Leave blank for auto generated

💓 Keep Alive 60

📄 Session ☒ Use clean session

Publish to a topic

- Flow
 - inject, mqtt out



- mqtt out
 - Topic : test/main
 - QoS : 1
 - You may choose to Retain message.

Edit mqtt out node

Delete Cancel Done

⚙ Properties

🌐 Server ProdSup2

📄 Topic test/main

⚙ QoS 1 Retain true

🏷 Name Name

Tip: Leave topic, qos or retain blank if you want to set them via msg properties.