MQTT

Summary of CoAP

- In CoAP, the client (the application) has to send a request to the server (the device)
  - Application has to pull the sensor data from the devices
- Advantage: Communication only done when sensor data is really needed by the client
- Disadvantage:
  - Wasting bandwidth if data is needed periodically (e.g. every 5 min).
  - Wasting bandwidth if several clients want the same data

- On the next slides, we see MQTT, a protocol for a completely different architecture...

MQTT

- MQTT = Message Queue Telemetry Transport
- A Machine-to-Machine (M2M) protocol
- Developed by IBM in 1999, standardized in 2013

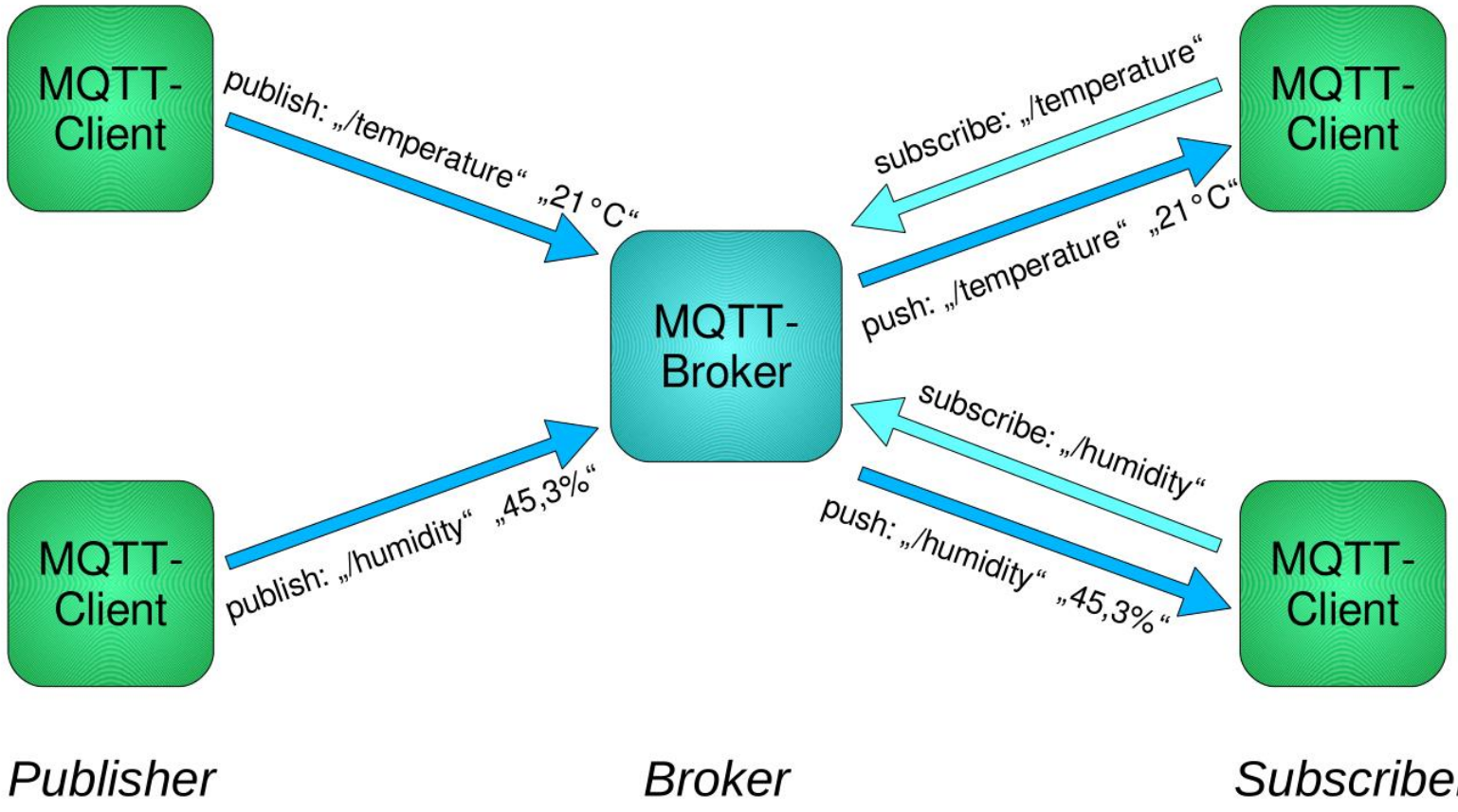
- On top of TCP/IP
  - Port 1883: MQTT
  - Port 8883: Secure MQTT (MQTT over SSL)

- Instead of a client-server pull architecture (e.g. HTTP or CoAP), we have a publisher-subscriber push architecture

Overview

The IoT devices

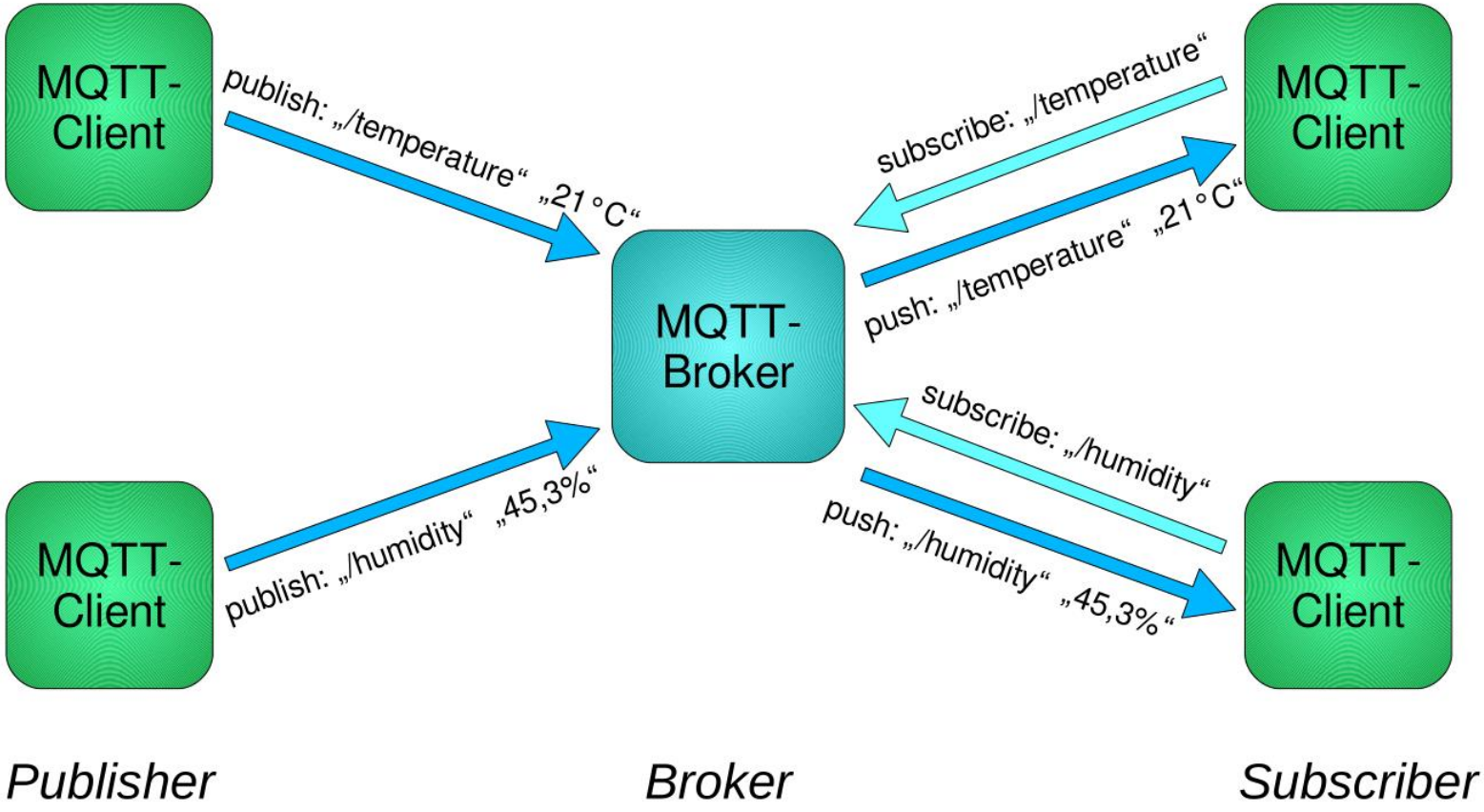
The applications using the sensor data



Picture by Uwe Berger, 2018.

Publishers

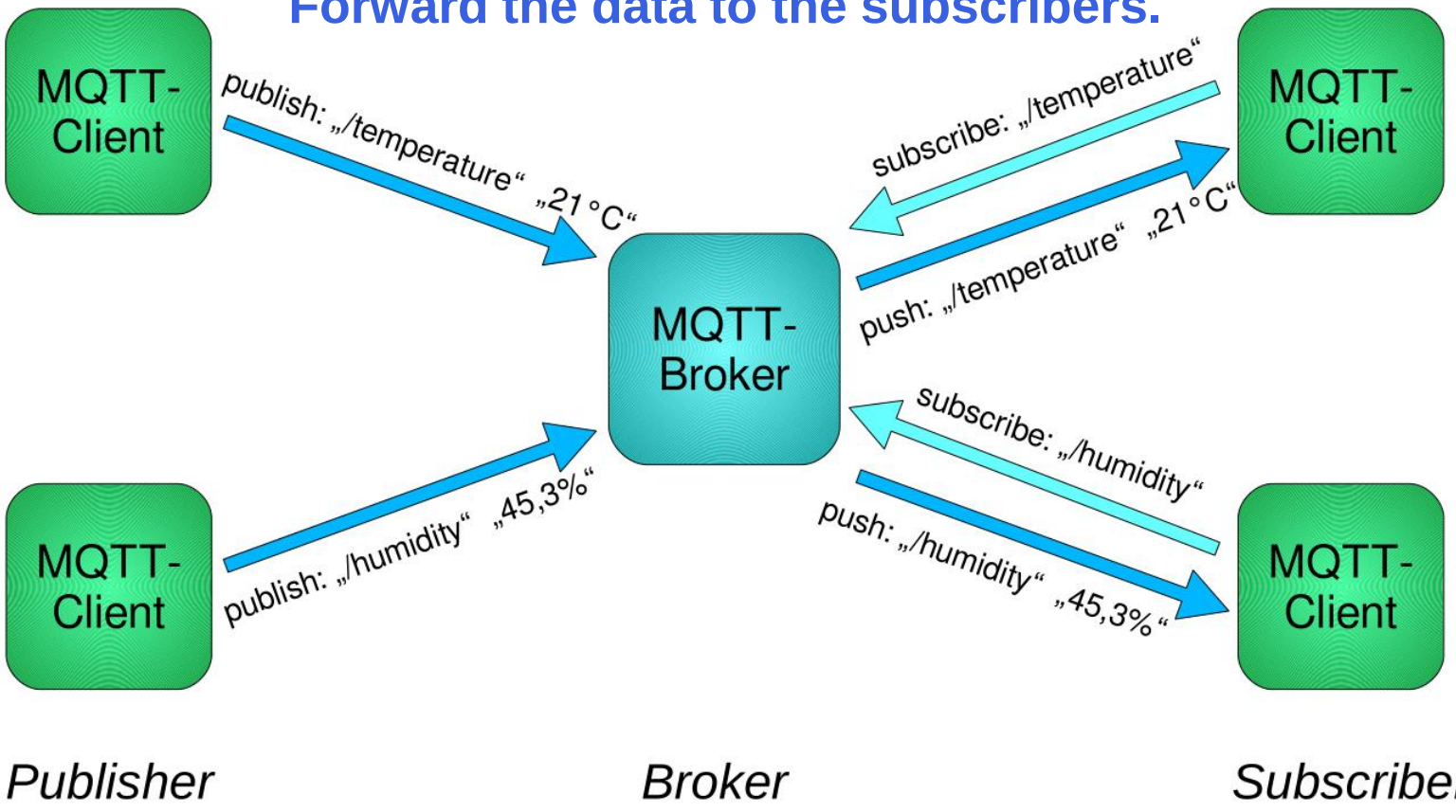
Publishers send data to the broker.
Data have a "topic": "/temperature", "/humidity"



Picture by Uwe Berger, 2018.

Broker

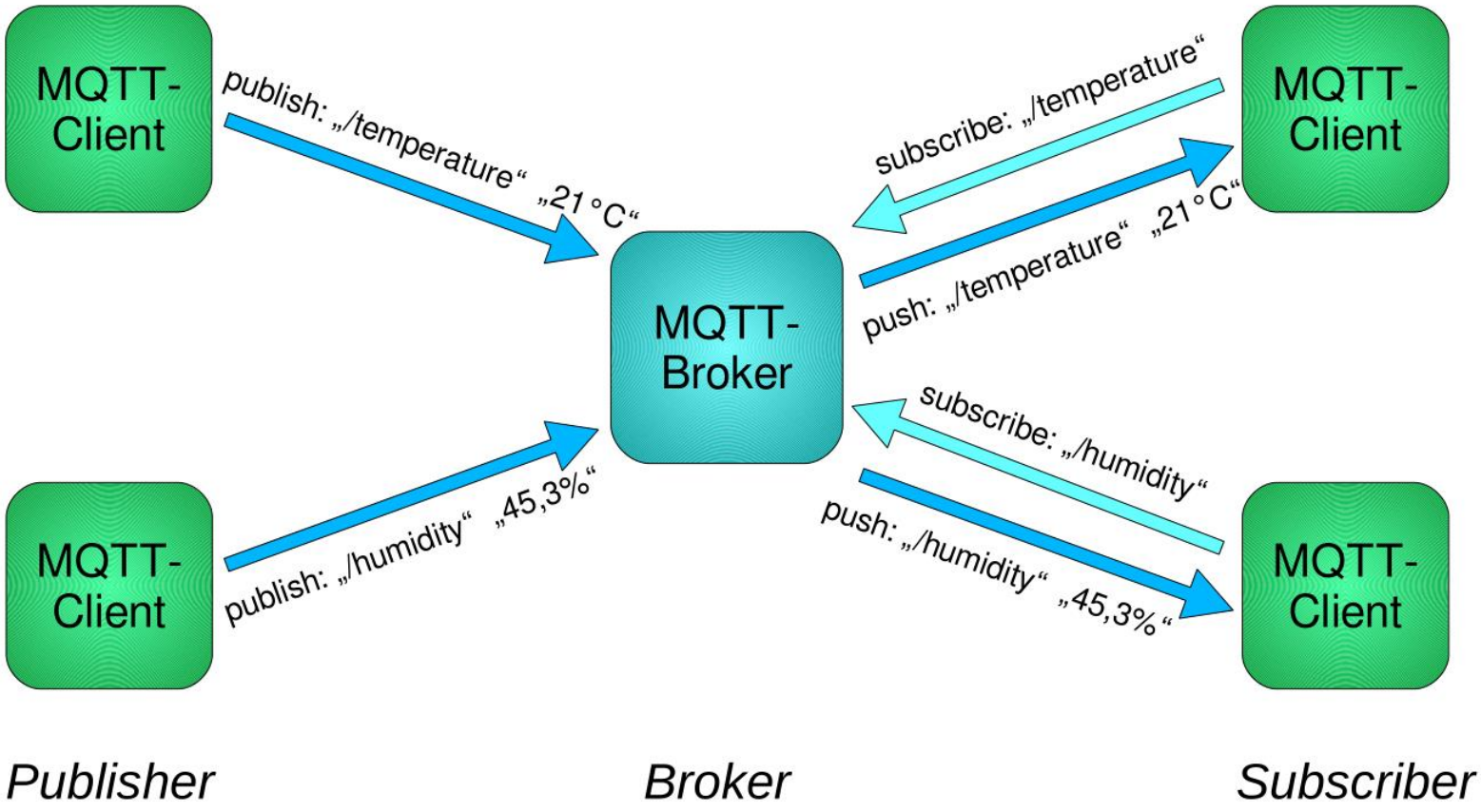
Brokers receive the data from the publishers.
Forward the data to the subscribers.



Picture by Uwe Berger, 2018.

Subscribers

Subscribers register at the broker with the "topics" there are interest in. Receive data from the broker.



Picture by Uwe Berger, 2018.

Topics

- Subscriber tell the broker for which topics they want to receive data
- Topics are organized in levels like filesystems:

```
sensors/temperature
sensors/humidity
ucl/barb/room10/temperature
ucl/barb/room21/temperature
Belgium/cars/1-ABC-123/speed
```

- Topic names are case-sensitive!
- Topics starting with "$" contain broker-related information (statistics, number of clients,...). Example:

```
$SYS/broker/messages/received
```

Topic Wildcards

- Clients can subscribe to similar topics using wildcards
- "+" stands for one topic level:

```
ucl/barb/+/temperature
```

matches

```
ucl/barb/room10/temperature
ucl/barb/room21/temperature
```

- "#" stands for one or more levels (only as last char):

```
ucl/#
```

matches

```
ucl/barb/room10/temperature
ucl/barb/room21/temperature
ucl/lavo/room51/temperature
```

Retained messages

- By default, messages are directly forwarded by the broker to the subscribers
  - A subscriber joining a topic at 13:10 will not get a message published at 12:50
- Messages marked as "Retained" are stored on the broker
  - Subscribers that join later will get them, too

Message format

- MQTT messages contain
  - a small fixed header (message type, flags, length)
  - optional header fields (depending on message type)
  - optional payload

- Topic names are UTF-8 strings
- No format specified for topic data. Can be anything. Text, picture, binary, encrypted data,...

- Very light-weight. Fixed header only 2 bytes if the message is short.

CONNECT message

- Used by the clients to connect to the broker. Broker replies with CONNACK.
- Contains
  - Unique Client ID (optional)
  - Username and password (optional)
  - Keep alive duration (optional)
  - Last Will topic and message (optional)

Last Will

- Publishers can configure a message that is automatically sent by the broker to the subscribers if the connection between the publisher and broker is lost
- Example for such a message:

```
"sensor/status OFF"
```

- If the broker doesn't see a new message from the client after keep-alive-duration $\times$ 1.5, it assumes that the client is disconnected $\rightarrow$ Last Will message sent
- Last Will is not sent if the client disconnects cleanly (with DISCONNECT message)
- Idle clients can send a PINGREQ message to keep the connection alive

PUBLISH message

- Used by the publisher or the broker to send data
- Contains
  - QoS level for this message
  - Retain flag
  - Topic name and data
  - Packet ID: used by QoS level 1 and 2

SUBSCRIBE message

- Used by the subscriber to subscribe to topics
- Contains:
  - Topic name (wildcards allowed)
  - Desired maximum QoS level

Quality of Service (QoS)

- Publishers and subscribers can define QoS for the communication with the broker

  Publisher → Broker

  Broker → Subscriber

- Three QoS levels:
  - QoS 0: "at most once"
    - Loss possible
  - QoS 1: "at least once"
    - No loss, but duplicates possible
  - QoS 2: "exactly once"
    - No loss, no duplicates

QoS Level 0

- Sender sends PUBLISH message
- Messages are not repeated. Losses possible.
- Cheap and unreliable

QoS Level 1

1. Sender sends PUBLISH message with packed ID
2. Receiver replies with PUBACK with same packet ID.
   Sender retransmits the message if PUBACK not received

QoS Level 2

1. Senders sends PUBLISH message with packet ID
2. Receiver saves packet ID. If the same PUBLISH message is received again, it can be detected by the packet id.
3. Receiver replies with PUBREC with same packet ID
4. Sender sends PUBCOMP. Receiver can discard the packet ID now.
5. Receiver replies with PUBCOMP. Sender can discard the original message now.

- Reliable, but high overhead

Implementation

- Many different client implementations for Java, Python, C, Javascript,...
- Lightweight broker implementations, like Eclipse Mosquitto
- Big distributed broker implementations like HiveMQ and CloudMQTT that can handle millions of clients

MQTT-SN

- MQTT is a simple protocol, but messages can be too big for 802.15.4 networks because of topic name
- MQTT-SN = MQTT for Sensor Networks
- Runs over UDP
- Messages can use 2-byte short topic name or a 1-byte topic ID instead of the full topic name
  - Clients can register topic names
  - Gateway can be preconfigured with list of topic names
- QoS Level 3 (or -1): Publisher can publish without first connecting
  - Useful for devices that sleep often → no connect messages necessary.

MQTT-SN Gateway

- MQTT-SN needs a gateway that translates between MQTT-SN and normal MQTT

IoT device <----------> Gateway <---------> Broker

MQTT-SN MQTT

- Clients can sleep to save power
  - The gateway stores messages in the meanwhile
  - Messages are delivered when the client wakes up