Networking and Interfacing IoT Platforms
Lab 3
Machine-to-machine communication & ZigBee

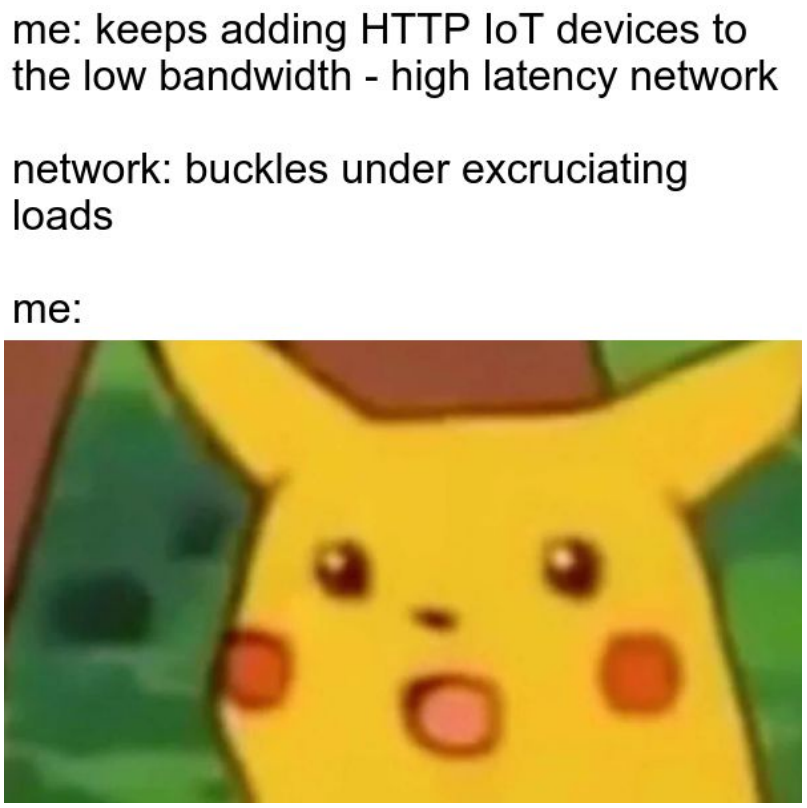
There are three parts to this assignment. Part one is a deep dive into the MQTT protocol, followed by part two which utilises MQTT in a practical assignment. Part three enhances part two by introducing ZigBee. Each part unlocks after completing the previous one. Once finished you should come and find a member of the teaching team to show off your work and to unlock the next part. **You have three weeks to complete this lab, each part should take you around one full lab week to complete.**

Note: Labs can be answered in Dutch or English, the important part is the contents. Paste lots of pictures/diagrams/… instead of writing walls of text. **Do include all your failures in the description as well as tips, recommendations or the help you received from the teaching team!** Remarks (Google Doc remarks) made by the teaching team must not be removed. Make a short movie after completion of a part and submit it as a separate file (no editing required!).

Part 1: Machine-to-machine communication

Assignment: MQTT broker

Up until now, all lab assignments mainly used HTTP(S) as their data protocol. The Internet of Things world, however, is not always a good fit for this protocol. That is why we often see MQTT being used for IoT projects.



MQTT stands for MQ Telemetry Transport, it is a client server publish/subscribe messaging transport protocol. Clients can publish application messages (i.e., data) to topics and subscribe to these topics to signal interest in receiving messages from other clients. The server, also known as a broker, accepts client connections, processes subscribe and unsubscribe events from clients and forwards application messages that match client subscriptions. The protocol is designed to be lightweight, open, simple and easy to implement.

For this assignment you will implement an MQTT broker using version 3.1.1 of the protocol in Python (CPython is enough; MicroPython compatibility is a nice extra). You will find the ISO/IEC specification on classroom; chapters 2,3 and 4 contain the most important details to get a minimal implementation up and running. When application messages are transported by MQTT, they have an associated Quality of Service (QoS) value of 0,1 or 2. For this assignment you do not need to implement QoS 2 and its associated control packets; however, your implementation should handle client publish and subscribe with QoS 2 correctly! Your implementation must only run on TCP/IP and does not need to consider security and/or authorization.

When you are done, come and get one of the teaching team members to demonstrate your implementation (max 10 minutes). Prepare a demo that shows off as many features as possible (e.g., different QoS levels, last will, subscription wildcards, error handling/robustness, …). Do not simply demonstrate the features, think about real life deployment scenarios that might use them. You can use an existing client implementation to test and demonstrate your broker. Keep in mind that we can and might test your implementation later on with an automated script to check if specification guidelines were followed correctly. Include a small readme document on how to start your application. By default your implementation must use the default ports as specified in the specification.

No creative extra and/or movie are required for this assignment. Only describe issues you encountered and any help/guidance you received from the teaching team.

To run the broker, use (from inside the broker directory):

```
>python3 main.py
```

Or run python directly and use:

```
from mqtt.mqtt_broker import MQTTBroker

# Change host to the IP of the local machine, or empty for localhost
broker = MQTTBroker(host="", port=MQTTBroker.PORT)
broker.start()
```

WARNING The implementation requires the Python `select` module to have `select.poll()` available that can be used on `socket` objects. This is **not** the case in Windows, so a *nix subsystem is required (e.g. Ububtu or WSL).
The implementation should work on the PyCom in MicroPython since only raw language features are used in addition to sockets.

Example Subscriber output:

```
root:broker# python3 main.py
[BROKER] Created at 127.0.0.1:1883
[BROKER] Starting to listen...
[BROKER] Accept client at 127.0.0.1:59347...
[BROKER] New connection with 127.0.0.1 on port 59347
[BROKER] CLIENT0@127.0.0.1:59347 requested CONNECT.
CLIENT0 Received <CONNECT conn=<clean, no will>, prot=b'MQTT', plvl=4, KeepAlive=60
s, id='b'mosq-TxvCfkFWyqDPWf1hsb''>
CLIENT0 renamed to 'mosq-TxvCfkFWyqDPWf1hsb'
mosq-TxvCfkFWyqDPWf1hsb Sent <CONNACK len=2>
[BROKER] Handling <mosq-TxvCfkFWyqDPWf1hsb@127.0.0.1:59347, Flags: <clean, no will>
, KeepAlive: 60s>
mosq-TxvCfkFWyqDPWf1hsb Received <SUBSCRIBE id=b'\x00\x02', topics=['dying' (0)]>
mosq-TxvCfkFWyqDPWf1hsb Sent <SUBACK id=b'\x00\x02', len=3>
mosq-TxvCfkFWyqDPWf1hsb is SUBSCRIBING to: 'dying' (0)
mosq-TxvCfkFWyqDPWf1hsb Subscribed to: ['dying' (0)]
mosq-TxvCfkFWyqDPWf1hsb Empty response?
[BROKER] Disconnecting mosq-TxvCfkFWyqDPWf1hsb: No packet received!
[BROKER] Destroying mosq-TxvCfkFWyqDPWf1hsb (0 left active)
```

Example Publisher output:

```
[BROKER] Accept client at 127.0.0.1:59454...
[BROKER] New connection with 127.0.0.1 on port 59454
[BROKER] CLIENT1@127.0.0.1:59454 requested CONNECT.
CLIENT1 Received <CONNECT conn=<clean, no will>, prot=b'MQTT', plvl=4, KeepAlive=60
s, id='b'mosq-glHLgrn3ouNf7MizL6''>
CLIENT1 renamed to 'mosq-glHLgrn3ouNf7MizL6'
mosq-glHLgrn3ouNf7MizL6 Sent <CONNACK len=2>
[BROKER] Handling <mosq-glHLgrn3ouNf7MizL6@127.0.0.1:59454, Flags: <clean, no will>
, KeepAlive: 60s>
mosq-glHLgrn3ouNf7MizL6 Received <PUBLISH<dup=0, QoS=0, ret=0> id=?, topic='test/me
', msg=b'published'>
[BROKER] PUBLISH to topic 'test/me': 'published'
mosq-glHLgrn3ouNf7MizL6 Received <DISCONNECT>
[BROKER] Disconnecting mosq-glHLgrn3ouNf7MizL6: Requested DISCONNECT.
[BROKER] Destroying mosq-glHLgrn3ouNf7MizL6 (0 left active)
```

The assignment folder `Opdracht_1` contains a video demonstrating a publisher and subscriber talking to each other, with various topics being matched and QoS values being used. All QoS settings were implemented, including the double exchange for QoS 2. When an ACK packet (or PUBREL, PUBREC, PUBCOMP etc.) is not send or received before the client is disconnected and the client is keeping its context (clean=0), the broker will resume sending or re-sending these packets.

A second video demonstrates our broker implementation working with a real client, a power socket that uses an MQTT subscription to know when to turn on or off. A MQTT publish client is used on the computer to publish ON and OFF packets, and after relaying these to the connected socket, we see the plugged-in lamp turn on or off.

Lab MQTT - Zigbee Deel 2

Een open source project dat als vervanging dient voor proprietary domotica gateways (vaak compatibel met slechts 1 technologie, zoals de Philips Hue Bridge) is zigbee2mqtt (zigbee2mqtt.io).

Dit project stelt je in staat om met goedkope hardware een software bridge te realiseren tussen een zigbee netwerk en een IP-gebaseerde back-end. Het project gebruikt daarvoor eenvoudige dongles die in staat zijn om zigbee berichten op te vangen en uit te sturen - gebaseerd op de Texas Instruments CC2531 chipset. Onder normale omstandigheden zouden we hiermee een volledige setup laten bouwen, maar gegeven de situatie is dit nu helaas niet mogelijk.

Daarom gaan we in het laatste deel van dit labo aan de slag met Bluetooth Low Energy. De feedback van de groepen wijst er op dat iedereen op zijn minst 1 toestel heeft dat hiervoor geschikt zou moeten zijn.

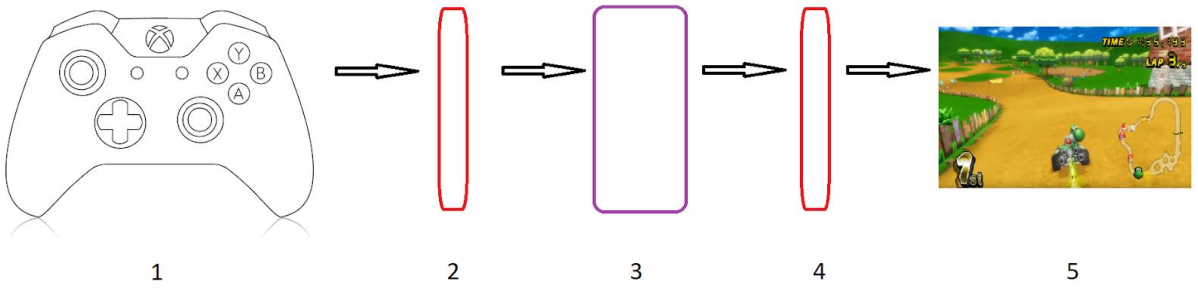
Concreet maak je in dit deel dus een software bridge die je BLE device koppelt aan je eerder ontwikkelde MQTT broker. Stel dat je een input device hebt, dan vorm je de gegevens van je input device om naar een JSON string en zorg je dat deze beschikbaar wordt via de broker. Je kan kijken in de Zigbee2MQTT documentatie voor inspiratie om een dergelijke message structuur op te bouwen (https://www.zigbee2mqtt.io/information/mqtt_topics_and_message_structure.html). In het Zigbee2MQTT project vind je ook een mogelijke indeling van de MQTT topics en hoe de devices daarin passen - het is niet verplicht deze te volgen, maar je kan er wel inspiratie uit halen. De situatie als je beschikt over een output device is analoog, alleen moet het device dan uiteindelijk BLE messages ontvangen om er mee aan de slag te kunnen. Let op, uiteraard moet je niet alle functionaliteit van Zigbee2MQTT implementeren; beperk je tot deze features die een bijdrage leveren tot het aantonen van het werken van je systeem - cf. het scenario hieronder.

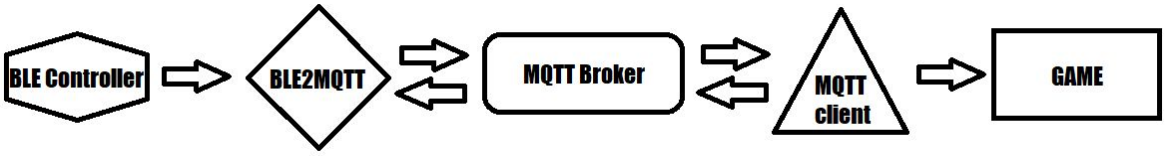
Naast je fysieke device maak je op zijn minst 1 virtueel (software-gebaseerd) device dat kan zorgen voor de (virtuele) aansturing of output. Als voorbeeld : stel dat je een BLE gebaseerde afstandsbediening hebt, maak dan een virtuele televisie die op de knoppen reageert. Heb je een BLE gebaseerde lamp, maak dan een on-screen schakelaar die de lamp kan aansturen.

Het systeem moet volledig compatibel zijn met de MQTT broker die je eerder geschreven hebt. Als tweede stap in dit labo vragen we je immers om de eigen MQTT broker te vervangen door een publieke variant. Er zijn tientallen van dergelijke services beschikbaar, kies er eentje en zorg dat je project hiermee ook compatibel is. Let er op dat sommige aanbieders een quotum hebben op het aantal berichten per uur/dag, kies dus zorgvuldig.

Gegeven de omstandigheden en de moeilijkere samenwerking zullen we voor dit labo 2 weken uittrekken (dus t.e.m. 3 april). Aan het einde verwachten we van jullie een live demo (week van 6 april, weliswaar via video conferencing), een korte presentatie (als begeleiding bij de live demo) alsook een rapportje van een 3-tal pagina's. Hierin beschrijf je vooral hoe het systeem werkt (packet structuur, hoe zitten topics in elkaar, welke functies ondersteunen je devices…); de bedoeling is niet om een zeer in-depth beschrijving van de werking te geven, focus op het aanhalen van de features die je implementatie ondersteunt.

Het idee: gamen in tijden van Corona





1. BLE Controller: Xbox One Controller verbonden met bluetooth
2. BLE2MQTT: MQTT Client die input events omzet naar MQTT-packets en published naar de broker
    - Verbonden met controller via BLE
    - Als een knop gebruikt wordt, stuurt hij een MQTT packet naar een specifieke topic voor die knop, vb `controller/stick/left/up`, `controller/button/X` enz.
3. MQTT Broker: Broker (kan eender welke broker zijn)
4. MQTT Client: MQTT Client die packets ontvangt en commando's omzet naar game-input.
    - Vraagt een subscription op de knoppen van de controller die hij wilt gebruiken, vb `controller/trigger/left`, `controller/button/B`, ... of een wildcard zoals `controller/stick/#`.
    - Zet deze ontvangen inputs om naar keyboard strokes of mapt deze naar een virtuele controller voor een spel.
5. Game: Ontvang virtuele keyboard presses of interfaced met virtuele controller.

Controller topics:

```
"""
Buttons/DPAD are [False, True],
Triggers are [0, 255],
Sticks are [-32768, 32767]
"""
X         = "button/X"
Y         = "button/Y"
A         = "button/A"
B         = "button/B"
LT        = "button/trigger/left"
RT        = "button/trigger/right"
LTHUMB    = "button/thumb/left"
RTHUMB    = "button/thumb/right"
LSHOULDER = "button/shoulder/left"
RSHOULDER = "button/shoulder/right"

LSTICK_X = "stick/left/X"
LSTICK_Y = "stick/left/Y"

RSTICK_X = "stick/right/X"
RSTICK_Y = "stick/right/Y"

DPAD_UP      = "dpad/up"
DPAD_RIGHT   = "dpad/right"
DPAD_DOWN    = "dpad/down"
DPAD_LEFT    = "dpad/left"

START = "button/start"
BACK  = "button/back"

States:
    ON  = "ON"
    OFF = "OFF"

All:
    BUTTONS   = "button/#"
    TRIGGERS  = "button/trigger/#"
    THUMBS    = "button/thumb/#"
    SHOULDERS = "button/shoulder/#"

    STICKS = "stick/#"
    LSTICK = "stick/left/#"
    RSTICK = "stick/right/#"

    DPAD = "dpad/#"
```

Als we een bepaalde button press ontvangen van de Xbox controller in de BLE relay, sturen we "ON" of "OFF" naar de bijbehorende topic van die knop (button, trigger, dpad). Voor de sticks, sturen we de lineaire waarde (tussen -32768, 32767).

Een subscribing client kan zich vervolgens subscriben op vb `"button/#"` om alle button presses te ontvangen en deze door te geven aan een eindapplicatie.

We hadden de topics ook andersom kunnen instellen, in de zin van een verdeling tussen links en rechts te maken, vb `controller/left/trigger`, `controller/right/button/A`… En een subscribing client zou dan kunnen subscriben op alle knoppen met `controller/+/button/#`. Maar we kozen uiteindelijk voor bovenstaande layout.

InputController

We ontvangen de input via een Xbox One Controller die verbonden is met de client-computer via Bluetooth. We hebben dit gedaan op een Linux-based client, met de gedachte dat we dit dan perfect kunnen porten naar een Raspberry-Pi.
Wanneer een input device met joysticks verbonden wordt met een Linux-systeem, wordt er een stream geopend op de locatie `/dev/input/jsX`, waarbij X aangeeft welk device het betreft.
De eerste stap bestaat uit het openen van de stream die toebehoort aan de controller. We doen dit door de file `'/proc/bus/input/devices'` uit te lezen, waarin alle verbonden apparaten worden opgelijst. Hieruit kunnen we afleiden welke stream we moeten uitlezen voor de input controller.
Nadat we de controller en jsX identifier gevonden hebben, openen we het `/dev/input/jsX` bestand en lezen daar telkens de inhoud van uit dat een event packet voorstelt.

Per gelezen input worden de volgende data uit de stream gelezen:

- Timestamp
  Tijd van event. Kan gebruikt worden om buffering op te zetten
- Value
  De waarde die ingelezen wordt. Voor een knop is dit 0 of 1. Voor axis kan deze waarde variëren tussen -32767 en 32767
- Type
  Geeft aan wat voor input er gelezen werd (button of axis)
- Code
  Geeft aan welke input gelezen is (welke knop, welke axis)

Momenteel is er wel een probleem dat de mapping van de verschillende codes kan verschuiven. Dit valt te wijten aan het feit dat Linux origineel geen controllers ondersteunt. Wanneer we echter de controller volledig disconnecten met het systeem en terug connecteren (via de ingebouwde bluetooth connectie) krijgen we de volgende mapping:

- Buttons (type == 1)
  - 0: A
  - 1: B
  - 3: X
  - 4: Y
  - 6: LB
  - 7: RB
  - 11: Start
  - 13: LTHUMB
  - 14: RTHUMB

- Axis (type == 2)
    - 0: Left stick - X axis
    - 1: Left stick - Y axis
    - 2: Right stick - X axis
    - 3: Right stick - Y axis
    - 4: Right trigger
    - 5: Left trigger
    - 6: DPad - X axis
    - 7: DPad - Y axis

Afhankelijk van welk type het is, nemen we de goede topic en vormen de value om naar iets wat we kunnen doorsturen in een MQTT-pakket. Voor knoppen worden 0 en 1, OFF en ON, voor triggers casten we de waarde tussen 0 en 255 naar een enkele byte en voor sticks casten we de -32768 tot 32767 waarden naar een signed byte reeks van lengte 4.

BLE2MQTT

De `BLE2MQTT` client opent de Bluetooth file stream in Linux, connect met de broker start met het uitlezen van de file stream. Elk valid inkomend event van de Bluetooth stream, wordt gepubliceerd op de corresponderende topic naar de broker. Aangezien elke knop wordt ingelezen, worden ze ook allemaal gepubliceerd.

Broker

We hebben ons systeem getest met onze eigen broker uit deel 1. Deze had Bram gehost op zijn server zodat William er verbinding mee kon maken. Na een test bleek onze implementatie echter erg traag te werken, wellicht voor het vele printen naar de console, maar alle functionaliteit werkt wel.

We hebben daarna ook een andere broker implementatie getest (Eclipse MQTT Broker). Die werkte veel sneller en hebben we daarna dan ook gebruikt voor de eigenlijke demo.

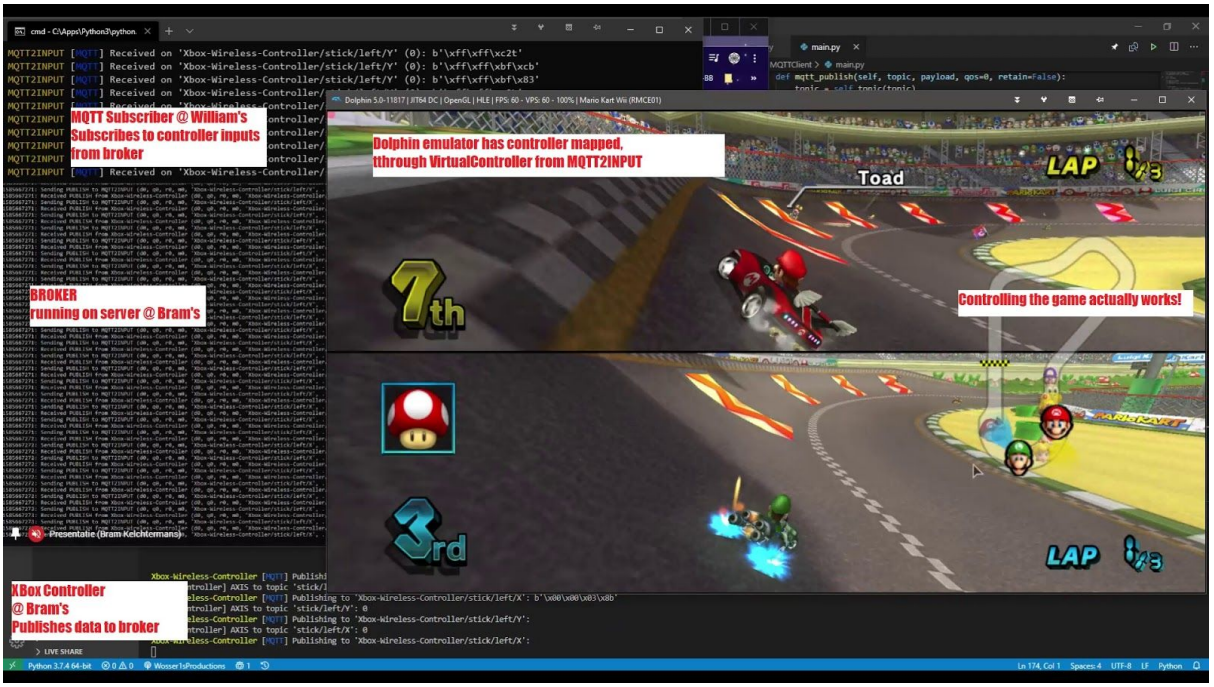
MQTT2INPUT

De ontvangende `MQTT2INPUT` client, opent een connectie met de broker en abonneert op enkele wildcard topics: `Controller.All.BUTTONS`, `Controller.All.STICKS` en `Controller.All.DPAD` (`<naam>/button/#`, `<naam>/stick/#` en `<naam>/dpad/#`). Daarbuiten emuleert hij ook een virtuele controller met pyxinput. Deze module gebruikt de ScpVBus driver om een virtuele controller te maken in Windows, die de module vervolgens kan aansturen. De driver is te vinden in de tools map.

Na het ontvangen van een bericht op één van de topics, wordt de waarde doorgestuurd naar de corresponderende virtuele knop op de controller. Voor de gewone knoppen, wordt de waarde True of False in plaats van ON en OFF, voor de sticks en triggers blijven de waarde gewoon hetzelfde en voor het dpad houden we bij welke knoppen reeds ingedrukt waren. Het dpad werkt namelijk met een bitmask voor elke knop, zodat verschillende knoppen tegelijkertijd ingedrukt kunnen blijven.

Omdat we een virtuele controller maken op deze manier, wordt deze ook effectief als input device gezien in Windows, en is dus bruikbaar voor elke applicatie die een controller als input kan krijgen.

Demo



Links onderaan de console van BLE2MQTT op de pc van Bram, links midden de output van de Linux MQTT broker die op de server van Bram draait en links boven de MQTT2INPUT client die lokaal bij William runt. Inkomende MQTT pakketten worden doorgegeven van de client naar de virtuele controller.

We hebben alle knoppen reeds gemapt in Dolphin emulator en in de screenshot hierboven controleren we allebei een personage uit Mario Kart Wii, waarbij William lokaal mouse en keyboard gebruikt en Bram remote zijn BLE XBox One controller. Een video wordt ook toegevoegd bij inzending.