Ghent University

Cloud Applications and Mobile (E736010)

Lab 4: Exploring Integration Patterns with Bluetooth, CoAP, and MQTT

Professor:

Prof. dr. ir. Sofie Van Hoecke

Assistants:

Dr. Jennifer B. Sartor

Ing. Cédric Bruylandt

2024 - 2025

UNIVERSITEIT GENT

Table of contents

1 Introduction

In this lab, We will look into the **gateway integration pattern** (see Figure 1), the **direct integration pattern** and the **cloud integration pattern**.

First, we will explore the **direct integration pattern**. Although the ESP32 does have WiFi capability, hosting an HTTP server directly requires a lot of power. Many battery-powered devices need to rely on low-power protocols to communicate, such as ZigBee or Bluetooth. We will use Bluetooth and Bluetooth Low Energy (BLE) to connect to our ESP32. Using Bluetooth, we can send sensor data to a smartphone or to another ESP32 (as shown in our IoT stack in Figure 2 and Figure 3, which is specific for this lab). Bluetooth is mostly used for short-distance communication between devices, for example, to connect a mouse wirelessly to a laptop, or to send data from a smartwatch to a computer or smartphone. This could also be used for mobile payments, tags to keep track of luggage locations in airports, autonomous vehicles, home automation systems, and remote monitoring of patients with heart conditions, to name a few use cases.

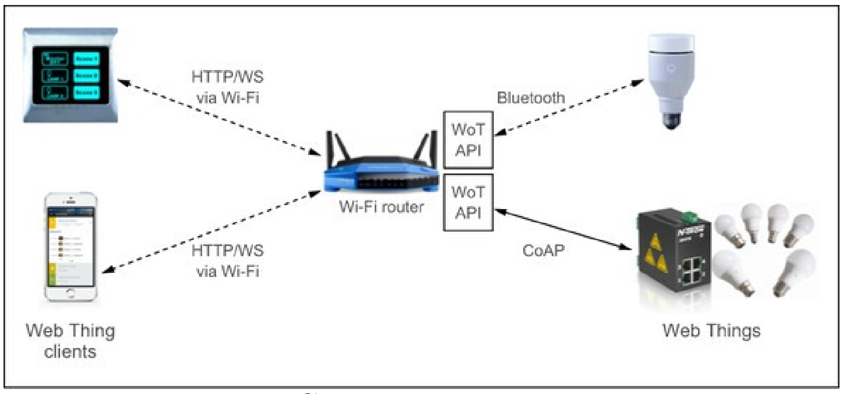


Figure 1: Gateway integration pattern

Next, we will experiment with the **gateway integration pattern**. We will use the Constrained Application Protocol (CoAP) protocol to communicate between our ESP32 microcontroller and the RPi (shown in the green "Sensors" box in Figure 2). The ESP32 has WiFi built-in, which makes it easier to communicate with other Things. We will set up a CoAP server on our RPi, and the ESP32 will act as the CoAP client. The ESP32 will gather sensor data and send it to our gateway: the RPi. We need our gateway to translate data coming over CoAP, which uses UDP, into a REST Web of Things (WoT) API that any HTTP client can use. The RPi can then send that data to the ingress API that we made in Lab 3. With this pattern, we can see that low-powered devices can save battery by passing their data on to a more powerful Thing that can make that data accessible to the world.

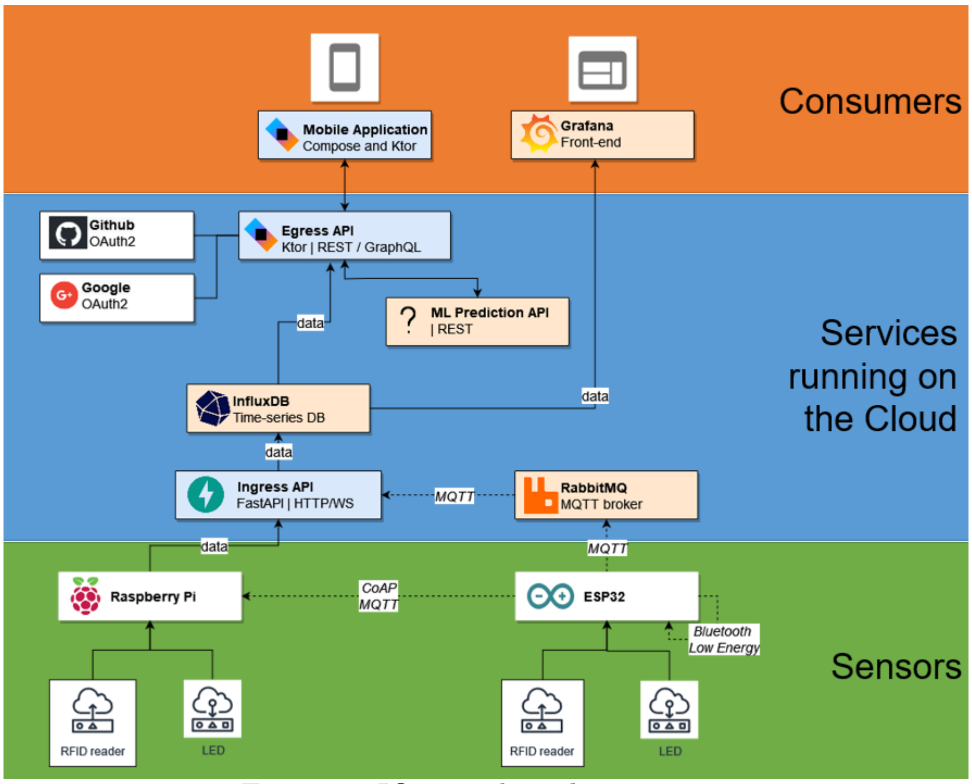


Figure 2: IOT-stack with categories

Finally we will explore the **cloud integration pattern**. First, we will get familiar with MQTT by having the ESP32 publish sensor data to an MQTT broker. We will set up that broker on the RPi, but the RPi still has to subscribe to the same topic to receive the sensor data from the broker. Then we'll have a new task that instead uses an MQTT broker set up in our cloud environment. The ESP32 will publish the sensor data to the MQTT broker running in the cloud, which requires our microcontroller to be accessible via internet protocols (TCP/IP) and be able to host an HTTP server. Luckily, the ESP32 is powerful enough to be able to do this. Then you will modify your ingress API that is running in the cloud to subscribe to that data and receive it from the MQTT broker, then store it in the database. With this pattern, we explore having our microcontroller send data to the cloud, not through the RPi but by using MQTT. The communication via MQTT can also be seen in Figure 3, as well as the MQTT broker running in the Cloud.

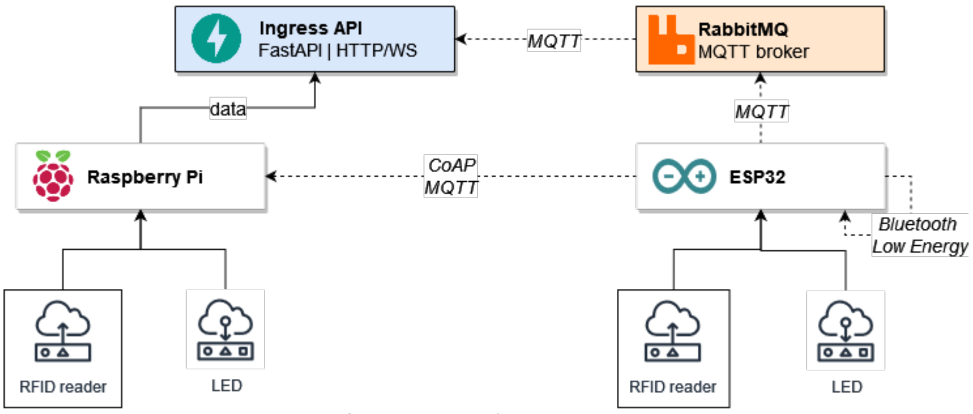


Figure 3: IOT-stack of what we do in Lab4

Within this lab session, you will perform several tasks that have to be documented in your lab report. Give the lab report a structure that mimics the structure of this document.

2 Goals

1. Send data from the ESP32 via Bluetooth Classic and Bluetooth Low Energy (BLE).
2. Send data from the ESP32 to the RPi using CoAP.
3. Let the RPi act as a gateway and pass incoming data from CoAP to your Ingress API.
4. Publish data from the ESP32 to a broker via MQTT and have the RPi subscribe to that data.
5. Modify our Ingress API to receive data from an MQTT broker in the cloud, which will receive data from the ESP32.
6. Implement integration patterns for connecting Things to the web.

3 Material

3.1 Own material

- Laptop & charger
- Smartphone & charger
- Ethernet cable (optional)
- **Micro-USB cable required for the ESP32**

3.2 Borrowed material

3.2.1 Material available in classroom

- Screen with HDMI (& power cable (USB to barrel jack))
- HDMI cable
- Keyboard
- Mouse

3.2.2 Borrowed material handed out

- Raspberry Pi 3 Model B(+)
- EU Power Supply (2.5A 5.1V)
- SD-card (16 GB) + (SD-adapter)
- ESP32 microcontroller (WROOM JOY-iT or DOIT)
- RFID reader RC522
- RFID tags
- Resistors (3x 1KΩ or 3x 1.2KΩ)
- Web of Things sensorkit
  - (Humidity and Temperature Sensor (DHT22))
  - (PIR Motion Sensor)
  - Breadboard
  - LEDs

  - 10K Ohm Resistors
  - 330 Ohm Resistors
  - M/M Jumper Wires
  - M/F Jumper Wires

4 Software

- VNC Viewer Installation below in Section 5.4, **only when we come together in the lab** because of the 2 week free trial.
- Bluetooth Smartphone app: Serial Bluetooth Terminal by Kai Morich (Android)
- (optional) Bluetooth Smartphone app for BLE: nRF Connect for BLE (Android) or nRF Connect for BLE (iOS)
- (optional) Bluepy library Installation below in Section 5.3
- CoAPthon library Installation below in Section 5.1. Install **CoAPthon3**!
- CoAP simple library Installation below in Section 5.2. Install **"CoAP simple library" by Hirotaka Niisato, version 1.3.13**, as explained below.
- Mosquitto Installation below in Section 5.1
- PubSub client for Arduino Installation below in Section 5.2
- **Optional** Paho-mqtt in Section 6.8 for bonus task8.

5 Preparation

5.1 CoAP and MQTT libraries to install on the RPi

We will set up a CoAP server on the RPi to listen for incoming data. For this we will use the CoAPthon library, but we will install the version that works with Python3. On your RPi, install this library:

```
pip3 install CoAPthon3
```

Then install the Mosquitto broker on the RPi to allow you to receive MQTT messages. You don't need to set up access with a username and password, no authentication is fine.

5.2 Arduino IDE Libraries to Install

In the Arduino IDE, go to Tools > Manage Libraries. Search for "CoAP simple library" by Hirotaka Niisato and **install version 1.3.13**. If you want to know more about this library, check out the Github page.

Finally, download the latest release of the PubSub client for Arduino from this GitHub repository. These steps explain how to import libraries from a zip file into the Arduino IDE. This enables the ESP32 to talk with the MQTT broker.

5.3 (Optional) Bluetooth Libraries to Install on the RPi

These libraries are only necessary if you do the bonus task4 below.

Re-synchronize the package index files from their sources by executing the following command on the RPi:

```
sudo apt-get update
```

Install the necessary software:

```
sudo apt-get install pi-bluetooth
sudo apt-get install bluetooth bluez blueman
```

Reboot your Raspberry Pi (Menu > Shutdown... > Reboot) or `sudo shutdown -r now`. Don't skip this step. If everything went well you should be able to see a Bluetooth manager (Menu > Preferences > Bluetooth manager).

You can install a library on the RPi to allow it to communicate with a Bluetooth Low Energy device. Install bluepy on the RPi.

5.4 VNC Viewer Installation

We recommend that you install **VNC Viewer** on your laptop so that you have a GUI view of your RPi instead of just the command line. This tool only offers a **2-week free trial**, so we recommend that you install this only when the lab actually starts and we do the practical session in person. This tool helps you set up Bluetooth if you want to do the optional task4 below. The guide on how to install this is in appendix A. With this free software you can establish a connection and control your computer remotely using a GUI interface. As long as your laptop and your RPi are connected to the same wifi network, you can control your RPi graphically from your laptop without having a separate screen, mouse, or keyboard.

6 Tasks

You do not have to do these tasks in order, as many of them don't depend on other tasks. However Task6 has to come after Task5 and Task9 should come after Task7. Don't forget to **take short videos to show you got each task working**.

6.1 #Task1 Send and receive data with Bluetooth Classic

We will now send data from the ESP32 via Bluetooth to your smartphone. **Note that Apple smartphones DO NOT support Bluetooth Classic.** For students with an iPhone, we recommend that you work together with a classmate with an Android phone for this task, or use Bluetooth Low Energy to communicate between your ESP32 and your iPhone.

You will need to install the serial library to complete these tasks. The code to communicate via Bluetooth with your ESP32 is shown in Listing 1. We first create an instance of the BluetoothSerial class, and begin communication. We also initialize normal serial communication's baud rate to 115200. Then, in the loop function, we read data from BluetoothSerial and print it to the Serial Monitor, and read data from the Serial Monitor and write it to BluetoothSerial. The idea is to send messages to and from your smartphone, so make sure you have a bluetooth app on your phone to test this with.

```
#include "BluetoothSerial.h"

/* Check if Bluetooth configurations are enabled in the SDK */
/* If not, then you have to recompile the SDK */
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

BluetoothSerial SerialBT;

void setup() {
  Serial.begin(115200);
  /* If no name is given, default 'ESP32' is applied */
  /* If you want to give your own name to ESP32 Bluetooth device, then */
  /* specify the name as an argument SerialBT.begin("myESP32Bluetooth"); */
  SerialBT.begin();
  Serial.println("Bluetooth Started! Ready to pair...");
}

void loop() {
  if (Serial.available())
  {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available())
  {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

Listing 1: esp32_bluetooth.ino

Within the Arduino IDE, choose the correct board and port under the Tools menu. We choose

”NodeMCU-32S”. Then upload the code to your ESP32 using the Arduino IDE. Open the Serial Monitor and set the correct baud rate.

Now turn on Bluetooth on your smartphone and scan for Bluetooth devices. You should see ESP32 listed under ”Available devices”. Your phone will ask if you want to pair with ”ESP32” and you say yes or OK. There is no password. Open the ”Serial Bluetooth Terminal” app and click on the three horizontal bars on the top left corner of the screen, as shown in Figure 4. Select the ’Devices’ tab, and select ESP32 from the list. Now click on the ’link’ icon at the top of the window to connect to the ESP32 via Bluetooth, shown on the left in Figure 5. The app will display the status as ’Connecting to ESP32...’, and if the connection succeeds, it will display ’Connected’, as in Figure 5.

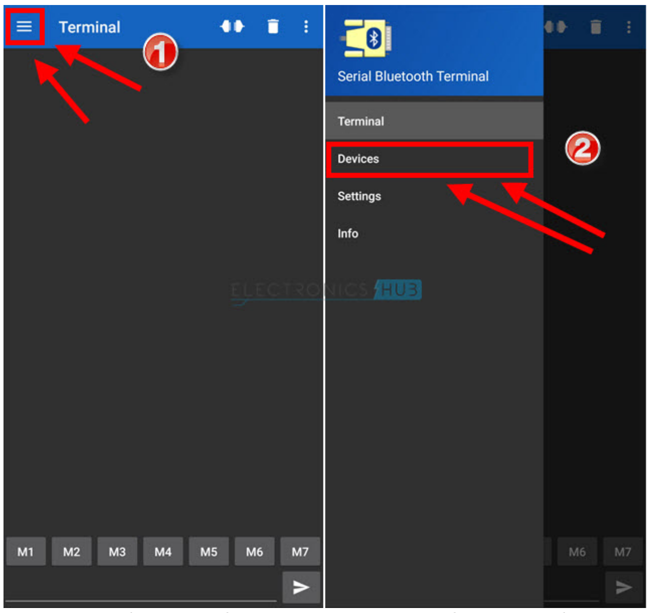
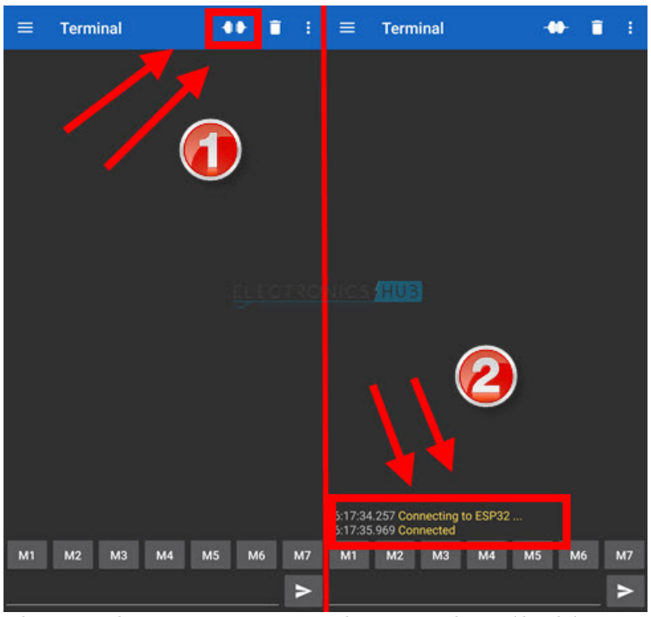


Figure 4: In Bluetooth app on smartphone, select Devices



```
5:17:34.257 Connecting to ESP32 ...
5:17:35.969 Connected
```

Figure 5: In Bluetooth app on smartphone, select ’link’ icon and connect.

At the bottom of the window in your smartphone app, there is a space where you can input text. Try to type a message to send to your ESP32, and click on the send button. The app then lists, next to a timestamp, what data you sent. Go back to your Serial Monitor in the Arduino IDE and make sure that data was received correctly by your ESP32. Now try to send data back from your ESP32 to your smartphone. Make sure you **take a small video** of this task being accomplished successfully.

6.2 #Bonus Task2 Control a LED on the ESP32 using Bluetooth Classic

Using the knowledge from Lab1 and Task1, modify the code that sends and receives data via Bluetooth in Listing 1 to have your smartphone control the LED on the ESP32. Your smartphone will send either a 1 or a 0 to the ESP32. When a '1' is sent, the ESP32 should turn on its LED, and when a '0' is sent, the light should be turned off.

6.3 #Task3 Send and receive data with Bluetooth Low Energy

Now we will make a Bluetooth Low Energy (BLE) connection between two ESP32 boards. **You will have to pair with another student for this task**. One ESP32 will be the server, and the other will be the client. The BLE server advertises characteristics that contain sensor readings that the client can read, and initiates the connection. The ESP32 client reads the values of those characteristics and displays them. Remember that each service and characteristic has a UUID (Universally Unique Identifier).

6.3.1 ESP32 BLE Server

We will have one ESP32 serve as the BLE server, which will advertise a service that contains a characteristic. That characteristic holds the value of our sensor data, a.k.a., a reading from the RFID reader. However, right now the value in the code is set to a "Hi" message.

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>

// See the following for generating UUIDs:
// https://www.uuidgenerator.net/

#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  BLEDevice::init("ESP32 AS A BLE");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  BLECharacteristic *pCharacteristic = pService->createCharacteristic(
                                         CHARACTERISTIC_UUID,
                                         BLECharacteristic::PROPERTY_READ |
                                         BLECharacteristic::PROPERTY_WRITE
                                       );

  pCharacteristic->setValue("Hi,other ESP32 here is your data");
  pService->start();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
  Serial.println("Characteristic defined!");
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(2000);
}
```

Listing 2: esp32_ble_server.ino

The code, shown in Listing 2, uses the BLE class to create a server, a service, and a characteristic. In the setup code, the value of the characteristic is set, which is the data to be communicated to the other Thing connected via BLE. Then the BLE server starts advertising.

Upload this code to one of the ESP32s. To test that the BLE is working, you can try using a BLE app on your smartphone to see if it can receive this "Hi..." string from the BLE server (optional). You would have to scan for BLE devices, connect with your ESP32 server, and then check its characteristic.

Next you will have to setup another ESP32 as the client to make sure this ESP32, working as the server, is working.

6.3.2 ESP32 BLE Client

A second ESP32 will be the client in our scenario. It will access data from the server. It should provide the UUIDs of the characteristic and service that it wants to connect to on the server, and then it can access the specified services.

In the client code below, we first have the Universally Unique Identifiers (UUID) for a characteristic and a service at the start of the code. Looking at the setup function, the code initializes a BLE device, and gets a scanner. Then the code sets a callback class for when a new device has been detected, and starts scanning. Once an advertising BLE server is detected, the "onResult" function is called, which checks if the found device has the service that we are looking for, and turns the doConnect flag to true.

In the loop function, if the doConnect flag is true, the connectToServer function is called. This function creates a BLE client and tries to connect him to the remote BLE server. Then it gets a reference to the service in the BLE server, and then a reference to the characteristic in the remote BLE server. Then the value of the characteristic is read and printed. Back in the loop function, if there is still a connection to the BLE server, the client writes a new value to the remote characteristic.

You can upload the full client code to your Arduino IDE and test that the ESP32 acting as the server can send data to the ESP32 acting as the client.

6.3.3 Sending RFID data from the BLE Server to the BLE Client

While this code demonstrates BLE communication between a BLE server and client, no actual sensor data is sent. Connect your RFID reader to the ESP32 as in Lab1 so it can detect when a tag is scanned. Modify the code of the server to send RFID tag readings. Change the client code to repeatedly read this sensor data from the server, and print it out. I recommend using the notifyCallback function (in the provided file, but not shown above).

6.4 #Bonus Task4 Send data from ESP32 with BLE to the RPi

If you want to distinguish yourself, send sensor data from one ESP32 to the RPi. You might have to change the BLE server script running on the ESP32 slightly.

In order to program sending or receiving data via BLE with the RPi, you will need to use the bluepy library. You need to have this library installed. Then, write a small Python script for the RPi that listens to the incoming data from the ESP32. For your Python program, we need to know some details about this BLE device, like its address. If you use a Bluetooth app, it will tell you the BLE device's address. In your Python program, you will need to create a Peripheral object with that address, something such as "3c:71:bf:0d:dd:6a".

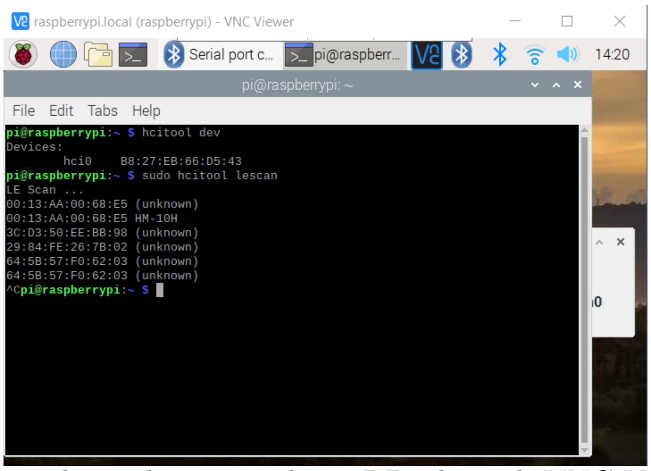
If you don't know the device's address, you can use the hcitool to access the bluetooth device, and potentially the gatttool (see Section 6.4.1) to detect the "services" running on the device. On the RPi, type

```
hcitool dev
```

to see if hcitool can see your device. Then run a low energy scan:

```
sudo hcitool lescan
```

That will give you a list of devices that it is detecting. If you don't see yours, try to unplug the ESP32 and re-plug it in, or turn Bluetooth on your RPi off and back on again. You will see the MAC address of the device and its name listed. You'll need the MAC address for the Python program. See Figure 6.



```
pi@raspberrypi:~ $ hcitool dev
Devices:
        hci0    B8:27:EB:66:D5:43
pi@raspberrypi:~ $ sudo hcitool lescan
LE Scan ...
00:13:AA:00:68:E5 (unknown)
00:13:AA:00:68:E5 HM-10H
3C:D3:50:EE:BB:98 (unknown)
29:84:FE:26:7B:02 (unknown)
64:5B:57:F0:62:03 (unknown)
64:5B:57:F0:62:03 (unknown)
^Cpi@raspberrypi:~ $
```

Figure 6: hcitool commands on RPi through VNC Viewer.

Now that you have more information about your BLE device, we can write our Python program to read data transmitted through it. You can look at the Documentation of the Peripheral class. Print the incoming sensor data on your RPi.

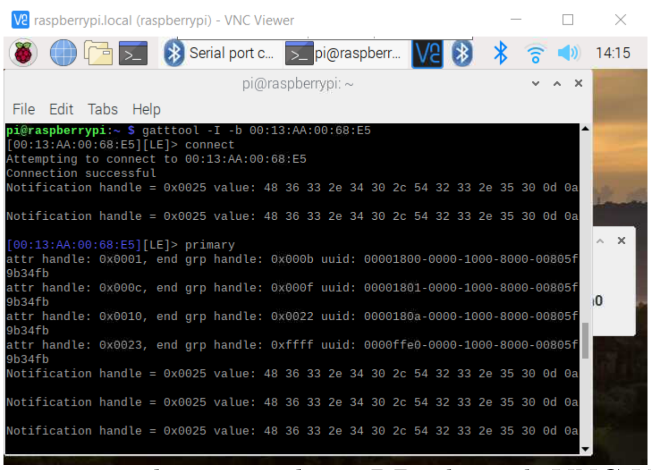
6.4.1 Informational: Writing to a BLE Device (not required)

While having the RPi read data via BLE only requires your BLE's MAC address, if you were to write data, you need more information from your BLE device. You would use gatttool, and you can get more information on how that tool is organized here.

You can open gatttool with the BLE device's address:
```
sudo gatttool -b <BLE_address> -I
```
then, if you want to connect, you would type
```
connect
```

If you can connect, the MAC address of your BLE device becomes purple. See Figure 7.

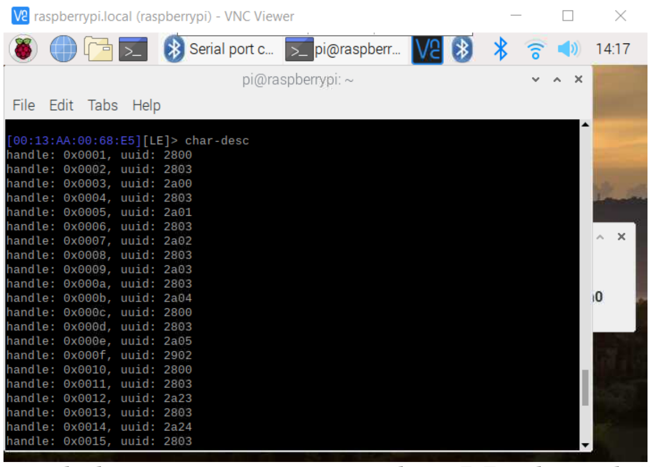
To get the primay UUIDs, or a list of available services running on the BLE device, you type
```
primary
```



```
pi@raspberrypi:~ $ gatttool -I -b 00:13:AA:00:68:E5
[00:13:AA:00:68:E5][LE]> connect
Attempting to connect to 00:13:AA:00:68:E5
Connection successful
Notification handle = 0x0025 value: 48 36 33 2e 34 30 2c 54 32 33 2e 35 30 0d 0a

Notification handle = 0x0025 value: 48 36 33 2e 34 30 2c 54 32 33 2e 35 30 0d 0a

[00:13:AA:00:68:E5][LE]> primary
attr handle: 0x0001, end grp handle: 0x000b uuid: 00001800-0000-1000-8000-00805f
9b34fb
attr handle: 0x000c, end grp handle: 0x000f uuid: 00001801-0000-1000-8000-00805f
9b34fb
attr handle: 0x0010, end grp handle: 0x0022 uuid: 0000180a-0000-1000-8000-00805f
9b34fb
attr handle: 0x0023, end grp handle: 0xffff uuid: 0000ffe0-0000-1000-8000-00805f
9b34fb
Notification handle = 0x0025 value: 48 36 33 2e 34 30 2c 54 32 33 2e 35 30 0d 0a

Notification handle = 0x0025 value: 48 36 33 2e 34 30 2c 54 32 33 2e 35 30 0d 0a

Notification handle = 0x0025 value: 48 36 33 2e 34 30 2c 54 32 33 2e 35 30 0d 0a
```

Figure 7: gatttool commands on RPi through VNC Viewer.

Another command lists the available handles, which are connection points that you can read and write access data to or from:
`char-desc`
A handle is a sequential number generated by the Bluez library which is tied to a specific characteristic. You can use the output of char-desc to relate the characteristic UUID to each open handle. See Figure 8.



```
[00:13:AA:00:68:E5][LE]> char-desc
handle: 0x0001, uuid: 2800
handle: 0x0002, uuid: 2803
handle: 0x0003, uuid: 2a00
handle: 0x0004, uuid: 2803
handle: 0x0005, uuid: 2a01
handle: 0x0006, uuid: 2803
handle: 0x0007, uuid: 2a02
handle: 0x0008, uuid: 2803
handle: 0x0009, uuid: 2a03
handle: 0x000a, uuid: 2803
handle: 0x000b, uuid: 2a04
handle: 0x000c, uuid: 2800
handle: 0x000d, uuid: 2803
handle: 0x000e, uuid: 2a05
handle: 0x000f, uuid: 2902
handle: 0x0010, uuid: 2800
handle: 0x0011, uuid: 2803
handle: 0x0012, uuid: 2a23
handle: 0x0013, uuid: 2803
handle: 0x0014, uuid: 2a24
handle: 0x0015, uuid: 2803
```

Figure 8: gatttool characteristic command on RPi through VNC Viewer.

If you were to write via BLE using bluepy, you would need to get a service of the peripheral using its UUID, which we can find in the output of the `primary` command above. Normally it is something like "0000ffe0-0000-1000-8000-00805f9b34fb". Then you can obtain the characteristic from that service, and write to it.

6.5 #Task5 Communicate sensor data with CoAP from ESP32 to RPi

Now we will communicate between our microcontroller, the ESP32, and the RPi using the **Con**strained **A**pplication **P**rotocol (CoAP). We will set up a CoAP server on the RPi to listen for incoming data. Then we will communicate sensor readings from the ESP32 using the CoAP protocol.

Make sure the RFID reader is connected to the ESP32 using the provided cables, as shown in Lab1. Then plug the ESP32 into your laptop via USB.

Open the given coapclient-students sketch (from Ufora) and look through the code. You have to enter the name of the WiFi network through which you will connect, and the password. You can see that we set up a CoAP response callback method for when the CoAP server acknowledges a message or communicates. The CoAP class is initialized with an instance of an UDP class:

```
WiFiUDP Udp;
Coap coap(Udp);
```

Inside the setup code, we initialize the baudrate for the serial monitor, setup WiFi, register the callback function with CoAP, and then start the CoAP client on our IoT device. Inside the loop function, we repeatedly send just the string "hello" to the CoAP server on the RPi.

You will modify this code to read scans of RFID tags and send that data to the RPi. Note that the function that we provide for you `array_to_string` will be helpful for getting the RFID ID into a string format.

On the RPi, you will write a simple Python program to act as the CoAP server. Check out how to write a very simple CoAP server with the CoAPthon library. You will write a simple class that extends from the Resource class within CoAPthon. This will be the resource that your CoAP client communicates with, sending sensor data to. You do not have to support all HTTP methods if you will not use them. Then create a CoAPServer class, similar to the example, and in main, initialize your CoAP server and have it start listening for clients. You can develop your code in PyCharm, and set it up with the remote interpreter on the RPi. Then run it, and compile and upload your ESP32 sketch to the device and verify they are communicating with each other.

6.6 #Task6 Send CoAP data from RPi to Ingress API

Implement a Python script that makes the RPi a gateway. The ESP32 listens to scans of RFID tags and transmits this data to the RPi via CoAP. Your task is to write a Python program on the RPi (or modify the one from Task5) to pass the data from the ESP32 to the Ingress API. This task expands on earlier tasks from this lab and from Lab 3 about microservices. Make sure you show that the data actually made it into the Ingress database.

6.7 #Task7 Communicate sensor data with MQTT from ESP32 to RPi

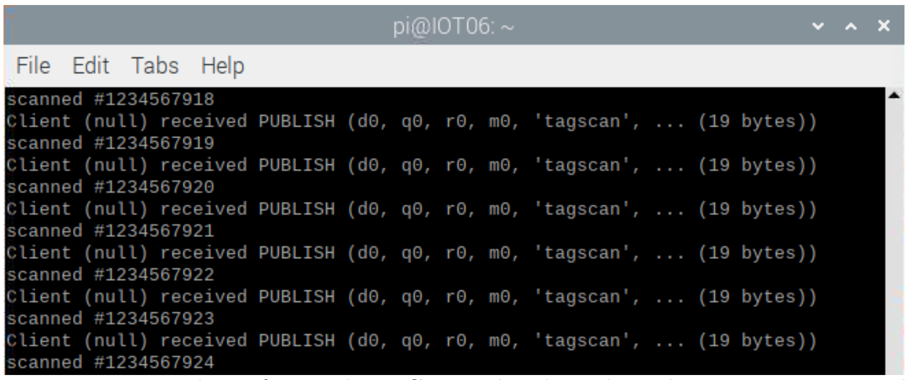
In this task, sensor data from the ESP32 will be transmitted via WiFi to an MQTT broker. That published data will then be stored in the MQTT broker until the RPi subscribes to it so it can receive the data. The ESP32 will publish to a topic and the RPi will receive the data by subscribing to the same topic. In this task, the RPi will serve as our MQTT broker.

To develop the code that will run on the ESP32, first take a look at the example code of how to talk to an MQTT broker that we provide for you. Change the code to communicate the RFID reader's tag readings to the following MQTT topic: `esp32/tagscan`.

You will have to change the mqtt_server to be the IP address of your RPi, and change the mqtt_port to 1883. This function might also come in handy: ltoa, which converts a long into a string.

You will use the Mosquitto broker on the RPi to check if you can receive the MQTT messages. Test the MQTT broker to check if everything goes according to plan.

Subscribe to `esp32/tagscan` in the terminal of the RPi. In the documentation of Mosquitto you can find out how to compose the subscribe command.



```
scanned #1234567918
Client (null) received PUBLISH (d0, q0, r0, m0, 'tagscan', ... (19 bytes))
scanned #1234567919
Client (null) received PUBLISH (d0, q0, r0, m0, 'tagscan', ... (19 bytes))
scanned #1234567920
Client (null) received PUBLISH (d0, q0, r0, m0, 'tagscan', ... (19 bytes))
scanned #1234567921
Client (null) received PUBLISH (d0, q0, r0, m0, 'tagscan', ... (19 bytes))
scanned #1234567922
Client (null) received PUBLISH (d0, q0, r0, m0, 'tagscan', ... (19 bytes))
scanned #1234567923
Client (null) received PUBLISH (d0, q0, r0, m0, 'tagscan', ... (19 bytes))
scanned #1234567924
```

Figure 9: Incoming data from the ESP32 displayed with Mosquitto on the RPi

6.8 #Bonus Task8 Control an LED via MQTT (Mosquitto)

Control the state of the ESP32's built-in LED with a publish message from the RPi using Mosquitto. In the documentation of Mosquitto you can find how to compose a publish command. Use topic `esp32/output` to control the state of the LED with the commands `on` and `off`. Confirm by publishing `led_is_on` or `led_is_off` to the `esp32/ledstatus` when you have changed the state of the LED.

Write a small Python script to control the state of the LED but with the Paho-MQTT library. The LED should turn on and off for three seconds.

6.9 #Task9 Modify your Ingress to subscribe to sensor data with MQTT

You will now modify your Ingress code from Lab 3 to connect with an MQTT broker in the cloud (RabbitMQ) and subscribe to data. The sensor data will be published from the ESP32 to the broker running in the cloud. For this, we will use the handy FastAPI-MQTT Python package. We have provided code on Ufora to get you started. Place the `mqtt.py` file in the same folder as your ingress project (where `main.py` is). You can look at the MQTT example code from this Github page of FastAPI-MQTT. You also need to modify a few small things in your ingress project:

1. Copy the code from `add-to-main.py` to your existing `main.py`. You will have to modify the initialization of your `app` to use the `lifespan` function.
2. Add the following package to your `requirements.txt` file: `fastapi-mqtt`.
3. RabbitMQ makes use of virtual hosts so that even if you use the same topic on two virtual hosts, they are separate. Thus you'll need to append your username to your virtual host. Your virtual host is called `vh<your-username>`, so the total username should look something like `vhpletinck-stef:pletinck-stef`.
4. The URL of your broker is "rabbitmq.database.ilabt.cloudandmobile.ilabt.imec.be" and the port number is 32712.

Then add the appropriate methods to deal with connecting with the MQTT broker, incoming messages, etc., based on the examples shown on the websites linked above.

You will have to change the code on your ESP32 to publish to a different broker, port, and use a username and password to access the broker. Here too, you will need to prepend your virtual host name to the username. To accomplish this, you can modify the code that you used for Task7. Look in the `reconnect` function for where the client connects. Uncomment the inclusion of the MQTT username and password as parameters to this client connect function call.

When you receive the data inside your ingress, you can deal with it the same way as when it is received on the POST to the `/data` REST endpoint. You can make sure the data is correctly sent to your database by going to http://influx.database.cloudandmobile.ilabt.imec.be when deploying your extended ingress API in the cloud.

7 Questions

1. Compare and contrast the different protocols you have explored in this assignment: Bluetooth Classic, Bluetooth Low Energy, CoAP, and MQTT. What scenario is each targeted for or good at? Be specific, give examples.

8 Summary

For some devices, it might not make sense to support HTTP or WebSockets directly, or it might not even be possible, such as when they have very limited resources like memory or processors, when they can't connect to the internet directly (such as your Bluetooth activity tracker), or when they're battery-powered. Those devices will use more optimized communication or application protocols and thus will need to rely on a more powerful **gateway** that connects them to the Web of Things, such as your mobile phone that would upload the data from your Bluetooth bracelet to the cloud, by bridging/translating various protocols.

However, if your IoT device is powerful enough, then it can **directly** communicate with other Things using powerful communication protocols and REST APIs. IoT devices (or a gateway itself) can also connect to services in the **cloud** that can process large amounts of data in a scalable way, and that offer more powerful features.

In this lab, we've explored several integration patterns: the gateway, direct, and cloud integration patterns.

9 Material you need to submit

Write a lab report elaborating on the solved tasks along with the explanation of the code. Give the lab report a structure that mimics the structure of this document. Include the answer to the question above in your report. You should take screenshots or short videos that show that you got each task working. Screenshots can be added to your report, and videos can be bundled into your zip file. If you had to do anything noteworthy in the preparation section or ran into problems, also include this in your report. Only the **.pdf** extension is allowed for the report. Name your lab report **LAB4_FamilyName_FirstName.pdf**. Archive the report together with the source code for the exercises and videos, and name this archive **Lab4_FamilyName_FirstName.zip**. Hand everything in using Ufora. You can hand everything in multiple times, only the last file will be saved.

Appendices

A Connect with VNC

We recommend that you install **VNC Viewer** on your laptop. With this free software you can establish a connection with another computer, in this case your RPi, and control it remotely. You can then have a GUI connection to your RPi without attaching an external screen, mouse, or keyboard. As long as your RPi is connected to the internet, VNC allows you to see what's going on on your RPi, graphically, and control it with your laptop's mouse and key strokes.

Of course you need VNC Server to run on your RPi for this to work. VNC Server is included with Raspbian but **you still have to enable it**. Enable VNC first on the RPi. Select Menu > Preferences > Raspberry Pi Configuration > Interfaces and make sure VNC is set to **Enabled**. Alternatively, run the command `sudo raspi-config`, navigate to Interfacing Options > VNC and select **Yes**.

Download the VNC Viewer for your OS.

Make an account and login on both your RPi and laptop. Check the mailbox of the e-mail account that you signed up with. VNC sends several e-mails to authorize access.