ESP32: Performance evaluation of network protocols

Giorgio Martino

IoT UniMoRe - October 2023

Abstract

The following document contains a performance evaluation of different network technologies and protocols using an IoT device.
The chosen device is a standard ESP32, which supports Bluetooth communication as well as Wi-Fi capabilities. An Ubuntu laptop has been used to act as second client or server, responding to ESP32 requests. A dashboard has also been developed to display real-time statistics of the different protocols is use.
The scope of the project is to test different communication protocols, and evaluate them

Contents

Listings

1 Introduction

The Internet of Things, IoT, is a collection of different devices and communication protocols, that form a growing network. The devices can be divided into two main categories:

- Sensors: these devices that can "sense", or better said read data from the environment. These may include temperature, humidity, accelerometer etc. . .
- Actuators: these devices that can convert a digital input into a physical output, for example LED lights, buttons, LCD screens etc. . .

- Microcontrollers: These are the devices that have the capabilities to communicate through different protocols, both between each other and with the Cloud. Their behaviour can be basic, merely forwarding data obtained through sensors to a Fog or Cloud node, or more advanced, performing a first layer of analysis or aggregation before sending the data.

  Microcontrollers can have an OS, like Raspberry PI, or not, like ESP32 or Arduino boards.

This project only focuses on communication between a microcontroller device, ESP32, and a fog or cloud node, represented by a laptop.
The technologies used to evaluate the performance are Bluetooth, Wi-Fi using CoAP protocol and Wi-Fi using HTTP protocol.
The scope of this project was to utilize these protocols and evaluate them in an analytical way, visually representing the obtained results.

2 Technologies analyzed

As stated before, two technologies have been used for the performance evaluation, Bluetooth and Wi-Fi. The Wi-Fi protocols used are CoAP and HTTP for the analysis, and MQTT to send analyzed data to the Dashboard.

2.1 Devices

2.1.1 ESP32

ESP32 has been chosen as the target device, impersonating what would be the Edge node. It is a SoC (System on Chip) device, and has full TCP/IP stack. It has a 32bit dual-core processor, and is equipped with Bluetooth and Wi-Fi.

It is a pretty standard device for prototyping, part of the reason why it has been chosen, there is a good amount of documentation online for many use cases, and it can be programmed both with Arduino IDE or VSCode, using PlatformIO plugin. In this performance evaluation, our ESP32 will act as an advanced edge node: it will send packets through different protocols, and perform some data aggregation before forwarding the results obtained to the dashboard.

All the implementations written for this device are in C++, naturally supported by the microcontroller.

2.1.2 Ubuntu laptop

An Ubuntu laptop has been used as Cloud node, in the scope of this project. Its role was mostly to act as a server or second client, allowing the ESP32 to analyze network performances over the whole round trip time.

Python has been used to write all the necessary code for these implementations, both because of its ease of use and extensive library support. Specifically, three different modules are present: a Bluetooth client, a CoAP server and an HTTP server.

2.2 Bluetooth

Bluetooth is a wireless, ”short-link” radio technology used to exchange data. Due to its low-range capabilities, it can only be used over short distances, usually up to 10 meters. It can be very useful in PANs (Personal Area Network), and it was included in the performance evaluation as being the only one not relying on Wi-Fi.

2.3 CoAP

CoAP (Constrained Application Protocol) is an open IETF standard protocol, designed for those devices that require a lightweight protocol, but want to use HTTP like verbs and URLs. It uses UDP protocol and is designed for IoT devices, which usually have limited memory, computing and/or battery power, low bandwidth.

Given the similarity between HTTP and CoAP I was very interested in how they would perform in terms of latency, packet loss and throughput. CoAP implementation is fairly easy, but still the most complex among the three (four counting MQTT) evaluated, both for the C++ client and Python server.

2.4 HTTP

HTTP is the *de-facto* standard in Wi-Fi network communication. It exposes resources as URLs (*http://domain/resource*), and declares some action verbs

(GET, POST, PUT, DELETE etc...)

HTTP is the most used protocol for web communications, which is why I wanted to include it in this research. As a Java Cloud developer myself, I was curious about how it would perform compared to Bluetooth and CoAP.

I'd say that, implementation-wise, it was by far the easiest to use. C++ implementation required very few lines of code, and setting up a Python server was pretty straight-forward.

3 Implementation

As part of the git repository where this project is stored, we can see two main folders: *cpp* and *python*.

cpp contains the C++ code implementation for the ESP32. There are different modules, most of them used for testing and prototyping (*bluetooth-arduino*, *coap-client*, *http-client*, *mqtt-publisher*), while the one used for performace evaluation is *performance*.

python contains the Python code implementation for the Ubuntu laptop. Different modules are present: *bluetooth-iot* is the implementation of the second bluetooth client, *coap-server* implements the CoAP server and *http-server* the HTTP one. *mqtt-client* has just been used for testing.

3.1 Bluetooth client to client

For the bluetooth C++ client, no external library was required. The way to initialize a Bluetooth connection, and send messages through it is simple enough. All that is needed is to declare a **BluetoothSerial** and use it.

```
BluetoothSerial SerialBT;

void setup() {
SerialBT.begin("DeviceName");
}

void loop() {
SerialBT.println("Message");
String response = SerialBT.readString();
}
```

Listing 1: Bluetooth C++ client

The python client was a bit more trivial, beginning with the installation of the *pybluez* library, needed to start a discovery of all bluetooth devices. Once found the ESP32, a connection through **BluetoothSocket** is started. Then it gets easier as it's just a matter of receiving and sending back an answer.

```
nearby_devices = discover_devices()
for blt_addr in nearby_devices:
    if "DeviceName" == lookup_name(blt_addr):
        target_address = blt_addr

service_matches = find_service(address=target_address)
socket = BluetoothSocket(bluetooth.RFCOMM)
socket.connect((host, port))

data = socket.recv(buf_size).decode()
socket.send("Message")
```

Listing 2: Bluetooth Python client

3.2 CoAP client to server

Implementing the CoAP C++ client requires a few more things than Bluetooth: an external library, *CoAP simple library*, is required, and a *WiFiUDP* interface is necessary as well. Of course there is the need to be connected to some network, which credentials are stored into *secrets.h*.

The last thing to do is to create a callback function, which role is to be called when a CoAP response is received. This function is then passed to the *coap* object, that can start looping and sending requests to the server.

```
#include "secrets.h"

WiFiUDP udp;
Coap coap(udp);
```

```
 void callback_response(CoapPacket &packet, IPAddress ip,
int port);

 void setup() {
   coap.response(callback_response);
   coap.start();
 }

 void loop() {
   coap.get(ip, 5683, "hello");
   coap.loop();
 }
```

Listing 3: CoAP C++ client

Again, the python server was a bit trickier. After importing a few required libraries, (*aiocoap*, *asyncio*) we can define a Resource Class that will expose the HTTP-like verbs. Then the Resource Class needs to be added and the server can be created.

```
 class HelloResource(resource.ObservableResource):
     async def render_get(self, request):
         payload = b'Message'
         return aiocoap.Message(payload=payload)

 async def main():
     root = resource.Site()
     root.add_resource(['hello'], HelloResource())
     await aiocoap.Context.create_server_context(bind=(
get_local_ip(), 5683), site=root)
     await asyncio.get_running_loop().create_future()

 if __name__ == "__main__":
     asyncio.run(main())
```

Listing 4: CoAP Python server

3.3 HTTP client to server

Very similarly to CoAP implementation, the HTTP C++ client requires a few things: the external library *ArduinoHttpClient* to be able to make

HTTP requests, and a network connection, Wi-Fi in this case. Then it's very straight-forward, the HTTPClient only needs to be initialized and it will be ready to send requests to the server.

```
#include "secrets.h"

HTTPClient http;

void setup() {
    http.begin("http://192.168.1.12:8000");
}

void loop() {
    int httpCode = http.GET();
    if (httpCode == HTTP_CODE_OK) {
      String page = http.getString();
    }
}
```

Listing 5: HTTP C++ client

Creating a Python server is extremely easy. Using FastAPI is as easy as it gets: basically it is only required to define an endpoint. Then you only need to start the server using *uvicorn* command line, specifying host and port.

```
app = FastAPI()

@app.get("/")
async def get_root():
    return "Message"
```

Listing 6: HTTP Python server

3.4 MQTT publisher

The MQTT publisher has been developed with the aim of progressively send computed statistics to the dashboard, to have a real-time view of all the metrics, for all evaluated protocols.

After importing the library *PubSubClient*, similarly to CoAP and HTTP, a client is declared and initialized. Then the loop can start, and messages can be published to the specified topic/s.

```cpp
PubSubClient client(ip, 1883, wifiClient);

void setup() {
    client.setServer(ip, 1883);
    client.loop();
}

void loop() {
    client.publish("topic/hello", "Message");
}
```

Listing 7: MQTT C++ publisher

4 Problems encountered

In this section I want to highlight a few issues encountered during the way, and how these were solved (if possible).

The first problem I stumbled upon was the network configuration of the Ubuntu laptop. There was a problem with the Linux kernel firewall, which by default dropped all incoming network requests, without any error message. A first work-around was to use ngrok to expose the local servers to the Internet. Then I managed to find and modify the firewall configurations, fixing the problem. Of course this problem did not affect Bluetooth.

Another issue was with the ESP32 flash memory size. Since I wanted to have a single C++ file, to test simultaneously Bluetooth, CoAP and HTTP, while also using MQTT to publish to the Node-RED dashboard, I had a few libraries imported in my PIO project.

This led to the program being too large to be uploaded into the flash memory of the ESP32. Since I was skeptic that my program could really be bigger than the maximum memory size, after some research I found about the following option, which changes the partition type allowing more space:

```
board_build.partitions = huge_app.csv
```

5 Performance evaluation

To evaluate performances, I wanted to automate everything as much as possible. The goal was, once everything was started, to be able to see a near real-time dashboard, including all metrics for every protocol analyzed.

This is why I decided to implement all the C++ code into one single program, and then deploy all the needed client/servers. The performances calculated are the following:

5.1 Latency

$$latency = totalTime/receivedMessages$$

Where, for each packet sent and received:

$$totalTime = totalTime + (receivedTime - sentTime)$$

Latency is calculated as the total time needed for a request to reach the server, and for the response to reach the client back.

If no packets were lost, the *receivedMessages* count would be equal to 5, meaning that every five messages that travel from the client to the server, and back to the client, statistics are published on the respective MQTT topic.

Needless to say, the lower the latency, the better.

5.2 Packet Loss

$$packetLoss = sentMessages - receivedMessages$$

Packet loss is the sum of packets that where sent but not received. Ideally this number is always zero, however, having 5 as the default count before publishing statistics, the maximum number is 5. After that all the counters and statistics are reset.

5.3 Throughput

$$throughput = (sizeof(payload) * sentMessages)/(totalTime/1000)$$

Where, for each packet sent and received:

$$totalTime = totalTime + (receivedTime - sentTime)$$

Note that, $totalTime/1000$ is necessary to ensure that *throughput* can be expressed as *bytes/second*.

To keep this measure consistent, every protocol sent the exact same message, providing the same bytes count per message. The message was a simple JSON like the following:

```
{
    "data": "Hello from ESP32"
}
```

Listing 8: JSON message