POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

IOT Challenge 3 - Exercise

Author: **Matteo Leonardo Luraghi**

Person Code: 10772886
Academic Year: 2024-25

Contents

1 | EQ1

Given my person code, the payload size is of size $L = 3 + 86 = 89$ *bytes*

Airtime values were computed using the tool from
www.thethingsnetwork.org/airtime-calculator, and are provided for SF7, SF8, and SF9 only, since for the other spreading factors the payload size exceeded the maximum allowed.

The computed airtimes are as follows:

- SF7: $\text{airtime}(SF7) = 174.3\,\text{ms}$
- SF8: $\text{airtime}(SF8) = 307.7\,\text{ms}$
- SF9: $\text{airtime}(SF9) = 554\,\text{ms}$

Each node transmits approximately one packet per minute, leading to a total of 50 packets per minute. This implies a total airtime consumption per minute of $50 \times \text{airtime}(SF)$, where the utilization of the channel per minute in each case is:

- SF7: $G_{\text{SF7}} = \frac{50 \times 174.3\,\text{ms}}{60000\,\text{ms}} = 0.14525$
- SF8: $G_{\text{SF8}} = \frac{50 \times 307.7\,\text{ms}}{60000\,\text{ms}} = 0.25642$
- SF9: $G_{\text{SF9}} = \frac{50 \times 554\,\text{ms}}{60000\,\text{ms}} = 0.46167$

The packet success rate can be approximated using the ALOHA model: $P_{\text{succ}} = e^{-2G}$

Using this model, the packet success rates for each spreading factor are:

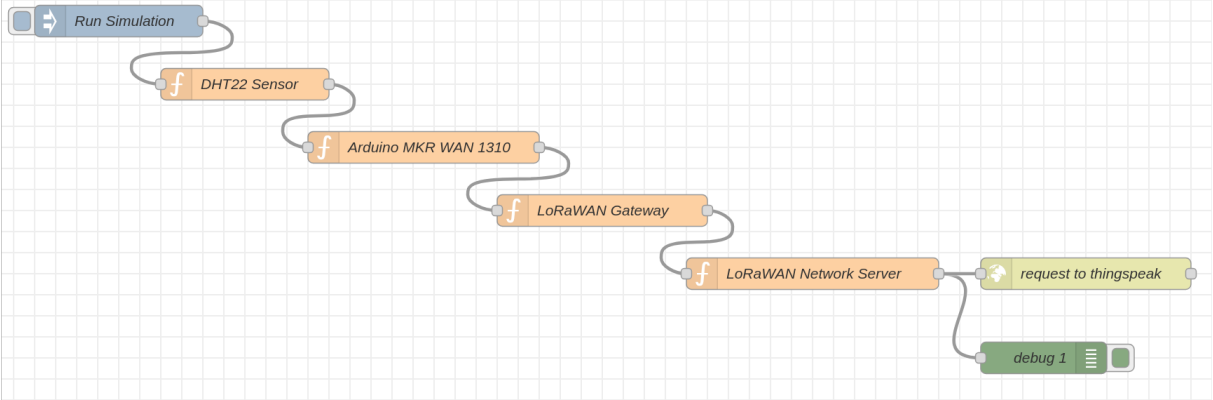
- SF7: $P_{\text{succ-SF7}} = e^{-2G_{\text{SF7}}} = 74.8\%$
- SF8: $P_{\text{succ-SF8}} = e^{-2G_{\text{SF8}}} = 59.9\%$
- SF9: $P_{\text{succ-SF9}} = e^{-2G_{\text{SF9}}} = 39.7\%$

Thus, the only spreading factor that achieves a packet success rate of at least 70% is **SF7**.

2 | EQ2

2.1. Node-RED Diagram

I designed the system based on the specified requirements using Node-RED:



The flow components are:

- `DHT22 Sensor`: function block that simulates the collection of temperature and humidity measurements by a DHT22 sensor
- `Arduino MKR WAN 1310`: function block that simulates sending the sensor's measurements via LoRaWAN to a nearby LoRaWAN gateway
- `LoRaWAN Gateway`: function block that simulates a LoRaWAN gateway, forwarding the sensor's data to the appropriate LoRaWAN network server
- `LoRaWAN Network Server`: function block that simulates a LoRaWAN network server, responsible for forwarding the sensor's data via an HTTP GET request to ThingSpeak
- `request to thingspeak`: http request block that simulates the actual GET request sent to the ThingSpeak API to save the data

The steps required to make this system fully operational are described in the following sections.

2.2. Connection to The Things Network

As described in the official documentation to connect the Arduino MKR WAN 1310 to The Things Network (TTN), we need to retrieve the device's Extended Unique Identifier (EUI), create an account on TTN and connect the board to a LoRaWAN network server.

Using the Arduino IDE, we must first install the 'MKRWAN' library. Then, by uploading the example sketch located at `File > Examples > MKRWAN > FirstConfiguration` and opening the Serial Monitor, we can obtain the device's EUI.

After creating an account on TTN and activating it, we can navigate to the TTN Console and click on "Create an application". Within the application dashboard, we select "End devices" and click "Add end device". We then fill in the required information about the Arduino board, including the device EUI. By clicking the "Generate" button, the console will provide the AppEUI and AppKey for our device.

Back in the Arduino IDE, after restarting the board, it will prompt for the AppEUI and AppKey. Once these values are entered correctly, the device will attempt to join the TTN network. If it is within the range of a compatible gateway, the connection will succeed and the message "Message sent correctly!" will appear in the Serial Monitor.

On the TTN Console, to forward data to ThingSpeak, we must configure a Webhook: under the `Integrations > Webhooks` tab in the TTN application, we can click "Add webhook", select ThingSpeak, enter the Channel ID and Write API Key, and TTN will automatically forward the uplink messages to ThingSpeak.

Lastly, The Things Network will need to format the data received by the board for ThingSpeak by using a custom payload formatter. This can be configured in the TTN Console under `Payload formatters > Uplink`, using the following JavaScript function:

```
function decodeUplink(input) {
  var str = String.fromCharCode.apply(null, input.bytes);
  var values = str.split(',');

  return {
    data: {
      field1: parseFloat(values[0]),
      field2: parseFloat(values[1])
    }
  };
}
```

This formatter splits the payload string into temperature and humidity values and assigns them to `field1` and `field2`, which TTN then forwards to ThingSpeak via the webhook integration.

2.3. Arduino Setup

Now that we've established a connection between the board and The Things Network, we can move forward with collecting sensor data by integrating the DHT22 sensor with the board. The Arduino will send the sensor's measurements to TTN via LoRaWAN.

Below is a sample implementation of the code for the board:

```
#include <MKRWAN.h>
#include <DHT.h>

// pin where the DHT22 sensor is connected
#define DHTPIN 2

DHT dht(DHTPIN, DHT22);
LoRaModem modem;

void setup() {
    // initialize the DHT sensor
    dht.begin();
    // initialize the modem by setting the frequency band for Europe
    // and joining TTN using the appEui and the appKey
    modem.begin(EU868);
    modem.joinOTAA(appEui, appKey);
}

void loop() {
    // read temperature and humidity
    float h = dht.readHumidity();
    float t = dht.readTemperature();
    // format the message's payload
    String payload = String(t, 1) + "," + String(h, 1);
    // send the measurements via LoRaWAN
    modem.beginPacket();
    modem.print(payload);
    modem.endPacket();
    // wait a minute between readings
    delay(60000);
}
```

3 | EQ3

I modified the provided `LoRasim.ipynb` notebook, saving the result as `Challenge3_LoRasim.ipynb`.

The main changes include the introduction of two functions:

- `simulate_figure_5`, which is the original `simulate` function, for simulating Figure 5 using `loraDir.py`

```
def simulate_figure_5(n_nodes, tx_rate, exp, duration):
    env = os.environ.copy()
    env["MPLBACKEND"] = "Agg"

    # Use subprocess.run to execute the command and capture output
    result = subprocess.run(
        [
            "python2",
            "lorasim/loraDir.py",
            str(int(n_nodes)),
            str(int(tx_rate)),
            str(int(exp)),
            str(int(duration)),
        ],
        env=env,
        capture_output=True,
        text=True,  # Capture output as text
    )

    # Print the output
    # print(result.stdout)
    # print(result.stderr)
```

- `simulate_figure_7`, adapted from the original `simulate` function to call `loraDirMulBS.py` and simulate Figure 7, allowing specification of the number of sinks and collision mode

```
def simulate_figure_7(n_nodes, tx_rate, exp, duration, n_sinks,
    collisions):
    env = os.environ.copy()
    env["MPLBACKEND"] = "Agg"

    # Use subprocess.run to execute the command and capture output
    result = subprocess.run(
        [
            "python2",
            "lorasim/loraDirMulBS.py",
            str(int(n_nodes)),
            str(int(tx_rate)),
            str(int(exp)),
            str(int(duration)),
            str(int(n_sinks)),
            str(int(collisions),)
        ],
        env=env,
        capture_output=True,
        text=True,  # Capture output as text
    )

    # Print the output
    # print(result.stdout)
    # print(result.stderr)
```

To simulate Figure 5, I used experiments 4, 3, and 5, corresponding respectively to SN3, SN4, and SN5 in the paper:

```
duration = 86400000
tx_rate = 1e6

nodes = (
    list(range(1,10)) +
    list(range(10,300,10)) +
    list(range(300,1000,100)) +
    list(range(1000, 1601, 100))
)

# simulate figure 5
for n_nodes in nodes:
  print(f"Simulating {n_nodes} nodes")
  simulate_figure_5(n_nodes=n_nodes, tx_rate=tx_rate, exp=4, duration=
      duration)
  simulate_figure_5(n_nodes=n_nodes, tx_rate=tx_rate, exp=3, duration=
      duration)
  simulate_figure_5(n_nodes=n_nodes, tx_rate=tx_rate, exp=5, duration=
      duration)
```

To simulate Figure 7, I used experiment 0 with collisions enabled and varied the number of sinks as specified in the paper:

```
sinks = [1, 2, 3, 4, 8, 24]

# simulate figure 7
for n_sinks in sinks:
  print(f"Simulating {n_sinks} sink{'s' if n_sinks>1 else ''}")
  for n_nodes in nodes:
    print(f"[{n_sinks}] - Simulating {n_nodes} nodes")
    simulate_figure_7(n_nodes=n_nodes, tx_rate=tx_rate, exp=0, duration=
        duration, n_sinks=n_sinks, collisions=1)
```

The results, obtained by plotting the corresponding `.dat` files, are shown in the following sections.

3.1. Figure 5

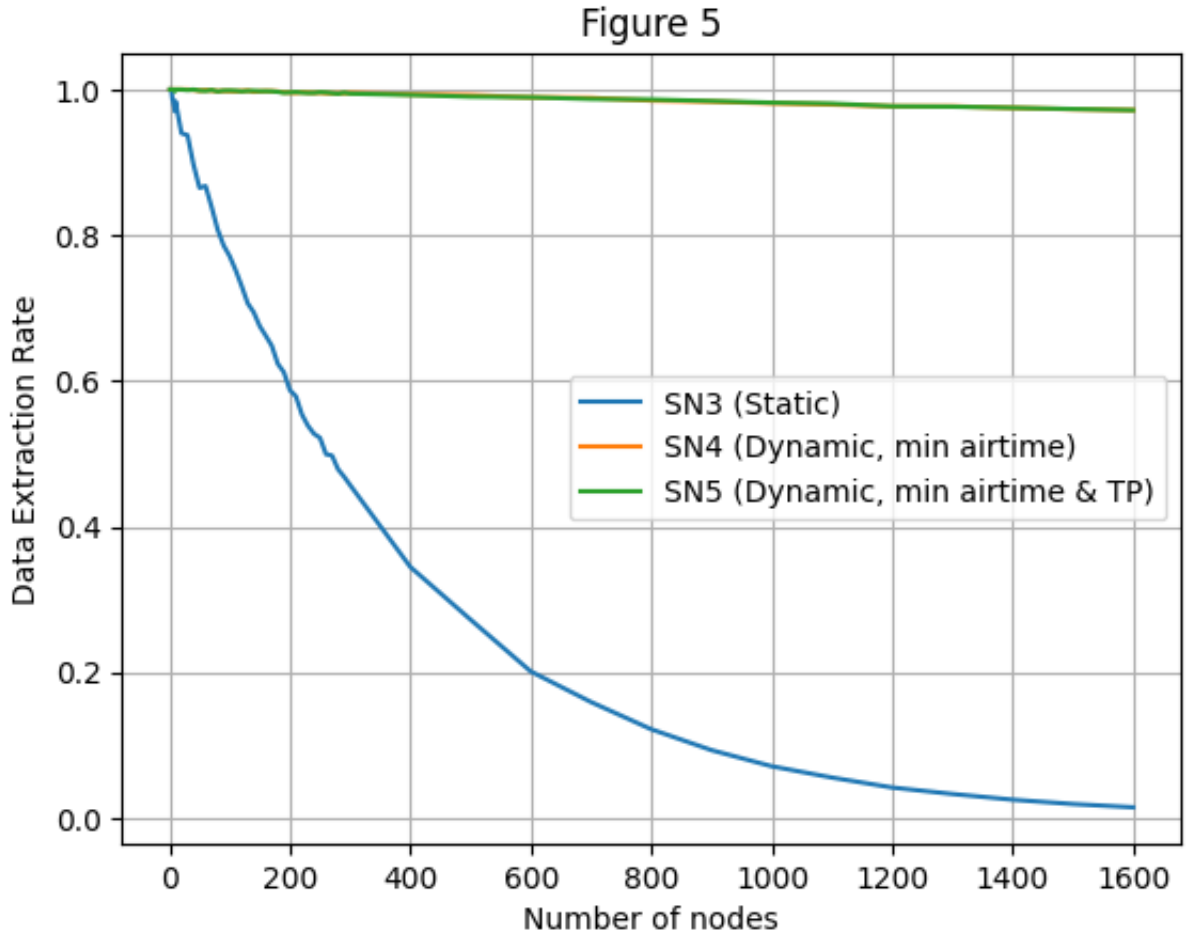


Figure 5
Data Extraction Rate
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200
1400
1600
Number of nodes
SN3 (Static)
SN4 (Dynamic, min airtime)
SN5 (Dynamic, min airtime & TP)

3.2. Figure 7

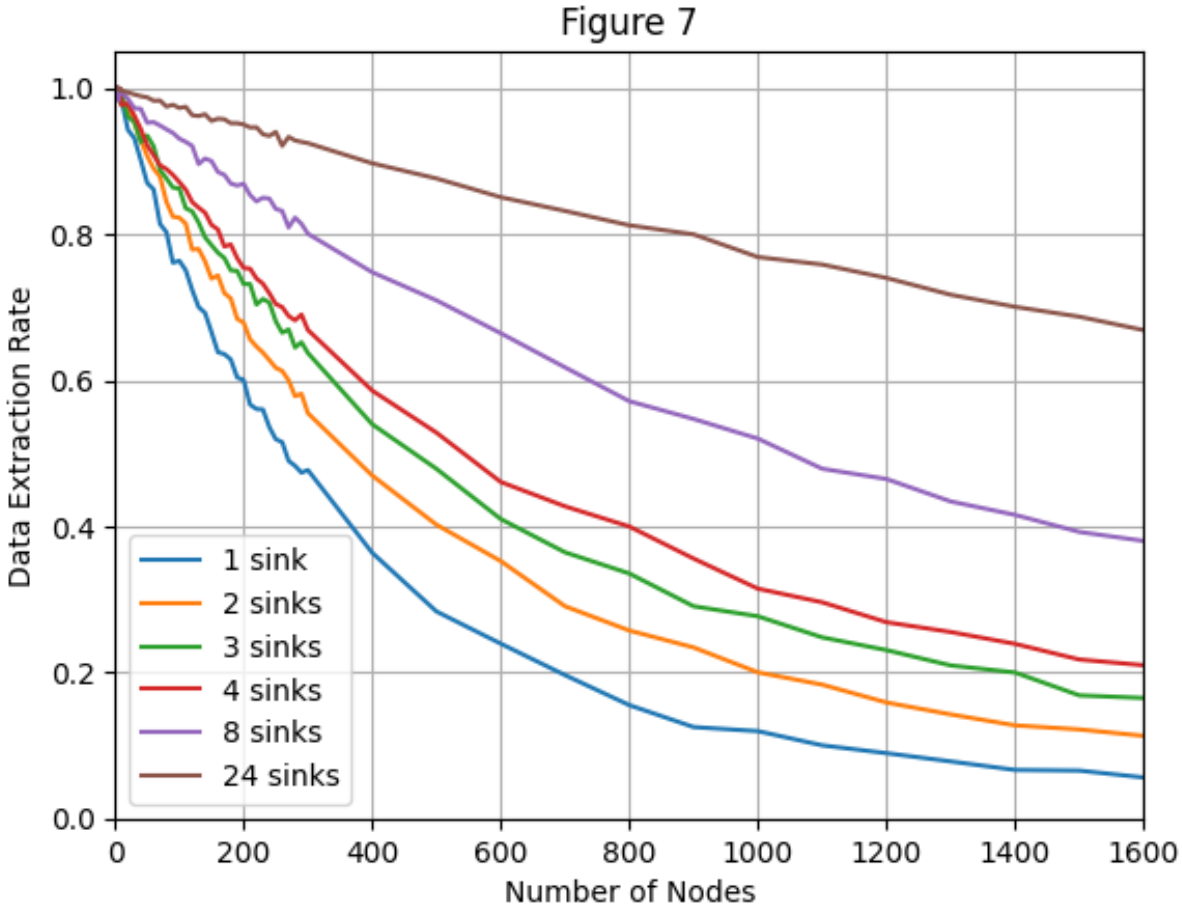


Figure 7
Data Extraction Rate
Number of Nodes
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200
1400
1600
1 sink
2 sinks
3 sinks
4 sinks
8 sinks
24 sinks