Internet of Things

Challenge n. 3: Theoretical exercise

Andrea Caravano, Alberto Cantele

Academic Year 2024–25

Contents

1 Question n. 1

1.1 Exercise text

A LoRaWAN network in Europe (carrier frequency 868 MHz, bandwidth 125 kHz) is composed by one gateway and 50 sensor nodes.
The sensor nodes transmit packet with payload size of L byte according to a Poisson process with intensity $\lambda = 1$ packet / minute.
Find the biggest LoRa SF for having a success rate of at least 70%.
For the payload size L of your packet, take it as follows:
Take XY = Last two digit of your person code (leader code)
$L = 3 + XY$ bytes

1.2 Problem analysis and preliminary computations

Let's approach the problem by delineating the known network parameters and identifying where we are aiming and therefore the missing pieces to complete the whole picture.

Given

$N = 50$ sensor nodes

$\lambda = 1$ packet/minute $= \frac{1}{60}$ packet/second

$L = 3 + XY = 3 + 60 = 63$ bytes

We are aiming at computing the Aloha success rate, as seen during the course lectures:

$$\text{Success Rate (SR)} = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t}$$

Where we are clearly missing t, the airtime of the packet.

1.3 The Airtime Calculator and boundaries in calculations

The Things Network, an open-source infrastructure aiming at providing networking coverage for LoRaWAN made available a calculator tool, returning the packet's airtime when given with the required problem data.

In particular, the following problem data are used:

Parameter	Value
L	63 bytes
Spreading Factor	SF7-SF9
Region	EU (868 MHz)
Bandwidth	125 kHz

Returning:

L	Spreading Factor (SF)	Region	Bandwidth	Resulting airtime
63	SF7	EU (868 MHz)	125 kHz	138,5 ms
	SF8			246,3 ms
	SF9			451,6 ms

The airtime calculator, however, reminds us of the limitations imposed to higher Spreading Factors, limiting the maximum size of the packet length to $L = 51$ bytes.

The following computations, therefore, are required to put an artificial limit to the payload size, as per requirements.

The provided problem data are now:

Parameter	Value
L	51 bytes
Spreading Factor	SF10-SF12
Region	EU (868 MHz)
Bandwidth	125 kHz

Resulting in:

L	Spreading Factor (SF)	Region	Bandwidth	Resulting airtime
51	SF10	EU (868 MHz)	125 kHz	698,4 ms
	SF11			1560,6 ms
	SF12			2793,5 ms

1.4 Resolution

We are now able to compute the Success Rate for all the problem configurations.

Success Rate (SR) $= \frac{S}{G} = e^{-2G} = e^{-2N\lambda t}$

Being the packet length and the Spreading Factors (on the spectrum) the only variable parameter among all configurations, they are chosen as a complete discriminant.

L	Spreading Factor (SF)	Computation
63	SF7	$SR_1 = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t} = e^{-2 \cdot 50 \cdot \frac{1}{60} \cdot 0{,}1385} = 0{,}7939 = 79{,}39\%$
	SF8	$SR_2 = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t} = e^{-2 \cdot 50 \cdot \frac{1}{60} \cdot 0{,}2463} = 0{,}6633 = 66{,}33\%$
	SF9	$SR_3 = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t} = e^{-2 \cdot 50 \cdot \frac{1}{60} \cdot 0{,}4516} = 0{,}4711 = 47{,}11\%$
51	SF10	$SR_4 = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t} = e^{-2 \cdot 50 \cdot \frac{1}{60} \cdot 0{,}6984} = 0{,}3122 = 31{,}22\%$
	SF11	$SR_5 = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t} = e^{-2 \cdot 50 \cdot \frac{1}{60} \cdot 1{,}5606} = 0{,}0742 = 7{,}42\%$
	SF12	$SR_6 = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t} = e^{-2 \cdot 50 \cdot \frac{1}{60} \cdot 2{,}7935} = 0{,}0095 = 0{,}95\%$

From which the requirement constraints (requiring $SR \geq 70\%$) clearly line out the first configuration being the only one compatible.

Moreover, we can also define an additional constraint on packet airtime, which is even more general and can be applied to any payload size, once the number of nodes is fixed.

$$SR = \frac{S}{G} = e^{-2G} = e^{-2N\lambda t} \geq 0{,}7 \Rightarrow e^{-2 \cdot 50 \cdot \frac{1}{60} \cdot t} \geq 0{,}7 \Rightarrow -2 \cdot 50 \cdot \frac{1}{60} \cdot t \cdot \ln(e) \geq \ln(0{,}7) \Rightarrow t \leq \frac{\ln(0{,}7)}{-2 \cdot 50 \cdot \frac{1}{60} \cdot 1}$$

$\Rightarrow t \leq 0{,}214\ seconds = 214{,}0\ ms$

From which we derive that our second configuration is, in fact, invalid.

1.5 Automation via code (Python)

1.5.1 Problem and constants declaration

```
n_nodes = 50  # nodes
Lambda = 1 / 60  # 1 packet over 1 minute = 60 seconds
# Maximum Spreading Factor allowed for a length L = 60 + 3 = 63 bytes is SF9: limiting
↪  L to 51 bytes, being the allowed maximum
# (see https://www.thethingsnetwork.org/airtime-calculator)
spreading_factors = ["SF7", "SF8", "SF9", "SF10", "SF11", "SF12"]
# (from SF10, L = 51 bytes is used, see notes)
t_airtimes = [0.1385, 0.2463, 0.4516, 0.6984, 1.5606, 2.7935]  # seconds
```

Where we identify the limit imposed to higher Spreading Factor configurations, as mentioned earlier.

1.5.2 Success Rate computation

```
# Success Rate computation is made using a loop to compute every computation with
↪ dfferent airtimes and a list to store the results
SR = []
for airtime in t_airtimes:
    SR.append(math.exp(-2 * n_nodes * Lambda * airtime))
```

Where we automatically apply the Aloha Success Rate formula to all the configurations, as mentioned.

1.5.3 Determine the maximum Spreading Factor matching the constraints

```
# The maximum Spreading Factor matching the Success Rate constraints is determined
↪ through a trivial loop iterating on all Success Rate results computed earlier
# Among them, we are looking for the highest one still allowing a matching Success Rate
↪ (>= 0.7)
matching_SR = -1
while matching_SR + 1 < len(SR) and SR[matching_SR + 1] > 0.7:
    # Implicit assumption is that the minimum Spreading Factor matches the constraints
    matching_SR += 1

best_SF = spreading_factors[matching_SR]
print(
    f"The resulting maximum Spreading Factor with a Success Rate matching the
    ↪ constraints (>= 0.7) is: {best_SF}"
)
```

Which results in the following output.

> The resulting maximum Spreading Factor with a Success Rate matching the constraints (>= 0.7) is: SF7

That confirms our theoretical expectations.

2 Question n. 2

2.1 Exercise text

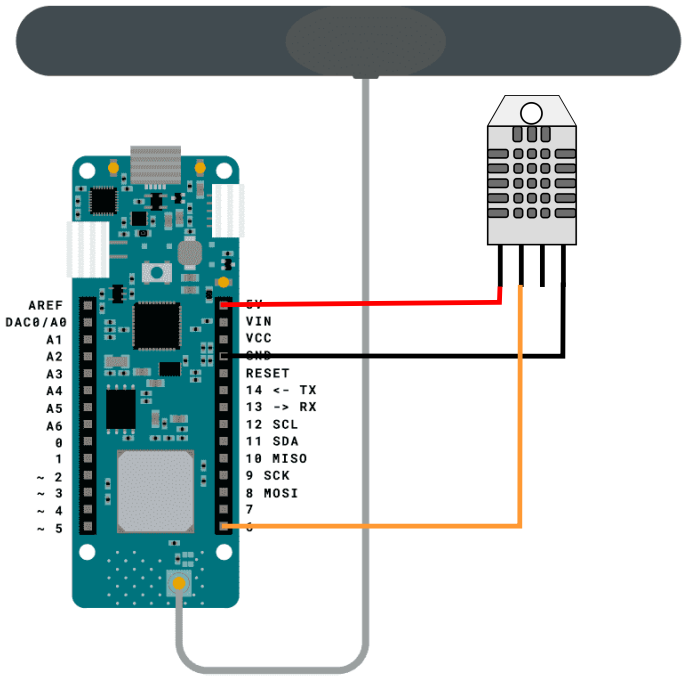
You have purchased an Arduino MKR WAN 1310 and wish to create a system that reads temperature and humidity data from a DHT22 sensor and sends this data wirelessly to ThingSpeak over LoRaWAN.

Design a complete system block diagram (sketch in Node-Red) and describe, in detail, the steps you would need to take to get the system fully operational.

2.2 The Arduino MKR WAN 1310 board and the DHT-22 sensor

First, let's start by preparing our local environment, connecting the Arduino LoRa-enabled board to the DHT-22 temperature and humidity sensor.

This translates to the following diagram.



Then, the Arduino IDE and related software should be installed and setup on the local development environment.

2.3 The Things Network

The Things Network (TTN) is an open-source infrastructure (and network server, the most intelligent part of a LoRa network) aiming at providing free LoRaWAN network coverage.

The project is actively maintained by a growing community across the world, on a voluntary basis.

Gateways are incrementally being deployed across the globe, providing an increasing network coverage.

The Things Network plays, more or less, the same role that a mobile radio network operator plays for mobile User Terminals: it provides the user with the connectivity and managerial services of their LoRaWAN provisions.

2.3.1 Authentication in LoRa-based networks

Being the communication medium intrinsically shared among network nodes, authentication and security or LoRa communications play a fundamental role in its implementation.

The baseline encryption scheme is AES-128, implemented through a couple of keys: the Network Session Key for authentication and integrity and the Application Session Key for data encryption.

Two authentication/activation methods are supported:

- Over-The-Air Activation (OOTA): The end-nodes are not initialized for any particular network: they will send a JOIN Request to a specific LoRa-based network (like The Things Network) and then receive a device address and an authorization token from which session keys are derived, starting from a root key pre-provisioned in the end-node by its manufacturer.

  This is the default and the preferred strategy.

- Activation by Personalization (ABP): End devices are personalized to work with a specific LoRa-based network, right from its manufacturer. Session keys are pre-provisioned by the manufacturer, alongside the device network address.

2.4 The Arduino board and The Things Network: integration overview

2.4.1 Register the board on the network

Before sending and receiving messages from The Things Network, the Arduino board must first be registered on the network.

For this, the board's `Device EUI` identifier must first be collected: the MKRWAN library embeds a specific function call (`modem.deviceEUI()`) that returns it.

Once the `Device EUI` has been collected, a free account on The Things Network must be created.

Then, devices are expected to be registered with an application to communicate with: in the registration procedure, the `Device EUI` is asked, alongside some other security and identification parameters.

2.4.2 Connect the board to the network and send/receive packets

Once registered, the board can Join The Things Network.

A packet being sent is delineated by a `modem.beginPacket()` and a `modem.endPacket()` and sent through `modem.print()`, while a packet being received is read through `modem.read()`, after checking if available.

In this step, some additional processing steps might be required to re-format the payload structure to adapt it to transmission rules.

When considering further communication optimizations, one may consider the characteristics lined out by Question n. 3, later.

2.5 ThingSpeak

2.5.1 The Things Network integration

As mentioned during the course lectures, the Network Server (provided by The Things Network, in our case) is where most of the intelligence of the LoRa architecture is.

It also provides comfortable integrations for both MQTT and HTTP protocol stacks, related, in particular, to ThingSpeak, for which the default, however, is HTTP.

2.5.2 Collection of the measurements

The DHT-22 temperature and humidity sensor is created defining a `DHT` object, clarifying the connection PIN and its model.

Subsequently, it is initialized in the `setup` phase through a `dht.begin()` function call.

Then, temperature measurements are collected with `dht.readTemperature()` and humidity measurements with `dht.readHumidity()`.

Finally, the board packages sensor readings into a byte array and hands it off to the data preparation and transmission routines.

2.5.3 Channel creation

A ThingSpeak channel is a container for a variety of details related to a particular data collection and the data itself.

Being a MATLAB product, it also offers many possibilities for deeper data analysis.

Data streams are identified by fields, which are related to a single measurement: in our case, we will need to create two fields, for both temperature and humidity values.

Upon channel creation, the channel's ID and its write API key should be provided to The Things Network's integration, as they will be used when forwarding data measurements gathered.

In this step, some additional processing steps might be required to re-format the payload structure to adapt it to the integration rules.

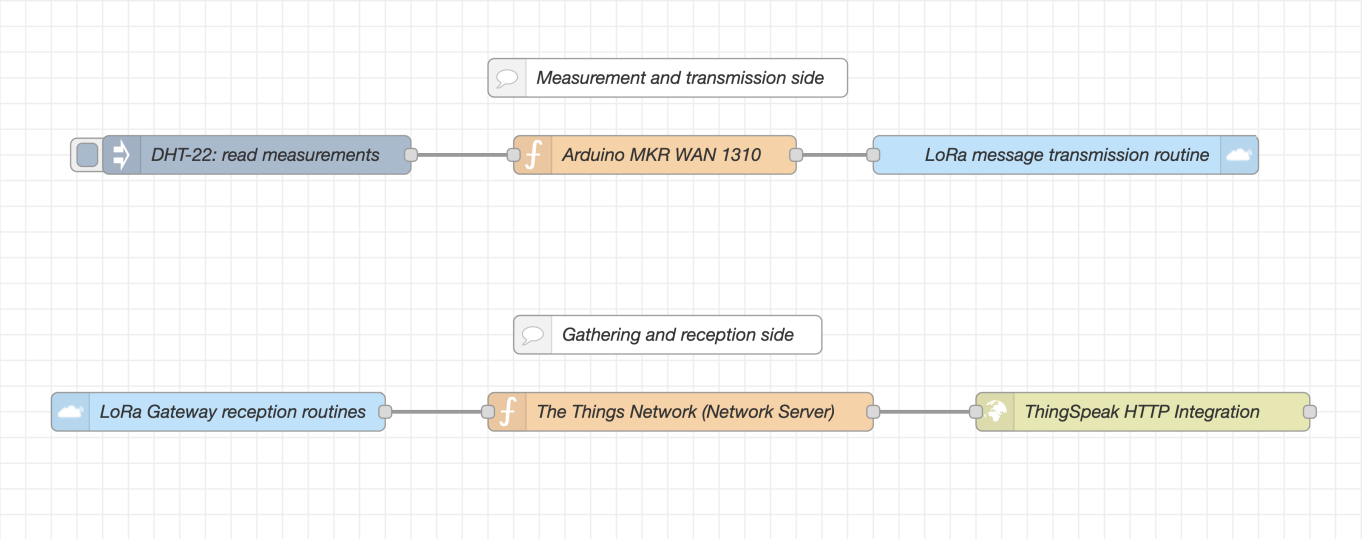
By default, the chosen protocol is HTTP, through POST requests.

Note, however, that integrations using MQTT are also possible.

2.6 Node-RED sketched communication diagram

In the following, the INJECT node represents the DHT-22 temperature and humidity sensor, while THE THINGS NETWORK's network server and the ARDUINO board are simulated by a function node.

Finally, LORA-specific uplink and downlink blocks close the communication loop.



3 Question n. 3

3.1 Exercise text

Using the paper "Do LoRa Low-Power Wide-Area Networks Scale?" by M. Bor et al. and the LoRa simulator available at LoRaSim, your task is to reproduce Figure 5 and Figure 7 from the paper.

1. Read the Paper: carefully study the relevant sections of the paper to understand the experimental setup, parameters, and key findings, especially those associated with Figures 5 and 7.
2. Explore LoRaSim: familiarize yourself with how the LoRaSim simulator works. Understand its configuration options and how to run experiments that model LoRa network behavior.
3. Reproduce the Figures
   (a) Use LoRaSim to replicate the simulations that produced Figure 5 and Figure 7.
   (b) Ensure your simulation parameters (e.g., number of nodes, spreading factors, traffic load, transmission power, etc.) match those used in the original experiments as closely as possible.
   (c) Present your results in the same format as the original figures for easy comparison.

3.2 Experiment sets

Let's start by first analyzing the paper: its aim is comparing some reasonably varied configurations of the LoRa operation modes, consisting of a mixture of Spreading Factors, Bandwidths, Coding Rates and Transmission Powers.

The outcomes of the chosen experiment sets line out the scaling characteristics of LoRa networks, investigating on capacity limits and developing models that describe the LoRa communication pattern and parametrize its behaviour.

To correlate the results to a real-world meaningful application and its scalability, a set of practical measurements has first been conducted on a real implementation, shaping a better understanding of the relationship among the varying parameters, shown in Figures 4, 5 and 7.

A CONFIGURATION SET is a set composed of Transmission Power, Carrier Frequency, Spreading Factor, Bandwidth, Carrier Rate, Transmission rate and Packet payload size.

For each cycle, we define the Data Extraction Rate (DER) parameter, which is related to the ratio of received messages to the total number of transmitted ones over a period of time, similarly to the concept of successful transmissions we introduced during laboratory lectures: being the DER intrinsically meaningful only from 0 to 1, the ideal DER value is the closest to 1 as possible.

Moreover, a Full Collision Check is performed, meaning that the full parametrical model of transmissions orthogonality is employed and checked for.

Finally, the chosen simulation duration is ~ 58 days.

In the following, the set of conducted experiments is described.

- Figure 4: A simple setup consisting of N nodes and one sink.

  A 20-byte packet is sent by each node every 16,7 minutes, to represent a realistic scenario.

  All nodes use the same configuration set, while three transmitter configurations are delineated:

  1. SN_1: The most robust LoRa transmitter settings, with the longest possible airtime (1712,13 ms).
     SN_1 has two definitions: one using Simple Channel Models and another one applying the LoRa-specific Channel Model.
  2. SN_2: The shortest possible airtime (7,07 ms).
  3. SN_3: The most common LoRaWAN deployment configuration setting.

  An exponential decrease of the DER is observed in most configurations with the growth of the number of nodes.

  The chosen split point is at $DER \geq 0{,}9$, to provide meaningful functionalities.

  In such a case, the default LoRaWAN configuration (SN_3) could host up to $N = 64$ nodes, while providing a communication range of ~ 100 *meters*.

  A similar behaviour is observed for SN_1, while the extreme case (SN_2) provides small reliability and limits the transmission range to ~ 37 *meters*.

- Figure 5: Dynamic communication parameter selection, N nodes and one sink.

  Nodes are randomly placed but are guaranteed to reach the sink if the strongest configuration setting is used.

  Again, a 20-byte packet is sent by each node every 16,7 minutes, to represent a realistic scenario.

  1. SN_4: Minimization of the airtime through Bandwidth, Spreading Factor and Carrier Rate configuration, but fixing the Transmit Power to 14 dBm.
  2. SN_5: Adds Transmit Power optimization on top of SN_4.

  Both SN_4 and SN_5, as shown later, provide a huge improvement to the default LoRaWAN deployment configuration (SN_3).

  In fact, up to $N = 1100$ can be supported, this time.

  However, these achievements turn out to be impractical, in a real-world scenario, as it relies on optimistic assumptions (no interference, little redundancy and no environment adaptation).

- Figure 7: Multiple nodes and sinks.

  Again, a 20-byte packet is sent by each node every 16,7 minutes, to represent a realistic scenario.

  The chosen experiment set, this time, consists of only SN_1, the same one described earlier.

  As multiple sinks are now present, a new placement strategy has to be applied.

  The chosen one relies on some simple geometrical considerations that are aimed at equally spacing a multiple number of sinks and, compatibly, the multiple nodes.

  Results, also replicated later, show that the increase in the number of sinks relates to a corresponding increase in the DER.

  Somewhat counter-intuitively, in fact, the increase in the number of sinks does not have a visible impact on reliability (saturation) of the nodes.

3.3 LoRaSim: the LoRa implementation simulator

As a compound to the provided paper, an automated tool has been developed to run a similar set of experiments, aimed at determining the final DER, given a varying number of nodes (and sinks) and tuning some configuration settings according to the experimental runs.

It consists of a group of PYTHON 2 scripts: the interesting ones, for our experimental runs, are `loraDir.py` and `loraDirMulBS.py`, that respectively cover the single and multi-sink cases.

The simulator accepts the following parameters:

- Nodes: number N of nodes to include.
- AvgSend: inverse of the transmission rate: average sending interval, in milliseconds.
- SimTime: simulation duration.
- BaseStations: number M of sinks to include.
- Collision: enables the FULL COLLISION CHECK, as mentioned earlier.
- Experiment: an integer index identifying the characteristics of each experimental run.

  The matching between experimental runs and the experiment sets delineated by the paper dissertation is also sketched.

  0. Use the settings with the slowest datarate (Spreading Factor = SF12, Bandwidth = 125 kHz, Coding Rate = 4/8), similarly to SN_1.
  1. Similar to experiment 0, but uses a random choice of three transmit frequencies. This experimental run does not match a complete experiment set.
  2. Use the settings with the fastest data rate (Spreading Factor = SF6, Bandwidth = 500 kHz, Coding Rate = 4/5), similarly to SN_2.
  3. Optimize the setting per node based on the distance to the gateway, similarly to SN_4.
  4. Use the settings as defined in LoRaWAN (Spreading Factor = SF12, Bandwidth = 125 kHz, Coding Rate = 4/5), similarly to SN_3.
  5. Similar to experiment 3, but also optimizes the transmission power, similarly to SN_5.

3.3.1 The updated version

Following the paper's release, a couple of bugs have been made clear by the scientific community, resulting in an update to the LORASIM simulator.

These updates were, in particular, aimed at correcting the use of logarithms in the path-loss model and revisiting the presumably too optimistic approach taken by the simulator in overestimating the network's throughput.

The paper's authors released an updated version of both the paper and the simulator.

However, since both versions can be interpreted as a solid result and being the outcomes unchanged, they are both taken into consideration when replicating them.

Note that, of course, the updated reference is the most accurate, with respect to the real-world implementation.

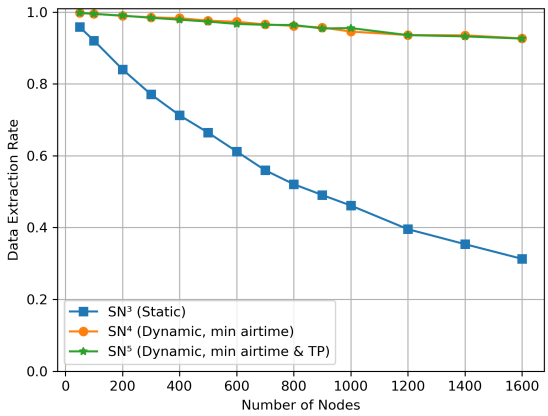
The old version is replicated starting from the updated one, as shown in the following.

```
# Update LoRaSim with its modified old version (see notes)
sudo apt-get install -y sed

cp lorasim/loraDir.py lorasim/loraDir-old.py
cp lorasim/loraDirMulBS.py lorasim/loraDirMulBS-old.py

```

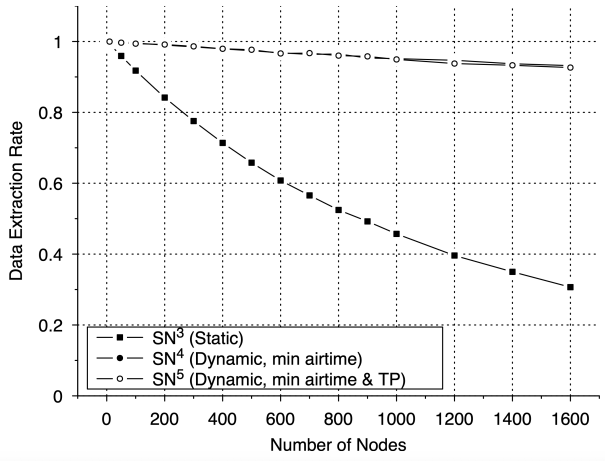
```
# Remove lines 116 and 117 (corresponding to the collisions sub-estimation)
sed -i '116,+1d;' lorasim/loraDir-old.py

# Replace natural logarithms with base-10 logarithm
sed -i 's/math.log10/math.log/g' lorasim/loraDir-old.py

# Remove lines 107 and 108 (corresponding to the collisions sub-estimation)
sed -i '107,+1d;' lorasim/loraDirMulBS-old.py

# Replace natural logarithms with base-10 logarithm
sed -i 's/math.log10/math.log/g' lorasim/loraDirMulBS-old.py
```

3.4 Experimental results

In the following, a comparison between the plot results of Figure 5 and 7 is provided for both the old and updated versions.



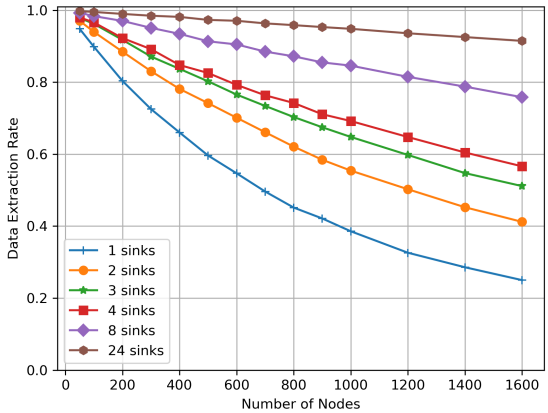
(a) Python simulation



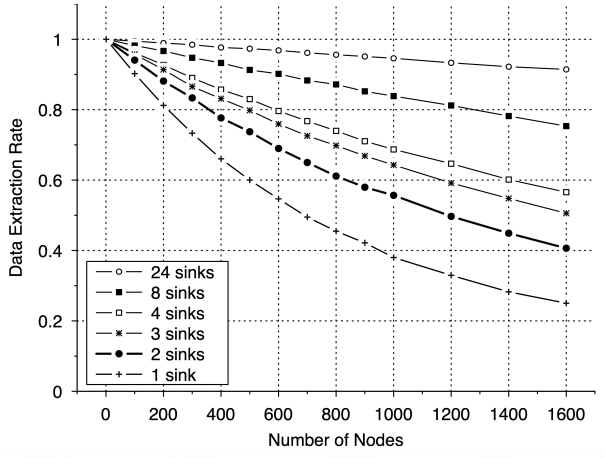
(b) Paper

Comparison: Figure 5, old version of the simulator

Both SN_4 and SN_5 provide a huge improvement to the default LoRaWAN deployment configuration. In fact, up to $N = 1100$ can be accomodated, according to the requirement imposing $DER \geq 0{,}9$. However, these achievements turn out to be impractical, in a real-world scenario, as it relies on optimistic assumptions (no interference, little redundancy and no environment adaptation).



(a) Python simulation

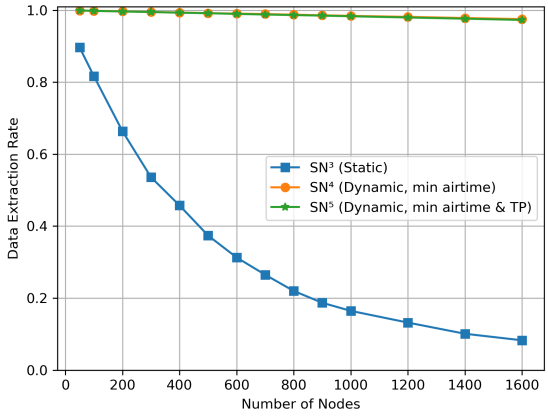


(b) Paper

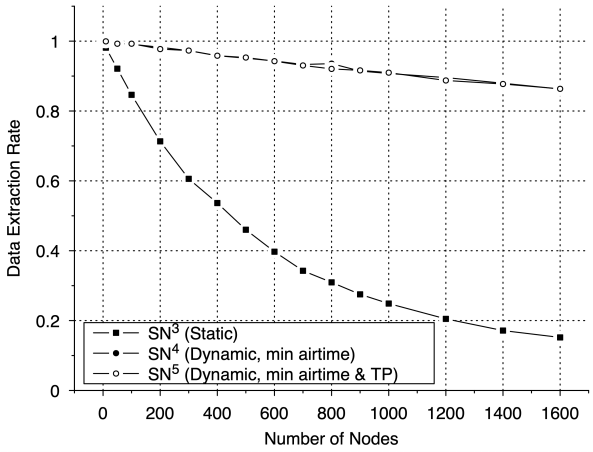
Comparison: Figure 7, old version of the simulator

Results show that the increase in the number of sinks relates to a corresponding increase in the DER. Somewhat counter-intuitively, in fact, the increase in the number of sinks does not have a visible impact on reliability (saturation) of the nodes.

The same also applies to the results produced by the updated version of the simulator.

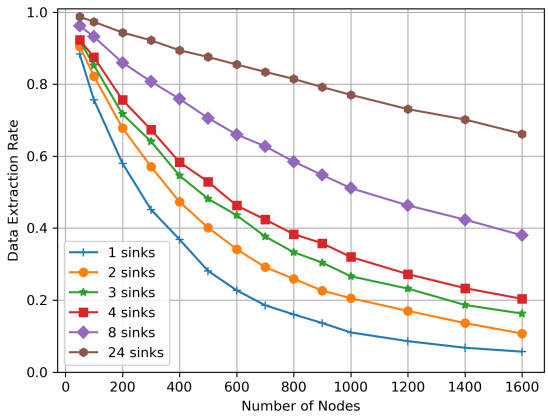


(a) Python simulation

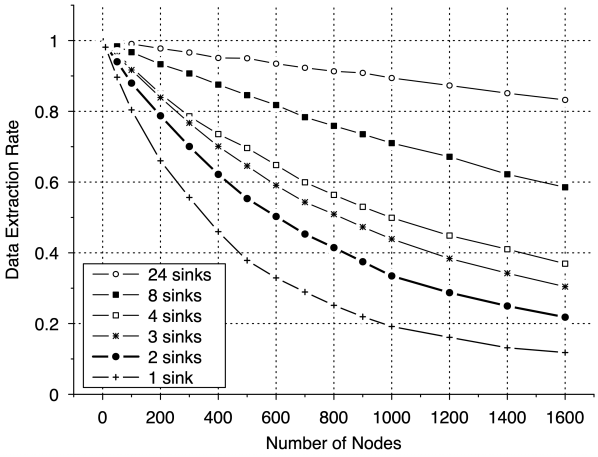


(b) Paper

Comparison: Figure 5, updated version of the simulator



(a) Python simulation



(b) Paper

Comparison: Figure 7, updated version of the simulator

3.5 Implementation details (in Python)

Common libraries import

```
import pandas as pd
import math
import matplotlib.pyplot as plt
import subprocess
import os
import re
import glob
import shutil
```

Simulation primitives

Note that the simulator's output is heavily verbose: it must be discarded.
Otherwise, for higher configurations, it will fill out the machine's memory and crash.

```python
# Simulation output directories (see notes)
UPDATED_VARIANT_DIR = "updated"
OLD_VARIANT_DIR = "old"


def simulate(n_nodes, tx_rate, exp, duration, collisions=True, old=False):
    env = os.environ.copy()
    env["MPLBACKEND"] = "Agg"

    # Old version discriminant (see notes)
    if old:
        old = "-old"
    else:
        old = ""

    # Final output directory (for simulation outputs)
    output_directory = OLD_VARIANT_DIR if old else UPDATED_VARIANT_DIR

    # Use subprocess.run to execute the command and capture output
    result = subprocess.run(
        [
            "python2",
            (
                f"/lorasim/loraDir{old}.py"
                if not COLAB
                else f"{os.getcwd()}/lorasim/loraDir{old}.py"
            ),
            str(int(n_nodes)),
            str(int(tx_rate)),
            str(int(exp)),
            str(int(duration)),
            str(int(collisions)),
        ],
        cwd=f"{os.getcwd()}/{output_directory}",
        env=env,
        # The simulator produces a heavily verbose output: it must be discarded.
        # Otherwise, for higher configurations, it will fill out the machine's memory!
        # The standard simulator's output is discarded, while the errors stream is
        ↪ kept.
        stdout=subprocess.DEVNULL,
        stderr=subprocess.STDOUT,
        # capture_output=True,
        # text=True,  # Capture output as text
    )

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


def simulate_multisink(
    n_nodes, tx_rate, exp, duration, base_stations, collisions=True, old=False
):
    env = os.environ.copy()
```

```
    env["MPLBACKEND"] = "Agg"

    # Old version discriminant (see notes)
    if old:
        old = "-old"
    else:
        old = ""

    # Final output directory (for simulation outputs)
    output_directory = OLD_VARIANT_DIR if old else UPDATED_VARIANT_DIR

    # Use subprocess.run to execute the command and capture output
    result = subprocess.run(
        [
            "python2",
            (
                f"/lorasim/loraDirMulBS{old}.py"
                if not COLAB
                else f"{os.getcwd()}/lorasim/loraDirMulBS{old}.py"
            ),
            str(int(n_nodes)),
            str(int(tx_rate)),
            str(int(exp)),
            str(int(duration)),
            str(int(base_stations)),
            str(int(collisions)),
        ],
        cwd=f"{os.getcwd()}/{output_directory}",
        env=env,
        # The simulator produces a heavily verbose output: it must be discarded.
        # Otherwise, for higher configurations, it will fill out the machine's memory!
        # The standard simulator's output is discarded, while the errors stream is
        ↪ kept.
        stdout=subprocess.DEVNULL,
        stderr=subprocess.STDOUT,
        # capture_output=True,
        # text=True,  # Capture output as text
    )

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


# Der in aloha defined as S/G = e^(-2G)
def aloha_der(n_nodes, t):
    rate = 1e-6
    return math.exp(-2 * n_nodes * rate * t)
```

Environment cleanup and management primitives

```
def reformat_multisink(directory="."):
    # Regular expression that matches files ending in BS{number}.dat
    pattern_file = re.compile(r".*BS\d+\.dat$")
```

```
4
5       # Get all .dat files
6       dat_files = glob.glob(os.path.join(directory, "*.dat"))
7
8       # Filter for the matching ones
9       files = [f for f in dat_files if pattern_file.match(f)]
10
11      for path in files:
12          # Read contents and locally store them
13          with open(path, "r") as file:
14              lines = file.readlines()
15
16          # Update status flag
17          status = False
18
19          for i, line in enumerate(lines):
20              if line.startswith("# "):
21                  # Remove the # extra space (for the format alignment)
22                  lines[i] = "#" + line[2:]
23                  status = True
24
25          # Update file contents with the reformatted one
26          if status:
27              with open(path, "w") as file:
28                  file.writelines(lines)
29
30
31  def simulation_outputs(directory="."):
32      # Get all .dat files
33      dat_files = glob.glob(os.path.join(directory, "*.dat"))
34      # And also computation outputs from the multi-basestation case
35      bs_file = glob.glob(os.path.join(directory, "basestation.txt"))
36      nodes_file = glob.glob(os.path.join(directory, "nodes.txt"))
37      # And finally output figures
38      output_figures = glob.glob(os.path.join(directory, "figure*.png"))
39      return dat_files + bs_file + nodes_file + output_figures
40
41
42  def cleanup_environment(directory="."):
43      for f in simulation_outputs(directory):
44          os.remove(f)
45
46
47  # Automatically invoke cleanup of the environment
48  cleanup_environment(f"./{UPDATED_VARIANT_DIR}")
49  cleanup_environment(f"./{OLD_VARIANT_DIR}")
50
51  try:
52      os.mkdir(f"./{UPDATED_VARIANT_DIR}")
53      os.mkdir(f"./{OLD_VARIANT_DIR}")
54  except:
55      pass
```

Simulation — Figure 5

```
# Problem data
# one packet of length L every 16,7 min = 16.7 * 60 * 1000 ms
transmission_rate = 16.7 * 60 * 1000
# 58 days simulation (see notes)
duration = 58 * 24 * 60 * 60 * 1000
# Enable the full collisions tracing model (see notes)
full_collision_model = True

n_nodes = [50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1200, 1400, 1600]

# Experiments set
experiments = [4, 3, 5]

# Iterate on all the interesting node configurations
for nodes in n_nodes:
    print(
        f"Simulating experiments n. {experiments[0]}, {experiments[1]} and
        ↪  {experiments[2]} on {nodes} nodes"
    )
    for exp in experiments:
        # Updated variant
        simulate(nodes, transmission_rate, exp, duration, full_collision_model, False)
        # Old variant
        simulate(nodes, transmission_rate, exp, duration, full_collision_model, True)
```

```
Simulating experiments n. 4, 3 and 5 on 50 nodes
Simulating experiments n. 4, 3 and 5 on 100 nodes
Simulating experiments n. 4, 3 and 5 on 200 nodes
Simulating experiments n. 4, 3 and 5 on 300 nodes
Simulating experiments n. 4, 3 and 5 on 400 nodes
Simulating experiments n. 4, 3 and 5 on 500 nodes
Simulating experiments n. 4, 3 and 5 on 600 nodes
Simulating experiments n. 4, 3 and 5 on 700 nodes
Simulating experiments n. 4, 3 and 5 on 800 nodes
Simulating experiments n. 4, 3 and 5 on 900 nodes
Simulating experiments n. 4, 3 and 5 on 1000 nodes
Simulating experiments n. 4, 3 and 5 on 1200 nodes
Simulating experiments n. 4, 3 and 5 on 1400 nodes
Simulating experiments n. 4, 3 and 5 on 1600 nodes
```

Extract simulation results files

```
data = []
data_old = []
for exp in experiments:
    data.append(pd.read_csv(f"{UPDATED_VARIANT_DIR}/exp{exp}.dat", sep=" "))
    data_old.append(pd.read_csv(f"{OLD_VARIANT_DIR}/exp{exp}.dat", sep=" "))
```

Data preparation

```
for d in data:
    d["der"] = (d["nrTransmissions"] - d["nrCollisions"]) / d["nrTransmissions"]

for d in data_old:
    d["der"] = (d["nrTransmissions"] - d["nrCollisions"]) / d["nrTransmissions"]

labels = [
    "SN3 (Static)",
    "SN4 (Dynamic, min airtime)",
    "SN5 (Dynamic, min airtime & TP)",
]

markers = ["s", "o", "*"]

# Explicit assertion: datasets and their labels are coherent in size
assert len(data) == len(data_old)
assert len(data) == len(labels)

# Example of the resulting data
data[0].head()
data_old[0].head()
```

Data plot, updated variant

```
plt.figure(dpi=300)  # higher resolution plot
for i in range(0, len(data)):
    # Plot labelled data sets: Number of nodes on the X axis and Data Extraction Rate
    ↪ on the Y axis
    plt.plot(data[i]["#nrNodes"], data[i]["der"], marker=markers[i], label=labels[i])
plt.xlabel("Number of Nodes")
plt.ylabel("Data Extraction Rate")
# Valid Y range is from 0 to 1 (an extra is included for representation spacing)
plt.ylim(0, 1 + 0.01)
plt.legend()
plt.grid()
plt.savefig(f"{UPDATED_VARIANT_DIR}/figure5.png")
plt.show()
```

Data plot, old variant

```
plt.figure(dpi=300)  # higher resolution plot
for i in range(0, len(data_old)):
    # Plot labelled data sets: Number of nodes on the X axis and Data Extraction Rate
    ↪ on the Y axis
    plt.plot(
        data_old[i]["#nrNodes"], data_old[i]["der"], marker=markers[i], label=labels[i]
    )
plt.xlabel("Number of Nodes")
plt.ylabel("Data Extraction Rate")
# Valid Y range is from 0 to 1 (an extra is included for representation spacing)
```

```
plt.ylim(0, 1 + 0.01)
plt.legend()
plt.grid()
plt.savefig(f"{OLD_VARIANT_DIR}/figure5.png")
plt.show()
```

Simulation — Figure 7

```
# Problem data
# one packet of length L every 16,7 min = 16.7 * 60 * 1000 ms
transmission_rate = 16.7 * 60 * 1000
n_sinks = [1, 2, 3, 4, 8, 24]  # base stations (sinks)
# 58 days simulation (see notes)
duration = 58 * 24 * 60 * 60 * 1000
# Enable the full collisions tracing model (see notes)
full_collision_model = True

n_nodes = [50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1200, 1400, 1600]

# Experiments set consists only of a common setting (see notes)
experiment = 0

# Iterate on all the interesting node (and sink) configurations
for sink in n_sinks:
    for nodes in n_nodes:
        print(
            f"Simulating experiment n. {experiment} on {nodes} nodes and {sink} sinks"
        )
        # Updated variant
        simulate_multisink(
            nodes,
            transmission_rate,
            experiment,
            duration,
            sink,
            full_collision_model,
            False,
        )
        # Old variant
        simulate_multisink(
            nodes,
            transmission_rate,
            experiment,
            duration,
            sink,
            full_collision_model,
            True,
        )

# Reformat results for the plot sub-computations, for both the updated and old cases
reformat_multisink(f"./{UPDATED_VARIANT_DIR}")
reformat_multisink(f"./{OLD_VARIANT_DIR}")
```

The output is suppressed, being heavily verbose and very similar to the one of Figure 5.

Extract simulation results files

```
data = []
data_old = []
for sink in n_sinks:
    data.append(
        pd.read_csv(f"{UPDATED_VARIANT_DIR}/exp{experiment}BS{sink}.dat", sep=" ")
    )
    data_old.append(
        pd.read_csv(f"{OLD_VARIANT_DIR}/exp{experiment}BS{sink}.dat", sep=" ")
    )
```

Data preparation

```
# Each sink configuration has its own dataset
labels = [f"{s} sinks" for s in n_sinks]

markers = ["+", "o", "*", "s", "D", "h"]

# Explicit assertion: datasets and their labels are coherent in size
assert len(data) == len(data_old)
assert len(data) == len(labels)

# Example of the resulting data
data[0].head()
data_old[0].head()
```

Data plot, updated variant

```
plt.figure(dpi=300)  # higher resolution plot
for i in range(0, len(data)):
    # Plot labelled data sets: Number of nodes on the X axis and Data Extraction Rate
    ↪ on the Y axis
    plt.plot(data[i]["#nrNodes"], data[i]["DER"], marker=markers[i], label=labels[i])
plt.xlabel("Number of Nodes")
plt.ylabel("Data Extraction Rate")
# Valid Y range is from 0 to 1 (an extra is included for representation spacing)
plt.ylim(0, 1 + 0.01)
plt.legend()
plt.grid()
plt.savefig(f"{UPDATED_VARIANT_DIR}/figure7.png")
plt.show()
```

Data plot, old variant

```
plt.figure(dpi=300)  # higher resolution plot
for i in range(0, len(data_old)):
    # Plot labelled data sets: Number of nodes on the X axis and Data Extraction Rate
    ↪ on the Y axis
    plt.plot(data_old[i]["#nrNodes"], data_old[i]["DER"], marker=markers[i],
    ↪ label=labels[i])
plt.xlabel("Number of Nodes")
```

```
plt.ylabel("Data Extraction Rate")
# Valid Y range is from 0 to 1 (an extra is included for representation spacing)
plt.ylim(0, 1 + 0.01)
plt.legend()
plt.grid()
plt.savefig(f"{OLD_VARIANT_DIR}/figure7.png")
plt.show()
```

3.6 Optionally: a Docker container

The LoRaSim simulator and its revised variant need installation, on the local system, of both Python 2, for the simulator and Python 3 for the notebook's execution.

Therefore, to provide a common abstraction layer, a specifically crafted Dockerfile is included in the repository, alongside a full Development Container environment, providing all the most common Jupyter notebooks requirements and LoRaSim.

When built, it can be used to serve as an interpreter for the project, integrating it with the embedded Python Conda environment.

```
FROM --platform=linux/amd64 quay.io/jupyter/base-notebook

#### INSTALL DEPENDENCIES ####
RUN conda update --all --yes \
    && conda install --yes notebook nest-asyncio pandas numpy scapy matplotlib
    ↪  scikit-learn

#### INSTALL DEPENDENCIES ####
USER root

# Python2 is no more available for Ubuntu 24.04, downloading and installing manually...
RUN apt-get update && export DEBIAN_FRONTEND=noninteractive \
    && apt-get -y install --no-install-recommends curl wget tar sed \
    && cd /tmp/ \
    && wget ... # Python 2 and its dependencies... (omitted for clarity)
    && apt-get -y install --no-install-recommends ./*.deb \
    && curl -o get-pip.py https://bootstrap.pypa.io/pip/2.7/get-pip.py

#### INSTALL LoRaSim AND ITS DEPENDENCIES ####
RUN curl -o lorasim.tgz https://www.lancaster.ac.uk/scc/sites/lora/lorasim-20170710.tgz
 ↪  \
    && tar -xvf lorasim.tgz \
    && mkdir /lorasim/ \
    && mv ./lorasim/* /lorasim/

#### UPDATE LoRaSim WITH ITS MODIFIED VERSION ####
RUN cp /lorasim/loraDir.py /lorasim/loraDir-old.py \
    && cp /lorasim/loraDirMulBS.py /lorasim/loraDirMulBS-old.py \
    # Remove lines 116 and 117 (corresponding to the collisions sub-estimation)
    && sed -i '116,+1d;' /lorasim/loraDir-old.py \
    # Replace natural logarithms with base-10 logarithm
    && sed -i 's/math.log10/math.log/g' /lorasim/loraDir-old.py \
    # Remove lines 107 and 108 (corresponding to the collisions sub-estimation)
    && sed -i '107,+1d;' /lorasim/loraDirMulBS-old.py \
    # Replace natural logarithms with base-10 logarithm
```

```
    && sed -i 's/math.log10/math.log/g' /lorasim/loraDirMulBS-old.py \
    # Permissions setup
    && chmod -R 777 /lorasim/

#### PATH SETUP ####
ENV PATH="$PATH:/home/$NB_USER/.local/bin"

#### INSTALL LoRaSim DEPENDENCIES ####
# Unprivileged notebook user
USER ${NB_UID}

RUN python2 /tmp/get-pip.py \
    && pip2 install -r '/lorasim/requirements.txt'

#### CLEAN UP ####
USER root

# Generally, Dev Container Features assume that the non-root user (in this case jovyan)
# is in a group with the same name (in this case jovyan). So we must first make that so.
# (see https://github.com/devcontainers-community/templates-jupyter-datascience-notebooks/)
RUN groupadd jovyan \
    && usermod -g jovyan -a -G users jovyan

RUN rm -rf /tmp/*

#### RESTORE UNPRIVILEGED USER ####
USER ${NB_UID}
```