IoT Challenge 3
Exercise Part

Giovanni Ni 10831328 (Leader)
Xinyue Gu 10840236

EQ1:

EQ1) 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%. Hint: use https://www.thethingsnetwork.org/airtime-calculator to compute the airtime of a packet.

Report the result in the form!!

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

e.g. personcode = 106929**11** -> **XY** = 11 -> **L =** 3 **+11 = 14**

In our case, since the leader's person code ends with 28, our payload size is 3+28 bytes = 31.

To solve this exercise we consider the aloha success rate formula studied during the lesson:

$$Succ.\,Rate = \frac{S}{G} = e^{-2G} = e^{-2N*\lambda*t}$$

Where N is the number of Nodes, in our case equals to 50

λ is the tx. Rate, in our case we have 1 packet/minute so 1/60 packet/second

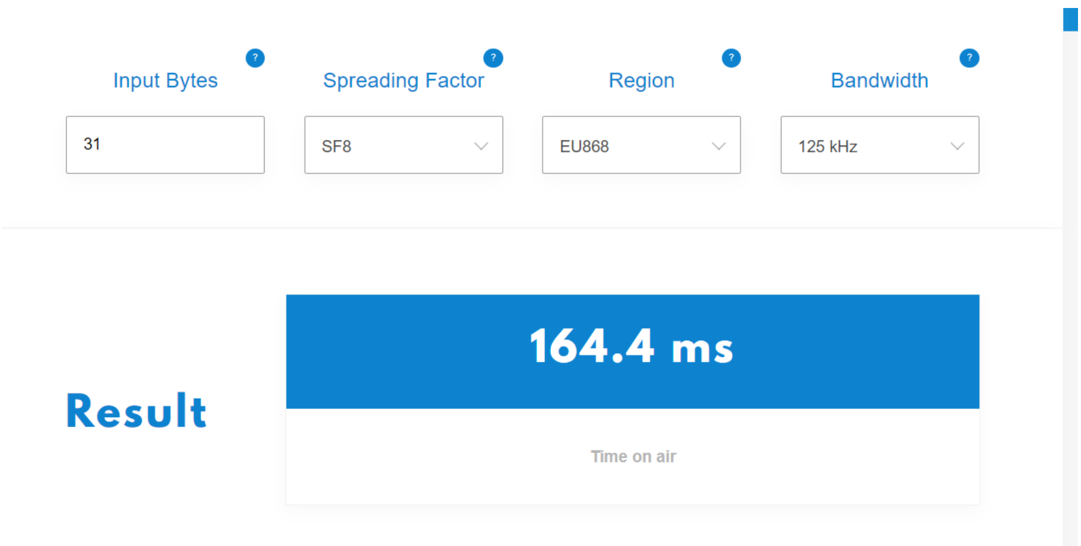
t is the packet airtime, which is unknown.

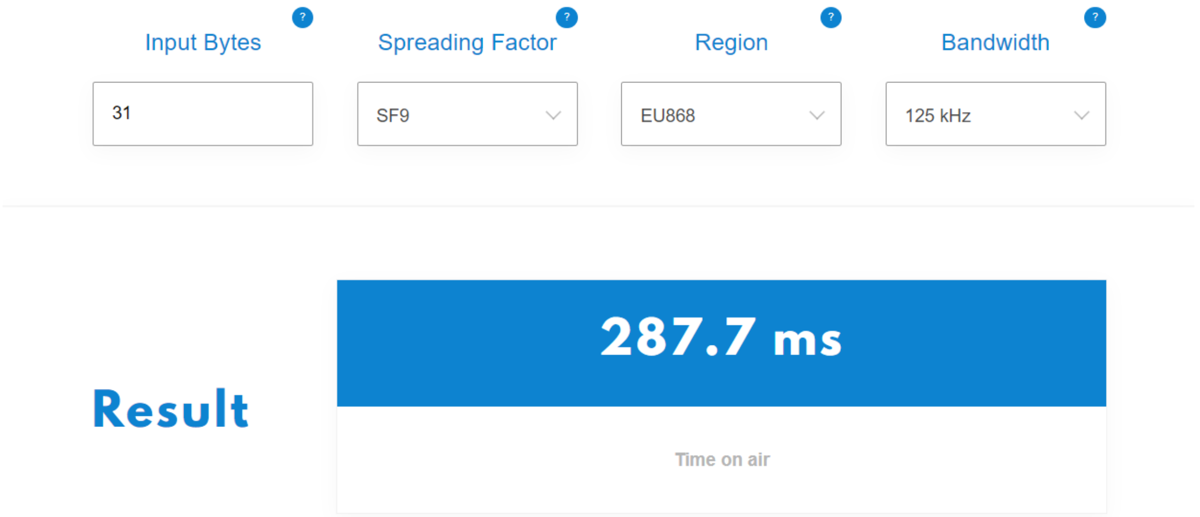
Using the formula we obtain that: $t = -\frac{\ln(SR)}{2*N*\lambda} = -\frac{\ln(0.7)}{100*\frac{1}{60}}$= 0.214s

We observe that if the packet airtime decrease, the success rate increases, it makes sense because intuitively less time on air → smaller collision window → less chance of two packets overlapping → better success.

So that means with t = 0.214s we have a Succ.Rate of 0.7, by increasing t the succ. rate would decrease, so 0.214s is the maximum value I can have.

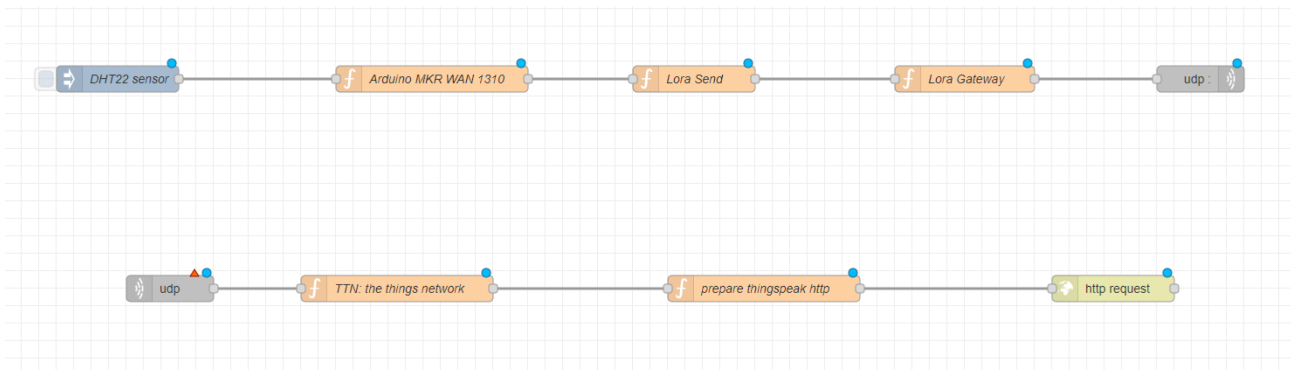
Now using the airtime-calculator, we observe that:



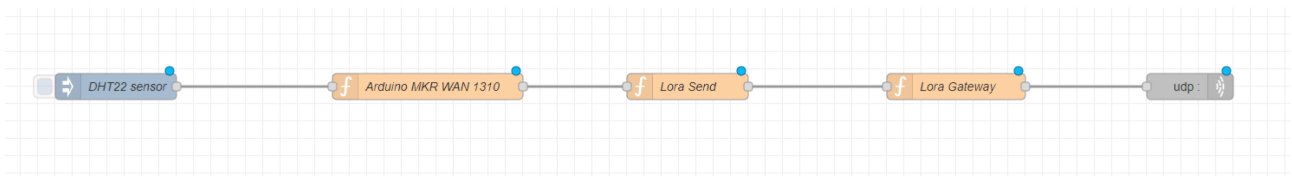


With the Spreading Factor 9 we obtain a t > 0.214s.
So in our case the biggest LoRa SF for having a success rate of at least 70% is SF8.

EQ2

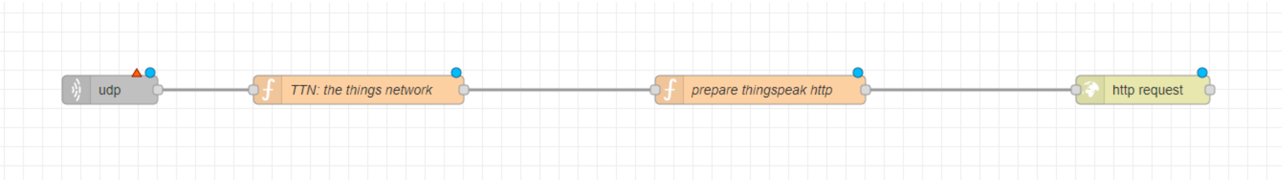


Our system has two parts: the Arduino side and the TTN: the things network side.



The Arduino side starts with the DHT22 sensor, which measures the temperature and humidity and sends the data, for example every 5 minutes, to the Arduino MKR WAN 1310 board.
The Arduino board reads the sensor values and forwards them to the LoRa transceiver (built into the MKR WAN 1310).
The LoRa transceiver packages the data into a LoRaWAN packet following the LoRaWAN protocol (adding headers frame counter)
This LoRaWAN packet is then transmitted wirelessly (over radio) to the nearby LoRa Gateway.

The LoRa Gateway receives the wireless LoRaWAN packet and wraps it inside a UDP packet, adding some extra metadata (like RSSI, SNR, timestamp, frequency, etc.).
The gateway is configured with the IP address and UDP port of the TTN router.
Then, through the UDP protocol — which is connectionless, lightweight, and faster than HTTP — the gateway forwards the packet to TTN.



Meanwhile, The Things Network (TTN) server is always listening on its UDP port, waiting to receive incoming packets from gateways.
When TTN receives a new packet, it decodes the LoRaWAN message to extract the original sensor data (temperature and humidity).
After decoding, TTN prepares a HTTP Webhook request containing the sensor data and sends it to ThingSpeak.
Finally, ThingSpeak receives the HTTP POST, stores the temperature and humidity values, and displays them in the channel.

EQ3

Figure 5 presents the second experiment from the paper.
We use LoRasim:

```
import subprocess

def simulate(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
    )
```

For Figure 5, we based our setup on the function provided in lorasim/loraDir.py, as it is suitable for scenarios where multiple nodes (N nodes) transmit to a single sink (which is the case here).
We set up the variables as follows:

- Tx_rate: A 20-byte packet is sent every 16.7 minutes. The 20-byte size is the default setting in LoRasim, and we set tx_rate to 1e6.
- CF: The carrier frequency is 868 MHz. We did not modify this setting, as it is already correctly set in loraDir.py.
- Duration: The original experiment in the paper uses 58 days; however, we set the duration to only 1 day (86400000 ms), because our simulator cannot handle such a long simulation time. Moreover, there seems to be little difference between simulating 58 days and 1 day for our purposes.
- Number of nodes: We increase the number of nodes as follows:

```
duration = 86400000
tx_rate = 1e6

for n_nodes in list(range(1, 10)) + list(range(10, 300, 10)) + list(range(300, 1601, 100)):
    print(f"Simulating {n_nodes} nodes")
    simulate(n_nodes, tx_rate, 4, duration)
    simulate(n_nodes, tx_rate, 3, duration)
    simulate(n_nodes, tx_rate, 5, duration)
```

Finally, for the experiment settings, we use exp4 for SN3, exp3 for SN4, and exp5 for SN5, based on the following details from loraDir.py:

```
if experiment==1 or experiment == 0:
        self.sf = 12
        self.cr = 4
        self.bw = 125

# for certain experiments override these
if experiment==2:
        self.sf = 6
        self.cr = 1
        self.bw = 500
# lorawan
if experiment == 4:
        self.sf = 12
        self.cr = 1
        self.bw = 125
```

```
if (experiment == 3) or (experiment == 5):
    minairtime = 9999
    minsf = 0
    minbw = 0

    print "Prx:", Prx

    for i in range(0,6):
        for j in range(1,4):
            if (sensi[i,j] < Prx):
                self.sf = int(sensi[i,0])
                if j==1:
                    self.bw = 125
                elif j==2:
                    self.bw = 250
                else:
                    self.bw=500
                at = airtime(self.sf, 1, plen, self.bw)
                if at < minairtime:
                    minairtime = at
                    minsf = self.sf
                    minbw = self.bw
                    minsensi = sensi[i, j]
    if (minairtime == 9999):
        print "does not reach base station"
        exit(-1)
```

After the simulation, we obtained the .dat files:

exp3.dat × exp4.dat exp5.dat

```
#nrNodes nrCollisions nrTransmissions OverallEnergy
1 0 91 0.309044736
2 0 185 0.492188928
3 0 243 0.453682944
4 0 330 0.859237632
5 0 425 0.7934784
6 0 491 1.028324352
7 0 599 1.50978432
8 0 691 1.290102528
9 0 725 1.729736448
10 0 869 2.133145344
20 2 1689 4.038718464
30 4 2507 5.925266688
40 2 3492 8.655727872
50 10 4265 9.653960448
60 2 5233 12.467364096
70 8 5985 14.849970432
80 18 6789 15.740938752
90 10 7733 17.526330624
100 12 8593 22.229889792
110 19 9554 21.909355776
120 20 10498 26.054130432
130 36 11268 25.615535616
140 28 12005 29.329496064
150 20 12872 30.888557568
160 37 13811 32.900093952
170 53 14738 36.156840192
180 50 15532 35.747762688
190 78 16589 39.319087104
```

exp3.dat | exp4.dat | exp5.dat

```
#nrNodes  nrCollisions  nrTransmissions  OverallEnergy
1  0  83  14.449999872
2  2  177  30.815059968
3  2  253  44.046385152
4  2  356  61.978312704
5  4  399  69.464457216
6  0  503  87.570481152
7  16  595  103.58734848
8  18  677  117.863251968
9  12  782  136.143372288
10  14  859  149.548793856
20  92  1775  309.0210816
30  158  2634  458.569875456
40  355  3396  591.231320064
50  555  4306  749.659029504
60  724  5093  886.672883712
70  1041  6105  1062.85842432
80  1282  6936  1207.53251942
90  1542  7576  1318.95420518
100  2066  8653  1506.45601075
110  2382  9502  1654.26384077
120  2767  10339  1799.98251418
130  3390  11405  1985.56925952
140  3575  12050  2097.8614272
150  4095  12857  2238.35720909
160  4609  13597  2367.18853325
170  5265  14674  2554.69033882
180  5757  15566  2709.98431334
190  6308  16340  2844.73491456
200  7219  17326  3016.39394918
```

exp3.dat | exp4.dat | exp5.dat

```
#nrNodes  nrCollisions  nrTransmissions  OverallEnergy
1  0  84  0.092671488
2  0  182  0.297321024
3  0  253  0.579984
4  0  344  0.700161408
5  0  413  0.8490336
6  0  529  0.974678016
7  0  612  1.09457856
8  0  682  1.043397504
9  0  776  1.162806528
10  0  843  1.653111552
20  2  1647  2.934328896
30  0  2601  5.100284352
40  8  3469  6.109875072
50  8  4341  7.746935616
60  2  5085  8.385372288
70  2  5893  11.431347264
80  10  6947  13.702473792
90  16  7691  13.269643776
100  8  8593  15.626239872
110  26  9691  15.469831296
120  19  10135  17.88523872
130  16  11240  20.90629824
140  18  12199  20.805533568
150  30  12951  24.03960096
160  56  13895  23.121468288
170  50  14777  26.627817792
180  54  15509  28.011558528
190  78  16503  29.307442176
200  66  17331  29.866023744
```

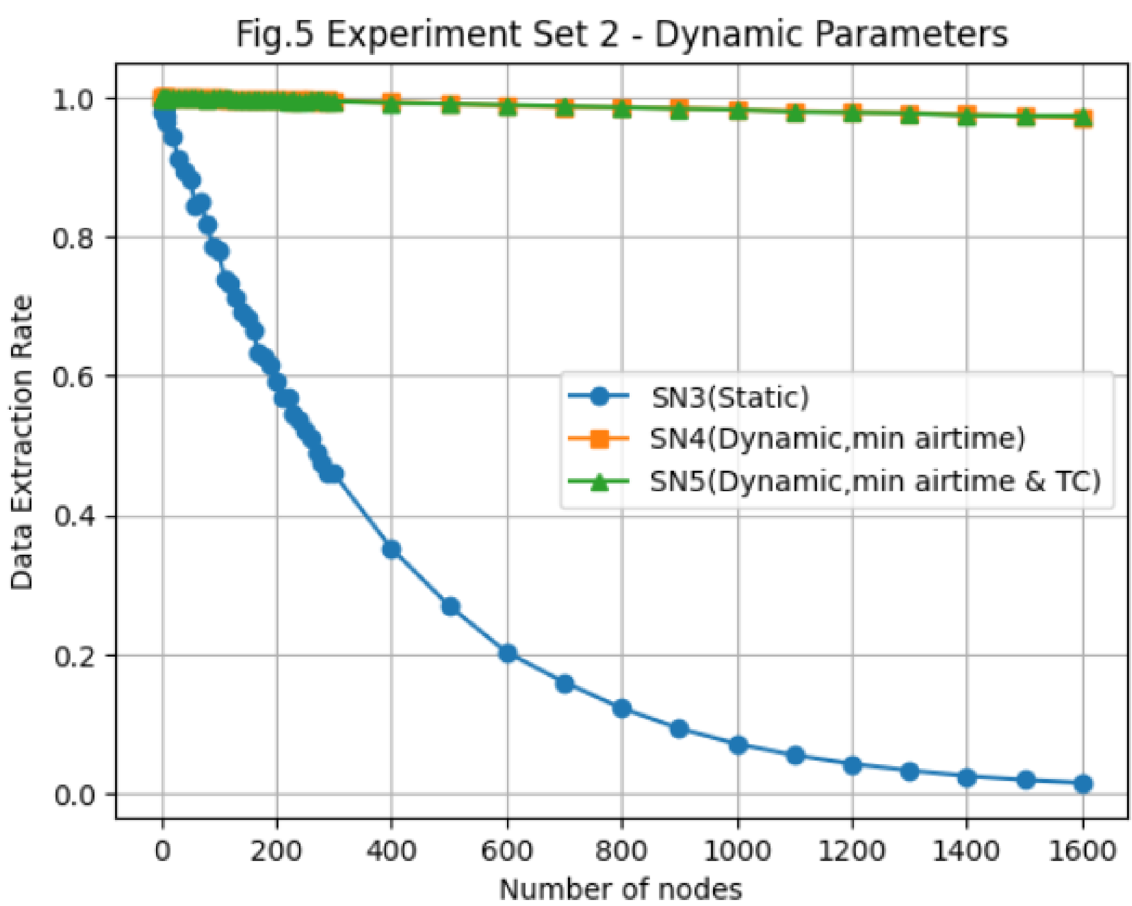
Then we calculate the Data Extraction Rate:

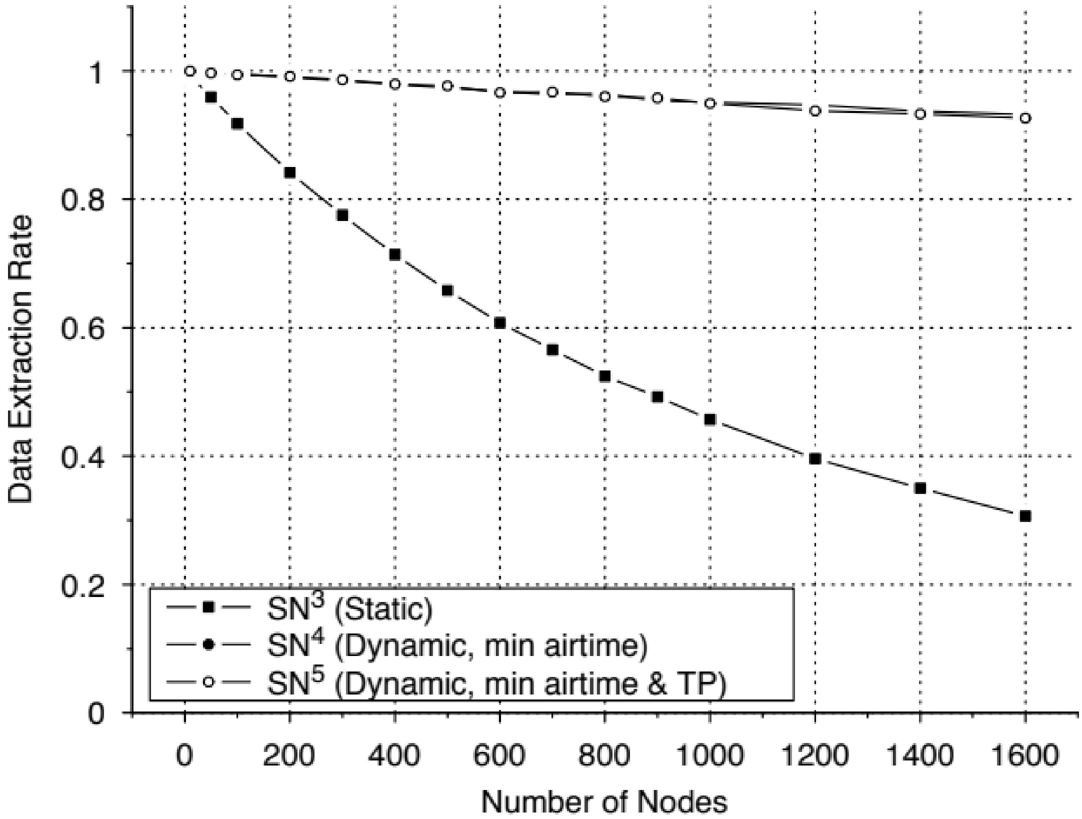
```python
sn3 = pd.read_csv("exp4.dat", sep=" ")
sn4 = pd.read_csv("exp3.dat", sep=" ")
sn5 = pd.read_csv("exp5.dat", sep=" ")
```

```python
sn3["der"] = (sn3["nrTransmissions"] - sn3["nrCollisions"]) / sn3["nrTransmissions"]
sn4["der"] = (sn4["nrTransmissions"] - sn4["nrCollisions"]) / sn4["nrTransmissions"]
sn5["der"] = (sn5["nrTransmissions"] - sn5["nrCollisions"]) / sn5["nrTransmissions"]
```

Finally, we plotted the data, and the figure looks very similar to the one in the paper.

```python
import matplotlib
import matplotlib.pyplot as plt

plt.plot(sn3["#nrNodes"], sn3["der"], label="SN3(Static)",marker='o')
plt.plot(sn4["#nrNodes"], sn4["der"], label="SN4(Dynamic,min airtime)", marker='s')
plt.plot(sn5["#nrNodes"], sn5["der"], label="SN5(Dynamic,min airtime & TC)", marker='^')
plt.title("Fig.5 Experiment Set 2 - Dynamic Parameters")
plt.xlabel("Number of nodes")
plt.ylabel("Data Extraction Rate")
plt.legend()
plt.grid()
plt.show()
```





In both cases, we observe that optimal allocation of settings in terms of airtime (and airtime plus TP) has a huge impact on achievable DER. With minimized airtime (SN4) and a DER > 0.9 requirement, well over N = 1600 nodes can be supported. This is a dramatic improvement compared to the N = 120 nodes achieved with the static, conservative settings used in LoRaWAN.

For Figure 7, we proceed in a similar way as for Figure 5, but in this case, we use the functions in lorasim/loraDirMulBS.py, because in this experiment the number of sinks also increases (not just a single sink as before).

Most of the variables remain the same, but we run a simulation for each number of sinks (1, 2, 3, 4, 8, 24). In particular, we set collision = 1, whereas by default it is 0. However, due to an unknown issue, passing the parameter collision = 1 in the “simulate” function does not work. Therefore, for this particular case, we manually edit loraDirMulBS.py and change the default value of “full_collision” from False to True.

```
# do the full collision check
full_collision = True

```

```python
import subprocess

def simulate(n_nodes, tx_rate, exp, duration,n_sinks):
        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))
                ],
                env=env,
                capture_output=True,
                text=True,    # Capture output as text
        )
```

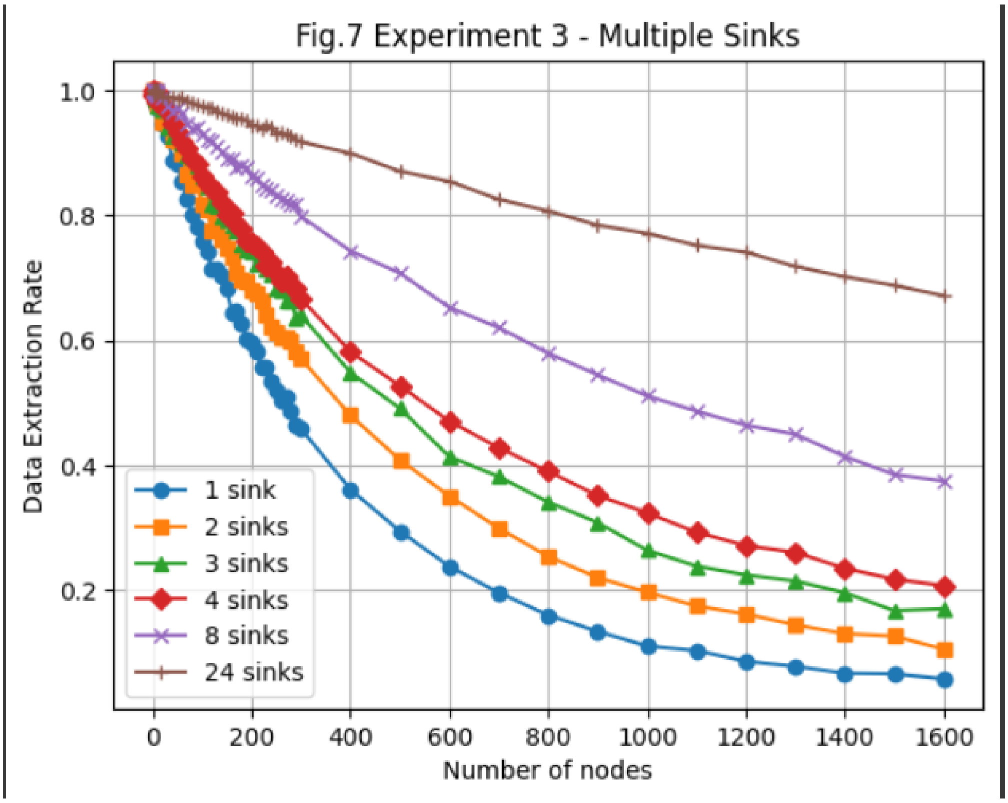
```python
duration = 86400000
tx_rate = 1e6

for n_nodes in list(range(1, 10)) + list(range(10, 300, 10)) + list(range(300, 1601, 100)):
    print(f"Simulating {n_nodes} nodes")
    simulate(n_nodes, tx_rate, 0, duration, 1)
    simulate(n_nodes, tx_rate, 0, duration, 2)
    simulate(n_nodes, tx_rate, 0, duration, 3)
    simulate(n_nodes, tx_rate, 0, duration, 4)
    simulate(n_nodes, tx_rate, 0, duration, 8)
    simulate(n_nodes, tx_rate, 0, duration, 24)
```

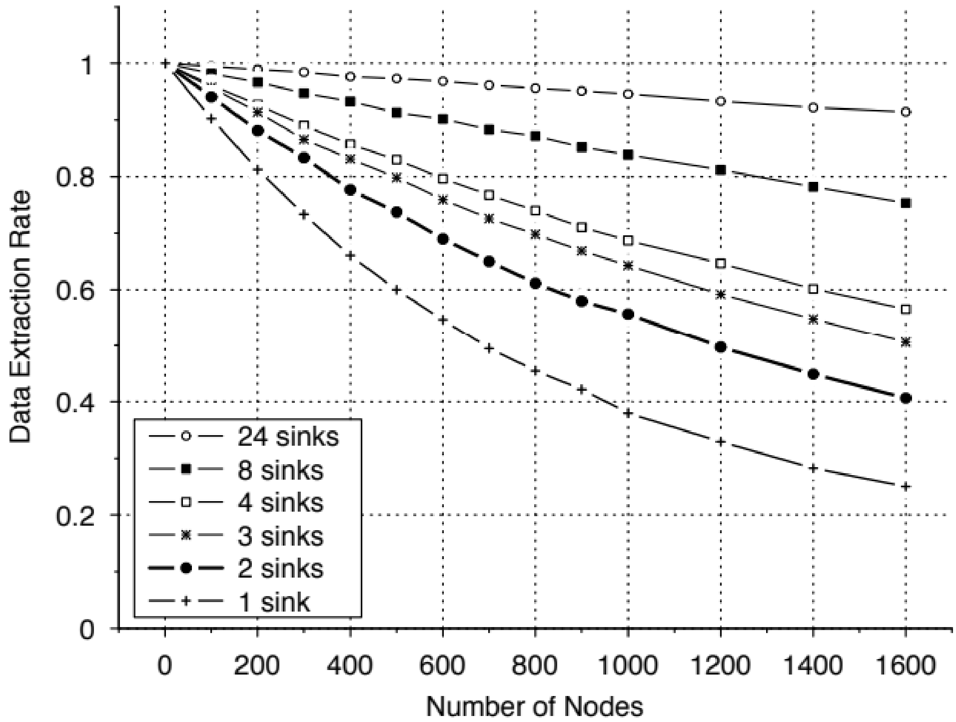
For all simulations we decided to use exp0.

```python
if experiment==1 or experiment == 0:
        self.sf = 12
        self.cr = 4
        self.bw = 125
```

Here, we do not need to calculate the DER separately, as it is already contained in the .dat file. We simply read the data from the file and plot it in a figure.
(There is a problem with the simulator file: there is a space between the “#” and “nrNodes”, so the system recognizes “#” as the number of nodes and “nrNodes” as the DER.)

Again, our plot is similar to the one in the paper:





With more sinks, the chances increase that a packet finds a sink where the capture effect works to its advantage. With an infinite number of sinks, each node could potentially find a sink and avoid packet loss.