LoRaWAN Network Implementation and Optimization

B. Karakostov
S5230837
University of Groningen
Groningen, Netherlands
b.karakostov@student.rug.nl

M. David
S5528984
University of Groningen
Groningen, Netherlands
m.david.5@student.rug.nl

R. Zare Hassanpour
Project Supervisor
University of Groningen
Groningen, Netherlands
r.zare.hassanpour@rug.nl

***Abstract*—This report presents the design and evaluation of a single-channel LoRaWAN network optimized for throughput and energy efficiency. We describe our experimental setup and investigate strategies for adaptive spreading factor allocation and transmit power control in the presence of inter-node interference utilizing Proportional-integral-derivative (PID) control. Our results demonstrate that careful parameter tuning can significantly improve performance in constrained single-channel scenarios.**

I. Introduction

LoRaWAN (Long Range Wide Area Network) is a low-power wide-area network protocol designed for wireless battery-operated devices in regional, national, or global networks. Built on top of the LoRa (Long Range) physical layer modulation, LoRaWAN defines the communication protocol and system architecture, while LoRa itself handles the physical layer that enables long-range links.

The technology operates in unlicensed ISM (Industrial, Scientific, and Medical) radio bands – typically 868 MHz (EU868) and 433 MHz (EU433) in Europe, 915 MHz in North America, and 433 MHz in Asia. In this project we focus on the 433 MHz band.

LoRaWAN[1] is well suited for Internet of Things (IoT) applications because it provides:

- Long-range communication, enabled by LoRa's chirp spread spectrum modulation;
- Low power consumption, thanks to optimized sleep modes and minimal transmission overhead;
- Secure communication via end-to-end AES-128 encryption as required by the specification.

The architecture follows a star-of-stars topology: end devices (sensors/actuators) communicate directly with gateways, which relay messages to a central network server over standard IP.

An important feature of LoRaWAN is the use of multiple spreading factors (SF7-SF12), which trade off data rate for range and robustness. In a production deployment, multi-channel gateways based on the Semtech SX1301/SX1302 concentrator can receive on several SFs and channels simultaneously. However, many low-cost or experimental setups rely on single-channel gateways, which can only listen on one SF and one frequency at a time. A situation this project directly addresses.

LoRaWAN defines three device classes with different capabilities and power profiles, summarized in Table 1.

TABLE I
LoRaWAN Device Classes

Class	Receive Windows	Power Consumption	Use Cases
Class A	Two short windows after each uplink	Lowest	Battery-powered sensors
Class B	Class A + scheduled receive slots	Medium	Actuators with predictable downlink
Class C	Continuous (except when transmitting)	Highest	Mains-powered devices, low-latency applications

Class A devices offer the lowest power consumption and are most suitable for battery-operated sensors. Class B devices enable more predictable downlink latency through time-synchronized beacons. Class C devices provide minimal latency but require continuous power supply.

The protocol uses several algorithms (notably Adaptive Data Rate – ADR) to optimize transmission parameters including spreading factor, bandwidth, and transmission power based on conditions. Spreading factors range from SF7 (fastest data rate, shortest range) to SF12 (slowest data rate, longest range). Table 2 shows the characteristics of each.

TABLE II
LoRa Spreading Factor Characteristics (125 kHz bandwidth, EU868)

SF	Data Rate (bps)	Sensitivity (dBm)	Relative Range	Characteristics
SF7	5470	−123	1×	Fastest, shortest range, lowest ToA
SF8	3125	−126	1.4×	Balanced speed and range
SF9	1760	−129	2×	Good compromise for urban areas
SF10	980	−132	2.8×	Reliable for medium-long range
SF11	440	−134.5	4×	Long range, high reliability
SF12	250	−137	5.6×	Maximum range, highest ToA

II. Motivation

In a standard multi-channel deployment, gateways can receive on multiple SFs and frequencies concurrently, so parameter allocation is relatively forgiving. On a single-channel gateway, however, all nodes must share one frequency and typically one SF at any given time, making collisions far more likely and parameter choices far more consequential. When multiple nodes transmit on the same channel, inter-node interference degrades signal quality, causing packet loss, retransmissions, and increased energy consumption.

This work focuses on that constrained scenario: a small multi-node LoRaWAN network operating over a single channel. The central question is: **how can we maximize data throughput while minimizing energy consumption when all nodes share a single receive channel and inter-node interference is unavoidable?**

III. Related Work

Several studies have examined LoRaWAN performance and the impact of parameter selection.

Magrin et al.[2] provide an analysis of how different LoRaWAN parameter settings: spreading factor, bandwidth, coding rate, and transmission power, affect network performance metrics such as packet delivery ratio, energy consumption, and fairness. Their simulation-based study highlights the strong interdependence between these parameters and the difficulty of finding a single optimal configuration.

Lavric and Popa[3] evaluate LoRaWAN scalability in large-scale wireless sensor networks, showing that alongside node density increases, collision rates grow significantly, particularly when all nodes use the same spreading factor.

On the energy side, de Andrade et al.[4] propose a dynamic SF reallocation strategy that accounts for the battery levels of individual nodes.

Kamarudin et al.[5] present a broader review of LoRaWAN performance, covering indoor and outdoor scenarios, and identify open issues including interference management, adaptive data rate behavior, and gateway capacity limitations. Their work provides context for understanding where single-channel setups fall short compared to multi-channel concentrators.

Our work builds on these findings by focusing specifically on the single-channel case. Rather than relying on simulation alone, we implement a real hardware testbed using an SDR-based gateway for full control over the physical layer.

IV. Experimental Setup

A. Objectives

We investigate how a LoRa network can optimize the trade-off between data rate and energy consumption by employing our own spin on adaptive transmit power control combined with dynamic spreading factor (SF) allocation. Each node will adjust its transmit power based on gateway feedback of the received signal-to-noise ratio (SNR), using a PD-based control loop to maintain reliable communication while minimizing energy use. Simultaneously, the network will utilize SF adaptation and coordinated timing to mitigate interference between nodes.

B. Network Hierarchy

Our setup implements the standard LoRaWAN star-of-stars topology, illustrated in Fig. 1. This architecture consists of end devices communicating directly with a central gateway via LoRa modulation, while the gateway connects to backend servers through standard IP networking (UDP).

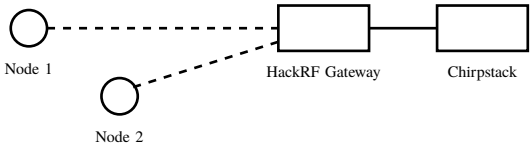

Fig. 1. Network Topology

C. Hardware Configuration

Our experimental setup consists of multiple components forming a complete LoRaWAN network. The gateway uses a software-defined radio (SDR) approach.

a) *Nodes:* STM32F401RE microcontroller with a hat (as shown in the GitHub repository[6]), consisting of:

- SX1278 LoRa transceiver module operating at 433 MHz;
- BME280 environmental sensor (temperature, humidity, pressure);
- Standard photoresistor for light measurement;
- An SSD1306 OLED screen for display.

b) *Gateway:* Our gateway is built around a HackRF One[7] software-defined radio (SDR), controlled via GNU Radio and the `gr-lora_sdr`[8], [9], [10] out-of-tree module.

The gateway software stack consists of:

- *GNU Radio flowgraph (Python)*: handles LoRa modulation/demodulation via `gr-lora_sdr` blocks and interfaces with the HackRF through `gr-osmosdr`;
- *Packet Forwarder*: implements the Semtech UDP packet forwarder protocol to relay received packets to ChirpStack;
- *Transmitter*: handles downlink by encoding and modulating LoRaWAN frames for transmission via the HackRF.

The whole system is containerized using Docker, with `gr-lora_sdr` built from source.

Receiver chain: Fig. 2 shows the GNU Radio Companion schematic of the receive chain. The actual implementation lives in `receiver.py`[11], where the flowgraph is constructed programmatically. The signal path is:

$$\begin{aligned}&\text{HackRF Source}\\&\rightarrow \text{Frame Sync} \rightarrow \text{FFT Demod} \rightarrow \text{Gray Demapping}\\&\rightarrow \text{Deinterleaver} \rightarrow \text{Hamming Dec.} \rightarrow \text{Header Dec.}\\&\rightarrow \text{Dewhitening} \rightarrow \text{CRC Verify} \rightarrow \text{Packet Sink}\end{aligned} \tag{1}$$

The `Frame Sync` block locks onto the LoRa preamble and synchronizes with the incoming chirps. After demodulation and decoding, packets are passed to a custom `LoRaPacketSink` block that forwards them to the packet forwarder via a callback. The oversampling factor is computed

as $\frac{S_r}{B_w}$, and the minimum buffer size is set to $\left\lceil \frac{S_r}{B_w} \cdot 2^{\mathrm{SF}} + 2 \right\rceil$, as each LoRa symbol spans 2^{SF} samples at a given bandwidth (and the +2 is a magic number).

SNR and RSSI Estimation: The `frame_sync` block has an optional second output port (type `float32`) that, when connected, enables per-packet SNR estimation from the preamble. Internally, the block decodes each preamble upchirp by multiplying with the ideal downchirp, takes the FFT, and computes:

$$\mathrm{SNR} = 10 \log_{10} \left(\frac{E_{\mathrm{peak}}}{E_{\mathrm{total}} - E_{\mathrm{peak}}} \right) \tag{2}$$

where E_{peak} is the energy in the strongest FFT bin and E_{total} is the total energy across all bins. The estimate is averaged over all usable preamble symbols. The block produces five floats per frame on port 1: SNR, CFO, STO, SFO, and an off-by-one indicator. A `SyncLogSink` block captures these values per SF chain.

Since the HackRF One is not a calibrated receiver, absolute RSSI cannot be measured directly. Instead, we extract it the `gr-lora_sdr` `fft_demod` block's `rssi` output.

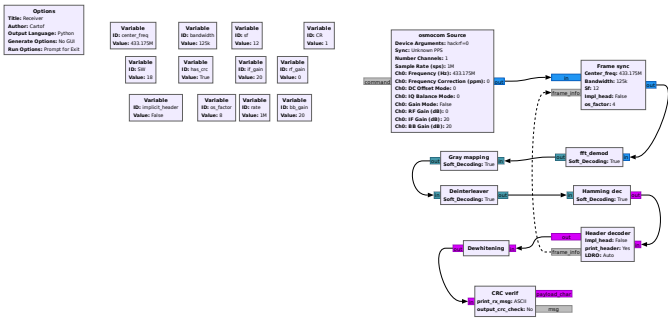

Fig. 2. GNU Radio Companion schematic of the LoRa receive chain

Transmitter chain: Fig. 3 shows the transmit chain schematic. Again, the real implementation is in `transmitter.py`[11]. The signal path mirrors the receiver in reverse:

$$\begin{aligned} \text{Whitening} &\rightarrow \text{Header} \rightarrow \text{Add CRC} \\ &\rightarrow \text{Hamming Enc.} \rightarrow \text{Interleaver} \\ &\rightarrow \text{Gray Demap} \rightarrow \text{Modulate} \\ &\rightarrow \text{HackRF Sink} \end{aligned} \tag{3}$$

Downlink packets arrive as raw bytes from ChirpStack (via `PULL_RESP`), get wrapped as PMT messages, and are posted to the whitening block's input port. The transmitter starts the flowgraph, waits for the frame to be fully transmitted, and then stops.

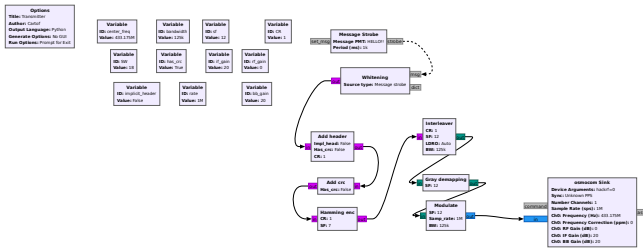

Fig. 3. GNU Radio Companion schematic of the LoRa transmit chain

Packet Forwarder: The packet forwarder bridges the GNU Radio flowgraph and ChirpStack. It implements the Semtech UDP protocol[12] (PUSH_DATA, PULL_DATA, PULL_RESP, etc.) and formats received packets as `rxpk` JSON objects with the appropriate metadata (frequency, data rate string like `SF12BW125`, RSSI, SNR). It also handles downlink scheduling by listening for `PULL_RESP` messages and routing them to the transmitter.

Multi-SF / Multi-BW Reception: We run multiple parallel receiver chains, each tuned to a different spreading factor, sharing the same HackRF source. Since LoRa spreading factors are orthogonal[1] , a single capture can be split into multiple demodulation pipelines. This mitigates the single-channel limitation by allowing simultaneous reception of packets transmitted with different SFs.

Half-Duplex Operation: The HackRF One cannot receive and transmit at the same time (its half-duplex), so when ChirpStack wants to send a downlink the packet forwarder has to stop the receiver, spin up the transmitter, wait for the frame to go out, and restart the receiver. We compensate for the timing mismatch by informing Chirpstack that we have handled downlink before actually doing so. In reality, the transmission is delayed by a user-defined constant (as USB and transmit/receive chain overhead differs based on hardware) On the node side, RX1 opens 5.5 s after TX and RX2 at 7.5 s, with a 250 ms scan guard.

Network Server: We run ChirpStack v4[11] as the network server, deployed in Docker alongside PostgreSQL 14, Redis 7, Mosquitto 2 (MQTT broker), and the ChirpStack Gateway Bridge. The gateway bridge is set up for the EU433 band with MQTT topics prefixed by `eu433/`. ChirpStack does device management, packet deduplication, MAC commands, and data routing. The built-in ADR is not disabled and need not be, as the PD controller sends data rate and power commands through the ChirpStack gRPC API directly.

Simulation: For testing at larger scale than our physical setup allows, we set up an ns-3 simulation environment with the LoRaWAN module from Magrin et al.[2] (`signetlabdei/lorawan`). The `setup.sh` script clones ns-3, grabs the LoRaWAN module and builds with examples and tests. This lets us compare our physical measurements with simulation results.

[1]Each SF corresponds to a different chirp rate, so signals with different SFs do not interfere with each other.

D. Methodology

a) *Channel Access: Slotted TDMA:* Since all nodes share one frequency channel, if two transmit at the same time a collision will arise. To avoid this we use a simple TDMA scheme. Each node gets a fixed slot index $s \in \{0, 1, ..., N_{\text{slots}} - 1\}$, and the n-th uplink from node s happens at:

$$t_{\text{tx}}(s, n) = t_0 + n \cdot T_{\text{interval}} + s \cdot \frac{T_{\text{interval}}}{N_{\text{slots}}} \tag{4}$$

Right now $T_{\text{interval}} = 60$ s and $N_{\text{slots}} = 2$. The slot offset is computed at compile time:

$$\text{TDMA\_SLOT\_OFFSET\_MS} = \frac{T_{\text{interval}}}{N_{\text{slots}}} \times s \tag{5}$$

Each node is offset by 30 seconds from the other, which leaves enough time for the uplink, both RX windows, and processing. There is no hardware clock sync – the nodes need to be started roughly at the same time, and the fixed-interval loop in the firmware handles the rest. We have not encountered any collisions with this setup.

b) *PD-Based Transmit Power and Spreading Factor Control:* Instead of using ChirpStack's built-in ADR, we wrote our own PD (proportional-derivative) control loop that adjusts each node's TX power and spreading factor independently. The controller is a C shared library (`libalgo.so`) called from a Python wrapper (`wrapper.py`). The wrapper subscribes to uplink events over MQTT and pushes downlink commands through the ChirpStack gRPC API.

Per-Node State: The controller keeps track of each node's state in a hash table (using `uthash`):

- Current spreading factor SF $\in [7, 12]$, starts at SF7;
- Current TX power $P \in [2, 17]$ dBm, starts at 14 dBm;
- Previous error e_{k-1}, starts at 0;
- Stability counter – how many frames in a row the node has had excess margin at minimum power.

SNR Target: Each spreading factor needs a different minimum SNR to demodulate reliably. We use these targets:

TABLE III
SNR DEMODULATION TARGETS PER SPREADING FACTOR

SF	7	8	9	10	11	12
Target SNR (dB)	−7.5	−10.0	−12.5	−15.0	−17.5	−20.0

Control Law: When an uplink arrives, the gateway has already estimated the SNR from the preamble (see SNR and RSSI Estimation above). The error signal is:

$$e_k = \text{SNR}_{\text{target(SF)}} - \text{SNR}_{\text{measured}} \tag{6}$$

Positive error means the link is weaker than desired, negative means there is extra margin. The PD law gives us a power adjustment:

$$\Delta P = K_p \cdot e_k + K_d \cdot (e_k - e_{k-1}) \tag{7}$$

with $K_p = 0.5$ and $K_d = 0.1$. We quantize to 2 dBm steps (what the SX1278 supports) and clamp to the allowed range:

$$P_{k+1} = \text{clamp}(P_k + \text{round}(\Delta P/2) \cdot 2, P_{\min}, P_{\max}) \tag{8}$$

SF Escalation and De-Escalation: We provide a heuristic that manages SF changes:

- *SF goes up (link stressed):* If measured SNR is below target, or even at max power the SNR margin would still be less than a comfort threshold ($C = 10$ dB), we increase SF up by one, set TX power back to $P_{\max}$, and reset the PD state (previous error and stability counter go to zero). This lets the controller start fresh at the better-fitting config.
- *SF goes down (too much margin):* If the error exceeds a margin threshold ($M = 5$ dB) and TX power has already been at $P_{\min}$ for at least $W = 3$ frames in a row, we drop SF by one. Power gets reset to $P_{\max}$ again and PD state is cleared, so the controller can re-converge at the faster data rate.

Closed-Loop Integration: The full loop can be summarized as follows (see Fig. 4):

1) Node transmits sensor data over LoRa.
2) HackRF gateway demodulates and sends the packet (with RSSI/SNR) to ChirpStack via UDP.
3) ChirpStack publishes the uplink to the MQTT broker.
4) Python wrapper gets the MQTT message, pulls out RSSI and SF, converts to internal DR (DR $= 12 -$ SF), calls the C `pid()` function.
5) The result (new DR and TX power) gets packed as 8 bytes ($2 \times$ int32, little-endian) and enqueued as a downlink on FPort 2 through ChirpStack's gRPC API. Any old queued downlinks get flushed first so the node always gets the latest command.
6) Node receives the downlink, unpacks the `State` struct, and calls `lora.reBegin(params)` which updates the target DR. Takes effect on the next uplink.

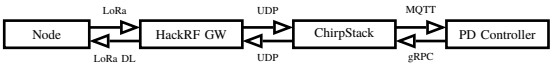

Fig. 4. Closed-loop control: uplink path (top), downlink path (bottom)

V. RESULTS

A. Field Test

To validate the gateway hardware and SNR/RSSI estimation pipeline, we carried out a walk test through the city centre. A single node transmitted at fixed TX power (2 dBm, SF7) while being carried to several locations at different distances from two gateway positions. Fig. 5 through Fig. 7 summarise the results.

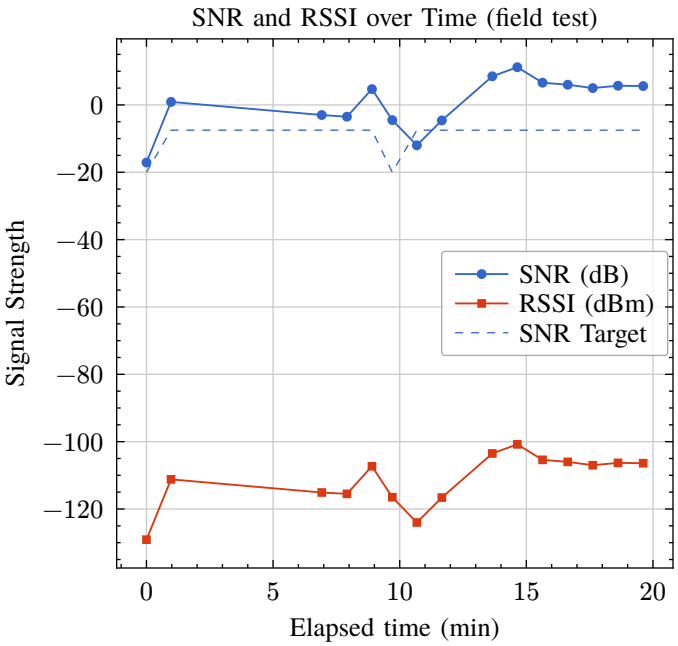

Fig. 5. Measured SNR and RSSI over elapsed time during the field test. Both gateway and node were mobile. Dashed line shows the SNR target for the current spreading factor.

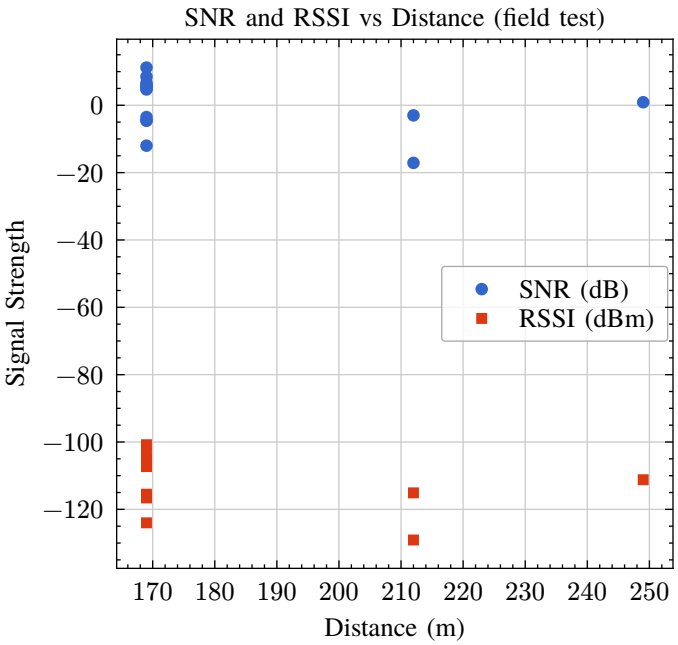

Fig. 6. SNR and RSSI as a function of estimated node-gateway distance. Distance is measured from the gateway's location (first listed node, or primera) to each node position.

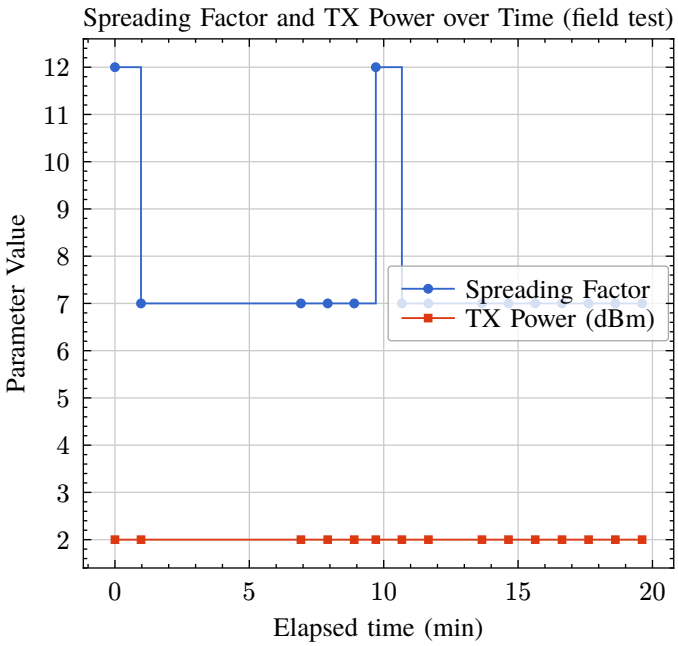

Fig. 7. Spreading factor and transmit power used by the node at each uplink during the field test.

B. *Simulation*

To evaluate the performance of our PD-based control strategy at larger scale, we set up an ns-3 simulation. The results shown here focus on a single node to clearly demonstrate the control behavior. Since our TDMA scheme (see Methodology) prevents collisions by assigning each node a dedicated time slot, the per-node dynamics are largely independent – additional nodes will exhibit similar adaptation patterns within their respective slots. Therefore, observing one node provides shows how the PD controller manages spreading factor and transmit power under varying link conditions.

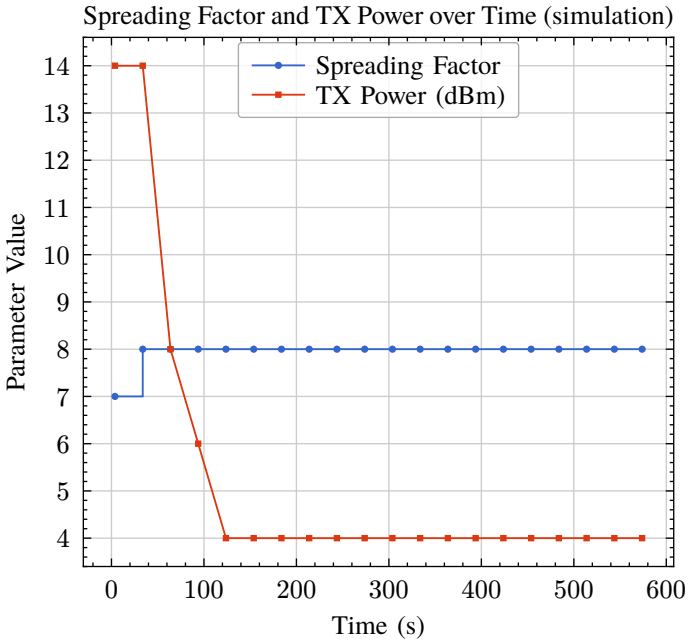

Fig. 8. Spreading factor and transmit power adaptation over time under PD control.

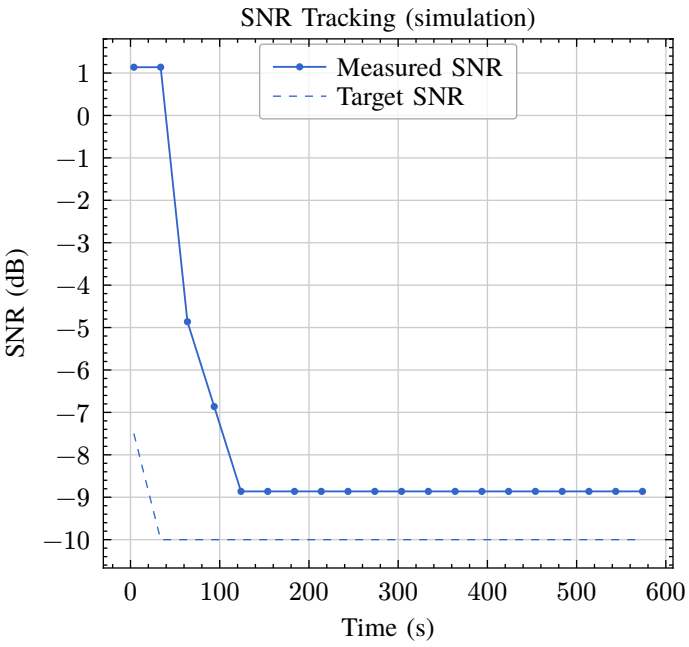

Fig. 9. SNR tracking performance showing measured SNR (solid) and target SNR (dashed). The PD controller adjusts TX power and SF to maintain SNR near the target.

VI. Discussion

A. Field Test Analysis

As shown in Fig. 7, the node transmitted at a fixed SF7 and 2 dBm throughout the entire test, which is clearly sub-optimal for the range of distances covered. Upon investigation, we traced this behaviour to the received-power calculation used by the ADR algorithm in the firmware. The SNR targets configured in the ADR implementation are conservative values on the order of -7 dB (one target per spreading factor), designed so that the algorithm increases SF or TX power when the measured SNR falls below the target. However, during our field test the observed SNR consistently remained around $+10$ dB – well above these thresholds – causing the algorithm to conclude that the link budget had ample margin. Hence, ADR never triggered an increase in spreading factor or transmit power, and the PID controller driving the adaptation continually drove both parameters toward their lowest (most energy-efficient) settings: SF7 and the minimum TX power of 2 dBm.

This indicates that the RSSI values reported by the gateway are higher than the true received signal level, likely due to an offset in the Rx power calculation. This is clearly a limitation of the HackRF One, which lacks accurate absolute power calibration and tends to report inflated RSSI values.

B. Simulation Analysis

The simulation shows the PD controller working as intended. Fig. 9 demonstrates that measured SNR tracks the target well, with fast response to changes and only small steady-state error (expected without an integral term).

Fig. 8 shows the two-stage adaptation strategy. When the link is strong, the controller minimizes both SF and TX power (SF7, low power) for maximum throughput and efficiency. As conditions degrade, it first increases TX power. When power hits its limit, it raises the spreading factor to maintain reliability.

Each SF change produces a step in the target SNR (dashed line in Fig. 9), and the controller tracks these new targets appropriately.

The simulation validates that PD control can balance throughput and reliability when RSSI measurements are accurate – unlike the field test where HackRF calibration issues prevented proper operation.

VII. Limitations & Future Work

There are a few limitations with what we have so far.

Since the HackRF cannot RX and TX at the same time, sending downlinks means we have to stop receiving. Combined with LoRaWAN's narrow RX windows (especially at high SFs where packets take a long time), we have hardcoded a ≈ 15 second delay in the loop. We also altered the underlying `RadioLib` to support this. The PD controller deals with this by always flushing old commands and sending the latest state, but it still slows down how fast the control loop converges.

Our TDMA scheme uses fixed slot assignments set at compile time with no clock sync. Works fine for two nodes, but scaling up would need some kind of dynamic slot allocation, maybe coordinated through downlink MAC commands.

The controller is PD, not PID – there's no integral component, which means it cannot fully eliminate steady-state error.

We tested with two physical nodes even though the original plan was three. A third node would stress-test the TDMA scheme more and give us better data on interference.

VIII. LLM Transparency

Parts of this project involved large language model assistance:

- This report was produced with the help (structuring and wording) of Claude Opus 4.6 (Anthropic).
- The TDMA logic and transmission chain implementation were heavily assisted by Claude Opus 4.6.

References

[1] Lora Alliance, "LoraWan specification." [Online]. Available: https://lora-alliance.org/wp-content/uploads/2021/11/LoRaWAN-Link-Layer-Specification-v1.0.4.pdf

[2] D. Magrin, M. Capuzzo, and A. Zanella, "A Thorough Study of LoRaWAN Performance Under Different Parameter Settings," *IEEE Internet of Things Journal*, 2019.

[3] A. Lavric and V. Popa, "Performance Evaluation of LoRaWAN Communication Scalability in Large-Scale Wireless Sensor Networks," *Wireless Communications and Mobile Computing*, 2018.

[4] I. H. de Andrade, L. H. M. K. Costa, and R. S. Couto, "Battery Life Optimization in LoRa Networks Using Spreading Factor Reallocation," *Ad Hoc Networks*, 2025.

[5] M. N. B. C. Kamarudin, A. B. Ayob, A. B. Hussain, and M. G. M. Abdolrasol, "Review of LoRaWAN: Performance, Key Issues and Future Perspectives," *Jurnal Kejuruteraan*, 2024.

[6] B. Karakostov and M. David, "Printed Circuit Board of the STM32 hat." [Online]. Available: https://github.com/RUG-BI-LORA-25/periphs

[7] Great Scott Gadgets, “HackRF One - Great Scott Gadgets.” [Online]. Available: https://greatscottgadgets.com/hackrf/one/
[8] J. Tapparel, O. Afisiadis, P. Mayoraz, A. Balatsoukas-Stimming, and A. Burg, “An Open-Source LoRa Physical Layer Prototype on GNU Radio,” 2020.
[9] J. Tapparel and A. Burg, “Design and Implementation of LoRa Physical Layer in GNU Radio,” 2024.
[10] J. Tapparel, “gr-lora_sdr GitHub Repository.” [Online]. Available: https://github.com/tapparelj/gr-lora_sdr
[11] B. Karakostov and M. David, “Codebase for the LoRaWAN gateway implementation.” [Online]. Available: https://github.com/RUG-BI-LORA-25/code
[12] Lora-net, “Semtech UDP Protocol Specification.” [Online]. Available: https://github.com/Lora-net/packet_forwarder/blob/master/PROTOCOL.TXT