Spring 2024

Final Report

Title of the project:	"Simulator for Time-Slotted LoRa Networks"
Team Members:	Assel Abzalova, Denis Ten, Abylay Kairatbek
Project Advisor/Co-Advisors	Assistant Professor Dimitrios Zormpas

Executive Summary

The introduction of the Time-Slotted (TS) LoRa protocol necessitated a new approach to protocol testing due to the scarcity of available nodes for large-scale experiments. To address this, we developed a versatile simulator that allows for comprehensive testing and analysis of TS LoRa networks under various conditions. The development involved several key tasks: we refactored the existing code to improve its structure and efficiency, removed the joining phase to eliminate initial connection delays, and eliminated the SACK packet to reduce network clutter. We also implemented a policy of allowing up to three additional retransmissions to enhance reliability before integrating the RL model. This model dynamically adjusts the number of retransmissions based on real-time network conditions, thus optimizing network performance and reducing unnecessary data transmission.

Introduction

In the realm of network development, practical experimentation is crucial for advancing new protocols. One such protocol, the newly introduced Time-Slotted (TS) LoRa protocol, requires experimental validation, which cannot be conducted in real-world settings due to a scarcity of available testing nodes. These nodes are inadequate for testing large networks comprising more than 10 physical objects. To address this challenge, a new simulator was developed to evaluate and understand the performance of Time-Slotted LoRa networks across various scenarios. The simulator allows users to set specific parameters such as node count, simulation duration, average wake-up time, and packet size, and incorporates essential functions like data transmission in designated time slots and management of desynchronization.

The simulator's design facilitates the active monitoring of critical metrics, including packet loss and energy usage, enabling detailed insights. Additionally, a reinforcement learning algorithm has been integrated to refine the optimization of packet retransmission frequencies. Ultimately, the simulator operates in two modes: a general mode that displays TS-Lora statistics, and an advanced mode equipped with the RL DQN model, which demonstrates potential energy savings by optimizing retransmission counts for each node.

Background and Related Work

Due to the specific challenges associated with testing large-scale networks, the creation of a simulator was identified as the only practical solution. In our search for an effective approach, we examined existing simulators, taking particular note of one for LoRa networks referenced in [1], which implemented an aloha-style experiment. However, since our TS-Lora protocol employs a time-slot method rather than aloha, it necessitated the development of new components such as frame classes.

Additionally, we considered improvements upon another simulator (for TS-Lora) discussed in [2]. Our project builds upon this foundation, focusing on addressing its several deficiencies:

- *Data Collisions.* The earlier version did not address collisions due to clock drift between time slots.
- *Connection Issues.* During the joining phase, about 90% of nodes failed to connect in setups with more than 20 nodes.
- *Initialization Failures.* Many nodes were not receiving their spreading factor (SF) settings at initialization.
- *Lack of Detailed Statistics.* There was no provision for comprehensive metrics like Packet Reception Ratio (PRR) and Packet Delivery Rate (PDR).
- *Scalability.* The codebase was not scalable as it was confined to a single file.

Our revised simulator aims to rectify these issues, offering a robust and scalable tool for more extensive and complex network testing.

Project Approach

The project shows the solution in the form of a simulator. The simulator was written in Python, SimPy discrete environment and is a console application, showing the detailed statistics at

the end of the run. It accepts 4 network parameters, including the number of nodes, simulation duration, average wake-up time, and packet size. Currently, our application receives parameters in the command line as follows: usage: *./main <number_of_nodes> <data_size(bytes)> <avg_wake_up_time(secs)> <sim_time(secs)>.* It also shows the visual representation of the physical location of the nodes and the gateway during the simulation:

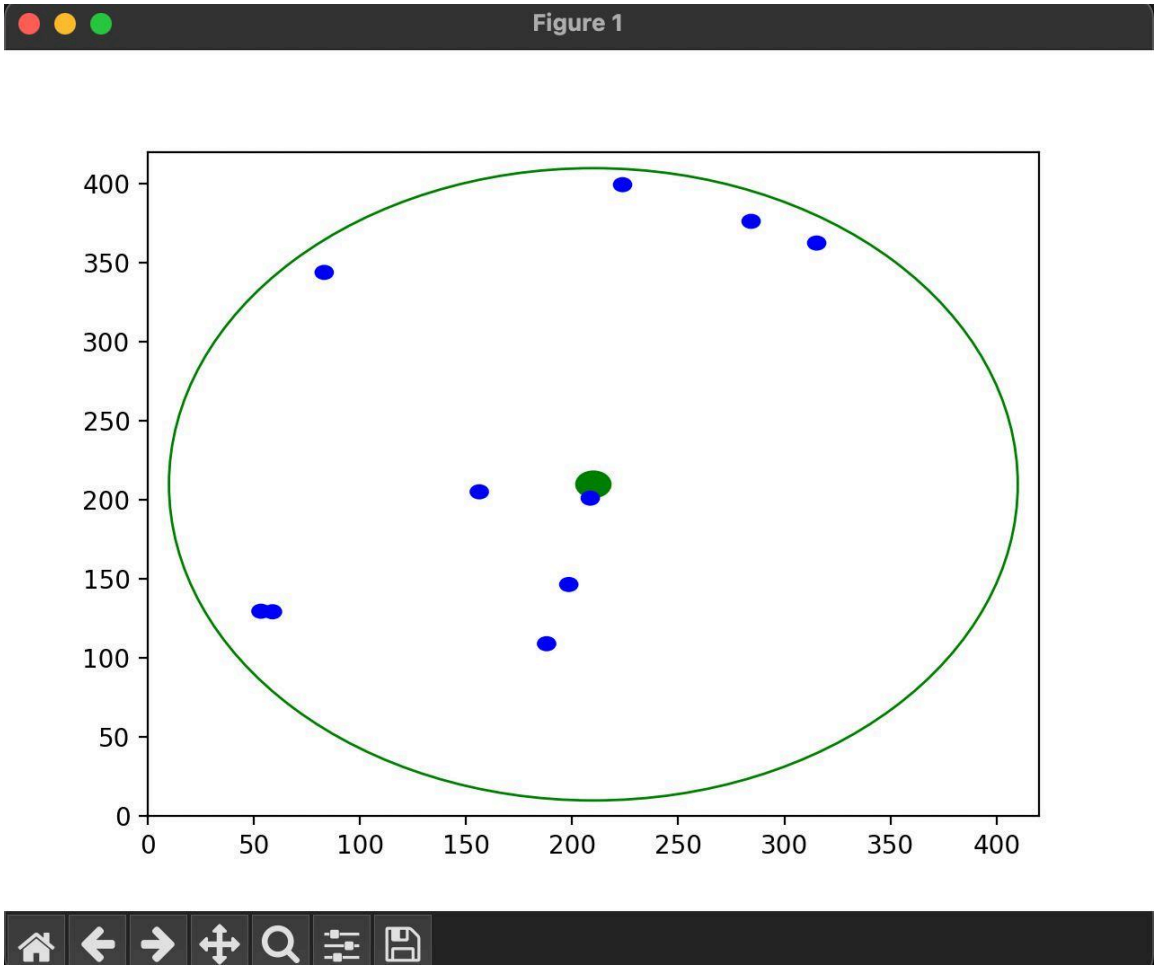

Figure 1. Visual demonstration of the nodes and the gateway locations

The work drew on the theoretical knowledge presented in existing literature [4][5][6]. For example, the Figure 2 below illustrates the Network star topology of Time Slotted LoRa. In this topology nodes communicate with a central gateway which is in turn connected to network and application servers for network management, node authentication, and data processing.

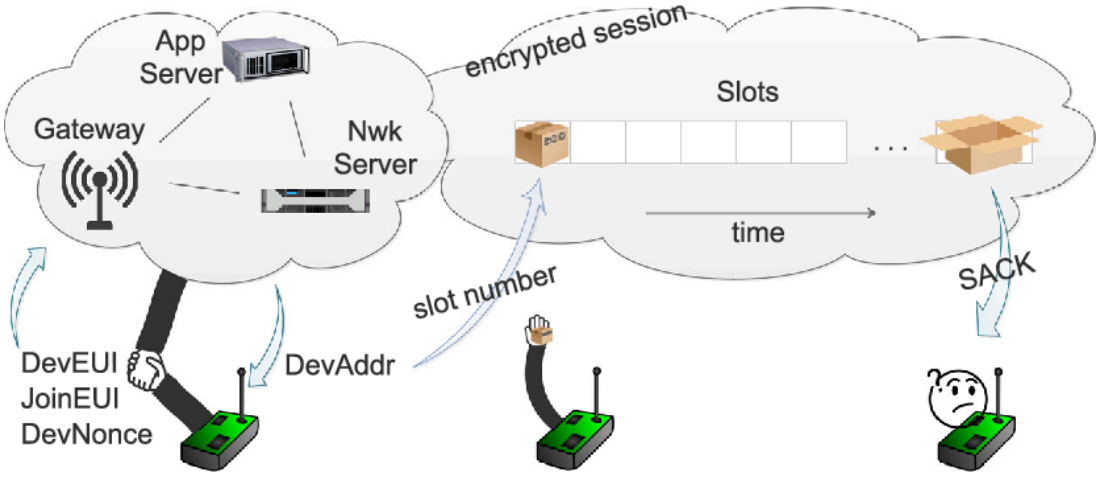

Figure 2. Network star topology of TS-LoRa

The Figure 2 shows a TS-LoRa network setup, showcasing a central gateway that serves as a hub for node communications [4]. This gateway liaises with a network server to coordinate operations and may connect to an application server that processes the incoming data.

For the joining phase, the simulation assumes that nodes are already connected to the network. This assumption allows the simulator to handle a high volume of nodes effectively, bypassing the initial registration process to focus on performance testing under load.

During the data transmission phase, nodes follow a predetermined schedule to wake up, gather sensor data, and transmit this information [5]. The system assigns each node a distinct time slot for its transmission to avoid any signal interference, thus ensuring a smooth and systematic flow of data within the network.

The TS-LoRa protocol incorporates a synchronization/acknowledgement phase (SACK) for time synchronization and the acknowledgment of data packets by the gateway. This confirms the successful reception of data transmitted from nodes and maintains the robustness of the communication process.

A timeline with designated slots demonstrates the time-division mechanism that controls transmission timings. Each node is given a slot number, which prescribes its specific transmission window, eliminating the risk of collision by ensuring that simultaneous transmissions do not occur.

During the first semester, Fall 2023, the project commenced with the implementation of a triple uplink transmission in our LoRa network simulator. This feature ensured that each data packet transmitted by a node would be sent three times, drastically improving the probability of

successful reception to 99.9%. We then addressed the inefficiency of the joining phase, which previously consumed half of the simulation time and struggled with scalability as node numbers increased, leading to high failure rates and collisions during node registration.

As for the second semester, Spring 2024, after developing the core components of the code and implementing triple uplink retransmissions, we explored reducing this number to improve resource efficiency. This concept formed the foundation for incorporating a reinforcement learning model, which would analyze various network parameters and autonomously determine the ideal number of uplinks or retransmissions in real scenarios.

Our approach used Reinforcement Learning for the control of the number of transmissions between the nodes and the gateway. Among all the available RL techniques, including SARSA, Q-learning, and Deep Q-learning, we have decided to use the last one (DQN - Deep Q-learning). This form of RL was selected because it can effectively manage the complex patterns and subtleties found in the network data.

The plan was to use parameters like SF (spreading factor), PRR (packet reception rate), RSSI (received signal strength indicator) and rate of successful transmissions as part of the model's state space. It was thought that a node's proximity to the gateway and its Spreading Factor (SF) significantly affect its packet transmission success rate. Nodes nearer to the gateway or with a lower SF typically have higher success rates, thereby reducing the need for multiple retransmissions.

The chosen deep reinforcement learning model initially required an environment. For this purpose, Gymnasium, a library well-suited for creating simulations for reinforcement learning, was selected due to its widespread use and support for a variety of scenarios needed for the project. The code utilizes standard functions provided by the Gymnasium library, including *`env.reset()`, `env.step(action)`, `env.render(mode='human', close=False)`, `env.close()`, `env.observation_space`,* and `env.action_space` to facilitate the manipulation of the simulation environment.

The defined observation space is a dictionary that includes packet reception rate (PRR), received signal strength indicator (RSSI), and spreading factor (SF) for each node in the network. These parameters were chosen to observe the environment because of their critical role in determining the optimal number of retransmissions for efficient network performance:

- PRR directly reflects the success rate of data transmission within the network.
- RSSI offers insight into the quality of the signal received by each node.
- SF provides information about the relative distance between the gateway and the node.

The code initially operated within an environment powered by SimPy, and the solution was enhanced by integrating the Gymnasium environment into the existing framework. The overall architecture of the project is illustrated in Figure 2.

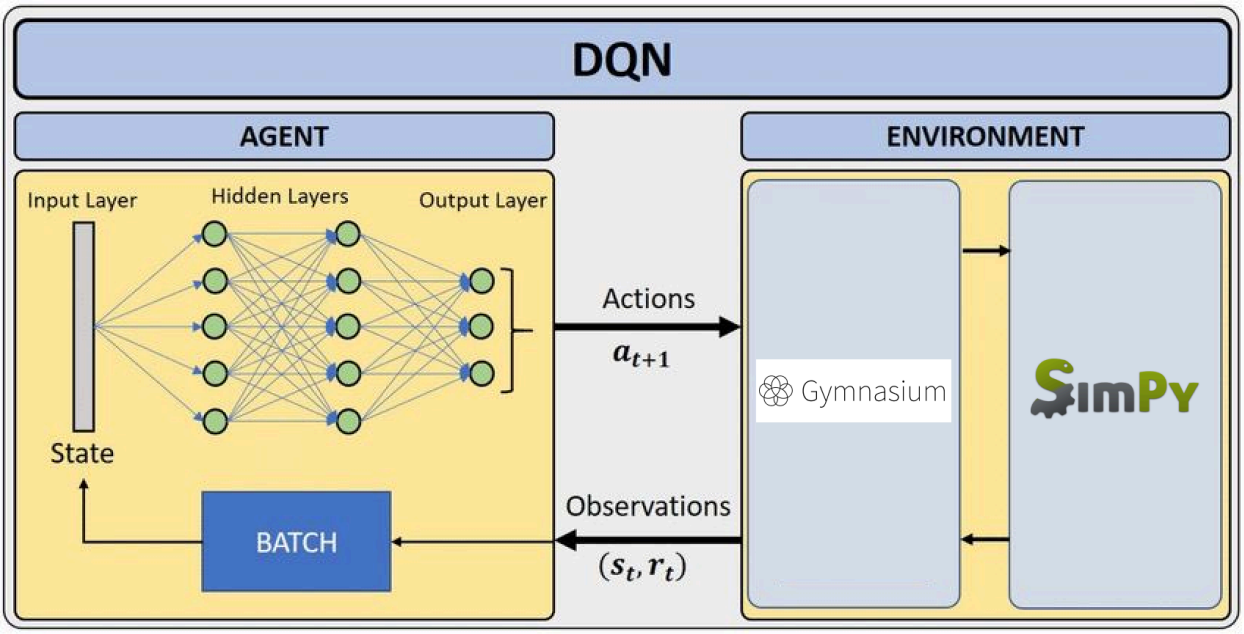

Figure 3. Architecture of the project's environment

The following UML diagram illustrates the structure of the project in more detail, including the classes with their fields, methods and how they are connected with each other:

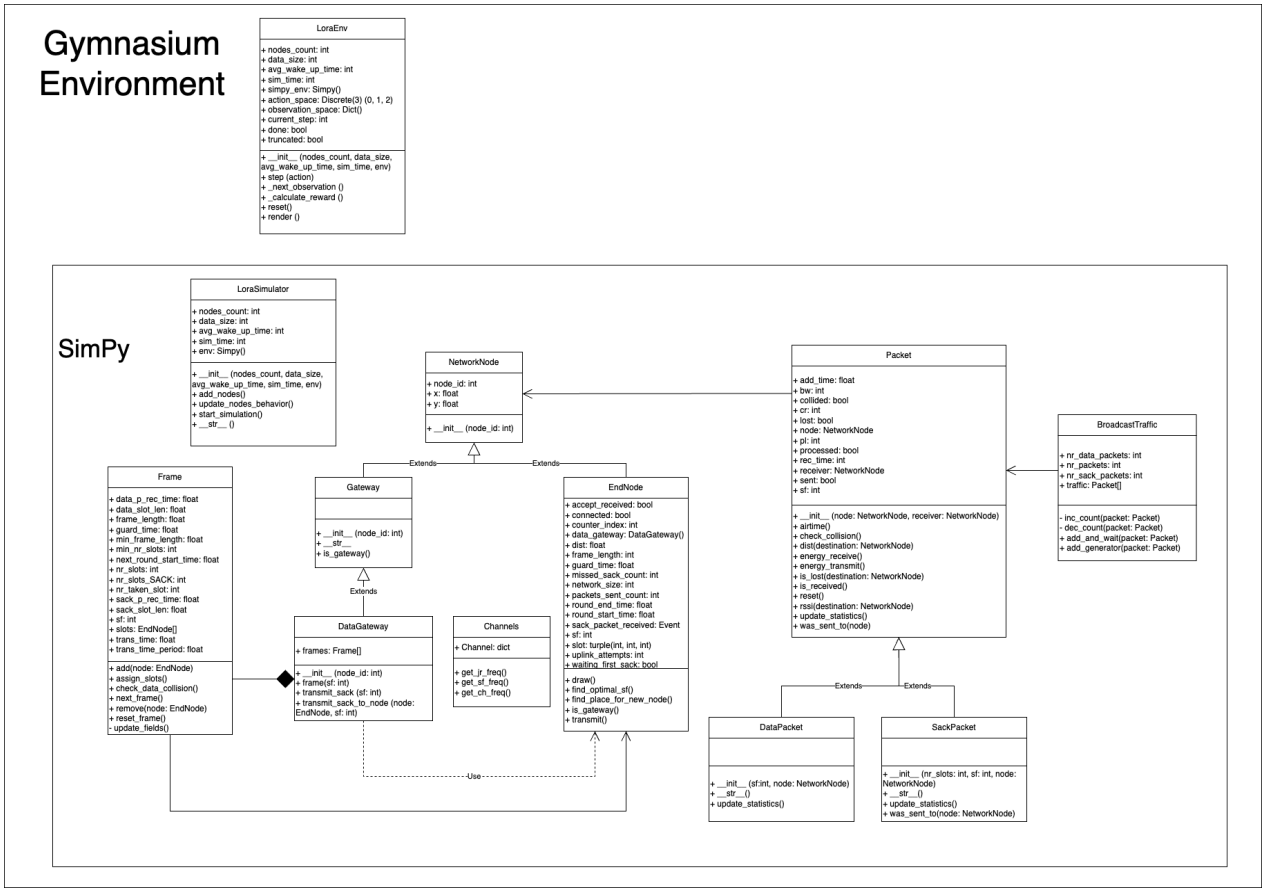

Figure 4. UML diagram of the simulator.

Project Execution

In terms of project execution, we divided the project work into 7 large tasks that we need to work on:

1. **Triple Uplink transmission:**

   Initially we built a triple uplink transmission on top of our previous simulator. In EndNode class for each node transmission slot we allocated 3 instead of 1. The triple uplink transmission allows to to make sure that the packet is received with the probability of 99.9%.

2. **Removing the joining phase:**

   The removal of the registration phase presented another critical issue that was addressed. In the previous version of the simulator, the registration phase consumed half of the simulation time, merely to enable nodes to join the network. As the number of nodes increased, a concurrent rise in the failure rates of nodes connecting to the network was

observed. Specifically, statistics from Figure 5 indicated that 16 out of 20 nodes failed to connect, and the collision count for join requests reached 1786—a substantially high number for a 20-node setup over a simulation duration of 3600 seconds. To rectify this scalability issue, the decision was made to eliminate the registration phase entirely.

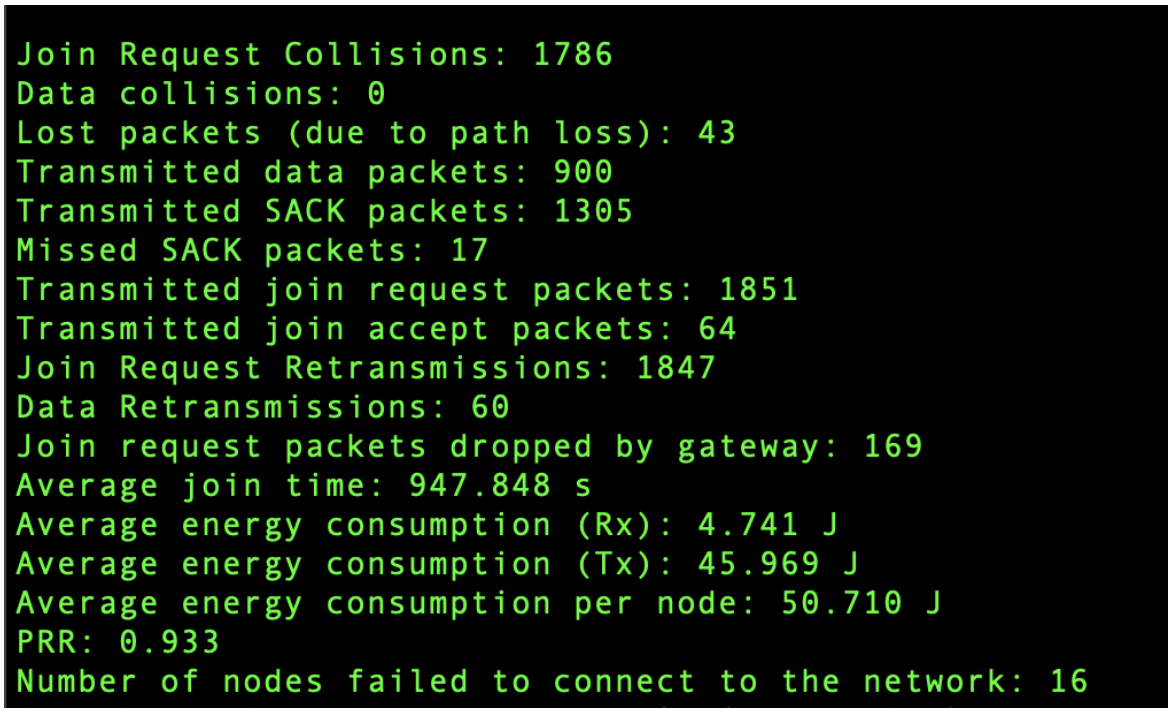

Figure 5. The joining phase statistics.

Initially, all classes containing JoinRequest and JoinAccept packets were identified, and the process to meticulously remove them was undertaken. Under the revised approach, each node was set to receive only a first SACK packet, which inadvertently led to error messages due to the round_end_time not being updated correctly, resulting in the environment time (env.now) exceeding the round_end_time. Negative delays, not permissible within the SimPy environment, necessitated a careful adjustment of timeout scheduling during the SACK packet transmission:

```
    def transmit(self, env):
        """
        Adjusted transmit function to accommodate the action chosen by the RL agent.
        """
        # print(env)
        while True:
            # calculating round start time
            yield env.timeout(random.uniform(0.0, float(2 * args.avg_wake_up_time)))
            if self.waiting_first_sack:
                yield self.sack_packet_received
                self.waiting_first_sack = False
                self.sack_packet_received = env.event()
            else:
                yield env.timeout(self.round_end_time - env.now)

            if self.round_start_time < env.now:
                log(env, f"[SACK-MISSED] {self}: missed sack packet")
                self.round_start_time = env.now + 1
                self.missed_sack_count += 1
                consts.nr_sack_missed_count += 1
            else:
                self.missed_sack_count = 0

            yield env.timeout(self.round_start_time - env.now)
```

Figure 6. Registration phase timeout.

Furthermore, an enhancement was implemented wherein all nodes were added to the frame and set to a status of connected at the initiation of the simulation. Previously, nodes only connected upon receiving a JoinAccept packet. This modification ensures that all nodes are connected before commencing their transmission activities, thus streamlining the process and enhancing network efficiency. These steps facilitated the simulation with a large number of nodes, significantly improving the scalability of our simulator.

3. Refactor the existing code

Before the integration of a new feature (Reinforcement Learning) into our simulator, the existing codebase required a comprehensive refactoring. The original code, comprising over 1000 lines solely within the main.py file, was not conducive to scalability or the addition of new features. As a result, a full refactoring was undertaken. Initially, the plan involved separating each class into its own file, but this approach encountered a problem known as "circular import," where two classes mutually reference each other.

To circumvent this issue, it was determined that classes with similar functionalities should be grouped into a single file. For example, classes such as 'NetworkNode', 'Gateway', 'DataGateway', and 'EndNode', which are related to nodes, were consolidated into a file named entities.py. Similarly, all other files were organized using this approach. Furthermore,

all constants were centralized into consts.py and utility functions, including those for collision detection, were moved to utils.py.

However, this restructuring led to challenges with global variables that needed to be accessible at the start of a simulation. To address this, a singleton file was created to store all instances of global variables. This restructuring resulted in a final version of the simulator that significantly enhanced its organization and scalability.

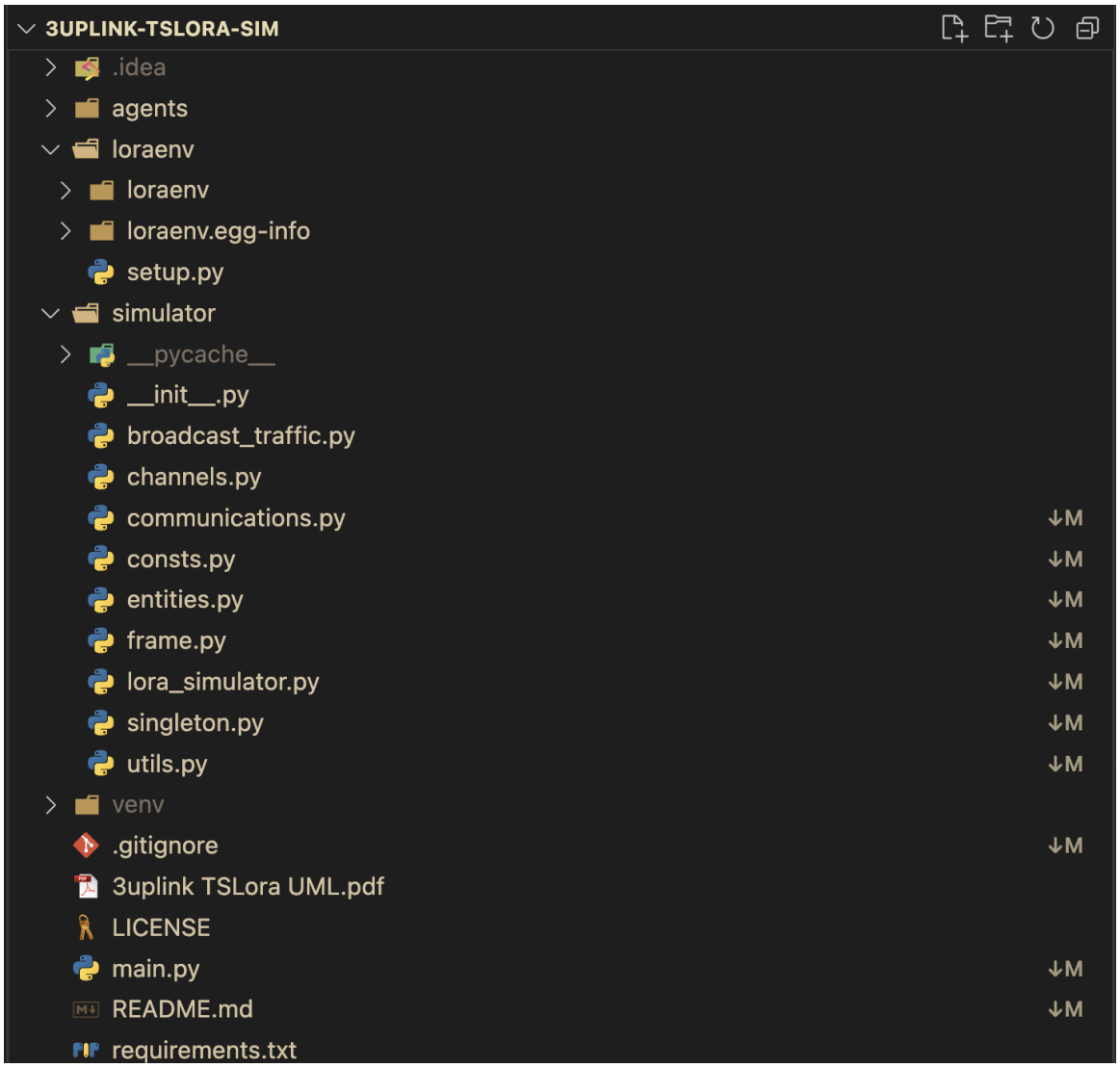

Figure 7. Final version of refactored code.

The final folder structure of the refactored code is illustrated in Figure 7. The logic related to simulation is organized within the 'simulator' folder, while the environment logic is housed in the 'loraenv' folder. This segmentation aids in clarifying and enhancing the scalability of the code, making it more manageable and adaptable for future expansions or modifications.

4. **Create Gymnasium environment**

   Following the comprehensive refactoring of the code, a Gymnasium environment was readily established to facilitate the training of our Reinforcement Learning model. The OpenAI Gymnasium library was selected due to its ease of use and comprehensive documentation. However, no existing environment within the documentation adequately suited our simulation needs, necessitating the development of a custom Gymnasium environment [3].

   The initial step involved determining the observation and action spaces, essential components for configuring the environment to accurately reflect the dynamics of our simulated network. The action space was defined as Discrete, with three possible actions (0, 1, and 2), each representing the number of transmissions (action + 1) that each node could make. For the observation space, a Dictionary was chosen, consisting of variables such as Spreading Factor (SF), Packet Reception Rate (PRR), and Received Signal Strength Indicator (RSSI), which are crucial for assessing the network's performance.

   Then we need to implement the basic reinforcement learning functions such as step(), reward(), observation() and reset(). The step() method advances the simulation by one step, corresponding to one second in the SimPy simulation environment. During this method, the specified action, which determines the number of transmission slots (ranging from 1 to 3), is applied to update the behavior of nodes. The simulation is then advanced by 1,000 milliseconds (or one second), during which packet transmissions and other network activities occur. After advancing the simulation, the method calculates the reward and checks if the simulation has reached its completion. The reward is calculated as the mean PRR minus a penalty term (defined by the lambda_value) multiplied by the number of uplinks. This formulation aims to maximize the PRR while minimizing the number of uplinks, thereby optimizing network efficiency and resource utilization.

   The observation() returns the current state of the simulation based on key network metrics. The reset() function reinitializes the environment to its base state, effectively resetting all modifications made during previous runs. This includes reestablishing the SimPy environment, resetting all network statistics, and re-adding nodes to the simulation. This method ensures that each run of the simulation starts from a consistent state, essential for accurate and repeatable experiments in reinforcement learning studies.

5. **Integration of Gymnasium environment with existing simulator (SimPy environment)**

   The next step after implementing the Gymnasium environment for our simulator to

introduce Reinforcement Learning later, was to integrate it with the existing environment in our project, SimPy environment. At this moment, our project was already utilizing an environment to schedule events, such as packet transmission, sack packet transmission, etc., hence it was difficult to combine two environments together. During this period, it was decided to wrap our SimPy environment with Gymnasium, so the workflow becomes the following: main script is launched, our custom Gymnasium environment for LoRa Network is launched, and after that, during each step taken in Gymnasium, our SimPy simulation is being ran (for the corresponding time period of one step in Gymnasium environment, in our case, 1 step in Gymnasium = 1 second in SimPy). This approach has eliminated all possible issues with two environments and allowed us to use our existing SimPy environment in the Gymnasium for future training of the Reinforcement Learning model.

6. **Integration of Reinforcement Learning (Deep Q-Network)**

   With the Gymnasium environment set and ready, the next task was to integrate Reinforcement Learning into the simulator. The principle behind using RL in our project, which is to dynamically adjust the number of uplinks (from 1 to 3) for the next transmission based on the current state of the nodes, was explained previously. For this purpose, the Deep Q-Network from StableBaselines3 was selected. The integration process was straightforward, as the environment with its action and observation spaces was already implemented. However, some parameters of the DQN agent required adjustments, such as the input policy, which was changed from single input to multi-input to accommodate our observation space – a dictionary containing three key metrics for all nodes: Packet Reception Ratio (PRR), Received Signal Strength Indicator (RSSI), and Spreading Factor (SF). Additionally, the default callback method from StableBaselines3 was imported and modified to allow the output of statistics and the drawing of graphs during usage.

7. **Training the DQN model**

   After successfully setting up the DQN model, the next objective was to train it to optimally determine the number of uplinks for each transmission, based on the current state of the nodes, including their Packet Reception Rate (PRR) and their previous uplink counts. The initial challenge involved designing an effective reward formula that considered various important variables. The initial reward formula simply calculated the difference between the mean PRR and the total uplink count, multiplying this difference by a small lambda value (0.0001), intended to penalize excessive uplink numbers while rewarding higher PRR values. However, this formula proved suboptimal for several reasons: it focused on total

statistics, which could lead the model to maintain rather than improve its performance due to already satisfactory values, and the total uplink count, which invariably increases, thus leading to a continually decreasing reward curve throughout the episode.

To address these shortcomings, we revised the reward formula and made several changes. This revised formula now accounts for changes in mean PRR and the increment in uplink counts, encouraging improvements in PRR while applying penalties more rationally – only for increases in uplink numbers, not their cumulative total. Additionally, the lambda value, which weights the uplink penalty, is dynamically adjusted based on a predefined target PRR: the penalty is more lenient when the PRR is below target and stricter when the PRR meets or exceeds the target. With this updated reward formula, the model was retrained and demonstrated enhanced performance.

Evaluation

The primary objective of the simulator was to allow large-scale testing of the Time-Slotted LoRa Protocol in a virtual environment before its real-world implementation, due to the complexity and scope of the setting. A Python-based simulator was therefore developed, featuring a wide range of customizable options such as the number of nodes, simulation duration, data packet size, and average node wake-up time. This variety in setup options ensures that the simulator fully meets the main goal of the project, allowing users to conduct tests under various conditions.

The next addressed problem involves the inefficient usage of resources and energy in a triple uplink technique, however it should be dynamically adjusted based on the current status of the network and the nodes. To evaluate the performance of our simulator and determine the efficiency of our solution, our updated simulator was evaluated independently and then compared side-by-side with the old simulator, which integrates Reinforcement Learning (RL). To assess the RL model's performance, its learning speed, convergence point of its learning curve, and its speed in finding the optimal policy for the environment, two graphs were plotted: one for the training phase of our DQN model (Total Reward vs Episodes) and one for the evaluation phase (Reward vs Step):

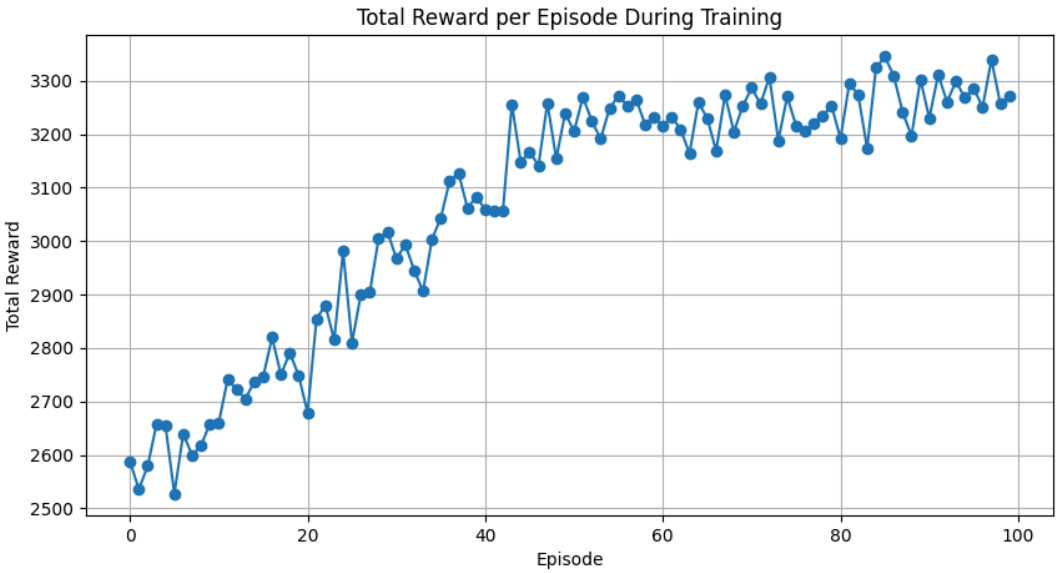

Figure 8. Training phase graph

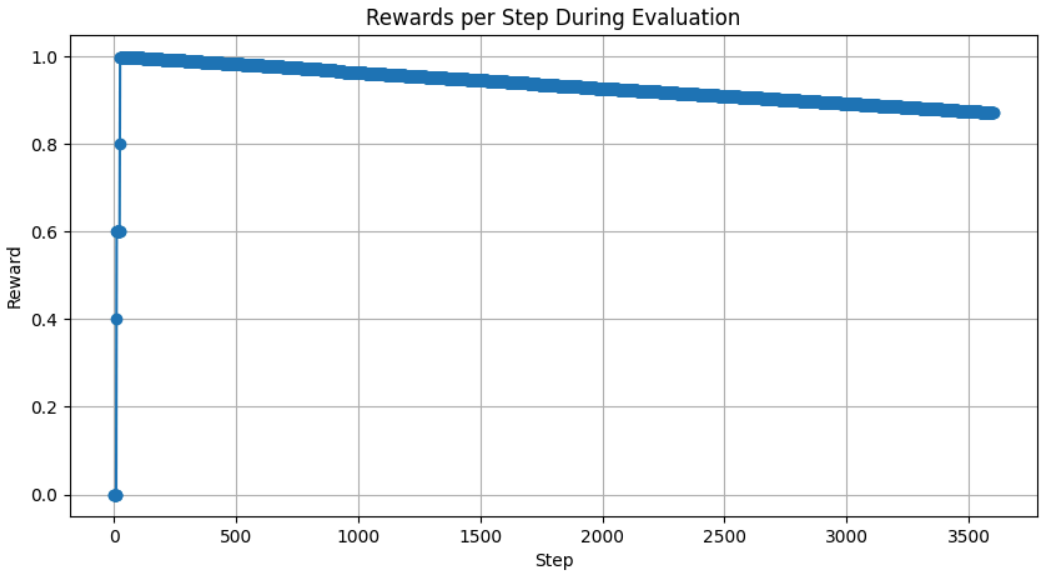

Figure 9. Evaluation phase graph

As can be seen from the graphs above, during the training phase, which lasted for 100 episodes, model started converging at the half of the training duration, with about ~3200 total reward, indicating that our reward formula is efficient and the DQN model can quickly learn the optimal policy. The second graph demonstrates how the model behaves in a real environment, operating without further learning and utilizing prior knowledge. Here, the graph illustrates that

the model quickly approaches and converges at the target reward value, signifying good performance.

Next, a comparison between the two simulator versions was conducted: the triple uplink and the dynamic (RL-driven) uplink. According to the simulator logic, similar results to those of the default simulator, with a high Packet Received Ratio (PRR), but fewer overall uplinks by nodes, were expected with RL integration. Consequently, two graphs were plotted for both versions of the simulator: Mean PRR vs Time and Total Uplink Count vs Time:

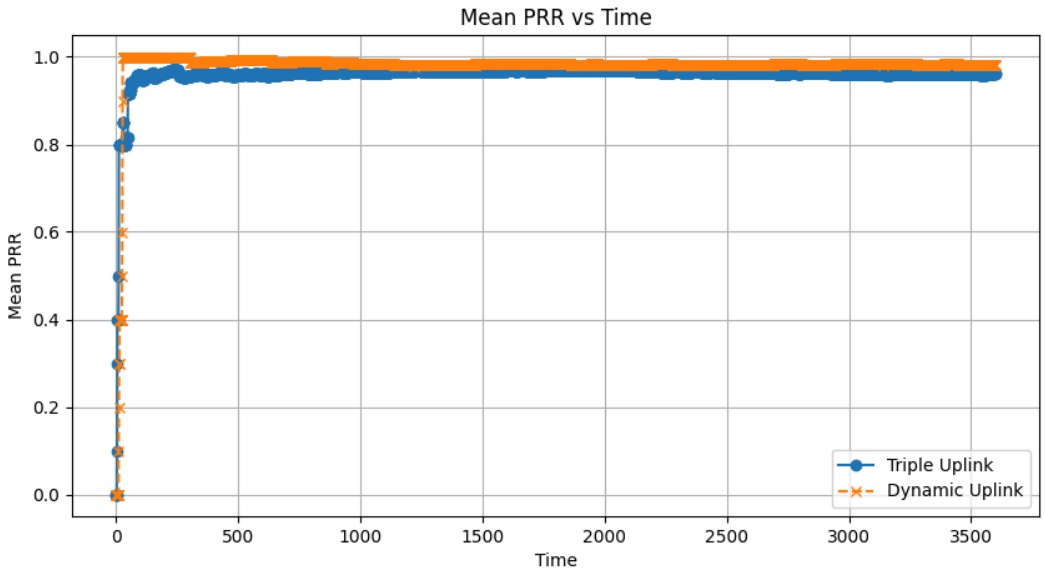

Figure 10. Mean PRR vs Time comparison

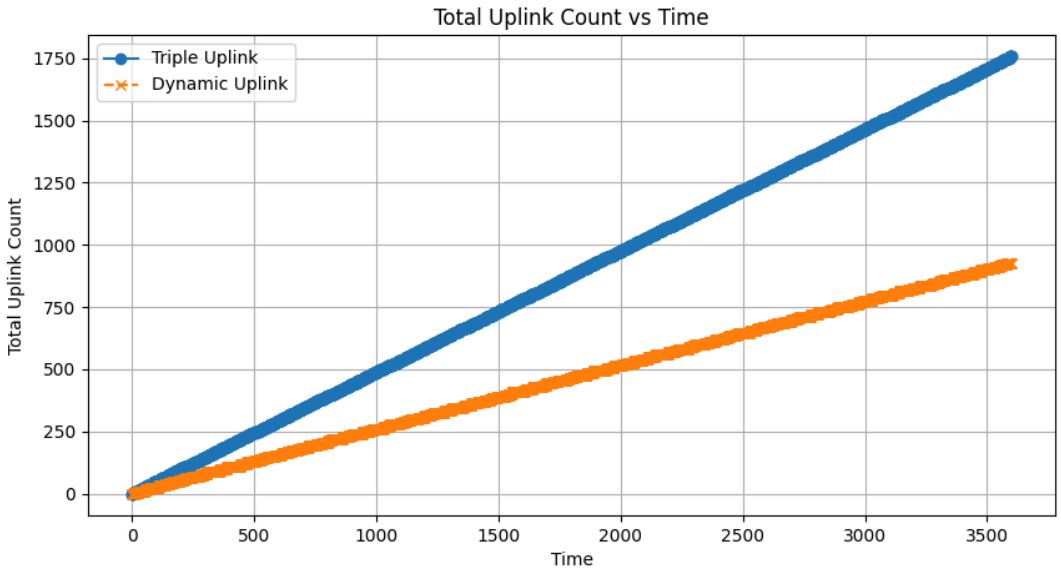

Figure 11. Total Uplink Count vs Time comparison

From the first figure, it is observed that the PRR vs Time graphs are relatively similar, since both versions simulate a real-world environment and aim for good conditions, hence the PRR is between 0.9 and 1.0 in both graphs. However, from the second figure, it is evident that the Total Uplink Count for our updated simulator grows slower than that of the default, triple uplink version. This indicates that our model is utilizing resources and energy more appropriately, opting for more uplink attempts only when necessary to maintain the target PRR, approximately 0.9.

Conclusion and Possible Future Work

In conclusion, the developed simulator successfully meets its initial objectives by providing a robust software solution that allows large-scale testing of the Time-Slotted LoRa Network Protocol. By employing a reinforcement learning-driven algorithm, the simulator enhances its resource and energy utilization efficiency through dynamic adjustments of uplink numbers based on real-time network conditions. This approach ensures optimized performance without the excessive use of network resources, aligning with the project's goals of improving efficiency and scalability.

For future work, there is a major opportunity to further enhance the simulator's functionality and accuracy. Currently, the reinforcement learning implementation governs the number of uplinks in the environment with a single policy that applies uniformly to all nodes in the

network. In other words, the RL model takes into account states of all the nodes as a whole and outputs a single action (number of uplinks for the next transmission) for all nodes in the simulation. This approach, while effective in general scenarios, might not optimally reflect the diverse conditions that individual nodes may encounter in more complex network configurations (with more nodes in the network, longer simulation time, etc.). To address this limitation, future updates of the project could focus on the development of a multi-agent reinforcement learning model. Such a model would allow each node to independently evaluate and decide its actions, such as the number of uplinks, based on its own state and local environmental variables. This improvement would not only enhance the granularity of control but also potentially increase the overall network efficiency by adapting more precisely to different conditions across the nodes. This approach aligns with the evolving trends in network management where decentralized decision-making processes can significantly improve both performance and adaptability in dynamic environments.

References

[1] M. Bor, “mcbor/lorasim,” GitHub, Oct. 13, 2022. https://github.com/mcbor/lorasim.

[2] MGL-coder, “MGL-coder/TS-LoRa-sim,” GitHub, Dec. 05, 2023. https://github.com/MGL-coder/TS-LoRa-sim.

[3] “Gymnasium Documentation,” *gymnasium.farama.org*. https://gymnasium.farama.org/tutorials/gymnasium_basics/environment_creation/

[4] D. Zorbas, K. Abdelfadeel, P. Kotzanikolaou, and D. Pesch, “TS-LoRa: Time-slotted LoRaWAN for the Industrial Internet of Things,” *Computer Communications*, vol. 153, pp. 1–10, Mar. 2020, doi: https://doi.org/10.1016/j.comcom.2020.01.056.

[5] D. Zorbas, “Design Considerations for Time-Slotted LoRa(WAN),” presented at the MaDeLoRa 2020: 1st Workshop on Massive LoRa Deployments: Challenges and Solutions, Feb. 2020. Available: https://hdl.handle.net/10468/9723.

[6] Z. Bagdauletkyzy, M. Galymgereyev, A. Abzalova, A. Kairatbek and D. Zorbas, "Poster: A Simulator for Time-Slotted LoRa Networks," 2023 IEEE Symposium on Computers and Communications (ISCC), Gammarth, Tunisia, 2023, pp. 1-3, doi: 10.1109/ISCC58397.2023.10218072.