SANTA CLARA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING

Date: October 31, 2026

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Sahana Prasanna Chandramohan [CSEN & ECEN] **Vikram Rakesh Jangid [ECEN]**

ENTITLED

High Altitude LoRa Networking (HALoN) for Crisis Communications

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
BACHELOR OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

Thesis Advisor [Kurt Schab (ECEN)]

Thesis Advisor [Krishna Kattiyan Ramamoorthy (CSEN)]

Thesis Advisor [Michael Schimpf (CSEN)]

Department Chair [Silvia Figueira (CSEN)]

Department Chair [Shoba Krishnan (ECEN)]

High Altitude LoRa Networking (HALoN) for Crisis Communications

by

Sahana Prasanna Chandramohan [CSEN & ECEN] Vikram Rakesh Jangid [ECEN]

Submitted in partial fulfillment of the requirements
for the degrees of
Bachelor of Science in Computer Science and Engineering
Bachelor of Science in Electrical and Computer Engineering
School of Engineering
Santa Clara University

Santa Clara, California
October 31, 2026

High Altitude LoRa Networking (HALoN) for Crisis Communications

Sahana Prasanna Chandramohan [CSEN & ECEN] Vikram Rakesh Jangid [ECEN]

Department of Computer Science and Engineering
Department of Electrical Engineering
Santa Clara University
October 31, 2026

ABSTRACT

Natural disasters, infrastructure failure, and politically motivated internet shutdowns can severely disrupt communication systems, preventing affected populations from coordinating emergency response, accessing information, and contacting others. Existing communication infrastructure relies heavily on centralized networks, fixed power availability, and costly deployments, making such systems vulnerable during periods of crisis. To alleviate these problems we propose High Altitude Long Range Networking (HALoN) for Crisis Communications, a rapidly deployable decentralized wireless mesh network designed for operation in infrastructure-denied environments. HALoN utilizes low-power long range (LoRa) communication and high-altitude deployment platforms such as balloons, drones and tall structures, among others, to create temporary communication coverage over affected regions.

The system introduces three primary contributions: a received signal strength indicator (RSSI) informed routing algorithm, a priority-aware Q-learning scheduling algorithm, and a feed network for a reconfigurable antenna system. The routing algorithm dynamically selects paths based on observed link quality to improve packet delivery reliability across unstable wireless links. The scheduling algorithm adaptively prioritizes packets according to urgency, retransmission history, congestion conditions and other factors in order to improve bandwidth utilization under constrained network conditions. The reconfigurable antenna feed network enables adaptable radiation characteristics to improve coverage flexibility and communication performance across varying deployment scenarios. Both the overall architecture and its implementation are relatively lightweight, but they have the potential to significantly improve communication survivability in disaster response, humanitarian aid, rural connectivity, and censorship-resistant networking applications. In this thesis, we analyze the implementation considerations associated with decentralized deployment, adaptive routing, intelligent scheduling, and reconfigurable wireless hardware, while also designing and developing these systems within a functional prototype network.

Table of Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 3
- 1.2 Objectives . . . 3
- 1.3 Approach . . . 4
  - 1.3.1 Network stack implementation . . . 4
  - 1.3.2 Routing implementation . . . 5
  - 1.3.3 Packet scheduling implementation . . . 5
  - 1.3.4 Link quality estimation . . . 6
  - 1.3.5 RSSI-Informed LSA Routing . . . 7
  - 1.3.6 Firmware implementation and testbed . . . 7
  - 1.3.7 Reconfigurable feed network implementation . . . 7
  - 1.3.8 Simulation environment . . . 8
  - 1.3.9 Evaluation metrics . . . 10

2 User Research **11**
- 2.1 Methods . . . 11
- 2.2 Stakeholder needs . . . 12
  - 2.2.1 User Stories . . . 13

3 Design and Rationale **14**
- 3.1 Design . . . 14
- 3.2 Use Case Overview . . . 15
- 3.3 Use Case Descriptions . . . 15
- 3.4 Use Case Diagram . . . 17
- 3.5 Class Diagram . . . 17
- 3.6 Sequence Diagram . . . 18
- 3.7 Activity Diagram . . . 19
- 3.8 Functional requirements . . . 19
- 3.9 Non-functional requirements . . . 23

4 RSSI-Based Link-State Routing **24**
- 4.1 Overview and Design Rationale . . . 24
- 4.2 Neighbour Discovery via HELLO Packets . . . 24
  - 4.2.1 RSSI-to-Cost Mapping . . . 25
- 4.3 Link State Advertisements . . . 26
  - 4.3.1 Payload Structure and Construction . . . 26
  - 4.3.2 Sequence Number Deduplication and Wraparound . . . 26
  - 4.3.3 Selective Row Clearing and the Topology Invariant . . . 27
- 4.4 Dijkstra's Algorithm in a Constrained Environment . . . 29
  - 4.4.1 Data Structures and Memory Budget . . . 29
  - 4.4.2 Algorithm and Next-Hop Table Construction . . . 29
- 4.5 LSA Flooding, Relay, and Topology Aging . . . 31
  - 4.5.1 Controlled Flooding via TTL . . . 31

4.5.2 Round-Robin LSA Relay . . . 32
4.5.3 Topology Aging and Node Expiry . . . 32

5 The Q-Learning Packet Scheduler 34
5.1 Motivation and Design Philosophy . . . 34
5.2 The Feature Space . . . 35
5.3 Q-Value as a Linear Feature Sum . . . 37
5.4 Epsilon-Greedy Action Selection . . . 39
5.5 Reward Function and Temporal-Difference Update . . . 40
5.5.1 Reward Function . . . 40
5.5.2 Temporal-Difference Error . . . 40
5.5.3 Interpreting the Update Semantically . . . 41
5.6 The Pending Update and Retransmission Mechanics . . . 42

6 Developing a Network Simulator 45
6.1 Goals and Scope . . . 45
6.2 Topology and Routing Model . . . 46
6.2.1 Geometric Connectivity . . . 46
6.3 The Python Q-Learner . . . 46
6.4 Packet Motion and Node Queue Dynamics . . . 49
6.5 Headless Simulation and Comparative Evaluation . . . 50
6.6 Validation Observations . . . 51

7 Developing a Reconfigurable Feed Network 52
7.1 Microstrip Sizing . . . 52
7.2 Endfire Pattern . . . 53
7.2.1 Power Divider . . . 53
7.2.2 Phase Shifter . . . 54
7.3 Broadside . . . 55
7.4 PCB Layout . . . 56

8 Firmware Integration 58
8.1 System Architecture . . . 58
8.2 Packet Wire Format and Serialisation Safety . . . 58
8.3 Main Loop Structure and Task Ordering . . . 59
8.4 The ACK-Driven Learning Loop . . . 61
8.5 ACK Routing and the `prev_hop` Fallback . . . 62
8.6 Duplicate Suppression with the Seen Cache . . . 62
8.7 Deterministic PRNG Seeding . . . 64
8.8 Serial Command Interface for Diagnostics . . . 64

9 System Evaluation 66
9.1 Internal Testing . . . 66
9.2 External Testing . . . 69
9.3 Unit Testing . . . 70
9.4 Integration Testing . . . 72

10 Implementation Plan 77
10.1 Timeline . . . 77
10.2 Project Risks . . . 77

11 Constraints and Standards 80
11.1 Constraints . . . 80
11.2 Standards . . . 82

12 Hazards **86**
12.1 Potential Hazards . . . 86
12.1.1 Batteries . . . 86
12.1.2 Radio Frequencies and Microwaves . . . 86
12.1.3 Power Tools Operation and Mechanical Foundation . . . 86
12.1.4 Soldering . . . 87

13 Societal Issues **88**
13.1 Ethical . . . 88
13.2 Social . . . 89
13.3 Political . . . 91
13.4 Economic . . . 93
13.5 Health and Safety . . . 93
13.6 Manufacturability . . . 97
13.7 Sustainability . . . 97
13.8 Environmental Impact . . . 98
13.9 Usability . . . 98
13.10 Lifelong learning . . . 99
13.11 Compassion . . . 99

14 Conclusion **101**

15 Acknowledgments **103**

16 Appendices **104**
16.1 Network Simulation Code . . . 104

List of Figures

1.1 Paradise Campfire Infrastructure Loss [1] . . . 1
1.2 Deployment of HALoN at High Altitude using different Platforms . . . 2
1.3 HALoN Node System Diagram . . . 4
1.4 Visualization of Data Flow and Processing done in HALoN Node . . . 5
1.5 Adaptive Weight Updating Cycle . . . 6
1.6 Visualization of Built-in Antenna Radiation that does not Reach Ground Users . . . 8
1.7 Network Simulator Visualization . . . 10

3.1 Use Case Diagram: Message Transmission . . . 17
3.2 UML Class Diagram of the HALoN System . . . 18
3.3 Packet Transmission Sequence Diagram . . . 19
3.4 Activity Diagram of Packet Handling in the HALoN Simulator . . . 20
3.5 Broadside Radiation Communication Pattern . . . 21
3.6 Broadside Radiation Communication Pattern Design . . . 21
3.7 Endfire Radiation Communication Pattern . . . 22
3.8 Endfire Radiation Communication Pattern Design . . . 22

7.1 Wilkinson power divider CAD in HFSS . . . 54
7.2 T-junction power divider CAD in HFSS . . . 54
7.3 Meandered microstrip line CAD in HFSS . . . 55
7.4 Endfire topology CAD in HFSS . . . 55
7.5 Broadside topology CAD in HFSS . . . 55
7.6 Electric schematic in KiCad . . . 56
7.7 PCB layout in KiCad . . . 57

9.1 Schematic of Feed Network PCB . . . 68

10.1 Gantt Chart of HALoN Project Timeline . . . 77

Chapter 1

Introduction

In the modern world, reliable communication infrastructure is foundational to emergency response, public safety, and community coordination. During crises, however, these systems are often among the first to fail. Natural disasters such as wildfires, hurricanes, earthquakes, and floods can destroy cell towers, sever fiber infrastructure, and overload surviving networks, leaving affected populations isolated at the exact moment communication becomes most critical. During the 2018 in Paradise, California, communication infrastructure failures severely impacted evacuation coordination and emergency response efforts as residents struggled to receive timely alerts and contact emergency services [2]. Figure 1.1 shows the destruction caused by the fires.



Figure 1.1: Paradise Campfire Infrastructure Loss [1]

Similarly, communication networks are vulnerable not only to environmental collapse, but also to political interference. During the September 2025 Taliban-imposed internet blackout in Afghanistan, millions experienced widespread communication disruption due to centralized control over telecommunications infrastructure, illustrating how governments or controlling entities can intentionally restrict access to information and connectivity during periods of instability [3].

Existing alternatives to traditional infrastructure each present significant limitations. Satellite communication systems such as Starlink can provide connectivity in remote or damaged regions, but they require expensive specialized

hardware, recurring subscription costs, and a reliable supply of electrical power, resources that are often unavailable during disasters [4]. Additionally, these systems remain dependent on centralized private providers that ultimately control regional service availability and access policies. Amateur (HAM) radio networks provide another decentralized communication option, but effective operation requires extensive training, licensing, coordination, and specialized equipment [5]. Participation in HAM radio is also increasingly niche, limiting accessibility for the general public during emergencies.

Keeping in mind these limitations, we propose a deployable, decentralized peer-to-peer communication system built using LoRa wireless technology. LoRa is a low-power wireless communication protocol based on a chirp spread spectrum modulation scheme, designed to transmit small packets of data over long distances while minimizing energy consumption [6]. Unlike high-throughput technologies such as Wi-Fi or cellular networks, LoRa prioritizes communication range and power efficiency over bandwidth, making it well-suited for infrastructure-constrained or disaster-stricken environments where only low-data-rate messaging is required [7]. "Long range" in this context refers to kilometer-scale communication distances that can extend to tens of kilometers under line-of-sight conditions, particularly when nodes are elevated above ground obstructions. HALoN leverages LoRa-enabled embedded systems attached to high-altitude deployment platforms such as balloons, drones, or temporary aerial relays to rapidly establish wide-area communication coverage over regions where conventional infrastructure is unavailable or denied, as seen in Figure 1.2.

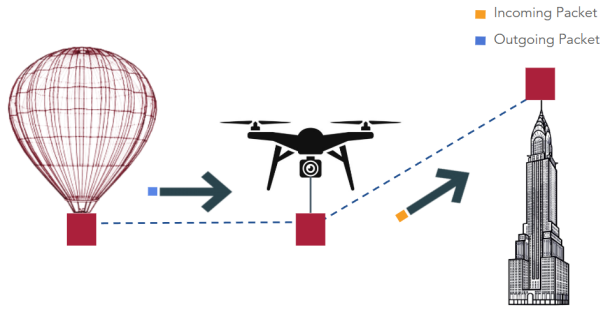


Figure 1.2: Deployment of HALoN at High Altitude using different Platforms

Each HALoN node operates as both a communication endpoint and a routing participant, forming a resilient mesh network capable of forwarding messages between users and ground stations without reliance on centralized infrastructure. By distributing network intelligence equally across the system, within each node, HALoN avoids single points of failure and remains operational even as nodes join, move, or disconnect from the network. The system further incorporates adaptive routing and reinforcement-learning-based packet scheduling to dynamically prioritize urgent traffic, mitigate congestion, and improve delivery reliability under unstable network conditions.

Beyond emergency response, HALoN has broader applications in rural and remote regions where communication

infrastructure is sparse, unreliable, or economically infeasible to deploy. By combining low-cost embedded hardware, long-range wireless communication, and rapidly deployable aerial platforms, HALoN aims to provide a scalable, energy-efficient, and citizen-accessible alternative for maintaining critical communications when conventional systems fail.

1.1 Problem Statement

To solve for the limitations addresed prior, we propose HALoN as a deployable decentralized communication system for infrastructure-free operation. The system is designed as a distributed multi-hop network in which each node functions simultaneously as an endpoint and routing participant, eliminating dependence on centralized infrastructure.

HALoN leverages elevated deployment to extend network reach and enable rapid area-wide connectivity in environments where ground infrastructure is unavailable or damaged. Communication is handled through adaptive routing at the node level, enabling dynamic path selection under changing network conditions.

In order to develop HALoN in this direction, we make the following contributions in this thesis:

- The design and implementation of HALoN, a deployable decentralized LoRa mesh network for disaster communication.
- An RSSI-informed link-state routing mechanism for adaptive path selection in unstable wireless environments.
- A decentralized Q-learning-based packet scheduler for fairness-aware airtime allocation in duty-cycle-constrained LoRa networks.
- A reconfigurable antenna feed network enabling adaptive radiation patterns for air-to-air and air-to-ground communication.

1.2 Objectives

HALoN is designed for operation under degraded and infrastructure-limited communication conditions. The system assumes intermittent connectivity, high packet loss, and constrained spectrum availability as baseline operating conditions. The primary objective is to support reliable transmission of short-form messages, including emergency alerts and positional data, over a multi-hop LoRa mesh network. High-bandwidth communication is intentionally excluded due to physical layer constraints, duty cycle limitations, and energy efficiency requirements inherent to the LoRa protocol. Reliability is defined in terms of bounded delivery behavior, including explicit success or failure feedback. The system is designed to avoid silent message loss by ensuring that transmission outcomes are observable to end users within a bounded time window.

Our second objective is fairness in resource allocation under constrained airtime. Due to duty cycle limitations, channel access is a scarce resource. The system must therefore allocate transmission opportunities in a manner that accounts for message urgency while avoiding systematic starvation of lower-priority traffic.

Our third objective is deployability on low-cost, widely available hardware without reliance on proprietary infrastructure or licensed spectrum. The system is designed for rapid deployment by non-specialist operators, including volunteer and community-based response groups.

Our final objective is to evaluate whether decentralized reinforcement learning can improve packet scheduling performance under dynamic network conditions. Each node independently adapts its scheduling policy based on local observations without centralized coordination, enabling distributed optimization of transmission decisions under changing link conditions.

1.3 Approach

HALoN is implemented as a fully embedded distributed system deployed on Adafruit Feather 32u4 microcontroller nodes with integrated RFM95 LoRa transceivers. All networking functionality is executed on-device in firmware, including routing, scheduling, packet handling, and neighbor management. No external controller is used during runtime.

1.3.1 Network stack implementation

At the physical and link layers, communication is performed using LoRa in the 915 MHz ISM band. Packets are constructed in firmware as fixed-format frames containing source ID, destination ID, sequence number, payload length, and message payload. Transmission is performed in half-duplex mode using the Semtech SX1276 radio interface. The high-level architecture of each node is detailed below in Figure 1.3:

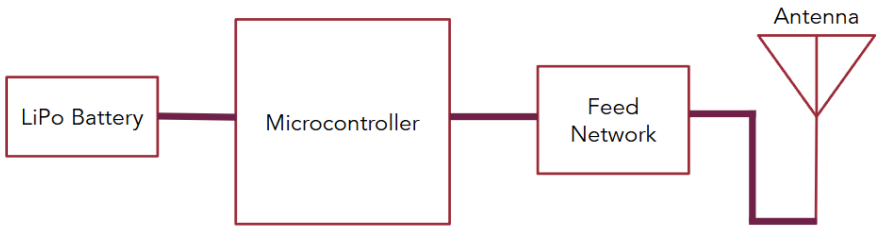


Figure 1.3: HALoN Node System Diagram

A lightweight medium access strategy is implemented using a time-division assumption combined with randomized backoff. This means that nodes do not rely on global clock synchronization; instead, transmission timing is

derived locally from the microcontroller system clock (millis-based scheduling). Each node independently determines when to dequeue and transmit packets based on its local notion of time and current queue state.

This creates a loosely coordinated data flow model in which packets propagate hop-by-hop through per-node transmission queues. As a result, end-to-end data movement emerges from local scheduling decisions at each node, with no centralized coordination or global scheduling authority. The receiving, processing and transmitting of data are local behaviors, as demonstrated in Figure 1.4.

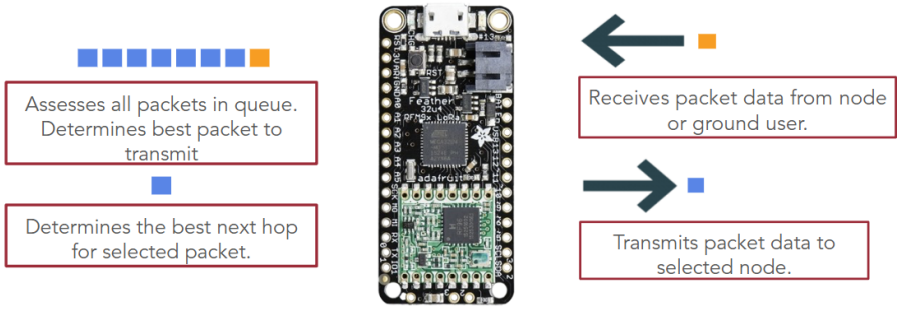


Figure 1.4: Visualization of Data Flow and Processing done in HALoN Node

1.3.2 Routing implementation

Routing is implemented as a link-state protocol. Each node periodically broadcasts neighbor discovery beacons to estimate link connectivity. Received signal strength (RSSI) is converted into a discrete link cost metric, which is stored in a local adjacency matrix.

Nodes exchange link-state advertisements (LSAs) containing their local neighbor tables. Each node maintains a full topology view constructed from received LSAs. When topology changes are detected (via timeout or update), the routing table is recomputed using Dijkstra's shortest-path algorithm over the adjacency matrix [8].

The routing output is stored as a next-hop table indexed by destination node ID. This table is used by the scheduler to forward packets hop-by-hop toward the destination.

1.3.3 Packet scheduling implementation

Each node maintains a bounded transmission queue stored as a static array in firmware due to memory constraints. Queue entries contain payload data, destination ID, priority level, retransmission count, and enqueue timestamp.

Three scheduling policies are implemented:

- First Come First Serve (FCFS) (baseline)
- Shorted Job First (SJF) (baseline approximation based on payload size and priority)
- Q-learning-based scheduler (primary contribution)

The Q-learning scheduler defines a discrete state space composed of quantized features: retransmission count, link quality bucket (derived from RSSI history), queue depth, packet priority, packet age, and payload size.

At each transmission opportunity, the scheduler evaluates all queued packets and selects one according to an ϵ-greedy policy [9]. The reward signal is computed after transmission based on acknowledgment success or failure, observed latency, and retransmission behavior.

Weights are updated using temporal-difference learning with fixed learning rate and discount factor. All updates are performed locally per node with no parameter sharing. The way in which weights are updated can be seen in Figure 1.5.

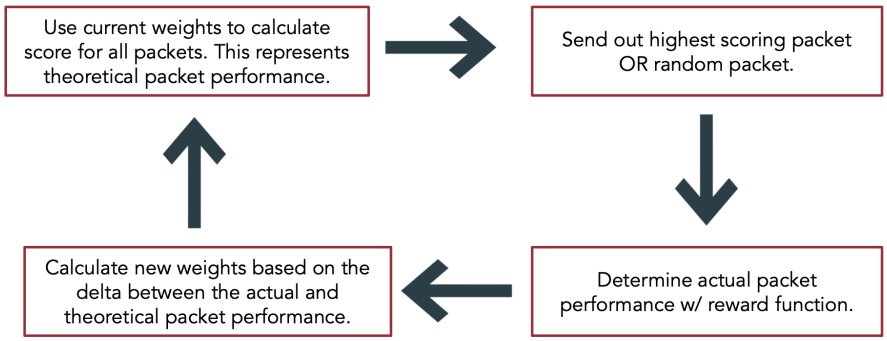


Figure 1.5: Adaptive Weight Updating Cycle

1.3.4 Link quality estimation

Link quality in HALoN is represented as a discrete link cost stored in a local adjacency matrix on each node. Each node maintains a bounded topology table in which link costs are explicitly assigned and updated through the routing layer when connectivity changes are detected.

A link between two nodes is represented symmetrically in a 2D cost matrix, where higher values correspond to weaker or less reliable links, and a special value indicates the absence of a usable connection. This abstraction allows the system to model link degradation and disconnection without requiring direct signal-strength sampling.

Because the implementation is constrained to low-power embedded hardware, link quality is treated as a structural property of the network graph rather than a continuously measured signal metric. This design reduces memory overhead and eliminates the need for analog signal processing while still allowing the routing algorithm to differentiate between viable and non-viable links.

The resulting topology representation serves as the input to the routing subsystem, where link costs directly influence shortest-path computation.

1.3.5 RSSI-Informed LSA Routing

HALoN uses a distributed shortest-path routing protocol based on Dijkstra's algorithm executed locally at each node over a shared topology representation. Each node maintains a local adjacency matrix encoding link costs to all other nodes within a fixed maximum network size. Links are symmetric, and absent connections are represented using an infinite cost value. Routing computation is triggered periodically or when topology changes occur. The algorithm computes shortest paths from the current node to all reachable destinations using a standard Dijkstra relaxation procedure with a visited-set optimization. Distances are stored as 16-bit integers to accommodate bounded path costs while minimizing memory usage on constrained microcontroller hardware.

Once shortest-path distances are computed, a next-hop table is constructed by backtracking predecessor pointers from each destination to determine the first hop along the optimal route. This next-hop table is then used during packet forwarding to determine the immediate neighbor to which a packet should be transmitted. The routing state is partially dynamic: links can be added or removed at runtime through explicit topology updates, which trigger local recomputation of the routing table. This allows the network to adapt to node mobility or intermittent connectivity without requiring global synchronization. Link cost updates are discrete and externally defined by the system's topology management logic rather than inferred continuously from physical signal measurements. This design choice prioritizes computational predictability and memory efficiency over fine-grained channel estimation, aligning with the constraints of low-cost embedded deployment.

1.3.6 Firmware implementation and testbed

All functionality is implemented on the Adafruit microcontroller boards. A multi-node testbench is used for evaluation, where nodes are deployed in controlled spatial configurations to emulate mesh behavior under varying distance, obstruction, and interference conditions.

Firmware is written in C++ using the Arduino framework. Constraints on SRAM (2.5 KB) and flash memory (32 KB) directly influence queue sizing, routing table limits, and feature discretization strategies.

1.3.7 Reconfigurable feed network implementation

The Adafruit Feather 32u4 includes a built-in RFM95 radio chip for wireless communication. While sufficient for most applications, HALoN must communicate with both ground users below and other nodes at altitude, two very different directions. A standard antenna radiates power in all directions equally, which is wasteful when the target is in a specific direction, leading to retransmissions and unnecessary power drain on an already power-limited system, this is visualized in Figure 1.6.

To address this, a switchable feed network is used to steer the antenna's radiation pattern based on who the node is communicating with. When talking to a ground user, the node uses an endfire pattern, which focuses energy downward.

When talking to other airborne nodes, it switches to a broadside pattern, which focuses energy horizontally. This allows each node to communicate efficiently in both directions without wasting power.

The reconfigurable feed network uses control signals from the microcontroller board to toggle between two states, state 1, sending data to users on the ground, or state 2, sending data to nodes in the sky. The reconfigurable feed network is simulated in Ansys HFSS, a full-wave electromagnetic solver used to evaluate high-frequency electronic components, part by part to gauge performance. In addition, dipole antennas for our design frequency of 900 MHz were designed and simulated, but due to time constraints they were not implemented.

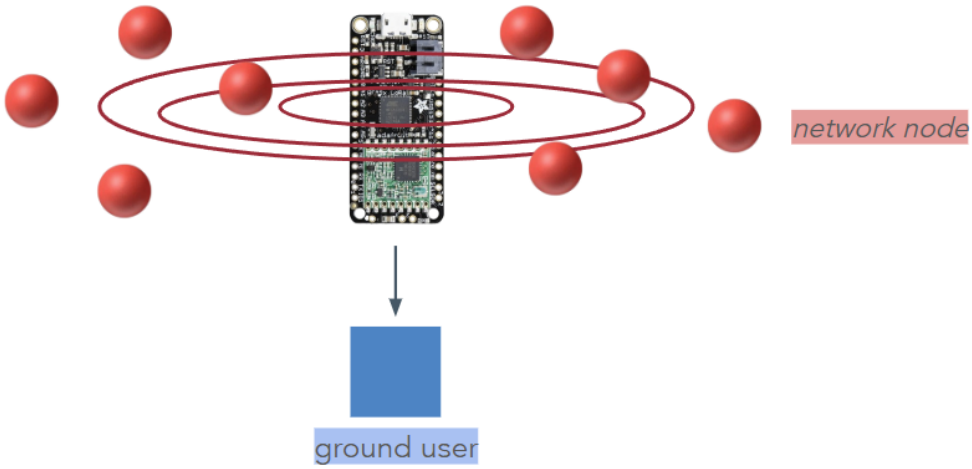


Figure 1.6: Visualization of Built-in Antenna Radiation that does not Reach Ground Users

The feed network consists of two signal paths, one for each state. State 1 uses a phase-shifted transmission line that introduces a 180° phase shift, while State 2 routes the signal directly to the output port. Once simulation results were satisfactory, the design was transferred to KiCad for PCB layout. The final board includes the feed network traces, an ADG919 RF switch from Analog Devices to toggle between states, pin headers for microcontroller interfacing, status LEDs for power and state indication, and SMA connectors for RF input and output. The completed PCB was then fabricated and validated.

1.3.8 Simulation environment

A Python-based discrete-event simulation environment is used to evaluate system behavior at scales beyond the constraints of the physical testbench implementation. The simulator models a time-stepped wireless mesh network in which nodes are embedded in a 2D plane, and connectivity is determined dynamically using a distance-based link model with a configurable maximum communication radius, it is seen in Figure 1.7. Links are reconstructed at runtime, and routing tables are computed using Dijkstra's shortest-path algorithm over the resulting connectivity graph.

Packet traffic is generated according to configurable injection processes, including randomized source-destination

pairs, heterogeneous packet sizes, priority classes, and stochastic arrival times over a bounded interval. Each packet carries metadata used by both routing and scheduling subsystems, including priority level, size, injection time, and retransmission history.

The communication model abstracts transmission as hop-by-hop movement along computed next-hop routes, with per-node transmission queues that emulate contention for a single shared radio interface. Node behavior includes a busy-time model that prevents simultaneous transmissions and introduces serialization delays proportional to packet size. Queueing behavior is explicitly modeled at each node, allowing packets to accumulate, wait, and be selected by a scheduling policy at each transmission opportunity.

Three scheduling policies are implemented and compared within the same simulation framework: First-Come First-Served, Shortest Job First, and a reinforcement learning-based Q-learning scheduler. The Q-learning scheduler operates over a discretized state space derived from packet and network features such as retransmission count, estimated link quality, queue occupancy, packet age, priority class, and packet size. Actions correspond to selecting a packet from the current node's transmission queue, with rewards shaped by delivery success and latency penalties.

The simulator supports both interactive execution and headless batch evaluation modes. In headless mode, identical traffic traces are replayed across scheduling policies to ensure controlled comparisons. This enables direct evaluation of algorithmic differences under identical network conditions without stochastic divergence between runs.

Simulation results are used to evaluate system performance under controlled variations in network density, traffic load, and topology structure, including conditions that cannot be reproduced on the physical hardware testbed due to limitations in node count, spatial scale, and sustained congestion duration. This includes stress testing under high queue buildup, intermittent connectivity, and uneven link quality distributions, allowing assessment of scheduler behavior in regimes that approximate post-disaster or infrastructure-degraded environments.

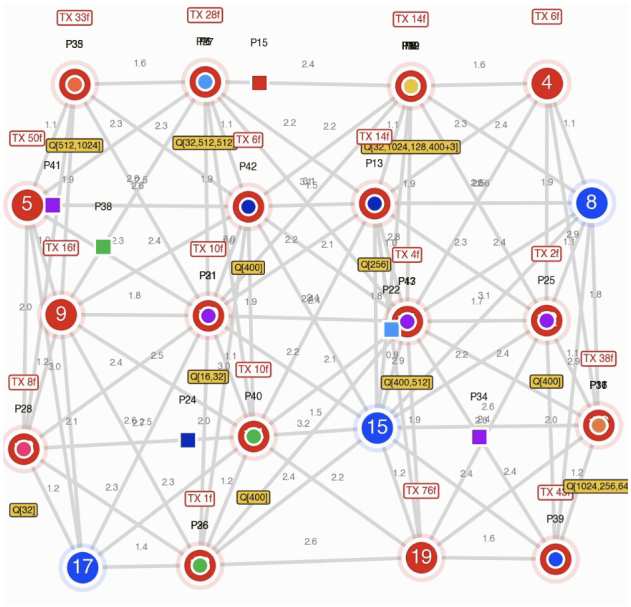


Figure 1.7: Network Simulator Visualization

1.3.9 Evaluation metrics

System performance is evaluated using packet delivery ratio, end-to-end latency, and queue waiting time, disaggregated per packet class to capture both reliability and fairness effects under heterogeneous traffic conditions.

Packet delivery ratio measures the reliability of the network at the transport level, capturing the fraction of successfully delivered packets relative to those generated [10]. This metric is critical in lossy wireless environments such as LoRa, where interference, duty cycle limitations, and multi-hop forwarding can introduce non-negligible packet loss.

End-to-end latency quantifies the temporal efficiency of the system by measuring the total time elapsed between packet generation and successful delivery. This metric captures the combined effects of routing path length, retransmissions, and medium access contention, and is particularly relevant for time-sensitive emergency messaging.

Queue waiting time per packet class isolates the effect of scheduling decisions on different traffic types. Unlike end-to-end latency, which aggregates network effects, queue waiting time reflects local resource contention at the node level. Disaggregating this metric by packet class enables evaluation of whether the scheduling policy introduces systematic delays for lower-priority or edge-case traffic, thereby providing a direct measure of fairness under constrained airtime conditions.

Chapter 2

User Research

2.1 Methods

HALoN is implemented as an embedded firmware system deployed on Adafruit Feather 32u4 nodes equipped with RFM95 LoRa transceivers. All core networking functionality, including packet formation, neighbor discovery, link-state advertisement exchange, routing table computation, and packet forwarding, is implemented directly in firmware and executed on physical hardware.

System evaluation is conducted primarily through a multi-node hardware testbench, in which multiple HALoN nodes are deployed and observed under controlled communication conditions. This testbed enables validation of routing behavior, queue dynamics, and end-to-end message delivery under realistic wireless constraints, including interference and packet loss.

Hardware constraints, including limited RAM, low processing throughput, and strict timing requirements, directly influence system design. Data structures and control flow are optimized to operate within these constraints, ensuring deployability on low-cost embedded platforms.

The routing subsystem maintains a local link-state database constructed from periodic neighbor discovery messages and link-state advertisements. Changes in topology trigger recomputation of shortest paths using Dijkstra's algorithm. Time-based expiration of routing entries ensures consistency with current network conditions.

The scheduling subsystem manages a per-node transmission queue. Baseline policies, including FCFS and SJF, are implemented for comparison against a reinforcement learning-based scheduler. The Q-learning scheduler operates over a discrete state space derived from packet and network features and selects transmission actions from the queue. Rewards are computed based on delivery success, latency, and retransmission outcomes, with policy updates performed using temporal-difference learning.

To complement physical testing, a Python-based simulation environment is used for scalability analysis under network conditions that exceed practical hardware deployment limits. The simulator models node density, link quality as a function of distance and signal strength, traffic patterns, and transmission timing, enabling controlled evaluation

of scheduling and routing behavior at scale.

Performance is evaluated using delivery ratio, end-to-end latency, and queue waiting time. Metrics are further disaggregated by packet priority and age to assess fairness across traffic classes.

Testing is structured across unit, integration, and system levels. Unit tests validate individual firmware modules, integration tests evaluate multi-node communication on the hardware testbench, and simulation tests explore large-scale and high-congestion scenarios.

2.2 Stakeholder needs

Stakeholder analysis is informed by established ethical frameworks, including duties of care, distributive justice, and rights-based design principles as articulated by the Markkula Center for Applied Ethics. These frameworks require that the system avoid harm to intended users, allocate limited communication resources in a justifiable manner, and provide accurate and transparent feedback regarding system capability and failure conditions.

The primary stakeholders are individuals in disaster-affected environments who rely on HALoN for emergency communication. These users operate under high uncertainty and limited situational awareness. In this context, undetected message loss constitutes a critical system failure, as it may lead users to incorrectly assume successful delivery. Accordingly, explicit acknowledgment of delivery success or failure is a required system property, rather than an implementation detail.

Emergency responders and coordination teams constitute a second stakeholder group. Their effectiveness depends on timely access to relevant communications. Scheduling policies that do not explicitly differentiate between urgent and non-urgent traffic effectively embed implicit prioritization decisions within system behavior. However, urgency-aware scheduling introduces fairness tradeoffs, as increased service for high-priority traffic may increase latency for lower-priority messages. These tradeoffs must be explicit and observable within system design.

Deployment and maintenance personnel, including volunteers and community emergency response teams, are stakeholders with respect to system usability and operational burden. Systems that require specialized expertise or complex configuration procedures shift deployment costs onto resource-constrained groups. In this context, deployability and documentation clarity are treated as design requirements consistent with professional responsibility guidelines such as the ACM Code of Ethics.

A further stakeholder group consists of users located at network peripheries or in geographically obstructed regions. These nodes are more likely to experience degraded link quality, higher retransmission rates, and increased queue delays. Scheduling policies that optimize solely for transmission efficiency may systematically deprioritize such nodes, resulting in uneven service quality across the network. This effect is particularly relevant in disaster contexts, where infrastructure-limited regions often overlap with historically underserved populations.

These stakeholder requirements introduce inherent tradeoffs. Urgency-based prioritization, fairness in resource allocation, and spectral efficiency are not simultaneously optimizable under duty-cycle-constrained, lossy wireless conditions. The objective of HALoN is therefore not to eliminate these tensions, but to make them explicit and structurally reflected in system behavior and evaluation.

2.2.1 User Stories

Based on our user research and the priority stakeholders involved in emergency communication during natural disasters, we define the following user stories. These stories capture core needs and expectations from civilians, responders, and deployers who will rely on the HALoN system.

Civilians in Affected Areas

- **As a civilian in a disaster zone,** I want to send short text messages to family members, **so that** I can confirm my safety when cell networks are down.
- **As a civilian in a disaster zone,** I want to receive local alerts broadcast through the network, **so that** I can stay informed about evacuation instructions.

Emergency Responders

- **As an emergency responder,** I want to coordinate with other teams through a reliable backup network, **so that** we can continue operations when infrastructure is destroyed.

System Operators / Deployment Personnel

- **As a deployment operator,** I want to launch and configure high-altitude LoRa nodes quickly, **so that** communication coverage can be restored within minutes.

Network Administrators

- **As a network administrator,** I want to monitor node-to-node connectivity and link quality, **so that** I can ensure continuous coverage across the affected area.

Chapter 3

Design and Rationale

3.1 Design

HALoN's architecture is layered, and each layer is shaped by the same underlying constraints and value commitments. At the physical and link layer, nodes communicate over LoRa in the 900 MHz ISM band, selected because it is unlicensed under FCC Part 15, removing the regulatory and financial barriers that licensed spectrum would impose on community-level deployment. Packets carry source and destination identifiers, packet type, sequence number, and TTL. Periodic HELLO messages maintain a neighbor table that records both connectivity and an estimate of link quality via received signal strength. The network layer implements a link-state routing protocol. Each node generates LSAs reflecting its current neighbor relationships and propagates them through the mesh. Nodes maintain a local topology database and recompute paths using Dijkstra's algorithm as that database changes. The result is a shared, consistent view of network structure that enables multi-hop routing without any centralized coordinator. This is a deliberate structural choice: a system intended for environments where infrastructure has failed should not itself depend on a central node whose failure would compromise the network. The transport layer is intentionally minimal. A lightweight acknowledgment mechanism confirms delivery at the destination, and retransmissions are bounded. When the retransmission limit is reached without an acknowledgment, the system reports failure rather than continuing to retry. This reflects the same commitment to honesty that runs through the transport layer's design in the ethics analysis: a bounded process that accurately reports failure provides the user with information they can act on. An unbounded process that never reports failure provides the illusion of persistence at the cost of accuracy. Message fragmentation handles payloads that exceed LoRa's maximum packet size. In the event of partial delivery, incomplete fragments are discarded rather than forwarded to the recipient. In a crisis context, a partial message, whether truncated location data or an incomplete instruction, may cause more harm than no message at all. Discarding fragments is a conservative choice made in the interest of data integrity. The transmission queue is where the most significant design effort is concentrated. Packets enter from local generation or from forwarding, and the scheduler selects which to transmit at each window. FCFS is implemented as the primary baseline. The Q-learning scheduler evaluates each packet

against a state vector encoding age, priority, retransmission count, queue depth, and link quality, and selects based on a value function updated from observed delivery outcomes. The age component serves as a starvation safeguard: the longer a message remains undelivered, the more strongly the scheduler is pushed toward transmitting it, regardless of other factors. This is a direct response to the distributive concern that a purely efficiency-driven scheduler would persistently disadvantage users on poor links [11]. Routing determines reachability; scheduling determines access. In a duty-cycle-limited network, these are different problems, and solving one without addressing the other produces a system that knows where messages should go but allocates the channel in ways that do not reflect the system's actual priorities. The scheduler is the site where HALoN's commitments to fairness and urgency-sensitivity are either realized or abandoned.

3.2 Use Case Overview

The use cases in this section were derived directly from the user stories in Chapter 2.3. Each user story describes a high-level goal from the perspective of a key stakeholder, and the corresponding use cases translate those goals into concrete system interactions. These use cases guide functional design, interface planning, and testing of the HALoN system.

3.3 Use Case Descriptions

Use Case 1: Send Emergency Message

Actor: Civilian in Affected Area

Preconditions:

- User device is within range of at least one HALoN node.
- Device is powered and has the HALoN communication interface available.

Main Flow:

1. User opens the emergency messaging interface.
2. User composes a short text message.
3. User sends the message.
4. System forwards the message to the nearest balloon node.
5. Node relays message across the network to the destination recipient or broadcast area.

Alternate Flow: If the node link is temporarily unreachable, the device stores the message and retries until a connection is re-established.

Postconditions: Message is delivered or queued for delivery.

Use Case 2: Receive Broadcast Alert

Actor: Civilian in Affected Area

Preconditions:

- User device is powered on.
- Device is within broadcast range of at least one node.

Main Flow:

1. System receives an alert from an emergency authority.
2. Nodes broadcast the alert across the network.
3. User device detects and displays the alert.

Postconditions: User is informed of emergency updates or safety instructions.

Use Case 3: Coordinate Response Operations

Actor: Emergency Responder

Preconditions:

- Responder device is authenticated and connected to the HALoN network.

Main Flow:

1. Responder opens the team coordination interface.
2. Responder sends location or status updates to the network.
3. Other responders receive and acknowledge the updates.

Alternate Flow: If the network is congested, messages are prioritized based on responder-level permissions.

Postconditions: Team operations remain coordinated despite infrastructure failure.

Use Case 4: Deploy HALoN Nodes

Actor: Deployment Operator

Preconditions:

- Balloon, microcontroller, and LoRa hardware are available and functional.

Main Flow:

1. Operator activates and configures the node.
2. Operator launches the balloon.
3. System establishes a link between the new node and existing nodes.
4. System confirms the node is active and part of the mesh network.

Postconditions: New node provides additional coverage in the disaster area.

Use Case 5: Monitor Network Health

Actor: Network Administrator

Preconditions: Admin has access to the HALoN monitoring dashboard.

Main Flow:

1. Admin logs into the monitoring system.
2. Admin views node status indicators (signal strength, packet loss, battery levels).
3. Admin identifies nodes with degraded performance.
4. Admin triggers maintenance alerts or recommends repositioning deployments.

Postconditions: Network performance is tracked and maintained.

3.4 Use Case Diagram

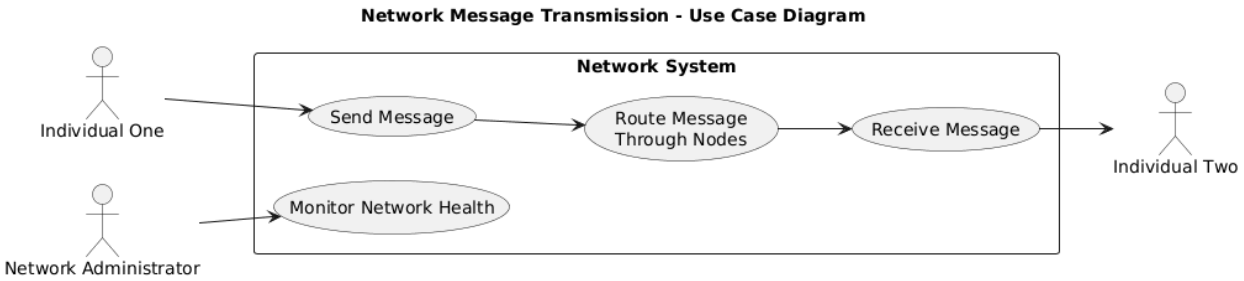


Figure 3.1: Use Case Diagram: Message Transmission

3.5 Class Diagram

The UML class diagram shown in Figure 3.2 presents the primary abstractions used in both the embedded firmware and the simulation environment.

At the firmware level, communication is centered around the `PKT` data structure, which encapsulates routing metadata, hop indices, and user payload data. The `FirmwareNode` abstraction coordinates packet injection, forwarding, and delivery while interfacing with the RH-RF95 LoRa radio driver.

The simulation environment introduces the `MessagePacket` and `NetworkSimulator` abstractions, enabling controlled experimentation with routing and scheduling policies while preserving conceptual alignment with the deployed firmware.

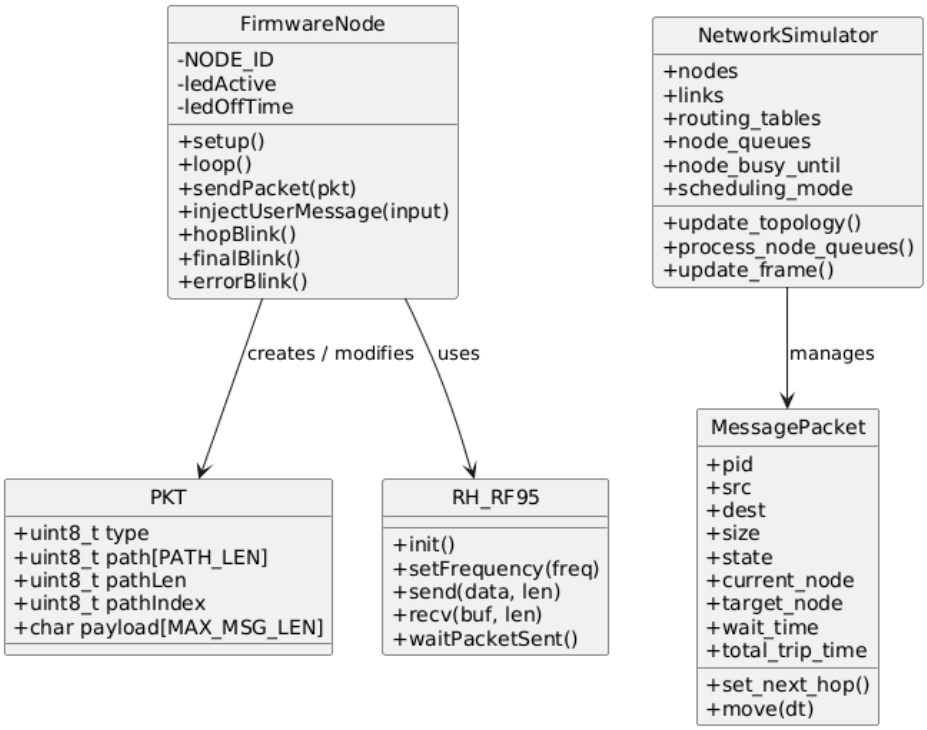


Figure 3.2: UML Class Diagram of the HALoN System

3.6 Sequence Diagram

The sequence diagram in Figure 3.3 illustrates the interactions involved in transmitting a message across the HALoN mesh network. A source node constructs a packet and transmits it via the LoRa radio interface. Intermediate nodes receive the packet, validate their position in the routing path, increment the hop index, and forward the packet to the next node. When the packet reaches its destination, it is delivered to the application layer.

This explicit hop-by-hop forwarding strategy avoids continuous route discovery traffic and ensures deterministic packet flow in constrained or unstable network conditions.

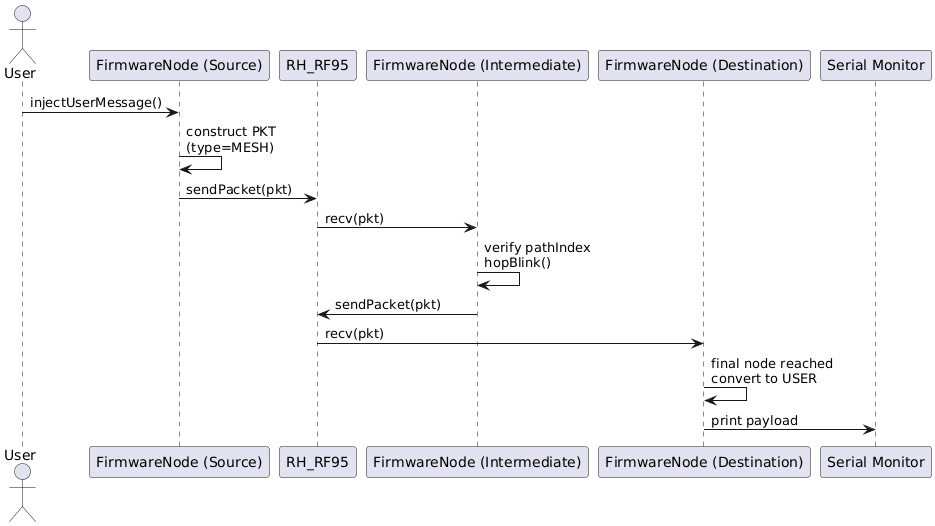


Figure 3.3: Packet Transmission Sequence Diagram

3.7 Activity Diagram

The activity diagram shown in Figure 3.4 models packet handling within the HALoN simulation environment. Packets are injected into the system, placed into node-level queues, scheduled according to the selected policy, and propagated through the network topology. As packets traverse the network, queueing delay and transmission latency are accumulated and recorded for performance evaluation.

This activity-driven model enables systematic comparison of scheduling strategies under varying network conditions.

3.8 Functional requirements

The functional requirements of the HALoN system define the essential capabilities required for reliable operation in emergency and infrastructure-degraded environments. These requirements are derived from the need to support intermittent connectivity, constrained hardware resources, and heterogeneous node deployment at both ground and aerial layers of the network.

First, the system must support reliable node-to-node communication over a multi-hop wireless mesh. Each LoRa-enabled microcontroller must be capable of transmitting, receiving, and forwarding packets to neighboring nodes, enabling messages to propagate beyond single-hop radio range. The target operational scale is approximately 20 nodes covering up to 50 square kilometers under sparse and variable connectivity conditions. This peer-to-peer forwarding capability is essential for maintaining network functionality when individual links fail or become temporarily unavailable.

Second, the system must support bidirectional ground-to-network communication. External users, including civil-

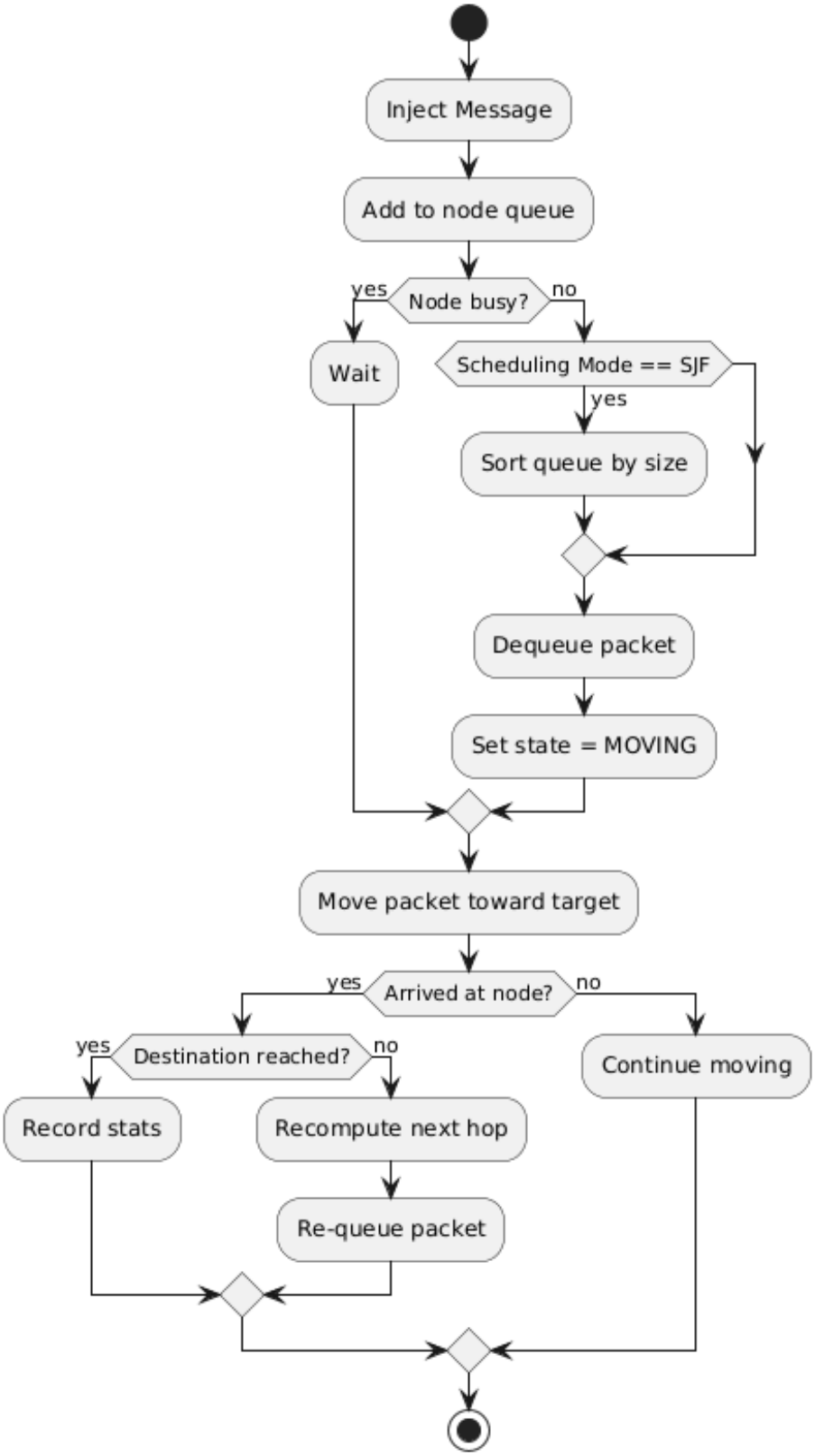


Figure 3.4: Activity Diagram of Packet Handling in the HALoN Simulator

ians and emergency personnel, must be able to inject messages into the network through ground-level interfaces. These messages are received by the nearest accessible node and integrated into the mesh routing and forwarding pipeline. From a system perspective, this requirement is abstracted as reliable ingress and egress of packets between external clients and the internal multi-hop network, without requiring centralized infrastructure.

Finally, each node must support adaptive antenna reconfiguration between broadside and endfire radiation modes to optimize communication performance across heterogeneous spatial layouts. This capability allows the system to dynamically trade off between wide-area connectivity and directional gain depending on network geometry. Broadside mode supports dense lateral connectivity across ground-level nodes, as seen in Fig. 3.5, and it is implemented using the diagram in Fig. 3.6.

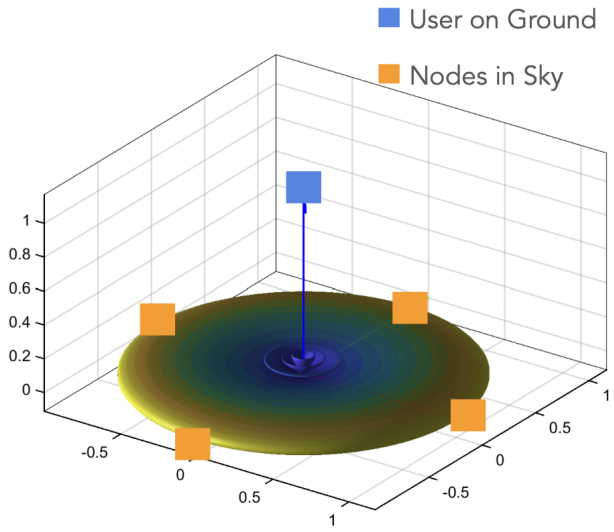


Figure 3.5: Broadside Radiation Communication Pattern

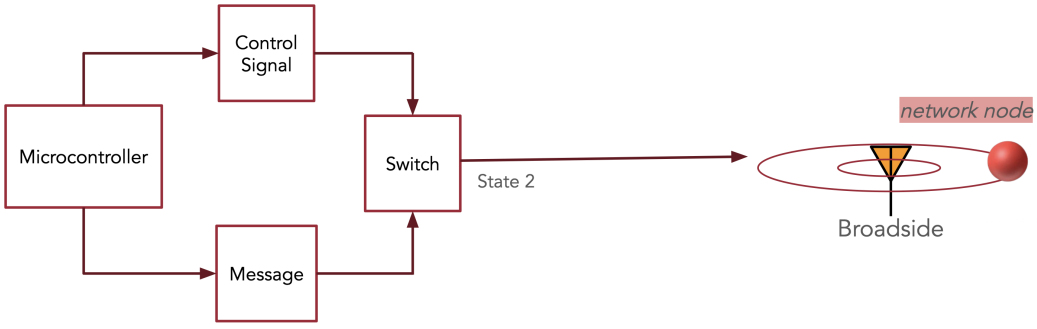


Figure 3.6: Broadside Radiation Communication Pattern Design

Endfire mode enables long-range directional links to elevated or aerial relay nodes, as seen in Fig. 3.7, and the implementation is in Fig. 3.8. This adaptability is required to maintain link quality and energy efficiency under varying

deployment conditions.

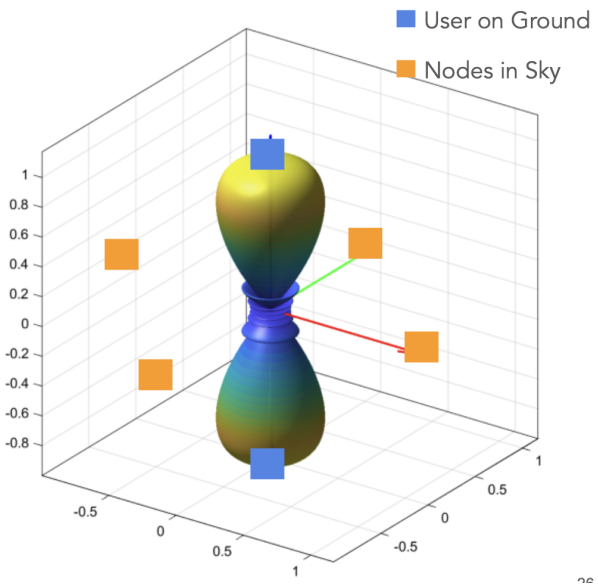


Figure 3.7: Endfire Radiation Communication Pattern

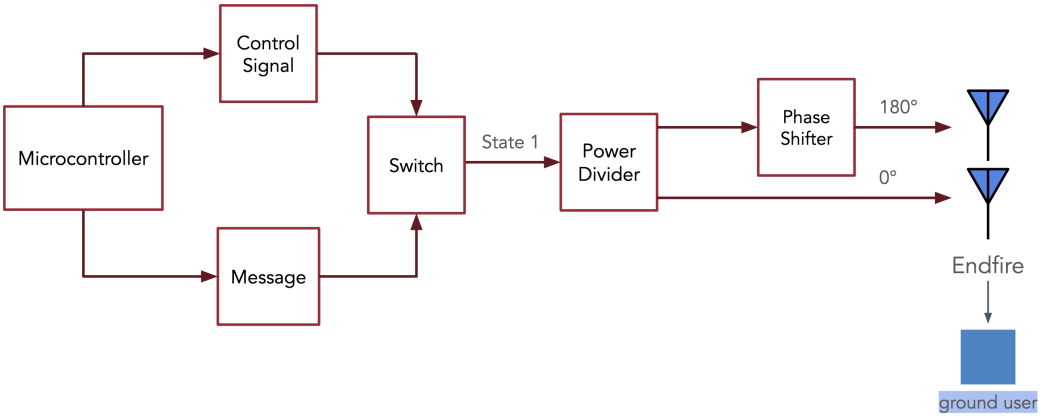


Figure 3.8: Endfire Radiation Communication Pattern Design

By dynamically switching between these two radiation modes, each node can adapt its physical-layer transmission characteristics to the spatial structure of the network. Broadside mode supports dense, distributed ground-level mesh connectivity, while endfire mode enables efficient long-range uplinks and downlinks to elevated infrastructure. This reconfiguration provides a hardware-level mechanism for adapting link quality and energy expenditure to the topology of the network rather than relying solely on routing or scheduling decisions.

3.9 Non-functional requirements

The non-functional requirements describe the attributes that the HALoN system must exhibit to ensure dependable performance in disaster environments, qualities that we discerned from analysis of disaster communications [12, 13]. Reliability is essential, as the system should continue functioning even when individual nodes fail or connection paths are lost, as is likely during an emergency or crisis situation: in other words, the mesh network must remain resilient to environmental disturbances and link quality. Power efficiency is also a critical requirement, with each node designed to operate on low-power microcontrollers and battery systems that allow for extended mission duration. Furthermore, the airborne hardware must exhibit environmental durability, withstanding the extreme temperatures, pressure changes, and mechanical stresses encountered at high altitudes and in crisis environments. Lastly, the system must incorporate strong security measures to protect communications from tampering or unauthorized access. Ideally, encryption or secure encoding methods should be employed to ensure message confidentiality and integrity across the network.

Chapter 4

RSSI-Based Link-State Routing

4.1 Overview and Design Rationale

Routing in a LoRa mesh must solve a harder problem than routing in a typical wired or Wi-Fi network. The topology changes continuously as nodes move and link qualities fluctuate with the radio environment; some nodes may appear and disappear entirely as balloons drift or batteries are exhausted. Classical distance-vector protocols propagate only reachability information rather than the full topology, making loop detection slow and convergence after topology changes unpredictable. Link-state protocols, by contrast, give every node a complete view of the network graph, from which each node independently runs Dijkstra's algorithm to compute consistent, loop-free routes [8]. Convergence after a topology change is bounded by the time to flood the changed link-state advertisement to all reachable nodes.

HALoN therefore adopts the link-state model. Each node periodically broadcasts a Link State Advertisement (LSA) describing its directly reachable neighbours and the cost of each link. Every node accumulates these advertisements into a local topology matrix and re-runs Dijkstra after each update. The design questions this chapter addresses are: how to translate raw RSSI and SNR measurements into meaningful link costs; how to prevent stale or duplicate advertisements from corrupting the topology; how to clear the topology matrix correctly when a fresh LSA arrives; and how to handle node expiry gracefully on a microcontroller where heap allocation and dynamic data structures are unavailable.

4.2 Neighbour Discovery via HELLO Packets

Topology discovery begins with periodic link probing. Each node broadcasts a `PKT_HELLO` packet at every `HELLO_INTERVAL` milliseconds. HELLO packets carry no payload; their sole purpose is to give listening nodes an opportunity to record fresh RSSI and SNR measurements for the sender. When a HELLO from node u arrives at node v, the firmware calls `neighbor_update(`u`, rssi, snr)`, which converts the measurements into a link cost $c(v, u)$ via the function described in Section 4.2.1 and stores the cost in the neighbour table with a timestamp.

4.2.1 RSSI-to-Cost Mapping

The RFM95 transceiver reports a raw RSSI value in dBm, typically in the range $[-140, -30]$. A stronger signal (less negative) corresponds to a more reliable link. The mapping from RSSI to link cost must be monotonically decreasing (stronger signal implies lower cost), strictly non-negative for compatibility with Dijkstra's algorithm, and quantised into a small number of levels to suppress sensitivity to momentary fluctuations in individual RSSI readings. We define the mapping $c : \mathbb{Z} \to \{1, 10, 30, 70, 100\}$ by

$$c(\text{rssi}) = \begin{cases} 1 & \text{rssi} \geq -60, \\ 10 & -80 \leq \text{rssi} < -60, \\ 30 & -100 \leq \text{rssi} < -80, \\ 70 & -110 \leq \text{rssi} < -100, \\ 100 & \text{rssi} < -110. \end{cases} \tag{4.1}$$

The cost values are deliberately non-uniform, and the design rationale for each gap is worth making explicit. The jump from 1 to 10 between the top two bins is small in absolute terms but sufficient for Dijkstra to prefer any strong link over any merely decent one regardless of path length. The jump from 30 to 70 at the -100 dBm threshold is more dramatic: it ensures that a two-hop path through two decent links ($2 \times 30 = 60$) is cheaper than a single hop over a bad link (70). This reflects the operational judgment that an extra relay hop is an acceptable cost to avoid a near-marginal radio path. The jump from 70 to 100 at -110 dBm marks the boundary of links that are essentially unreliable under realistic conditions and should be avoided unless absolutely no alternative exists.

The SNR measurement acts as a penalty modifier. If SNR falls below the threshold SNR_{min} (set to 0 dB in the default configuration), a constant additive penalty is applied to the RSSI before mapping, effectively downgrading the quality estimate to account for interference conditions that raw signal strength alone does not capture. This is important in urban environments where adjacent-channel interference from consumer electronics can degrade link quality without substantially lowering the received power.

When a neighbour's HELLO cost changes or when a neighbour is heard for the first time, transitioning its stored cost from `INF_COST` to a finite value, the firmware immediately updates the local topology matrix and triggers a route recomputation. A hysteresis threshold `COST_CHANGE_THRESHOLD` prevents an oscillating link from triggering a continuous flood of LSAs: only changes that exceed this threshold cause the node to advertise a fresh LSA to its neighbours. The handler is shown in Listing 4.1.

Neighbours that stop sending HELLOs are expired by `neighbor_expire()`: any neighbour not heard from within `HELLO_TIMEOUT` is removed, its link cost is set to `INF_COST`, and a route recomputation is triggered. If the expired neighbour was the only path to some portion of the network, the routing table will reflect this immediately after the

```
static void handle_hello(const Packet* pkt, int8_t rssi, int8_t snr) {
    uint8_t old_cost = neighbor_get_cost(pkt->src_id);
    neighbor_update(pkt->src_id, rssi, snr);
    uint8_t new_cost = neighbor_get_cost(pkt->src_id);

    if (new_cost != old_cost) {
        routing_set_link(NODE_ID, pkt->src_id, new_cost);
        routing_compute();
        // only flood a new LSA if the change is significant.
        uint8_t diff = (new_cost > old_cost)
                     ? (new_cost - old_cost) : (old_cost - new_cost);
        if (old_cost == INF_COST || diff >= COST_CHANGE_THRESHOLD)
            lsa_set_send_flag();
    }
}
```

Listing 4.1: HELLO handler with cost-change detection (`halon_firmware.ino`).

recomputation, and messages queued for those destinations will fail fast rather than being sent into a broken path.

4.3 Link State Advertisements

4.3.1 Payload Structure and Construction

A Link State Advertisement encodes the originating node's identity, an 8-bit outgoing sequence counter, and the list of currently active neighbours with their costs. The payload layout used by `lsa_build()` is:

Byte 0	1	2	3	4	5	6	⋯
origin	seq	n	nbr_0	cost_0	nbr_1	cost_1	⋯

where n is the neighbour count and the remainder of the payload contains n two-byte pairs. The total payload length is therefore $3 + 2n$ bytes. With `MAX_NEIGHBORS` = 8, the maximum payload length is $3 + 16 = 19$ bytes, comfortably within LoRa's maximum payload at all standard spreading factors (Listing 4.2).

4.3.2 Sequence Number Deduplication and Wraparound

When node v receives an LSA with origin u and sequence number σ_{rx}, it must decide whether the advertisement is fresh or stale. The LSA module maintains a per-origin array `last_seq[MAX_NODES]`, initialised to `0xFF` as a "never seen" sentinel. The acceptance condition is

$$\text{accept} \iff \texttt{last_seq}[u] = \texttt{0xFF} \quad \vee \quad (\text{int8_t})(\sigma_{\text{rx}} - \texttt{last_seq}[u]) > 0. \tag{4.2}$$

The cast to `int8_t` in (4.2) is the critical correctness detail, and understanding why requires thinking carefully about unsigned arithmetic wraparound. When the outgoing sequence counter `my_seq` increments past 255, it wraps to 0. In unsigned 8-bit arithmetic, the fresh sequence number 0 compared against the last accepted value 255 gives

```
void lsa_build(Packet* out) {
    uint8_t ids[MAX_NEIGHBORS];
    uint8_t costs[MAX_NEIGHBORS];
    uint8_t n = neighbor_get_all(ids, costs);

    uint8_t* p = out->payload;
    p[0] = NODE_ID;
    p[1] = my_seq++;
    p[2] = n;
    for (uint8_t i = 0; i < n; i++) {
        p[3 + 2*i]     = ids[i];
        p[3 + 2*i + 1] = costs[i];
    }
    out->payload_len = 3u + n * 2u;

    out->src_id   = NODE_ID;
    out->dst_id   = BROADCAST_ID;
    out->next_hop = BROADCAST_ID;
    out->type     = PKT_LSA;
    out->ttl      = TTL_DEFAULT;
}
```

Listing 4.2: LSA construction from current neighbour table (`lsa.cpp`).

$0 - 255 = -255$, which when truncated to 8 bits yields 1, a positive number, which would appear to indicate that 0 is newer than 255. That happens to be correct in this case, but the signed interpretation is what makes it correct *systematically* for all possible wraparound scenarios. The signed 8-bit difference int8_t$(0 - 255)$ = int8_t$(1) = 1 > 0$ correctly identifies sequence 0 as newer than sequence 255. More generally, the signed comparison handles all cases where the true difference is in $(-128, 128)$, which is guaranteed as long as no two consecutive sequence numbers sent by the same node differ by more than 127 in the true count, a condition easily satisfied at a 30-second periodic LSA interval.

The reception and update logic is given in Listing 4.3.

4.3.3 Selective Row Clearing and the Topology Invariant

A subtle but consequential correctness issue arises in how an accepted LSA updates the global topology matrix. The topology is stored as a flat two-dimensional array

$$T \in \{0, \ldots, 255\}^{N \times N}, \qquad T[u][v] = \text{cost of the link } u \to v \text{ as most recently advertised by node } u.$$

The invariant to be maintained is:

$$T[u][v] = c(u \to v) \quad \text{as reported in node } u\text{'s most recent accepted LSA,} \tag{4.3}$$

with $T[u][v]$ = `INF_COST` for links not mentioned in u's advertisement, and $T[w][u]$ for $w \neq u$ determined solely by w's most recent LSA.

```
bool lsa_receive(Packet* pkt) {
    if (pkt->payload_len < 3) return false;
    uint8_t* p     = pkt->payload;
    uint8_t origin = p[0];
    uint8_t seq    = p[1];
    uint8_t n      = p[2];

    if (origin == 0 || origin > MAX_NODES)           return false;
    if ((uint8_t)(3u + n * 2u) > pkt->payload_len)   return false;

    uint8_t idx = origin - 1u;
    if (last_seq[idx] != 0xFF && (int8_t)(seq - last_seq[idx]) <= 0)
        return false;          // stale or duplicate: drop silently

    last_seq[idx]       = seq;
    lsa_last_heard[idx] = millis();

    routing_clear_row(origin);
    for (uint8_t i = 0; i < n; i++) {
        uint8_t nbr_id = p[3 + 2*i];
        uint8_t cost   = p[4 + 2*i];
        if (nbr_id == 0 || nbr_id > MAX_NODES) continue;
        routing_set_link(origin, nbr_id, cost);
    }
    return true;
}
```

Listing 4.3: Sequence number check and topology update (`lsa.cpp`).

An earlier version of the firmware used `routing_set_link(`u`, `v`, INF_COST)` to clear all of u's links before applying the new LSA. This is incorrect: `routing_set_link` applies symmetrically, setting both $T[u][v]$ and $T[v][u]$. If node w had previously advertised a link to u (setting $T[w][u]$ to a finite cost), the symmetric clear would silently destroy $T[w][u]$, violating the invariant for row w even though no new LSA from w has arrived. The topology entry for $w \to u$ would disappear, causing Dijkstra to compute routes that bypass that link even though w still considers it active.

The correct approach is *selective row clearing*: when an LSA from origin u is accepted, only row u of the topology matrix is zeroed, leaving all other rows exactly as they were. The firmware provides two distinct clearing operations for two distinct situations, shown in Listing 4.4.

`routing_clear_row` is used when processing an incoming LSA: only the originating node's advertised links are affected, and all other nodes' views of their links remain intact. `routing_clear_node` is reserved for node expiry, where the node is being treated as entirely absent: not only should its own advertised links disappear, but all references to it as a destination or intermediate relay in other nodes' rows should also be removed.

```
void routing_clear_row(uint8_t node_id) {
    if (!valid_id(node_id)) return;
    uint8_t idx = node_id - 1u;
    for (uint8_t v = 0; v < MAX_NODES; v++) {
        if (v == idx) continue;
        topo[idx][v] = INF_COST;
    }
}

void routing_clear_node(uint8_t node_id) {
    if (!valid_id(node_id)) return;
    uint8_t idx = node_id - 1u;
    for (uint8_t v = 0; v < MAX_NODES; v++) {
        topo[idx][v] = INF_COST;        // clear outgoing (row)
        topo[v][idx] = INF_COST;        // clear incoming (column)
    }
}
```

Listing 4.4: Row clearing versus full node clearing (`routing.cpp`).

4.4 Dijkstra's Algorithm in a Constrained Environment

4.4.1 Data Structures and Memory Budget

The topology matrix `topo[MAX_NODES][MAX_NODES]` uses `uint8_t` entries, totalling $20 \times 20 = 400$ bytes. The sentinel `INF_COST = 0xFF = 255` marks absent links; the maximum legitimate link cost is 100, well below 255, so the sentinel is unambiguous. Node IDs are 1-indexed externally and stored 0-indexed internally, so node ID n is stored at index $n - 1$. The next-hop table `next_hop[MAX_NODES]` stores one 1-indexed node ID per destination, using 20 bytes.

Distance values during Dijkstra are stored as `uint16_t`. The argument for this choice follows directly from the parameters. Link costs lie in $\{1, 10, 30, 70, 100\}$, and paths are at most $N_{\max} = 20$ hops long. The maximum possible shortest-path cost is therefore

$$c_{\max} \times N_{\max} = 100 \times 20 = 2000 \leq 65535 = 2^{16} - 1.$$

Using `uint8_t` for distances would overflow at any path cost exceeding 255, which could happen on a three-hop path of decent links ($3 \times 100 = 300 > 255$), producing silently incorrect routing decisions. Using `uint16_t` prevents this at the cost of only $20 \times 2 = 40$ additional bytes of stack usage during the computation. The stack variables for the algorithm are allocated automatically at function entry and freed at return, so there is no persistent memory cost.

4.4.2 Algorithm and Next-Hop Table Construction

The algorithm initialises all distances to ∞ (represented as `0xFFFF`), sets the source distance to zero, and iterates N times. On each iteration it extracts the unvisited node with minimum distance, marks it visited, and relaxes its

```
void routing_compute(void) {
    uint16_t dist[MAX_NODES];
    uint8_t  prev[MAX_NODES];
    bool     visited[MAX_NODES];

    for (uint8_t i = 0; i < MAX_NODES; i++) {
        dist[i] = 0xFFFF; prev[i] = 0xFF; visited[i] = false;
    }
    uint8_t src = NODE_ID - 1u;
    dist[src]   = 0;

    for (uint8_t iter = 0; iter < MAX_NODES; iter++) {
        uint8_t u = 0xFF; uint16_t min_d = 0xFFFF;
        for (uint8_t i = 0; i < MAX_NODES; i++)
            if (!visited[i] && dist[i] < min_d)
                { min_d = dist[i]; u = i; }
        if (u == 0xFF) break;    // all reachable nodes settled
        visited[u] = true;

        for (uint8_t v = 0; v < MAX_NODES; v++) {
            if (topo[u][v] == INF_COST) continue;
            uint16_t candidate = dist[u] + topo[u][v];
            if (candidate < dist[v])
                { dist[v] = candidate; prev[v] = u; }
        }
    }
```

Listing 4.5: Dijkstra's algorithm: shortest-path computation (`routing.cpp`).

outgoing edges. The next-hop table is populated after all distances are finalised by walking the predecessor chain `prev[]` backward from each destination to the source to find the first-hop neighbour. The shortest-path computation is shown in Listing 4.5, and the next-hop table construction in Listing 4.6.

The algorithm uses a linear scan for the minimum-distance node, giving $O(N^2)$ overall complexity. With $N = 20$, this involves at most $20 \times 20 = 400$ iterations of the inner relaxation loop and $20 \times 20/2 = 200$ comparisons in the minimum scan, totalling roughly 600 simple integer operations. On the Cortex-M0+ running at 16 MHz, this completes in well under one millisecond. A priority-queue implementation would reduce the complexity to $O((N + E) \log N)$, but for $N = 20$ the constant factor savings are irrelevant and the heap allocation required by a priority queue is unavailable.

The predecessor chain walk in the final loop is $O(N)$ per destination, and since there are at most $N - 1$ destinations the total post-processing cost is $O(N^2)$, matching the complexity of the main loop. The walk is correct because the predecessor array `prev[]` forms a tree rooted at the source: following `prev[]` from any destination always terminates at `src` (if the destination is reachable) or at a node with `prev[curr] = 0xFF` (if the chain is broken, indicating an unreachable node). The routing table is populated only for the former case.

```
    memset(next_hop, NO_ROUTE, sizeof(next_hop));
    next_hop[src] = NODE_ID;          // route to self: trivial

    for (uint8_t i = 0; i < MAX_NODES; i++) {
        if (i == src || dist[i] == 0xFFFF) continue;
        uint8_t curr = i;
        while (prev[curr] != 0xFF && prev[curr] != src)
            curr = prev[curr];
        if (prev[curr] == src)
            next_hop[i] = curr + 1u;  // convert 0-indexed back to node ID
    }
}
```

Listing 4.6: Dijkstra's algorithm: next-hop table from predecessor chain (`routing.cpp`).

```
static void handle_lsa(Packet* pkt, int8_t rssi, int8_t snr) {
    (void)snr;
    uint8_t origin = (pkt->payload_len >= 1) ? pkt->payload[0] : 0;
    uint8_t ncount = (pkt->payload_len >= 3) ? pkt->payload[2] : 0;
    bool changed = lsa_receive(pkt);
    if (changed) {
        routing_compute();
        Serial.print(F("LSA from N")); Serial.print(origin);
        Serial.print(F(" nbrs="));    Serial.print(ncount);
        Serial.print(F(" rssi="));    Serial.println(rssi);
    }
    if (changed && pkt->ttl > 1) {
        pkt->ttl--;
        pkt->prev_hop = NODE_ID;
        radio_send(pkt);
    }
}
```

Listing 4.7: LSA receive and conditional relay (`halon_firmware.ino`).

4.5 LSA Flooding, Relay, and Topology Aging

4.5.1 Controlled Flooding via TTL

When node v accepts a fresh LSA from origin u and the TTL of the received packet exceeds 1, node v decrements the TTL, sets itself as `prev_hop`, and rebroadcasts the packet. This constitutes a controlled flood: each relay hop decrements the TTL by one, so the advertisement propagates at most `TTL_DEFAULT` $-$ 1 hops beyond the originator. The sequence number check at each receiver ensures the same LSA is never processed twice by the same node, and since each relay sets its own `prev_hop`, the propagation path is traceable for debugging. The combination of TTL limiting and sequence deduplication guarantees termination of the flood and eliminates routing loops in the control plane. The receive-and-relay handler is shown in Listing 4.7.

```
static uint8_t relay_idx = 0;

bool lsa_build_relay(Packet* out) {
    uint8_t self = NODE_ID - 1u;
    for (uint8_t tries = 0; tries < MAX_NODES; tries++) {
        relay_idx = (uint8_t)((relay_idx + 1u) % MAX_NODES);
        if (relay_idx == self)              continue;
        if (last_seq[relay_idx] == 0xFF)    continue;  // never heard
        if (lsa_last_heard[relay_idx] == 0) continue;  // expired

        uint8_t origin = relay_idx + 1u;
        uint8_t ids[MAX_NEIGHBORS], costs[MAX_NEIGHBORS];
        uint8_t n = routing_get_neighbors(origin, ids, costs);
        if (n == 0) continue;

        uint8_t* p = out->payload;
        p[0] = origin; p[1] = last_seq[relay_idx]; p[2] = n;
        for (uint8_t i = 0; i < n; i++) {
            p[3 + 2*i] = ids[i]; p[3 + 2*i + 1] = costs[i];
        }
        out->payload_len = 3u + n * 2u;
        out->src_id   = origin;
        out->dst_id   = BROADCAST_ID;
        out->next_hop = BROADCAST_ID;
        out->type     = PKT_LSA;
        out->ttl      = TTL_DEFAULT;
        return true;
    }
    return false;   // nothing to relay this cycle
}
```

Listing 4.8: Round-robin relay construction (`lsa.cpp`).

4.5.2 Round-Robin LSA Relay

In addition to forwarding received LSAs, each node periodically re-transmits one other node's stored LSA from its own topology knowledge, cycling through all known origins in round-robin order. This relay mechanism serves nodes that missed the original flood due to a transient connectivity gap: instead of waiting for the originator's next periodic advertisement, such a node can recover missing topology information from its neighbours' relay transmissions. The relay is built from the locally stored topology data rather than from a cached copy of the original packet, ensuring it reflects the most recent accepted link information (Listing 4.8).

4.5.3 Topology Aging and Node Expiry

The timestamp `lsa_last_heard[idx]` records when each origin's most recent LSA was accepted. On each main-loop iteration, `lsa_expire_origins()` checks every entry against the current time. An origin u is expired if and only if it has been heard from before (i.e. its `lsa_last_heard` is nonzero) and the elapsed time since last contact

exceeds LSA_ORIGIN_TIMEOUT:

$$\text{expire } u \iff \texttt{lsa_last_heard}[u] \neq 0 \quad \wedge \quad (\texttt{now} - \texttt{lsa_last_heard}[u]) > \texttt{LSA_ORIGIN_TIMEOUT}. \tag{4.4}$$

An expired origin is removed via `routing_clear_node()`, its `last_seq` entry is reset to `0xFF`, and a route recomputation is triggered. The reset of `last_seq` to `0xFF` is important: it means that if the node returns and resumes broadcasting, its fresh advertisements will be accepted immediately regardless of their sequence numbers, because the sentinel condition in (4.2) bypasses the comparison for never-seen origins. Without this reset, a returning node whose counter has wrapped would have its fresh LSAs rejected as stale until its sequence number exceeded the pre-expiry value potentially leaving it invisible to the routing layer for an entire counter cycle of 256 advertisements.

Chapter 5

The Q-Learning Packet Scheduler

5.1 Motivation and Design Philosophy

LoRa is a shared, half-duplex radio medium. When one node transmits, every other node within range must listen or wait: the channel is occupied for the entire duration of the airtime, and no two nodes in mutual radio range can transmit simultaneously without collision [14, 15]. In a network of N nodes sharing a single channel, the time available for any given node to transmit is fundamentally scarce and contested. The question of which packet to send next is therefore not a bookkeeping detail but a consequential decision that shapes the end-to-end latency, fairness, and delivery rate of the whole network.

In an emergency mesh, this scarcity is made worse by a heterogeneity of message types that simple scheduling disciplines cannot handle well. A first-come-first-served (FCFS) policy treats a low-priority status update identically to a life-critical alert: both are served solely by their position in the arrival queue. Shortest-job-first (SJF) maximises instantaneous channel throughput by preferring smaller packets, but systematically starves large messages and is entirely blind to their semantic content. Neither policy can adapt. They carry no concept of link quality, no awareness of how long a message has been waiting, and no mechanism to improve their behaviour over time as they observe outcomes [16, 17, 18].

The HALoN scheduler is designed to do better on all three fronts. It maintains a learned scoring function over a compact feature space derived from each packet's current properties and the state of the queue. At every transmission opportunity it selects the packet with the highest score, updating the scoring function after each outcome so that successful transmission strategies are reinforced and failing ones are penalised. The framework is Q-learning with linear function approximation, chosen specifically because the weight table and its update rule fit comfortably in the RAM of an Arduino-class microcontroller and can be executed in microseconds, while still capturing the most predictive structure in the scheduling problem.

5.2 The Feature Space

The first design decision in any reinforcement learning system is the state representation. A representation that is too fine-grained leads to a space too large to generalise across; one that is too coarse discards information the agent needs to distinguish good decisions from bad ones. For an embedded scheduler whose total SRAM budget is 32 KB and whose weight table must coexist with topology matrices, routing tables, and packet buffers, both pressures are real and neither can be ignored.

The chosen representation maps each packet p, together with the current queue depth q and wall-clock time t, into a six-dimensional feature vector

$$\mathbf{f}(p, q, t) = \big(B_r(p),\ B_\ell(p),\ B_q(q),\ B_\pi(p),\ B_a(p, t),\ B_z(p)\big), \tag{5.1}$$

where each component is a bucket index drawn from a small ordered set. The six features, their bucketing functions, and the rationale for each are developed below.

Retransmission count B_r. Let $r_p \in \mathbb{Z}_{\geq 0}$ denote the number of times packet p has been retransmitted without a successful acknowledgement. The retransmission bucket is

$$B_r(p) = \begin{cases} 0 & r_p = 0, \\ 1 & r_p = 1, \\ 2 & r_p \geq 2. \end{cases} \tag{5.2}$$

A high retransmission count is evidence of link trouble, receiver unavailability, or channel congestion. Continuing to retry such packets aggressively wastes channel time that could instead be spent on first-attempt packets with a higher probability of success. The bucketing deliberately collapses all counts $r_p \geq 2$ into a single bucket: the distinction between a packet on its third and fifth retransmission is not practically meaningful for scheduling, and keeping the feature space small improves generalisation across similar situations.

Link quality B_ℓ. The scheduler maintains an 8-bit sliding delivery window $h_d \in \{0, \ldots, 255\}$ for each destination d, encoding the outcomes of the last $n_d \leq 8$ transmissions as a packed bitfield. The empirical success rate is $\hat{\rho}_d = \text{popcount}(h_d)/n_d$. When sufficient history exists, the link quality bucket is

$$B_\ell(p) = \begin{cases} 0\ \text{(BAD)} & \hat{\rho}_d < 0.6, \\ 1\ \text{(DECENT)} & 0.6 \leq \hat{\rho}_d < 0.9, \\ 2\ \text{(STRONG)} & \hat{\rho}_d \geq 0.9. \end{cases} \tag{5.3}$$

```
static void record_link_outcome(uint8_t dst_id, bool success) {
    if (dst_id < 1 || dst_id > MAX_NODES) return;
    uint8_t idx = dst_id - 1u;
    uint8_t n   = link_history_n[idx];
    if (n < 8) {
        if (success) link_history[idx] |=  (1u << n);
        else         link_history[idx] &= ~(1u << n);
        link_history_n[idx] = n + 1;
    } else {
        link_history[idx] = (uint8_t)(link_history[idx] >> 1);
        if (success) link_history[idx] |= 0x80u;
    }
}
```

Listing 5.1: Sliding delivery window update (`scheduler.cpp`).

When no delivery history is available for destination d, the scheduler falls back to the routing layer's link cost for the next hop on the path to d, with thresholds aligned to the RSSI-to-cost buckets described in Chapter 4. The initial scheduling estimate thereby degrades gracefully from empirical measurement to route-cost approximation as history accumulates. The firmware implements the sliding window as a single `uint8_t` per destination, updated on each outcome via a shift-and-insert, as shown in Listing 5.1.

The shift-and-insert requires only two operations per update and no division. The success rate $\hat{\rho}_d$ at read time is computed by a popcount loop over the 8-bit value. Total storage cost is one byte per destination (the bitfield) plus one byte per destination (the count n_d), for 40 bytes over 20 destinations, a cost negligible relative to the overall SRAM budget.

Queue depth B_q. Let $q = |Q|$ be the number of packets currently waiting for transmission. The depth bucket is

$$B_q(q) = \begin{cases} 0 \text{ (LOW)} & q \le 2, \\ 1 \text{ (MEDIUM)} & 3 \le q \le 5, \\ 2 \text{ (HIGH)} & q \ge 6. \end{cases} \tag{5.4}$$

A deep queue signals congestion pressure: the node has more pending messages than it can drain quickly, and the risk of packet drops due to a full queue rises. When the queue is deep, the scheduler should favour smaller, higher-priority messages to accelerate draining and reduce peak backlog.

Message priority B_π. Each message is assigned a semantic priority level at injection by the application. Four levels are defined:

$$B_\pi(p) \in \{0 \text{ (LOW)},\ 1 \text{ (MEDIUM)},\ 2 \text{ (HIGH)},\ 3 \text{ (CONTROL)}\}. \tag{5.5}$$

Control packets like route-refresh signals, synchronisation messages and ACK's carry the highest weight and are not subject to demotion by learning. The weight update rule's learning rate $\alpha = 0.10$ is small enough that eroding the initial priority advantage of a CONTROL packet would require a sustained sequence of failures that, in a correctly functioning network, should not occur.

Message age B_a. Let $\tau_p = t - t_{\text{enq}}(p)$ be the elapsed time in seconds since packet p entered the queue at the current node. The age bucket is

$$B_a(p,t) = \begin{cases} 0 \text{ (FRESH)} & \tau_p < 2, \\ 1 \text{ (AGING)} & 2 \le \tau_p < 6, \\ 2 \text{ (OLD)} & \tau_p \ge 6. \end{cases} \tag{5.6}$$

Without an age term, a low-priority packet that arrives when the queue is congested could wait indefinitely behind a continuous stream of higher-scoring arrivals. The age bonus is a starvation safeguard: any packet that has waited six or more seconds receives the maximum age credit regardless of its priority or link quality and must be served within a bounded number of additional arrivals.

Message size B_z. In LoRa's spreading-factor-based modulation, on-air duration scales approximately linearly with payload length; a 512-byte message may occupy the channel for several seconds at the spreading factors used in long-range configurations. Larger packets therefore inflict a larger opportunity cost on all other queue members. With size s_p in bytes,

$$B_z(p) = \begin{cases} 0 \text{ (SHORT)} & s_p \le 64, \\ 1 \text{ (MEDIUM)} & 65 \le s_p \le 512, \\ 2 \text{ (LONG)} & s_p > 512. \end{cases} \tag{5.7}$$

The size penalty is a fairness mechanism: it prevents a single large message from monopolising the channel and reduces head-of-line blocking for shorter messages queued behind it.

5.3 Q-Value as a Linear Feature Sum

Standard tabular Q-learning associates a scalar $Q(s,a)$ with every (state, action) pair. Characterising the state of a packet queue precisely would require tracking the exact feature values of every packet in the queue simultaneously, an exponential state space that is entirely infeasible on a microcontroller. Even restricting to the six bucketed features of a single packet, the number of distinct feature combinations is $3 \times 3 \times 3 \times 4 \times 3 \times 3 = 972$. Across a queue of up to 16 packets that may each land in any of these combinations, the implicit joint state space is astronomically large. More critically, a tabular approach would provide no generalisation whatsoever: observing that a packet with bucket

```
// Initial weights (Python FEATURE_SCORES x 100)
static int16_t w_retx[3]  = {   0, -30, -70 };
static int16_t w_link[3]  = { -50,   0,  50 };
static int16_t w_queue[3] = { -70, -30,   0 };
static int16_t w_prio[4]  = {   0,  40,  70, 100 };
static int16_t w_age[3]   = {   0,  30,  70 };
static int16_t w_size[3]  = {   0, -30, -70 };

static int16_t compute_q(const TXEntry* e, uint8_t depth) {
    return w_retx [bucket_retx(e->retransmits)]
         + w_link [bucket_link(e->dst_id)]
         + w_queue[bucket_queue(depth)]
         + w_prio [e->priority]
         + w_age  [bucket_age(millis() - e->enqueue_time_ms)]
         + w_size [bucket_size(e->payload_len)];
}
```

Listing 5.2: Weight vectors and Q-value computation (`scheduler.cpp`).

vector $(0, 2, 1, 3, 0, 0)$ was successfully delivered tells the table nothing about how to score a packet with bucket vector $(0, 2, 1, 2, 0, 0)$, even though the difference is a single priority bucket.

Linear function approximation resolves both problems simultaneously. Each feature is associated with an independent weight vector of size equal to the number of buckets for that feature. The Q-value of a packet is the sum of the weights indexed by the packet's active buckets.

$$Q(\mathbf{f}) = w^{(r)}_{B_r} + w^{(\ell)}_{B_\ell} + w^{(q)}_{B_q} + w^{(\pi)}_{B_\pi} + w^{(a)}_{B_a} + w^{(z)}_{B_z}. \tag{5.8}$$

The total parameter count is $3+3+3+4+3+3 = 19$ scalar weights. Storing each as a 16-bit integer requires exactly 38 bytes. The linear structure means that a weight update following a "bad link" outcome automatically adjusts the score of every future packet that reaches the scheduler via a poor link, regardless of its other features. This generalisation is precisely what allows the scheduler to adapt without storing separate Q-values for every feature combination that has ever been encountered.

The firmware stores each weight as an `int16_t` in fixed-point format scaled by 100, so that the floating-point conceptual value $w = 0.50$ is stored as the integer 50. This allows all Q-value arithmetic to use integer addition with no floating-point instructions (Listing 5.2).

Reading these initial values against (5.8) encodes the design intuition directly. A fresh, first-attempt, high-priority, short packet on a strong link in a high queue scores $0 + 50 + 0 + 70 + 0 + 0 = 120$. The same packet with two retransmissions and a bad link scores $-70 + (-50) + 0 + 70 + 0 + 0 = -50$. The relative ordering of these two extreme cases is sharp, and learning refines the intermediate cases to match the actual observed network behavior.

Note that the queue depth `depth` is passed into `compute_q` as an argument rather than read from a global, because during a dequeue pass the queue depth is constant: the pass is evaluating all candidates simultaneously before removing

any of them. All candidates are therefore scored against the same queue-depth context, and the relative ordering among them is determined entirely by their individual features.

Table 5.1 summarises the initial weight values and their interpretations.

Feature	Bucket 0	Bucket 1	Bucket 2	Bucket 3
Retransmissions	ZERO (+0)	ONE (−30)	TWO+ (−70)	—
Link quality	BAD (−50)	DECENT (+0)	STRONG (+50)	—
Queue depth	LOW (−70)	MEDIUM (−30)	HIGH (+0)	—
Message age	FRESH (+0)	AGING (+30)	OLD (+70)	—
Message size	SHORT (+0)	MEDIUM (−30)	LONG (−70)	—
Priority	LOW (+0)	MEDIUM (+40)	HIGH (+70)	CTRL (+100)

Table 5.1: Initial Q-weight values (stored as integers scaled by 100). Higher values favor selection. The signs encode the hand-designed intuition: strong links, fresh packets, high priority, and small size are all positive signals; retransmissions, bad links, and large size are negative.

5.4 Epsilon-Greedy Action Selection

With a Q-value defined for every packet in the queue, the selection rule follows the standard epsilon-greedy policy. With probability ε the scheduler selects a packet uniformly at random (*exploration*); with probability $1 - \varepsilon$ it selects the packet with the maximum Q-value (*exploitation*).

$$p^* = \begin{cases} \text{Uniform}(Q) & \text{w.p. } \varepsilon, \\ \arg\max_{p \in Q} \; Q(\mathbf{f}(p, |Q|, t)) & \text{w.p. } 1 - \varepsilon. \end{cases} \tag{5.9}$$

The exploration rate is fixed at $\varepsilon = 0.10$ and is not decayed over time. The decision not to decay ε is deliberate and worth explaining. In most reinforcement learning applications, decaying ε is motivated by the assumption that the environment is stationary: once the agent has learned the optimal policy, further exploration wastes resources. But the environment of a deployed mesh network is emphatically non-stationary. Link qualities fluctuate as nodes move and radio conditions change; new nodes join and depart; the distribution of message priorities shifts over the deployment lifetime. A permanently nonzero ε ensures the scheduler never commits irrevocably to weights learned under a topology that no longer reflects reality, while a rate of 10% keeps the exploration overhead small enough that the greedy arm dominates in steady state.

The firmware uses the C standard library `rand()` for the Bernoulli draw and the uniform candidate selection, shown in Listing 5.3.

```c
static uint8_t select_idx(void) {
    if ((rand() % 100) < EPSILON_PCT) {
        return (uint8_t)(rand() % q_count);   // explore: pick uniformly
    }
    uint8_t  best   = 0;
    int16_t  best_q = -32000;
    for (uint8_t i = 0; i < q_count; i++) {
        int16_t qv = compute_q(&q[i], q_count);
        if (qv > best_q) { best_q = qv; best = i; }
    }
    return best;   // exploit: pick highest Q-value
}
```

Listing 5.3: Epsilon-greedy selection (`scheduler.cpp`).

5.5 Reward Function and Temporal-Difference Update

5.5.1 Reward Function

The immediate reward for a transmission outcome is

$$r = \begin{cases} +100 & \text{success (ACK received)}, \\ -50 & \text{failure (ACK timeout)}, \end{cases} \quad - \; 5 \cdot \min(\lfloor \tau_a \rfloor,\ 60), \tag{5.10}$$

where τ_a is the age of the packet in seconds at the time the outcome is observed, clipped at 60 seconds. The age penalty in the reward is as important as the base signal. Without it, the scheduler would be indifferent between a packet delivered quickly and one delivered slowly after many retransmissions, as, after all, they both earn +100. With the age penalty, a message that finally succeeds on its third attempt after sitting in the queue for 20 seconds earns $100 - 5 \times 20 = 0$, the same as a timeout. This encodes the operational reality that a message delivered too slowly may have no value in a time-critical emergency context.

5.5.2 Temporal-Difference Error

Let $Q_{\text{cur}} = Q(\mathbf{f}_{\text{sel}})$ denote the Q-value of the selected packet, computed from the feature vector $\mathbf{f}_{\text{sel}}$ captured at the moment of selection. After the outcome is observed, let Q' be the queue state with the dequeued packet removed, and define

$$Q_{\text{next}} = \begin{cases} \max\limits_{p \in Q'} Q(\mathbf{f}(p, |Q'|, t)) & |Q'| > 0, \\ 0 & |Q'| = 0. \end{cases} \tag{5.11}$$

The one-step TD error is

$$\delta = r + \gamma \cdot Q_{\text{next}} - Q_{\text{cur}}, \tag{5.12}$$

```
static void td_update(const PendingUpdate* pu, bool success) {
    uint32_t age_s = (millis() - pu->enqueue_ms) / 1000UL;
    if (age_s > 60) age_s = 60;

    int16_t reward = success ? 100 : -50;
    reward -= (int16_t)(5 * age_s);

    int16_t cur_q = w_retx [pu->b_retx]  + w_link [pu->b_link]
                  + w_queue[pu->b_queue] + w_prio [pu->b_prio]
                  + w_age  [pu->b_age]   + w_size [pu->b_size];

    int16_t next_q = 0;
    if (q_count > 0) {
        next_q = -32000;
        for (uint8_t i = 0; i < q_count; i++) {
            int16_t qv = compute_q(&q[i], q_count);
            if (qv > next_q) next_q = qv;
        }
    }

    int32_t td    = (int32_t)reward
                  + ((int32_t)GAMMA * next_q / 100L)
                  - (int32_t)cur_q;
    int16_t delta = (int16_t)((int32_t)ALPHA * td / 100L);
    if (delta >  50) delta =  50;
    if (delta < -50) delta = -50;
```

Listing 5.4: TD update: reward and error computation (`scheduler.cpp`).

with discount factor $\gamma = 0.90$. The weight update applies δ uniformly to each active feature bucket of the selected packet.

$$w_k^{(f)} \leftarrow \text{clip}\Big(w_k^{(f)} + \text{clip}(\alpha\delta,\ -50,\ 50),\ -1000,\ 1000\Big), \tag{5.13}$$

for each feature f with active bucket k, and with learning rate $\alpha = 0.10$. The inner clip on the weight delta limits any single update to 50 units in the fixed-point integer scale (i.e. 0.50 in floating-point), preventing a single extreme outcome from destabilising the weight table. The outer clip prevents indefinite weight growth under sustained success or failure in a particular bucket. All intermediate arithmetic is promoted to 32 bits to prevent overflow on the product $\gamma \cdot Q_{\text{next}}$, which can reach $0.90 \times 1000 \times 6 = 5400$ in the scaled representation, well exceeding the 16-bit signed range. The reward and error computation is shown in Listing 5.4, and the application of the delta to each active bucket in Listing 5.5.

5.5.3 Interpreting the Update Semantically

The mathematics of the TD update can obscure a fairly intuitive mechanism, and it is worth dwelling on what actually happens to the weights under representative outcomes. Consider a scheduler that selects a fresh, first-attempt, high-priority packet on a strong link with a shallow queue. The initial Q-value of this packet is $0+50+(-70)+70+0+0 = 50$. Suppose the packet is delivered successfully within one second. The reward is $100 - 5 \times 1 = 95$. If the queue is empty

```
    w_retx [pu->b_retx]  += delta; clamp_weight(&w_retx [pu->b_retx]);
    w_link [pu->b_link]  += delta; clamp_weight(&w_link [pu->b_link]);
    w_queue[pu->b_queue] += delta; clamp_weight(&w_queue[pu->b_queue]);
    w_prio [pu->b_prio]  += delta; clamp_weight(&w_prio [pu->b_prio]);
    w_age  [pu->b_age]   += delta; clamp_weight(&w_age  [pu->b_age]);
    w_size [pu->b_size]  += delta; clamp_weight(&w_size [pu->b_size]);
}
```

Listing 5.5: TD update: applying the delta to active buckets (`scheduler.cpp`).

after dequeue, $Q_{\text{next}} = 0$ and $\delta = 95 + 0 - 50 = 45$. The per-weight delta is $\text{clip}(0.10 \times 45, -50, 50) = 4.5$, which the integer arithmetic truncates to 4. Each of the six active-bucket weights (`ZERO`, `STRONG`, `LOW`-queue, `HIGH` priority, `FRESH`, `SHORT`) increases by 4 units. The Q-value of similar packets rises, making the greedy arm more likely to select them in future.

Now consider the same packet type transmitted over a bad link that times out. The reward is $-50-5\times$(timeout age), which could easily reach -100 for an aged packet. With $Q_{\text{cur}} = 50$ and $Q_{\text{next}} \approx 50$ (similar packets remain in the queue), $\delta = -100 + 0.9 \times 50 - 50 \approx -105$. The delta is clipped to -50, and each of the active buckets decreases by 5 units. Repeated failures on supposedly strong links gradually erode the link-quality advantage, driving the scheduler to rely more on priority and age as the dominant signals when link quality is unreliable.

5.6 The Pending Update and Retransmission Mechanics

A dequeue event does not immediately trigger a weight update, because the outcome of the transmission is not yet known. The scheduler captures the feature vector at dequeue time and stores it in a `PendingUpdate` struct that persists until the ACK arrives or the timeout fires. This struct is the sole record bridging the selection decision and the learning event (Listing 5.6).

Only one pending update may be active at a time. The firmware's main loop enforces this by checking `scheduler_pending()` before any dequeue attempt: a second packet is not sent until the pending update for the first is resolved. When a timeout fires without an ACK, `try_requeue()` re-inserts the packet with an incremented retransmission counter and a freshly recorded enqueue timestamp (Listing 5.7).

The fresh enqueue timestamp resets the age clock for the retransmitted copy, so the age bucket reflects how long the packet has waited in this particular attempt rather than accumulating age across all retransmissions indiscriminately. The retransmission counter increment separately encodes the history of prior failures, so both signals remain independently informative in the Q-value computation. On the next dequeue pass, the retransmitted packet competes with all other queue members under its updated feature vector, where the elevated retransmission bucket now depresses its Q-value relative to fresh first-attempt candidates.

When an ACK for destination d arrives, any re-enqueued retransmission of a message to that destination must be

```
typedef struct {
    uint8_t  pkt_id;
    uint8_t  dst_id;
    uint8_t  b_retx, b_link, b_queue, b_prio, b_age, b_size;
    uint32_t enqueue_ms;
    bool     valid;
} PendingUpdate;

bool scheduler_dequeue(TXEntry* out) {
    if (q_count == 0) return false;
    uint8_t idx = select_idx();

    pending.dst_id     = q[idx].dst_id;
    pending.b_retx     = bucket_retx (q[idx].retransmits);
    pending.b_link     = bucket_link (q[idx].dst_id);
    pending.b_queue    = bucket_queue(q_count);
    pending.b_prio     = (uint8_t)q[idx].priority;
    pending.b_age      = bucket_age  (millis() - q[idx].enqueue_time_ms);
    pending.b_size     = bucket_size (q[idx].payload_len);
    pending.enqueue_ms = q[idx].enqueue_time_ms;
    pending.valid      = true;

    *out            = q[idx];
    last_sent       = q[idx];
    last_sent_valid = true;

    for (uint8_t i = idx; i < q_count - 1u; i++)
        q[i] = q[i + 1u];
    q_count--;
    return true;
}
```

Listing 5.6: Pending update record and dequeue with feature capture (`scheduler.cpp`).

```
static bool try_requeue(void) {
    if (!last_sent_valid)                          return false;
    if (last_sent.retransmits >= MAX_RETRANSMITS) { last_sent_valid = false; return
        false; }
    if (q_count >= QUEUE_SIZE)                    { last_sent_valid = false; return
        false; }
    TXEntry* e         = &q[q_count++];
    *e                 = last_sent;
    e->retransmits++;
    e->enqueue_time_ms = millis();    // reset age clock for this attempt
    last_sent_valid    = false;
    return true;
}
```

Listing 5.7: Re-enqueue on retransmission (`scheduler.cpp`).

```
void scheduler_cancel_retransmit_dst(uint8_t dst_id) {
    if (last_sent_valid && last_sent.dst_id == dst_id)
        last_sent_valid = false;

    for (uint8_t i = 0; i < q_count; ) {
        if (q[i].retransmits > 0 && q[i].dst_id == dst_id) {
            for (uint8_t j = i; j < q_count - 1u; j++)
                q[j] = q[j + 1u];
            q_count--;
        } else { i++; }
    }

    if (pending.valid && pending.dst_id == dst_id)
        pending.valid = false;
}
```

Listing 5.8: Three-location ACK cancellation sweep (`scheduler.cpp`).

purged across all three locations where it could reside, as shown in Listing 5.8.

The three-location sweep handles the inherent race between the ACK arrival and the retransmission pipeline. A delayed ACK may arrive after the timeout has already fired and re-enqueued the packet; without the queue scan in case 2, the retransmitted copy would be transmitted a second time, causing a duplicate delivery at the destination.

Chapter 6

Developing a Network Simulator

6.1 Goals and Scope

Testing a multi-node mesh protocol on physical hardware is expensive and difficult to control. Each experiment requires programming multiple microcontrollers, arranging them in space, coordinating timed message sequences, and interpreting results that are confounded by radio noise, physical movement, and the inherently stochastic behaviour of the LoRa channel. More fundamentally, when a packet fails to arrive, there is no easy way to determine whether the cause was a protocol error, a transient channel impairment, or a hardware fault: the physical variables are inseparable in a real deployment.

A well-constructed simulator separates these concerns by implementing the scheduling and protocol logic in a fully observable, deterministic environment where the same packet trace can be replayed identically under different scheduling disciplines. It serves two complementary purposes in the HALoN development process. The first is protocol validation: verifying that the Python implementation of the Q-learning scheduler produces weight trajectories matching the firmware when given the same packet sequence, confirming that the two codebases are isomorphic and that any behavioural difference observed during physical deployment is a hardware or radio effect rather than a logic error in either implementation. The second is comparative evaluation: measuring the quantitative performance difference between the Q-learning scheduler and classical disciplines under controlled, identical conditions, providing empirical grounding for the design argument that learned scheduling is worth its implementation complexity.

The simulator, `network_simulator_ls.py`, is an interactive Python application built on Matplotlib and NetworkX. It supports a configurable multi-node topology with real-time drag-and-drop node repositioning, three scheduling modes selectable via radio buttons, a slider-driven packet injection rate and node count, and a headless comparison mode that runs all three disciplines against an identical packet trace and renders a six-panel statistical comparison.

6.2 Topology and Routing Model

6.2.1 Geometric Connectivity

The simulator represents network nodes as points in a two-dimensional coordinate space. Two nodes u and v are connected if and only if their Euclidean distance satisfies

$$d(u, v) = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} \leq D_{\max},$$

where $D_{\max}$ = `MAX_LINK_DIST` is a configurable threshold. The default value of $D_{\max} = 3.2$ was selected empirically: with the default five-node layout, values below approximately 3.0 produce a disconnected graph that splits into two islands, which is an uninteresting degenerate case; values much above 4.0 produce a fully connected graph in which every node can reach every other directly, eliminating the multi-hop routing problem that the simulator is designed to study.

This geometric model is a deliberate simplification of the RSSI-based cost function used in the firmware. In the simulator, all active links are treated as equal-cost by the routing layer, and Dijkstra's algorithm minimises hop count rather than the weighted sum of RSSI-derived costs defined in equation (4.1). This is intentional: it isolates scheduling behaviour from routing artefacts, ensuring that a latency difference between two scheduling disciplines reflects the scheduling policy and not an incidental difference in route selection.

Routing tables are recomputed from scratch via NetworkX's `single_source_shortest_path` on every topology change, as shown in Listing 6.1.

The routing table for each node n maps each destination d to the first-hop neighbour on the shortest path, or `None` if d is unreachable. This is exactly the same semantic as the firmware's `next_hop` table.

6.3 The Python Q-Learner

The Python `QLearner` class is a deliberate line-by-line translation of the C scheduler logic into Python, with identical bucket thresholds, identical initial weights, and an identical TD update rule. The bucket enumeration and bucketing functions mirror the firmware (Listing 6.2):

The initial weight dictionary, shown in Listing 6.3, matches the firmware's integer values after dividing by 100.

The Q-value computation, epsilon-greedy selection, and TD update are shown in Listing 6.4.

This is isomorphic to the C `td_update` function in all structurally important respects: the same formula for δ (equation (5.12)), the same learning rate $\alpha = 0.10$ and discount $\gamma = 0.90$, the same outer weight clamp (to $[-1.0, 1.0]$ in floating-point, equivalent to $[-100, 100]$ in the $\times 100$ integer scale). The Python version omits the firmware's inner clip on the weight delta (clip($\alpha\delta, -50, 50$)) because Python floats do not overflow and the inner clip is a concession to fixed-point arithmetic rather than an algorithmic requirement. This difference makes the Python weights slightly less

```
def update_topology():
    global links
    G = nx.Graph()
    G.add_nodes_from(nodes.keys())
    links = []
    for i in nodes:
        for j in nodes:
            if i >= j: continue
            d = ((nodes[i][0]-nodes[j][0])**2 +
                 (nodes[i][1]-nodes[j][1])**2)**0.5
            if d <= MAX_LINK_DIST:
                links.append((i, j))
                G.add_edge(i, j, weight=d)
    for n in nodes:
        try:
            paths = nx.single_source_shortest_path(G, n)
            routing_tables[n] = {
                d: (p[1] if len(p) > 1 else None)
                for d, p in paths.items()
            }
        except Exception:
            routing_tables[n] = {d: None for d in nodes}
```

Listing 6.1: Geometric topology and shortest-path routing update (`network_simulator_ls.py`).

```
def retransmission_bucket(r):
    if r == 0: return Retransmission.ZERO
    if r == 1: return Retransmission.ONE
    return Retransmission.TWO_PLUS

def link_quality_bucket(history):
    if not history: return LinkQuality.DECENT   # no data: assume decent
    rate = sum(history[-10:]) / len(history[-10:])
    if rate < 0.6: return LinkQuality.BAD
    if rate < 0.9: return LinkQuality.DECENT
    return LinkQuality.STRONG

def queue_bucket(q):
    if q <= 2: return QueueLevel.LOW
    if q <= 5: return QueueLevel.MEDIUM
    return QueueLevel.HIGH

def age_bucket(age):         # age in seconds
    if age < 2: return Age.FRESH
    if age < 6: return Age.AGING
    return Age.OLD

def size_bucket(s):          # size in bytes
    if s <= 64:  return Size.SHORT
    if s <= 512: return Size.MEDIUM
    return Size.LONG
```

Listing 6.2: Feature bucketing functions in Python (`network_simulator_ls.py`).

```
INITIAL_WEIGHTS = {
    'retransmission': {
        Retransmission.ZERO:      0.00,
        Retransmission.ONE:      -0.30,
        Retransmission.TWO_PLUS: -0.70,
    },
    'link': {
        LinkQuality.BAD:    -0.50,
        LinkQuality.DECENT:  0.00,
        LinkQuality.STRONG:  0.50,
    },
    'queue': {
        QueueLevel.LOW:    -0.70,
        QueueLevel.MEDIUM: -0.30,
        QueueLevel.HIGH:    0.00,
    },
    'priority': {
        Priority.LOW:      0.00,
        Priority.MEDIUM:   0.40,
        Priority.HIGH:     0.70,
        Priority.CONTROL:  1.00,
    },
    'age':  { Age.FRESH:  0.00, Age.AGING:  0.30, Age.OLD:   0.70 },
    'size': { Size.SHORT: 0.00, Size.MEDIUM:-0.30, Size.LONG:-0.70 },
}
```

Listing 6.3: Initial weight dictionary (`network_simulator_ls.py`).

```
class QLearner:
    def __init__(self, alpha=0.10, gamma=0.90, epsilon=0.10):
        self.alpha, self.gamma, self.epsilon = alpha, gamma, epsilon
        self.weights = {k: dict(v) for k, v in INITIAL_WEIGHTS.items()}

    def q_value(self, features):
        return sum(self.weights[k][v] for k, v in features.items())

    def select_packet(self, queue, queue_len, st):
        if random.random() < self.epsilon:
            return random.choice(queue)
        return max(queue,
                   key=lambda p: self.q_value(
                       encode_features(p, queue_len, st)))

    def update(self, features, reward, next_queue, st):
        next_q = 0.0
        if next_queue:
            next_q = max(
                self.q_value(encode_features(p, len(next_queue), st))
                for p in next_queue)
        td = reward + self.gamma * next_q - self.q_value(features)
        for k, v in features.items():
            self.weights[k][v] = max(-1.0, min(1.0,
                self.weights[k][v] + self.alpha * td))
```

Listing 6.4: QLearner class: Q-value, selection, and update (`network_simulator_ls.py`).

```
def move(self, dt):
    if self.state == 'QUEUED':
        self.wait_time += dt          # accumulate queue wait
        return
    tx, ty = nodes[self.target_node]
    dx, dy = tx - self.pos[0], ty - self.pos[1]
    dist   = (dx**2 + dy**2)**0.5
    if dist < PACKET_SPEED:
        # Arrived at target node.
        self.pos          = [tx, ty]
        self.current_node = self.target_node
        if self.current_node == self.dest:
            self.arrived         = True
            self.state           = 'ARRIVED'
            self.total_trip_time = sim_time - self.injection_time
            self.success_history.append(1)
            completed_stats.append(self)
        else:
            self.state = 'QUEUED'
            node_queues[self.current_node].append(self)
            self._set_next_hop()
    else:
        self.pos[0] += (dx / dist) * PACKET_SPEED
        self.pos[1] += (dy / dist) * PACKET_SPEED
```

Listing 6.5: Per-frame packet movement and hop completion (`network_simulator_ls.py`).

conservative in their step sizes, a known and documented deviation.

6.4 Packet Motion and Node Queue Dynamics

In-flight packets are represented by `MessagePacket` objects that track position, state, accumulated wait time, total trip time, retransmission count, and the sliding success history used for link quality bucketing. On each animation frame, the `move()` method advances the packet toward its target node at a constant speed `PACKET_SPEED` per frame, as shown in Listing 6.5.

The re-enqueue at an intermediate node faithfully models the store-and-forward behavior of the firmware: a packet can only depart a relay node when that node's local scheduler selects it, not immediately upon arrival. This means that queue depth and waiting time are realistic per-hop quantities, not just end-to-end averages.

The per-node scheduling runs once per frame for each node (Listing 6.6).

The `node_busy_until` counter models channel occupancy time: larger packets occupy the node for more frames before the next transmission is permitted. The formula `max(20, size // 10)` gives small packets a minimum of 20 frames and scales linearly above 200 bytes, approximating LoRa's spreading-factor-dependent air-time behaviour without the full physical layer model.

```
def process_node_queues(frame):
    for nid in list(nodes.keys()):
        if node_busy_until.get(nid, 0) <= frame:
            q = node_queues.get(nid, [])
            if not q: continue
            if scheduling_mode == 'SJF':
                q.sort(key=lambda p: p.size)
                pkt = q.pop(0)
            elif scheduling_mode == 'Q-Learning':
                pkt      = ql_agent.select_packet(q, len(q), sim_time)
                features = encode_features(pkt, len(q), sim_time)
                q.remove(pkt)
                # Reward includes age penalty matching equation (3.2).
                reward   = 1.0 - 0.05 * (sim_time - pkt.injection_time)
                ql_agent.update(features, reward, q, sim_time)
            else:                              # FCFS
                pkt = q.pop(0)
            pkt.state            = 'MOVING'
            # Channel occupancy time proportional to packet size.
            node_busy_until[nid] = frame + max(20, pkt.size // 10)
```

Listing 6.6: Node queue draining under three scheduling modes (`network_simulator_ls.py`).

```
return {
    'total_injected': n_inj,
    'total_arrived' : n_arr,
    'delivery_rate' : n_arr / n_inj if n_inj else 0,
    'avg_wait'      : sum(p.wait_time        for p in local_done) / n_arr,
    'avg_latency'   : sum(p.total_trip_time  for p in local_done) / n_arr,
    'max_latency'   : max(p.total_trip_time  for p in local_done, default=0),
    'min_latency'   : min(p.total_trip_time  for p in local_done, default=0),
    'throughput'    : n_arr / local_t,
    'sim_time'      : local_t,
    'per_packet'    : pp,          # per-packet latency and wait records
}
```

Listing 6.7: Headless simulation return structure (`network_simulator_ls.py`).

6.5 Headless Simulation and Comparative Evaluation

Quantitative comparison between scheduling disciplines requires running all three against the *identical* packet trace. The `run_headless(algo_name)` function executes the full simulation loop without rendering, using a saved copy of the packet template as input. The function terminates when all packets have been delivered or the frame limit is reached, and returns the statistics dictionary shown in Listing 6.7.

The "Compare All" button triggers `run_headless` for FCFS, SJF, and Q-Learning in sequence on the same saved template, then renders a six-panel comparison window with bar charts for average latency, average queue wait, throughput, and delivery rate; a per-packet latency trace showing how each algorithm handles individual messages; and a latency range plot showing the maximum and minimum per algorithm.

6.6 Validation Observations

The simulator was used for two concrete validation tasks. First, weight trajectories after a fixed 20-packet sequence were compared between the Python simulator's `ql_agent.weights` and the firmware's `weights` serial command output after injecting the same sequence via the `test` command. This comparison uncovered a bug in an early version of the firmware's `td_update`: the intermediate product `GAMMA * next_q` was computed in 16-bit arithmetic, which overflowed on deep queues with high-Q-value candidates. The Python simulator produced the correct weight trajectory (because Python floats never overflow) while the firmware diverged, making the bug immediately visible. The fix restored agreement between the two implementations.

Second, the comparative evaluation consistently showed that the Q-learning scheduler achieves a better balance between average latency and latency variance than either FCFS or SJF on mixed-priority traces. FCFS exhibits the highest average latency for high-priority packets because they are not promoted above earlier-arriving low-priority ones. SJF achieves the lowest overall average latency at the cost of the highest variance and starvation of large packets under sustained congestion. The Q-learning scheduler's latency distribution is intermediate in mean but significantly tighter in spread: the age bonus prevents indefinite starvation, the link-quality signal suppresses futile transmissions on failing paths, and the priority field ensures that control and high-priority messages are served promptly even when the queue is deep. The relative advantage of Q-learning over FCFS grows with packet diversity: for homogeneous queues (all same priority, same size), all three algorithms converge to similar performance, which is the expected result and serves as a sanity check on the implementation.

Chapter 7

Developing a Reconfigurable Feed Network

The Adafruit Feather 32u4 boards come with the RFM95 radio chip built in, enabling it to communicate with other nodes. This is suitable for most applications, but HALoN requires communicating effectively with users on the ground and other nodes in the sky. Sending power downwards like this is inefficient and will require many retransmissions and consume more power, putting strain on an already power-limited system. The solution is a feed network that toggles between two different states to direct power in the direction of whomever the system is trying to communicate with. A radiation pattern is a representation of how an antenna or source emits or receives electromagnetic energy in space, and each state will have its own. For nodes in the sky, in order to communicate to the ground user they will use an endfire radiation pattern, and to communicate with other nodes in the sky they will use a broadside radiation pattern.

7.1 Microstrip Sizing

The feed network would be on a printed circuit board (PCB), and the traces would be microstrip. At radio frequencies (RF), signals travel as electromagnetic waves rather than instant electrical currents. Thus, transmission lines are used to guide these waves, manage impedance, and prevent destructive reflections that cause power loss and signal distortion. On a PCB microstrip transmission lines are commonly used, and they have design equations [19] to determine important parameters. For the characteristic impedance there is the following equation

$$Z_0 = \begin{cases} \dfrac{60}{\sqrt{\varepsilon_e}} \ln\left(\dfrac{8d}{W} + \dfrac{W}{4d}\right) & \text{for } W/d \leq 1 \\ \dfrac{120\pi}{\sqrt{\varepsilon_e}\,[W/d + 1.393 + 0.667 \ln(W/d + 1.444)]} & \text{for } W/d \geq 1 \end{cases} \tag{7.1}$$

In this equation $\frac{W}{d}$ is the width to height ratio of the copper trace on the substrate, and that can be found using

$$\frac{W}{d} = \begin{cases} \dfrac{8e^A}{e^{2A}-2} & \text{for } W/d < 2 \\ \dfrac{2}{\pi}\left[B - 1 - \ln(2B-1) + \dfrac{\varepsilon_r - 1}{2\varepsilon_r}\left\{\ln(B-1) + 0.39 - \dfrac{0.61}{\varepsilon_r}\right\}\right] & \text{for } W/d > 2 \end{cases} \tag{7.2}$$

where ϵ_r is the relative permittivity of the PCB substrate and A and B are defined as follows

$$A = \frac{Z_0}{60}\sqrt{\frac{\epsilon_r + 1}{2}} + \frac{\epsilon_r - 1}{\epsilon_r + 1}\left(0.23 + \frac{0.11}{\epsilon_r}\right) \tag{7.3}$$

$$B = \frac{377\pi}{2Z_0\sqrt{\epsilon_r}}. \tag{7.4}$$

Plugging in our material parameters of the PCB substrate, FR-4 Epoxy with $\epsilon_r \approx 4.4$, we get the values in Table 7.1. The 70 Ω sizing is there for any transitions from 50 Ω to 100 Ω.

Impedance	Sizing (mm)
50 Ω	2.25
100 Ω	0.66
70 Ω	1.57

Table 7.1: Impedances and Corresponding Trace Widths

7.2 Endfire Pattern

7.2.1 Power Divider

To produce an endfire radiation pattern, two signals of equal amplitude but 180° phase difference must be fed to the antenna elements such that they constructively interfere in the desired direction. The input signal must therefore be divided in half, with one output subsequently phase shifted by 180°. Two fundamental topologies exist for power division in microwave engineering: the T-junction and the Wilkinson power divider [19]. The T-junction is straightforward to implement but offers poor isolation between output ports and presents impedance matching challenges. The Wilkinson power divider provides superior port isolation and a better impedance match, but at the cost of greater design complexity.

The initial design employed a Wilkinson power divider, as shown in Fig. 7.1. However, the quarter-wave arms of the Wilkinson divider must each be $\frac{\lambda}{4}$ in length. At the operating frequency of 900 MHz, the free-space wavelength is approximately 33 cm, placing each arm at roughly 8 cm resulting in a PCB footprint too large to be practical. The T-junction was therefore selected as the power divider topology for this design.

For our design, the T-junction was fairly trivial to implement, a 50Ω trace splitting into two 100Ω traces, as seen in Fig. 7.2. S-parameters, discussed in detail in Chapter 9, were used to validate the design in simulation. For a two-way

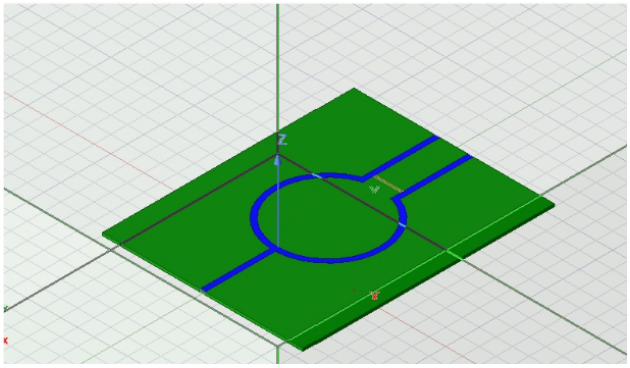


Figure 7.1: Wilkinson power divider CAD in HFSS

power divider, the ideal transmission coefficients $|S_{21}|$ and $|S_{31}|$ are each -3 dB, corresponding to a 50% power split. Simulation results yielded -3.6 dB at port 2 and -3.5 dB at port 3, which is sufficient for this application.

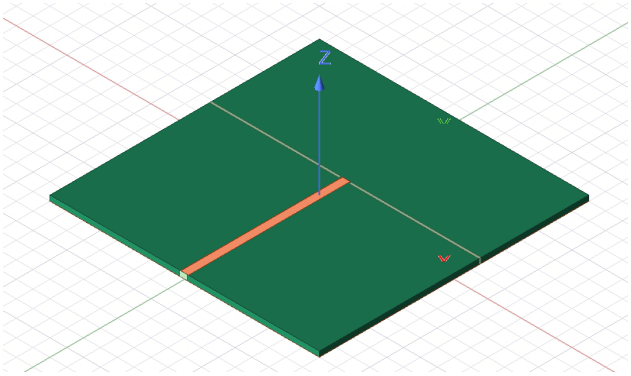


Figure 7.2: T-junction power divider CAD in HFSS

7.2.2 Phase Shifter

With power divided into two transmission lines, one of them now has to incur a 180° phase shift. At high frequency, adding length to transmission lines effects the phase of the signal, because the wavelength of the signal become comparable to the physical length of the line [19]. Specifically, it is a well-known result that to change phase by 180°, line of length $\frac{\lambda}{2}$ must be added. At our design frequency, a $\frac{\lambda}{2}$ is equivalent to about 8 cm, which would make the PCB very large if it was a straight line. A common design technique is employed in such scenarios, meandering the line allows the same physical length to be added but in less space as seen in Fig. 7.3.

This phase shift must be relative to the other output line of the power divider. With knowledge of the design, it was then onto simulation. In order to fine-tune the phase shift, the power divider and phase shift topologies were combined and trace lengths were parameterized. This allowed for parametric sweeps of the length to find the exact length that would yield the desired phase shift between both outputs. The CAD to simulate this is shown in Fig 7.4.

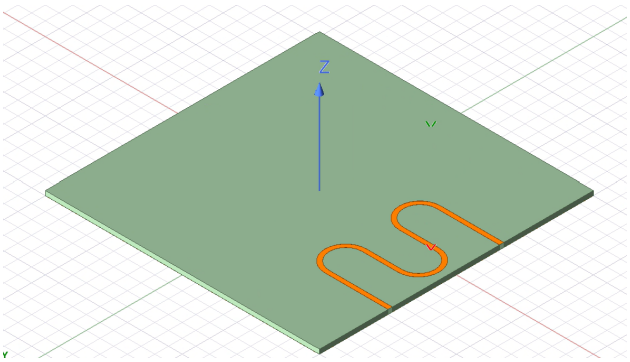


Figure 7.3: Meandered microstrip line CAD in HFSS

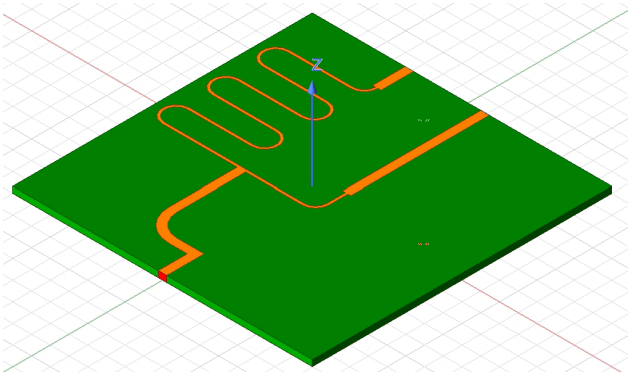


Figure 7.4: Endfire topology CAD in HFSS

7.3 Broadside

Broadside radiation does not require any constructive interference, and thus to implement it on microstrip it is simply a transmission line to guide the wave from the input to the output, seen in Fig. 7.5. Simulating this was straightforward and to evaluate it, the S_{21} parameter was measured.

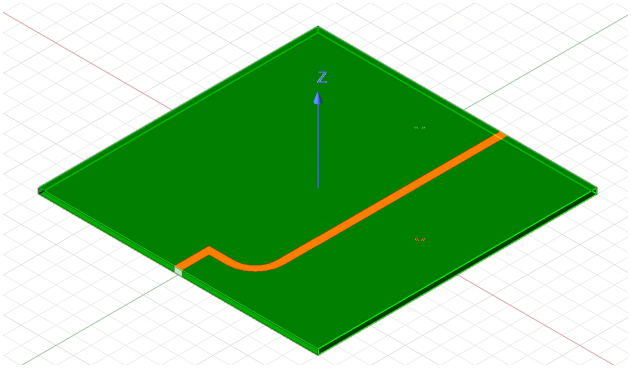


Figure 7.5: Broadside topology CAD in HFSS

7.4 PCB Layout

Once the simulation of the topologies completed, it was onto adding the extra components to make this a functional device. To interface our signal with the traces SMA ports were necessary; to switch between states a RF switch integrated circuit (IC) was required; and to determine the status of the board LEDs were essential. In order to combine the trace layout and these additional components, KiCad was utilized, as seen in Fig. 7.6. The main difference between the PCB and the simulated CAD is the switch and the SMA connectors. Simulating the switch accurately would be very time consuming, and although beneficial, outside of the scope of such a project.

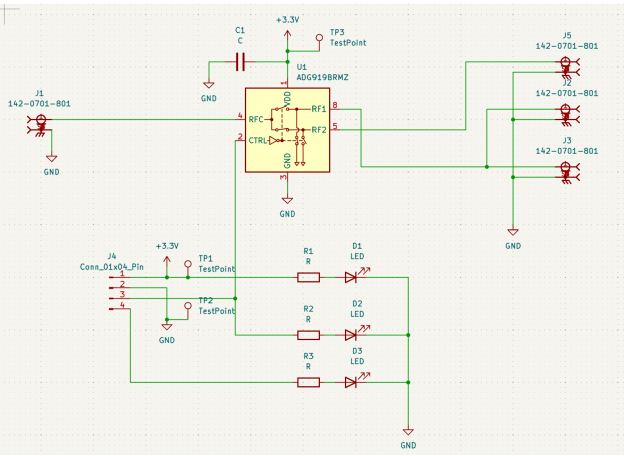


Figure 7.6: Electric schematic in KiCad

Additionally, the SMA connectors were not directly modeled to include their shape, but HFSS has a way to model the input signal from such a connector, and this was used in the simulation. The components were all laid out and the traces were imported from HFSS to be the proper size. Once everything passed our design review process, the board was ordered through PCBWay. The final board is in Fig. 7.7.

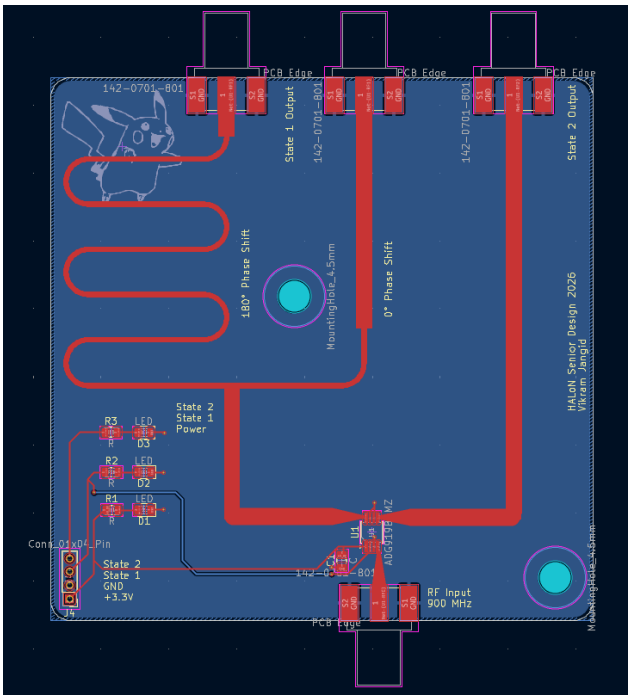


Figure 7.7: PCB layout in KiCad

Chapter 8

Firmware Integration

8.1 System Architecture

The HALoN firmware runs on an Adafruit Feather 32u4 microcontroller paired with an RFM95 LoRa radio transceiver operating at 915 MHz. The firmware is organised as a collection of loosely coupled C modules: `packet`, `neighbor`, `lsa`, `routing`, `scheduler`, `msglog`, `metrics`, `dashboard`, and `testmode`. Each module owns its own static state, exposes a small public API, and makes no assumptions about how other modules manage their memory. A module can be exercised in isolation either through the firmware's serial command interface or by calling its API directly in a test harness. The simulator mirrors this structure by implementing the scheduler and routing logic independently of the radio and hardware abstractions, so that a discrepancy between simulator and firmware behaviour pinpoints which module contains the inconsistency.

The main firmware file `halon_firmware.ino` acts as an orchestrator: it holds no state of its own beyond timing counters and LED timers, and it calls module APIs in a carefully ordered sequence on every iteration of the Arduino `loop()` function. The ordering is not arbitrary; it reflects data dependencies between components and the timing requirements of the half-duplex radio driver.

8.2 Packet Wire Format and Serialisation Safety

All inter-node messages share a fixed 8-byte header followed by a variable-length payload:

Offset	0	1	2	3	4	5	6	7
Field	`src_id`	`dst_id`	`next_hop`	`prev_hop`	`pkt_id`	`ttl`	`type`	`payload_len`

The `next_hop` field carries the 1-indexed ID of the immediate next relay. This means each forwarding node replaces `next_hop` with the result of its own `routing_next_hop(dst_id)` call, routing the packet one hop at a time without requiring the original sender to know the full path. The `prev_hop` field records the most recent transmitting node, used as a fallback for ACK routing when the return path through the routing table is not yet available (see Section 8.5).

```
bool pkt_deserialize(const uint8_t* buf, uint8_t len, Packet* out) {
    if (len < PKT_HEADER_LEN) return false;

    out->src_id      = buf[0];  out->dst_id      = buf[1];
    out->next_hop    = buf[2];  out->prev_hop    = buf[3];
    out->pkt_id      = buf[4];  out->ttl         = buf[5];
    out->type        = buf[6];  out->payload_len = buf[7];

    uint8_t claimed = out->payload_len;

    if (claimed > (uint8_t)(MAX_PAYLOAD + 8))     return false;
    if ((uint16_t)PKT_HEADER_LEN + claimed > len) return false;

    if (claimed > 0)
        memcpy(out->payload, &buf[PKT_HEADER_LEN], claimed);
    return true;
}
```

Listing 8.1: Deserialisation with ordered bounds validation (`packet.cpp`).

Deserialisation includes a two-stage bounds check that prevents a malformed or corrupted packet from causing a buffer overrun. The check order is important: the first test must confirm that the claimed payload length is not absurdly large before promoting the sum to a wider integer type in the second test, as shown in Listing 8.1.

If the first check were omitted and `claimed` were, say, 250 with `PKT_HEADER_LEN` = 8, the expression `(uint8_t)(8 + 250)` would wrap to 2, which is less than any reasonable `len`, causing the second check to pass spuriously and triggering a `memcpy` of 250 bytes into a buffer of only `MAX_PAYLOAD` bytes. The two-stage check is therefore not redundant; each stage protects against a distinct class of malformed input.

8.3 Main Loop Structure and Task Ordering

The `loop()` function is non-blocking and executes continuously. Each iteration is a linear sequence of conditional actions, with millisecond-resolution timing gates controlling periodic tasks. The receive and topology-maintenance phase is shown in Listing 8.2, and the transmission phase in Listing 8.3.

The ordering of these tasks is the product of reasoning about data dependencies and timing constraints. Receive always runs first so that ACKs, which reset the ACK timeout timer and free the scheduler to dequeue the next packet, are processed as quickly as possible after arrival. Topology maintenance (neighbour expiry, LSA expiry, route recomputation in steps 2 and 3) follows immediately, ensuring that any route table update caused by a newly received LSA is complete before the data transmission step (step 10) queries `routing_next_hop`. Only after the routing state is consistent does the firmware attempt to transmit. This sequencing ensures the next hop selected for a DATA packet reflects the current best-known topology rather than a stale table from before the most recent LSA arrived.

The guard condition `!scheduler_pending()` in step 10 is particularly important. The scheduler enforces at

```
void loop() {
    uint32_t now = millis();

    handle_rx();

    { uint8_t expired[MAX_NEIGHBORS];
      uint8_t n = neighbor_expire(expired);
      if (n > 0) {
          for (uint8_t i = 0; i < n; i++)
              routing_set_link(NODE_ID, expired[i], INF_COST);
          routing_compute();
          lsa_set_send_flag();
      }
    }

    if (lsa_expire_origins() > 0) routing_compute();

    outbox_expire();
    metrics_expire();

    if (scheduler_check_timeout())
        Serial.println(F(">> RETRANSMIT queued (no ACK)"));

    if (now - last_hello_ms >= HELLO_INTERVAL)
        { send_hello(); last_hello_ms = now; }

    if (lsa_needs_send() ||
        (now - last_lsa_ms) >= LSA_INTERVAL) {
        send_lsa(); lsa_clear_send_flag(); last_lsa_ms = now;
        Packet relay; relay.pkt_id = next_pkt_id();
        if (lsa_build_relay(&relay))
            { relay.prev_hop = NODE_ID; radio_send(&relay); }
    }

    testmode_tick();
```

Listing 8.2: Main loop orchestration (`halon_firmware.ino`), condensed: receive and topology maintenance.

```
    if (!scheduler_empty() && !scheduler_pending()
        && (now - last_tx_ms) >= TX_COOLDOWN) {
        TXEntry entry;
        if (scheduler_dequeue(&entry)) {
            uint8_t pid = send_data_packet(entry.dst_id, entry.payload,
                entry.payload_len, (uint8_t)entry.priority,
                entry.retransmits);
            if (pid == 0xFF)
                { scheduler_set_tx_id(0); scheduler_record_outcome(0, false); }
            last_tx_ms = now;
        }
    }

    dashboard_tick();
}
```

Listing 8.3: Main loop orchestration (`halon_firmware.ino`), condensed: transmission and dashboard.

```
static uint8_t send_data_packet(uint8_t dst_id,
                                const uint8_t* payload, uint8_t len,
                                uint8_t priority, uint8_t retransmits) {
    uint8_t nh = routing_next_hop(dst_id);
    if (nh == NO_ROUTE) {
        Serial.print(F("!! No route to ")); Serial.println(dst_id);
        return 0xFF;
    }
    Packet p;
    p.src_id      = NODE_ID;   p.dst_id      = dst_id;
    p.next_hop    = nh;        p.prev_hop    = NODE_ID;
    p.pkt_id      = next_pkt_id();
    p.ttl         = TTL_DEFAULT;
    p.type        = PKT_DATA;
    p.payload_len = (len <= MAX_PAYLOAD) ? len : MAX_PAYLOAD;
    for (uint8_t i = 0; i < p.payload_len; i++) p.payload[i] = payload[i];

    outbox_add(p.pkt_id, dst_id, priority, (const char*)payload, p.payload_len);
    outbox_set_tx(p.pkt_id);
    metrics_record_sent(p.pkt_id);
    led_pulse_blue();
    radio_send(&p);
    scheduler_set_tx_id(p.pkt_id);
    return p.pkt_id;
}
```

Listing 8.4: DATA packet construction, transmission, and ID binding (`halon_firmware.ino`).

most one outstanding unacknowledged DATA packet at a time. Dequeueing a second packet while the first awaits acknowledgement would overwrite the `PendingUpdate` record and the `last_sent` buffer, destroying the tracking state needed for both retransmission and weight update. The guard prevents this at the main-loop level, so the scheduler itself never needs to handle the case of a second concurrent pending update.

8.4 The ACK-Driven Learning Loop

The integration between ACK handling and the Q-learning scheduler is the central feedback loop of the firmware. Understanding it requires tracing the full lifecycle of a DATA packet from enqueue to learning update. When a DATA packet is dequeued, the scheduler captures the pending update record (Section 5.5). The firmware constructs the packet struct, registers it in the outbox and metrics modules, and transmits it via the radio driver. The packet ID is reported back to the scheduler via `scheduler_set_tx_id()` immediately after the radio send, as shown in Listing 8.4.

When an ACK arrives, `handle_ack()` extracts the original packet ID and destination node from the ACK payload. The weight update is applied *before* retransmission cancellation (Listing 8.5).

The ordering of `scheduler_record_outcome` before `scheduler_cancel_retransmit_dst` preserves the consistency of the learning signal. `record_outcome` reads the current queue state to compute Q_{next} (equation (5.12)). If retransmissions were cancelled first, the queue would be smaller, and Q_{next} would reflect a different, artificially re-

```
static void handle_ack(Packet* pkt) {
    if (pkt->dst_id == NODE_ID) {
        if (pkt->payload_len >= 2) {
            uint8_t orig_id  = pkt->payload[0];
            uint8_t dst_node = pkt->payload[1];
            outbox_set_delivered(orig_id);
            metrics_record_ack(orig_id);
            scheduler_record_outcome(orig_id, true);
            scheduler_cancel_retransmit_dst(dst_node);
        }
    } else if (pkt->ttl > 1 && pkt->next_hop == NODE_ID) {
         uint8_t nh = routing_next_hop(pkt->dst_id);
        if (nh == NO_ROUTE) return;
        pkt->next_hop = nh; pkt->prev_hop = NODE_ID; pkt->ttl--;
        radio_send(pkt);
    }
}
```

Listing 8.5: ACK handling with learning update ordering (`halon_firmware.ino`).

duced queue state. Updating weights first and then cancelling ensures the learning signal is conditioned on the queue state at the actual moment of ACK receipt, which is the most accurate and representative context for the TD update.

8.5 ACK Routing and the `prev_hop` Fallback

Routing an ACK from the destination back to the original sender requires a path in the return direction. In a well-converged network, this path exists in the routing table. However, if the source's LSA has not yet reached the destination's routing table the normal routing query returns `NO_ROUTE` and the ACK cannot be sent.

The firmware handles this with the `prev_hop` fallback: the DATA packet's `prev_hop` field records the ID of the node that forwarded it. That node is, by construction, a direct neighbour of the destination and a valid candidate for the first hop of the ACK's return journey (Listing 8.6).

This fallback is not merely a defensive measure: it is the mechanism that allows the ACK-driven learning loop to function correctly in a network that has not yet fully converged. Without it, the source node would time out waiting for an ACK, re-enqueue the packet, and the scheduler would receive a spurious failure signal that would incorrectly penalise a transmission that actually succeeded.

8.6 Duplicate Suppression with the Seen Cache

Because LoRa is a broadcast medium, a DATA packet transmitted by one node is received by every node within range, including nodes that are not the intended `next_hop`. Without duplicate suppression, a node that overhears a packet could forward it again, causing duplicate deliveries and wasted channel time. The firmware maintains a circular cache of recently seen (source, packet-ID) pairs, shown in Listing 8.7.

```
static void send_ack(uint8_t orig_pkt_id, uint8_t orig_src_id,
                     uint8_t reverse_hop) {
    uint8_t nh = routing_next_hop(orig_src_id);
    if (nh == NO_ROUTE) {
        if (reverse_hop != 0 && reverse_hop != BROADCAST_ID
            && reverse_hop != NODE_ID) {
            nh = reverse_hop;
        } else {
            Serial.print(F("ACK drop (no route to N"));
            Serial.print(orig_src_id); Serial.println(')');
            return;
        }
    }
    Packet p;
    p.src_id     = NODE_ID;  p.dst_id  = orig_src_id;
    p.next_hop   = nh;       p.prev_hop= NODE_ID;
    p.pkt_id     = next_pkt_id();
    p.ttl        = TTL_DEFAULT;  p.type = PKT_ACK;
    p.payload[0] = orig_pkt_id;
    p.payload[1] = NODE_ID;
    p.payload_len= 2;
    radio_send(&p);
}
```

Listing 8.6: ACK routing with prev_hop fallback (`halon_firmware.ino`).

```
typedef struct { uint8_t src; uint8_t id; } SeenEntry;
static SeenEntry seen[SEEN_CACHE_SIZE];
static uint8_t   seen_idx = 0;

static bool seen_check_and_add(uint8_t src, uint8_t id) {
    for (uint8_t i = 0; i < SEEN_CACHE_SIZE; i++)
        if (seen[i].src == src && seen[i].id == id) return true;
    seen[seen_idx].src = src;
    seen[seen_idx].id  = id;
    seen_idx = (uint8_t)((seen_idx + 1u) % SEEN_CACHE_SIZE);
    return false;
}

static void handle_data(Packet* pkt) {
    if (seen_check_and_add(pkt->src_id, pkt->pkt_id)) return;

    if (pkt->dst_id == NODE_ID) {
        led_pulse_green();
        inbox_add(pkt->pkt_id, pkt->src_id,
                  (const char*)pkt->payload, pkt->payload_len);
        send_ack(pkt->pkt_id, pkt->src_id, pkt->prev_hop);
    } else if (pkt->ttl > 1 && pkt->next_hop == NODE_ID) {
        led_pulse_blue();
        uint8_t nh = routing_next_hop(pkt->dst_id);
        if (nh == NO_ROUTE) return;
        pkt->next_hop = nh; pkt->prev_hop = NODE_ID; pkt->ttl--;
        radio_send(pkt);
    }
}
```

Listing 8.7: Seen cache and DATA packet dispatch (`halon_firmware.ino`).

```
srand((unsigned int)((uint32_t)NODE_ID * 2654435761UL ^ micros()));

delay((uint32_t)NODE_ID * 300UL);
```

Listing 8.8: PRNG seeding and boot stagger (`halon_firmware.ino`).

The circular eviction policy means entries age out of the cache after `SEEN_CACHE_SIZE` subsequent packets have been received and recorded. With `SEEN_CACHE_SIZE` = 16 and transmit cooldowns measured in hundreds of milliseconds, in-flight duplicates are reliably suppressed: any re-broadcast of the same packet by a neighbouring node would arrive within a few milliseconds, well before 16 new unique packets had evicted the original entry.

8.7 Deterministic PRNG Seeding

The epsilon-greedy scheduler uses `rand()` for its exploration draws. If all nodes use the same PRNG seed, their exploration decisions are identical and correlated: all nodes would explore or exploit the same candidates simultaneously, undermining the diversity of scheduling decisions across the mesh and eliminating the statistical benefit of independent exploration. The firmware seeds the PRNG at boot with a value unique to each node and incorporating hardware timing entropy:

$$\text{seed} = \underbrace{\texttt{NODE_ID} \times 2{,}654{,}435{,}761}_{\text{hash}} \oplus \underbrace{\texttt{micros()}}_{\text{hardware entropy}}, \tag{8.1}$$

where the constant $2{,}654{,}435{,}761 \approx 2^{32}/\varphi$ is the Knuth multiplicative hash constant and φ is the golden ratio. Multiplication by this constant spreads the small integer node IDs (1 through 20) across the full 32-bit integer space, ensuring that adjacent node IDs produce wildly different initial PRNG states. The XOR with `micros()` adds entropy from the hardware microsecond timer at boot, so two nodes with the same ID but different power-on times (as may occur during development or after a partial network reset) also produce different exploration sequences. The seeding and boot stagger are shown in Listing 8.8.

The `delay` after seeding staggers the initial HELLO broadcasts by $300 \times$ `NODE_ID` milliseconds. Without this stagger, all nodes would attempt their first HELLO at approximately the same time after power-on, causing collisions that would delay initial neighbour discovery and LSA convergence. The stagger ensures node 1 broadcasts first while nodes 2 through N listen and record the HELLO, then node 2 broadcasts, and so on, so that the first round of HELLO exchanges completes without collision in a newly powered network.

8.8 Serial Command Interface for Diagnostics

The firmware exposes a text-based serial command interface that allows developers to interrogate and drive the network during testing. The most important commands for development and field diagnostics are summarised in Table 8.1.

Command	Effect
`send <dst> <pri> <msg>`	Enqueue a message with the specified destination, priority level (`l`/`m`/`h`/`c`), and payload string.
`test <N> <dst>`	Enqueue N test messages to destination `dst`, used for throughput and latency benchmarking.
`test time <s> <dst>`	Continuously inject test messages to `dst` for `s` seconds.
`teststats`	Print cumulative delivery metrics (packets sent, ACKs received, average latency, loss rate).
`weights`	Print the current Q-learning weight vectors in the $\times 100$ integer format. Used to verify that firmware weights match the Python simulator after a given packet sequence.
`status`	Print the neighbour table, routing table, current queue depth, and the full learned topology (all link costs accumulated from received LSAs).
`lsa`	Force an immediate LSA broadcast, useful for manually triggering topology propagation in a stalled network.

Table 8.1: Serial command interface summary.

The `status` command is particularly valuable for debugging routing issues: its topology output prints the full 20×20 link-cost matrix as viewed by the local node, including link costs that were learned from other nodes' LSAs rather than directly measured. Comparing `status` output across multiple nodes reveals topology asymmetries or LSA propagation failures that would be invisible from any single node's perspective.

Chapter 9

System Evaluation

The High Altitude LoRa Network (HALoN) is evaluated through structured validation of both functional and non-functional requirements. HALoN is a crisis-communication system designed to provide resilient, priority-aware message delivery in disaster environments. The system consists of two primary subsystems developed in parallel: a software subsystem responsible for scheduling and routing decisions, and a hardware subsystem centered around a reconfigurable antenna architecture.

Because HALoN is intended for deployment in high-risk, time-sensitive scenarios, evaluation is treated as a risk-mitigation process rather than a post-development formality. Testing is structured to reduce technical uncertainty, mitigate schedule risk, and ensure alignment with system-level goals of reliability, prioritization, adaptability, and usability.

Functional requirements are evaluated through verification of successful multi-hop message delivery, priority-aware scheduling behavior, rerouting under node failure, and correct antenna performance at the intended operating frequency. Non-functional requirements such as latency, delivery rate, gain, directionality, and usability are evaluated through quantitative measurement and controlled testing environments.

Evaluation is divided into internal testing, external testing, unit testing, and integration testing, each targeting a different layer of system risk.

9.1 Internal Testing

Internal testing consists of structured validation performed by the project team to evaluate correctness, robustness, and performance prior to user exposure.

Software Validation

The software subsystem governs message scheduling across the network. Because HALoN prioritizes communication originating near disaster zones, scheduling correctness directly impacts mission effectiveness.

Three scheduling approaches are evaluated:

- First Come First Serve (FCFS)
- Shortest Job First (SJF)
- Priority-Aware Q-Learning

These algorithms are tested within a custom-developed network simulator that emulates realistic multi-node LoRa communication environments. The simulator collects quantitative metrics including:

- Average network latency
- Number of injected packets
- Number of delivered packets
- Delivery rate
- Average queue wait time per node

Performance across these metrics determines which algorithm best satisfies system-level objectives. The highest-performing algorithm is deployed onto the microcontroller platform. This process mitigates the risk of deploying an algorithm that performs well theoretically but degrades under congestion or topology variability.

Hardware Validation

S-parameters, or scattering parameters provide a rigorous framework for characterizing the behavior of linear, time-invariant networks at high frequencies. S-parameters are defined in terms of incident and reflected traveling waves at each port, making them directly measurable with a vector network analyzer and well-suited for characterization in matched transmission-line environments. For an N-port network, the scattering parameter S_{ij} is defined as the ratio of the wave emerging from port i to the incident wave at port j, with all other ports terminated in the reference impedance Z_0:

$$S_{ij} = \frac{b_i}{a_j}|_{a_k=0, k\neq j} \tag{9.1}$$

For our designed network, there are three ports, making the following matrix:

$$[S] = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{bmatrix} \tag{9.2}$$

The particular parameters pertinent to this work are the transmission coefficients S_{21}, S_{31} and the reflection coefficient S_{11}. S_{21} and S_{31} characterize the forward power transmission through the network and it is a complex quantity.

Specifically, its magnitude captures insertion loss performance while its phase directly encodes the electrical delay introduced by the network. This makes S_{21} and S_{31} critical parameters for validating both the power division performance of the power divider and the phase shift introduced by the phase shifter, where the target phase offset can be verified by comparing the arguments of S_{21} and S_{31}. S_{11}characterizes the impedance match at the input port, quantifying the fraction of incident power reflected due to impedance mismatch.

In Ansys HFSS, extracting S-parameters is the primary way we measured performance. For each of the topologies, the input and output ports were measured and analyzed.

This lets us get our desired performance before combining everything into the complete design for the endfire branch. Beginning with the power divider, we ideally want a 50% split between ports 2 and 3, which is -3 in decibels, and for the phase shifter, we expect about a 180° phase shift between ports 2 and 3.

For the broadside branch, the validation is simpler as the main metric we care about is how much power gets to the output. Using Fig 9.1 as reference, we care about the S_{41} parameter. As this is just a transmission line, the magnitude of S_{41} is observed in simulation and physical measurement.

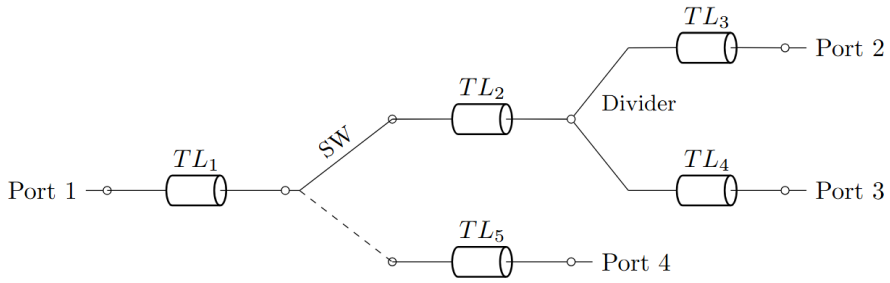


Figure 9.1: Schematic of Feed Network PCB

The PCB can be schematically represented as a four port network, shown in Fig. 9.1. The VNA used to take these measurements only has two ports, so to accurately measure the performance, a sequence of two-port measurements were taken with the other ports terminated with 50Ω loads. For the results in Table 9.1, the values are taken from the S_{21} measurement from each port.

Measurement	Simulation	Physical PCB
$\angle S_{21}$	−34.71°	−77.52°
$\angle S_{31}$	139.67°	104.92°
$\lvert S_{21}\rvert$	-3.72 dB	-5.48 dB
$\lvert S_{31}\rvert$	-3.34 dB	-5.51 dB
$\lvert S_{41}\rvert$	-0.286 dB	-3.35 dB

Table 9.1: Data Values from Simulation and Physical Measurement of Reconfigurable Feed Network

Analyzing the phase, a discrepancy of approximately 35° − 45° is observed between simulation and physical measurement at both ports 2 and 3. This offset is attributed primarily to the permittivity tolerance of the FR-4 substrate,

which ranges from 4.2 to 4.8, compared to the HFSS default value of 4.4 used in simulation. Variation in ϵ_r directly affects the phase velocity along the traces, altering the electrical length of the network and introducing a systematic phase offset at each port. Despite this, the total phase difference between ports 2 and 3 is 174.38° in simulation and 182.44° in physical measurement, both within approximately 8° of the target 180° shift. This confirms that the endfire radiation pattern will still form as intended, as the functional phase relationship between the output ports is preserved.

In the magnitude measurements of ports 2 and 3, a discrepancy of approximately 1.76-2.17 dB is observed between simulation and physical measurement, corresponding to only 28% of the input power being delivered at each port. The S_{41} result is particularly illustrative in diagnosing the source of this loss. A simple transmission line would ideally exhibit negligible insertion loss, yet the physical measurement shows approximately 3.06 dB of loss, indicating that the FR-4 substrate is the dominant loss mechanism rather than the complexity of the divider or phase shifter topology. Additional contributors to the observed discrepancy include SMA connector launch parasitics, solder joint resistance, and trace edge roughness, all non-idealities inherent to the fabrication process that are not captured in simulation. Taken together, these results confirm that while absolute power delivery is reduced relative to simulation, the network meets its primary functional requirement of producing a 180° phase difference between output ports, enabling endfire beam formation.

9.2 External Testing

External testing focuses on operational usability under non-technical conditions. Because HALoN is intended for disaster deployment where users may lack technical expertise, usability is critical.

A structured instruction document will be developed describing system setup and message transmission procedures. Individuals with no prior technical background will be asked to follow these instructions and attempt to send messages across the network without assistance.

During testing, we evaluate:

- Ability to successfully initiate communication
- Frequency and type of user errors
- Clarity and completeness of documentation
- Time required to complete message transmission

Observed failure points or confusion will be documented and used to refine system interface design and instructional clarity.

9.3 Unit Testing

Unit testing verifies isolated components of the HALoN system prior to full system integration. Because HALoN contains both simulation software and embedded firmware, unit testing is applied differently across the two subsystems.

For the scheduling simulator, unit tests verify correctness of the queueing and scheduling logic that determines packet transmission order. These tests ensure that packet insertion, removal, priority handling, and latency accounting operate deterministically under controlled inputs. Verifying these behaviors in isolation reduces the risk that errors in scheduling logic propagate into full network simulations.

For the embedded LoRa firmware, unit tests focus on deterministic logic that does not require direct radio transmission. Hardware-dependent components such as RF communication and interrupts are difficult to test in isolation without full hardware integration, so testing concentrates on packet construction, routing state transitions, and timing-based behaviors within the firmware.

The Arduino firmware unit tests are implemented using the *AUnit* testing framework, which allows test cases to run directly on the microcontroller. The following tests validate key components of the HALoN routing firmware.

Packet Parsing Test

This test verifies that a user message injected through the serial interface correctly parses routing information and payload data. Given an input such as `"123 Hello"`, the firmware should construct a mesh packet whose path array contains nodes 1, 2, and 3, with the payload set to `"Hello"`.

```
test(InjectUserMessage_ParsesPathAndPayload) {
    String input = "123 Hello";
    injectUserMessage(input);

    assertEqual(packet.pathLen, 3);
    assertEqual(packet.path[0], 1);
    assertEqual(packet.path[1], 2);
    assertEqual(packet.path[2], 3);
}
```

Invalid Input Rejection Test

This test ensures that malformed serial inputs that do not contain a routing path and payload separator are rejected. If a message such as `"123Hello"` is received, the firmware should ignore the input and avoid transmitting a packet.

```
test(InjectUserMessage_InvalidFormatIgnored) {
    String input = "123Hello";
    injectUserMessage(input);

    assertFalse(packetWasSent);
}
```

Hop Progression Test

This test validates the mesh routing logic that advances a packet along its predefined path. When a node receives a mesh packet whose next hop matches its node identifier, the firmware should increment the packet's path index before forwarding it.

```
test(MeshPacket_HopProgression) {
    PKT packet;
    packet.type = PKT_TYPE_MESH;
    packet.path[0] = NODE_ID;
    packet.pathLen = 2;
    packet.pathIndex = 0;

    if (packet.path[packet.pathIndex] == NODE_ID) {
        packet.pathIndex++;
    }

    assertEqual(packet.pathIndex, 1);
}
```

Final Node Conversion Test

This test verifies that when a packet reaches the final node in its routing path, it is converted from a mesh-routing packet to an end-user delivery packet.

```
test(FinalNode_ConvertsToUserPacket) {
    PKT packet;
    packet.type = PKT_TYPE_MESH;
    packet.pathLen = 1;
    packet.pathIndex = 1;

    if (packet.pathIndex >= packet.pathLen) {
        packet.type = PKT_TYPE_USER;
    }

    assertEqual(packet.type, PKT_TYPE_USER);
}
```

LED Timing Test

This test confirms that the hop indicator LED correctly deactivates after its timing interval expires. Proper LED state management ensures that visual feedback remains synchronized with packet forwarding behavior.

```
test(LedTurnsOffAfterDuration) {
    ledActive = true;
    ledOffTime = millis() - 1;

    updateLed();

    assertFalse(ledActive);
}
```

Together, these tests validate the correctness of packet construction, routing progression, message handling, and timing behavior within the HALoN firmware. By verifying these components independently of the full network

environment, unit testing reduces debugging complexity and increases confidence in the reliability of the embedded communication protocol.

9.4 Integration Testing

Integration testing evaluates how independently validated components function together as cohesive pipelines. While unit tests verify a single function with controlled inputs, integration tests exercise the full call chains that execute during real operation: `injectUserMessage()` calling into `sendPacket()`, the receive handler driving `hopBlink()` and then forwarding, and the final-node path triggering type conversion alongside serial output. Failures that only emerge from component interactions—such as state corruption between calls, incorrect LED timing under concurrent radio load, or payload loss across the relay pipeline—are the specific targets of integration testing.

Because the LoRa radio (`rf95`) requires physical hardware, a lightweight stub layer replaces `rf95.send()` and `rf95.recv()` so that all six integration tests run directly on the microcontroller using the *AUnit* framework, identical to the unit tests. The stubs capture transmitted packets into a shared `capturedTx` struct and inject pre-built receive packets via `stubRecv()`. This approach exercises the full software pipeline without a live radio link while still testing real firmware logic. A `resetState()` helper clears all shared state between tests to prevent cross-test contamination.

Hardware Stubs and Shared State

The following declarations replace the hardware-dependent radio calls during testing. `sendPacket()` is redefined to capture the transmitted packet rather than transmit over the radio. `stubRecv()` injects a caller-configured `rxPacket` into the receive path. `resetState()` clears all shared variables before each test.

```
static PKT    capturedTx;
static bool   txWasCalled    = false;
static PKT    rxPacket;
static bool   rxReady        = false;
static String capturedSerial = "";

void sendPacket(PKT &packet) {
    capturedTx  = packet;
    txWasCalled = true;
}

bool stubRecv(uint8_t *buf, uint8_t *len) {
    if (!rxReady) return false;
    memcpy(buf, &rxPacket, sizeof(PKT));
    *len    = sizeof(PKT);
    rxReady = false;
    return true;
}

void resetState() {
    txWasCalled    = false;
    memset(&capturedTx, 0, sizeof(PKT));
```

```
    memset(&rxPacket,   0, sizeof(PKT));
    rxReady        = false;
    ledActive      = false;
    ledOffTime     = 0;
    capturedSerial = "";
}
```

Integration Test 1: Inject → Parse → Send Pipeline

This test verifies the full `injectUserMessage()` pipeline as an integrated chain: a serial string drives path parsing, payload copy, and finally `sendPacket()`. Unlike the unit test that only checked individual struct fields, this test also asserts that `sendPacket()` was actually called, confirming the parse and transmit stages are correctly coupled.

```
test(Integration_InjectMessage_CallsSendWithCorrectPacket) {
    resetState();
    String input = "123 Hello";
    injectUserMessage(input);

    assertTrue(txWasCalled);
    assertEqual((int)capturedTx.type, (int)PKT_TYPE_MESH);
    assertEqual((int)capturedTx.pathLen,   3);
    assertEqual((int)capturedTx.path[0],   1);
    assertEqual((int)capturedTx.path[1],   2);
    assertEqual((int)capturedTx.path[2],   3);
    assertEqual((int)capturedTx.pathIndex, 0);
    assertEqual(String(capturedTx.payload), String("Hello"));
}
```

Integration Test 2: Malformed Input → Pipeline Abort

This test verifies that an invalid serial string causes the entire pipeline to abort before `sendPacket()` is reached. The unit test confirmed the parser returns early on a missing space separator; this test confirms that early return correctly suppresses the transmission stage as well.

```
test(Integration_MalformedInput_DoesNotCallSend) {
    resetState();
    injectUserMessage(String("123Hello")); // missing space separator

    assertFalse(txWasCalled);
}
```

Integration Test 3: Receive → Hop Logic → LED → Forward

This test simulates a mesh packet arriving at an intermediate relay node. It exercises the full receive-to-forward pipeline: `stubRecv()` injects the packet, the hop check verifies the node ID, `hopBlink()` activates the LED, `pathIndex` is incremented, and `sendPacket()` is called with the updated packet. All four components are tested together in one integrated chain.

```
test(Integration_RelayHop_IncrementsIndexAndBlinksThenForwards) {
    resetState();
```

```
    rxPacket.type      = PKT_TYPE_MESH;
    rxPacket.pathLen   = 2;
    rxPacket.pathIndex = 0;
    rxPacket.path[0]   = NODE_ID;
    rxPacket.path[1]   = 2;
    strncpy(rxPacket.payload, "TestMsg",
            sizeof(rxPacket.payload) - 1);
    rxReady = true;

    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);
    if (stubRecv(buf, &len)) {
        PKT packet;
        memcpy(&packet, buf, len);
        if (packet.type == PKT_TYPE_MESH &&
            packet.path[packet.pathIndex] == NODE_ID) {
            hopBlink();
            packet.pathIndex++;
            if (packet.pathIndex < packet.pathLen)
                sendPacket(packet);
        }
    }

    assertEqual((int)capturedTx.pathIndex, 1);
    assertEqual((int)capturedTx.type, (int)PKT_TYPE_MESH);
    assertTrue(ledActive);
    assertTrue(txWasCalled);
    assertEqual(String(capturedTx.payload), String("TestMsg"));
}
```

Integration Test 4: Receive → Final Node → Type Conversion → Output

This test simulates a packet arriving at the last node in the routing path. The packet type must convert from PKT_TYPE_MESH to PKT_TYPE_USER, the final-hop LED must fire, the payload must be written to serial output, and a USER packet must be transmitted. All four actions are verified together, making this the highest-value integration test as it covers the complete delivery confirmation pipeline at the destination node.

```
test(Integration_FinalNode_ConvertsAndDelivers) {
    resetState();
    rxPacket.type      = PKT_TYPE_MESH;
    rxPacket.pathLen   = 1;
    rxPacket.pathIndex = 0;
    rxPacket.path[0]   = NODE_ID;
    strncpy(rxPacket.payload, "CrisisAlert",
            sizeof(rxPacket.payload) - 1);
    rxReady = true;

    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);
    if (stubRecv(buf, &len)) {
        PKT packet;
        memcpy(&packet, buf, len);
```

```
        if (packet.type == PKT_TYPE_MESH &&
            packet.path[packet.pathIndex] == NODE_ID) {
            packet.pathIndex++;
            if (packet.pathIndex >= packet.pathLen) {
                finalBlink();
                packet.type = PKT_TYPE_USER;
                stubSerialPrint(packet.payload);
                sendPacket(packet);
            }
        }
    }

    assertEqual((int)capturedTx.type, (int)PKT_TYPE_USER);
    assertTrue(ledActive);
    assertTrue(capturedSerial.indexOf("CrisisAlert") >= 0);
    assertTrue(txWasCalled);
}
```

Integration Test 5: `hopBlink()` + `updateLed()` Timing Integration

This test verifies that `hopBlink()` and `updateLed()` interact correctly through their shared `ledActive` and `ledOffTime` state, as they would across iterations of the main firmware loop. Backdating `ledOffTime` simulates a subsequent loop iteration occurring after the blink duration has elapsed, confirming the two functions integrate correctly across calls.

```
test(Integration_HopBlink_ExpiresOnNextLoopUpdate) {
    resetState();
    hopBlink();
    assertTrue(ledActive);

    ledOffTime = millis() - 1;
    updateLed();

    assertFalse(ledActive);
}
```

Integration Test 6: Sequential Messages Maintain Independent State

This test verifies that two back-to-back calls to `injectUserMessage()` produce fully independent packets with no state bleed between invocations. The function must reset `pathLen` and `pathIndex` before processing each new input so that the routing path from the first call does not persist into the packet built by the second.

```
test(Integration_TwoSequentialMessages_IndependentState) {
    resetState();

    injectUserMessage(String("12 Alpha"));
    assertTrue(txWasCalled);
    assertEqual((int)capturedTx.pathLen, 2);
    assertEqual(String(capturedTx.payload), String("Alpha"));

    txWasCalled = false;
```

```
    memset(&capturedTx, 0, sizeof(PKT));

    injectUserMessage(String("345 Beta"));
    assertTrue(txWasCalled);
    assertEqual((int)capturedTx.pathLen, 3);
    assertEqual(String(capturedTx.payload), String("Beta"));
    assertEqual((int)capturedTx.path[0], 3);
    assertEqual((int)capturedTx.path[1], 4);
    assertEqual((int)capturedTx.path[2], 5);
}
```

Together, these six integration tests validate the correctness of the HALoN firmware pipelines that span multiple components. By replacing only the hardware radio calls with stubs while preserving all real firmware logic, the tests expose integration failures that unit tests cannot detect: incorrect coupling between parse and transmit stages, broken LED-to-loop synchronization, state corruption across sequential message injections, and payload loss across the relay pipeline. All tests execute on the microcontroller using the *AUnit* framework, enabling automated pass/fail reporting over the serial monitor without requiring a live radio environment.

Chapter 10

Implementation Plan

10.1 Timeline

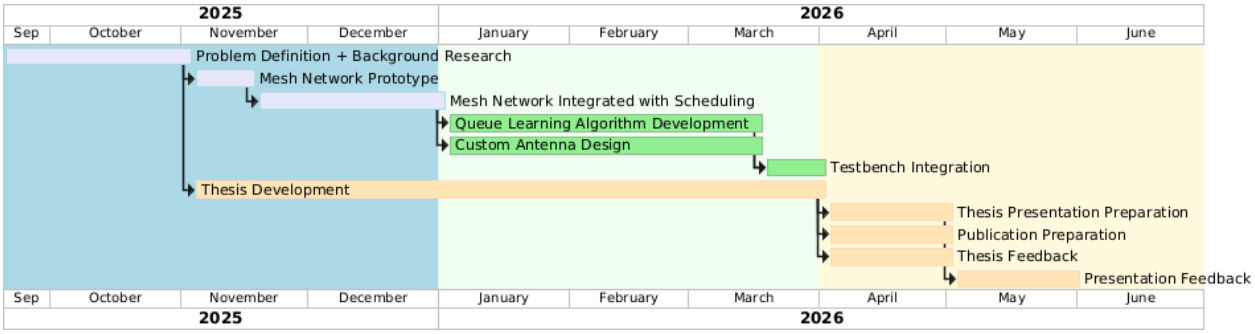


Figure 10.1: Gantt Chart of HALoN Project Timeline

10.2 Project Risks

Risk I: Simplex Communication in a Duplex-Oriented System

A major risk in this project is integrating a simplex communication channel into a system that is typically designed assuming duplex communication. The Adafruit Feather LoRa boards operate in simplex mode, meaning a node can either transmit or receive at any given time, but not both. Because of this constraint, scheduling algorithms that rely on preemption or overlapping communication are fundamentally incompatible with the hardware.

To mitigate this, HALoN uses a non-preemptive scheduling algorithm combined with time division multiplexing (TDM). Nodes are assigned fixed transmission windows, and all other nodes remain in receive mode during that window. This approach aligns with TDMA-style scheduling used in low-power wireless sensor networks and avoids the need for preemption altogether.

Risk II: Distributed Clock Synchronization for TDM

The solution to Risk I directly introduces Risk II. In order for TDM to function correctly, nodes must have a shared understanding of when their transmission slots begin and end. In a distributed wireless system, achieving perfect clock synchronization is not feasible. Transmission delays are non-deterministic and vary with distance and environmental conditions, meaning clocks cannot be initialized simultaneously or remain perfectly aligned over time.

Rather than attempting strict global synchronization, HALoN uses coarse-grained time alignment with tolerance for error. A reference node periodically broadcasts a synchronization beacon defining the start of a scheduling frame. Nodes align their local slot counters relative to this frame rather than to an absolute global clock. Small guard intervals are inserted between time slots to absorb clock drift and propagation delay, allowing correct operation under bounded synchronization error.

Risk III: Priority, Fairness, and Starvation in Crisis-Aware Scheduling

Scheduling policy selection is a key risk given HALoN's role as a crisis-response communication system. Messages originating closer to the epicenter of a crisis, or from nodes experiencing greater severity, must be prioritized to ensure timely delivery. However, strict priority scheduling introduces the risk of starvation for lower-priority messages.

To address this, HALoN implements a priority-aware queue learning scheduler. The scheduler maintains three queues, one per priority level, where priority is determined by proximity to the crisis. While higher-priority queues are serviced more frequently, messages in lower-priority queues are promoted after spending a bounded amount of time in the system. This aging mechanism prevents starvation while preserving urgency-aware scheduling.

Risk IV: Limited Compute and Memory Resources

Another major risk is the limited memory and computational capability of the Adafruit Feather boards. Early versions of the system design assumed the availability of long training phases, large state representations, or complex models, all of which are unrealistic on constrained embedded hardware.

To mitigate this, the Q-learning algorithm was redesigned to rely only on metrics that naturally arise from system behavior, such as queue occupancy, transmission success, and observed latency. The state space is intentionally small, training occurs in real-time, and no large matrices or external models are required, making the approach feasible for embedded deployment.

Risk V: Power Constraints and Transmission Efficiency

Finally, the Feather boards operate under strict power constraints. Using omnidirectional antennas results in significant power loss due to energy being radiated in directions where no receiving node exists. This inefficiency is particularly problematic in emergency deployments, where redeployment is not possible.

To address this, HALoN incorporates a reconfigurable directional antenna that focuses transmission energy along specific vectors. By limiting radiation to relevant directions, the system reduces power consumption while improving effective range and link reliability.

Chapter 11

Constraints and Standards

11.1 Constraints

Hardware Memory and Computational Resources

Every HALoN node runs on an ATmega32u4 microcontroller clocked at 8 MHz, with 32 KB of flash and 2 KB of SRAM [20]. These characterstics become limiting when accounting for everything that must reside in memory simultaneously: the neighbor table, LSA records, a topology graph, a Q-table, acknowledgment timers, amongst other structures. To make this feasible, every data structure in HALoN carries a hard size cap: dynamic memory allocation was ruled out entirely, as the overhead and risk of corrupting other crucial memory are incompatible with our tight memory system. These are the decisions made as a direct consequence of the hardware platform, and they shaped the algorithmic design of every subsystem in HALoN.

Simplex (Half-Duplex) Radio Operation

The RFM95 LoRa radio can either transmit or receive at any given moment (known as simplex), never both simultaneously [20]. While a node is transmitting, it is completely deaf to the channel and cannot detect incoming packets. It has many consequences, including that a node cannot receive an acknowledgment during the same transmission that triggered it; that a node that receives from multiple planes risks both the loss of messages and the loss of transmission. HALoN addresses these consequences through time-division multiplexing, where nodes take structured turns on the channel so that every transmission occurs into a listening network, and ACK receive windows are explicitly scheduled after each outgoing packet. The half-duplex limitation is an inherent property of the radio hardware, and the entire network structure is designed around it.

FCC Regulatory Constraints on ISM-Band Transmission

HALoN transmits in the 902-928 MHz ISM band, placing it under FCC jurisdiction via Title 47 of the Code of Federal Regulations, Part 15. The rule with the most direct impact on HALoN's behavior is 47 CFR § 15.247: for frequency-

hopping systems using channels narrower than 250 kHz, no single channel may be occupied for more than 0.4 seconds out of any 20-second window [21]. This limit constrains how long HALoN can dwell on any frequency and bounds the maximum burst length before a channel hop is required. Separately, 47 CFR § 15.5 requires that Part 15 devices accept interference from licensed users [22], meaning HALoN must accept that packets may corrupted by other spectrum users. The retransmission logic exists in part because due to this condition. Any real-world deployment in the United States also requires that the radio hardware carry FCC certification, a requirement the RFM95 module already satisfies.

FAA Airspace Regulations for Unmanned Free Balloon Operations

HALoN's balloon-borne relay nodes fall under 14 CFR Part 101, Subpart D, which governs unmanned free balloon operations in U.S. airspace. The most operationally significant requirement is 47 CFR § 101.35: operators must notify the nearest FAA Air Traffic Control facility between 6 and 24 hours before any launch, and the notice must include the planned launch time and location, forecast trajectory, estimated time to cruising altitude, and the physical characteristics of the balloon and payload [23]. For a disaster response system, this is a meaningful constraint, but perhaps more so for a student-development team. Balloon relay nodes cannot be deployed on short notice in response to a sudden-onset event, and the activation of any balloon backbone requires advance planning and regulatory coordination. Section 101.37 adds hardware requirements as well, including radar reflectors for operations near controlled airspace. In practice, this constraint prevented the testing of the HALoN system in the ideal air-borne topology for which it had been designed: instead, the nodes have been deployed on the ground in smaller areas, with limited transmission power to scale.

Economic Constraint: Per-Node Cost and Budget

HALoN was developed within the fixed budget of a senior design project, and cost per node is a core design requirement independent of that limitation. The system is intended for deployment in resource-constrained disaster response environments where expensive commercial infrastructure is not a viable option, so affordability is a functional requirement, not merely a convenience. The Adafruit Feather 32u4 with integrated LoRa radio was selected in part because it is among the most cost-effective platforms available for long-range wireless mesh operation [20]. Operating in the unlicensed ISM band eliminates recurring spectrum licensing costs. The simulation-first evaluation strategy, built around a custom Python network simulator, was also a product of this constraint: procuring and developing a large physical testbed was not within budget. Beyond the project itself, keeping per-node cost low is an equity consideration. The communities most likely to lose communications infrastructure in a major disaster are often those least able to afford premium recovery equipment, and a system that is too expensive to deploy where it is needed most provides little practical value.

11.2 Standards

FCC 47 CFR Part 15: Unlicensed Radio Frequency Devices

Part 15 of Title 47 of the Code of Federal Regulations establishes the legal framework under which HALoN nodes may transmit in the 902–928 MHz ISM band without an individual operator license [21]. Section 15.247 defines the technical envelope for intentional radiators in this band: up to 1 watt of conducted output power for frequency-hopping systems with at least 50 channels, minimum 6 dB bandwidth requirements, and the channel occupancy time limits that directly constrain HALoN's hopping behavior. Section 15.5 establishes the baseline condition for all Part 15 operation, requiring that devices accept interference from licensed users and cause no harmful interference to licensed services [22]. These requirements influenced the choice of radio module, the spreading factor and bandwidth configuration, and the design of the retransmission and hopping logic. Compliance with Part 15 is a legal prerequisite for any U.S. deployment, and all of HALoN's transmission parameters were selected with those limits as a hard ceiling.

Semtech SX1276: LoRa Physical Layer Specification

The RFM95 radio module on the Adafruit Feather 32u4 is built on the Semtech SX1276 chipset, whose datasheet defines the LoRa physical layer that HALoN's radio stack is programmed against [14]. LoRa uses Semtech's patented chirp spread spectrum modulation, encoding data by sweeping a carrier across a defined bandwidth according to a pattern determined by the spreading factor. The SX1276 supports spreading factors SF6 through SF12 and bandwidths of 125, 250, or 500 kHz, with each combination trading data rate for link budget and receiver sensitivity. HALoN's physical layer parameters, including the spreading factor and bandwidth values used for mesh links, are selected from the options the SX1276 defines. The packet structure, CRC behavior, interrupt timing, and register map documented in the datasheet are the specification that HALoN's firmware is written against for all radio operations.

LoRa Alliance LoRaWAN Link Layer Specification v1.0.4 (TS1-1.0.4)

LoRaWAN is the primary open standard for wide-area LoRa networks, and version 1.0.4 of the Link Layer Specification defines the MAC frame format, device classes, acknowledgment mechanisms, and regional frequency plans [7]. HALoN does not implement LoRaWAN's star-of-stars topology due to the centralized nature of that network structure, but the US915 regional parameters the specification defines, 64 uplink channels across 902.3–914.9 MHz at 125 kHz bandwidth and 8 downlink channels at 500 kHz bandwidth, directly informed HALoN's channel plan. The spreading factor and coding rate choices in HALoN's physical layer are consistent with what TS1-1.0.4 specifies for the US915 band. Grounding the physical layer configuration in the LoRaWAN regional specification ensures that HALoN's radio behavior is technically consistent with established LPWAN practice and would be compatible with existing LoRaWAN gateway infrastructure if broader network integration were ever required.

IETF RFC 2328: Open Shortest Path First (OSPF) Version 2

RFC 2328, published by the Internet Engineering Task Force in April 1998, defines OSPF Version 2, the link-state routing protocol that HALoN's routing layer is architecturally modeled on [24]. OSPF operates through four mechanisms: neighbor discovery via periodic HELLO packets, Link State Advertisement generation and network-wide flooding, maintenance of a synchronized topology database at every router, and shortest-path computation using Dijkstra's algorithm. HALoN's routing layer follows this same structure. Nodes exchange HELLOs to discover neighbors, generate LSAs that propagate through the mesh, maintain a local topology database from received LSAs, and run Dijkstra to compute paths to every other node. The RFC 2328 documentation serves as the architectural reference for evaluating the correctness of HALoN's neighbor discovery, flooding, and path selection behavior.

FAA 14 CFR Part 101, Subpart D: Unmanned Free Balloons

Title 14 CFR Part 101, Subpart D governs unmanned free balloon operations in the U.S. National Airspace System and applies directly to HALoN's balloon-borne relay nodes [23]. Section 101.31 defines the scope of the subpart, covering all unmanned free balloon operations in the United States. Section 101.35 establishes pre-launch notification requirements: operators must submit a notice to FAA ATC that includes the launch time and location, forecast trajectory, estimated time to cruising altitude, and the physical dimensions and weight of the balloon and payload. Section 101.37 specifies equipment requirements, including radar reflectors for operations in or near controlled airspace. Every element of deploying a HALoN balloon node in a real operational context, the regulatory filings, the launch timeline, and the payload hardware configuration, must conform to these provisions. No launch may proceed without the notification process in § 101.35 being completed within the specified window.

IEEE 802.11s: Wireless Mesh Networking Amendment

IEEE 802.11s defines wireless mesh networking as an amendment to the IEEE 802.11 standard, finalized in 2011 and incorporated into IEEE 802.11-2012 [25]. The standard specifies how stations establish peer links, discover neighbors, maintain distributed topology state, compute multi-hop paths, and forward frames through intermediate nodes. Path selection is handled by the Hybrid Wireless Mesh Protocol, and routing cost is quantified by the Airtime Link Metric, which accounts for radio link conditions in path selection decisions. HALoN operates over LoRa links rather than 802.11 radios, so the standard does not govern HALoN's hardware or MAC layer. The relevance is architectural: HALoN's mesh layer performs the same functions that 802.11s formalizes, and evaluating HALoN's design against a ratified standard for the same class of system provides a meaningful external reference point for assessing its correctness and completeness.

Arduino Hardware and Software Platform

The Adafruit Feather 32u4 conforms to the Arduino open-source hardware and software standard, which defines the board form factor, bootloader protocol, pin naming conventions, and C++ library API that HALoN's firmware is built against [26]. Writing firmware within the Arduino framework means that hardware access goes through a standardized abstraction layer, third-party libraries follow the Arduino library specification, and the build and flash toolchain is the standard Arduino AVR toolchain. This gave HALoN access to library ecosystems such as the the RadioHead library for RFM95 radio control and the standard `Wire` and `SPI` libraries for peripheral communication, without requiring custom low-level drivers. Adherence to the Arduino platform standard also ensures that the HALoN firmware toolchain is fully reproducible: any developer with the Arduino IDE and the Adafruit board support package can build and flash HALoN without specialized embedded tooling.

ISO/IEC 14882: C++ Programming Language

HALoN's firmware is written in C++ because the Arduino framework is built dominantly upon it [27]. The Arduino abstraction layer, its library API, and the third-party libraries HALoN depends on are all written in C++, making the language a consequence of the platform choice. The ATmega32u4 platform imposes additional restrictions beyond the language standard: dynamic memory allocation, runtime type information, and C++ exceptions are disabled at the compiler level, as none are compatible with our limited RAM. However,other characteristics of the langugage, such as support for structs, stronger typing and compile time constants provided a kind of scaffolding for the codebase. Furthermore, all memory in HALoN is statically allocated, and sizing decisions are enforced at compile time (necessary in our project due to our limited memory system).

Python 3: Simulation and Data Analysis

HALoN's network simulator, performance analysis pipeline, and topology visualization tools are implemented in Python 3, following the language specification maintained by the Python Software Foundation [28]. Python was selected for the simulation environment because its high-level data structures and extensive scientific computing ecosystem allow complex network behavior to be modeled and analyzed efficiently. The simulator captures HALoN's routing protocol, Q-learning scheduler, and retransmission logic in software, enabling network-scale experiments that could not be conducted on physical hardware within project constraints. NumPy supports the array operations required for simulation state management, and Matplotlib generates the performance plots presented throughout this report. Implementing the simulator in Python also ensures that the evaluation methodology is fully reproducible without any hardware dependencies.

Ansys HFSS: Electromagnetic Simulation

Ansys HFSS is a full-wave electromagnetic simulation environment that solves Maxwell's equations using the finite element method and is the standard tool in the RF and antenna engineering community for high-frequency design work [29]. HFSS was used to model and optimize the antenna elements for HALoN's balloon-borne relay nodes operating in the 902–928 MHz ISM band. The simulation workflow included defining antenna geometry and feed structure, assigning material properties and boundary conditions, and solving for far-field radiation patterns, input impedance, and S-parameters across the operating band. Conducting these simulations prior to fabrication allowed the antenna design to be validated and refined without the time and cost of iterating on physical prototypes. HFSS results are well-established in the RF engineering literature as a reliable basis for antenna design decisions.

MathWorks MATLAB: Numerical Verification and Graphing

MATLAB was used in two capacities during HALoN's development: as an independent verification environment for the Python simulator's numerical outputs, and as a tool for generating a subset of the performance figures presented in this report [30]. Key performance metrics, including packet delivery ratio, end-to-end latency distributions, and Q-table convergence behavior, were reproduced in MATLAB independently of the Python pipeline. Consistency between the two environments provided confidence that the simulation results were numerically correct: any discrepancy would have indicated an implementation error in the primary simulator.

Chapter 12

Hazards

12.1 Potential Hazards

A formal Student Project Hazard Assessment was completed in compliance with Santa Clara University Environment, Health and Safety (EHS) protocols prior to the initiation of any project work. All activities were conducted in SCDI 4027.

12.1.1 Batteries

Lithium polymer (LiPo) cells were used to supply power to the LoRa radio module, associated firmware logic, and the antenna subsystem during testing. Primary failure modes include thermal runaway, electrolyte leakage, and combustion under conditions of overcharge, over-discharge, mechanical deformation, or short-circuit. Risk was mitigated through strict adherence to manufacturer datasheets, operation within rated voltage and current envelopes, and storage in approved lab containers. LiPo Battery Safety Training was completed through the SCU Maker Lab prior to battery handling.

12.1.2 Radio Frequencies and Microwaves

The project involved the design, fabrication, and active testing of custom feed network operating at 900 MHz within the LoRa ISM band. RF exposure risk arises from both near-field and cumulative far-field exposure during antenna characterization and packet transmission trials. Secondary risk involves unintentional out-of-band emissions or co-channel interference with licensed or unlicensed systems operating in the 900 MHz band. Controls included operation exclusively within the RF Lab, compliance with FCC Part 15 power limits and LoRaWAN duty-cycle restrictions, and adherence to antenna datasheet specifications

12.1.3 Power Tools Operation and Mechanical Foundation

A wooden testbench platform was constructed to fixture the node hardware during integration and range testing. Fabrication required the use of power cutting and shaping tools, introducing risks of laceration, electric shock, and exposure

to airborne particulates. Controls included mandatory PPE (safety glasses, appropriate footwear and clothing), adherence to tool-specific operating procedures, and completion of lab-specific training before use of any shop equipment.

12.1.4 Soldering

Antenna feed connections and RF module integration required manual soldering of components to Adafruit Feather 32u4 boards. Soldering operations introduce risk of thermal burns from the iron tip or molten solder, and inhalation of flux fumes containing rosin and potentially activated compounds. Controls included use of a designated fume-extracted soldering station, appropriate PPE, and completion of lab-specific soldering training prior to any assembly work.

Chapter 13

Societal Issues

13.1 Ethical

The analysis that follows applies the Markkula Ethics Center's Framework for Ethical Decision Making, its Ethical Toolkit for Engineering Practice, and the professional ethical standards of the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) to HALoN's three architectural layers. For each layer, the analysis proceeds in a three pronged structure: first, an identification of the ethical lenses most applicable to that layer and why they are most relevant; second, an examination of how HALoN's design choices respond to these ethical considerations; and third, a dedicated ethical risk sweep, a systematic account of remaining risks, outstanding obligations, and conditions under which the design could produce harm rather than benefit. Following this layer-by-layer analysis, we address conditions under which HALoN might fail, as well as the potential uses of the system by actors whose values conflict with those we have designed it to consider.

The pursuit of engineering, although often presented as morally neutral, is far from such: the product of our work is impactful, the process of it– conception, design, iteration– even more so. While technical systems are often framed in terms of mathematics and technical theory as though these are objective truths, every engineered system embodies the decisions of the engineers who have had a hand in its making. In her article, "Computer Systems: Moral Entities But Not Moral Agents," Dr. Deborah Johnson states: "the very idea of technology is the idea of things that are human-made... technology is humanity at work." In software and networking systems in particular, these human-made judgments are amplified by scale, thus shaping the behaviors, opportunities, and vulnerabilities of entire populations. Therefore, the questions that engineers since time immemorial have had to ask are whether a system can be built, whether it should be built in a particular way, and who bears the consequences when it is not built responsibly.

In producing artifacts of particular functionalities, engineers and designers are facilitating particular behaviors. When an engineer selects a hardware platform or determines system behavior under particular conditions, they make choices that carry implications for values such as access, fairness, autonomy, and dignity, whether they realize it or not. The choices made for our thesis project, High Altitude Long-Range Networking (HALoN) for Crisis Communi-

cations, also affect the capacity of individuals to participate meaningfully in communication infrastructure on which an increasing proportion of contemporary life depends. In developing HALoN, we are committed to ensuring equitable access to communications for vulnerable communities in periods of crisis, but understand we must design our solution with these stakeholders and their needs in mind. In doing so, we acknowledge that, sometimes, human values conflict, even amongst themselves (security and freedom, honesty and kindness, individualism and collectivism, etc.). The values we have centered, and the decisions we have made around them, are a reflection of us, the HALoN development team, and our moral priorities.

13.2 Social

Networking technologies illustrate clearly the value-bearing nature of technical design. The TCP/IP stack is often discussed as a neutral abstraction, with clean separation across physical, network, transport, and application layers. Yet each layer embeds choices about access, reliability, privacy, and priority that have real consequences for users, consequences that are not distributed evenly. Decisions made at the physical layer constrain possibilities at layers above; implementation of the internet layer determines whose voices can be heard and when; design choices at the transport layer shape what users can reliably communicate under crisis conditions; at the application layer, decisions further shape how users interact with the system and what information is ultimately conveyed.

HALoN is a crisis communication system designed to provide temporary, rapidly deployable connectivity in environments where traditional infrastructure has failed or is unavailable. In other words, it is designed to operate as a minimum viable communication network required to support human coordination, safety, and dignity in moments of disruption. At a more technical level, we maintain the notion of decentralization in the topology of the network, as HALoN uses a peer-to-peer mesh topology, without gateway or server nodes. In other words, no one node is entirely dependent on any other in order to communicate, ensuring that the network maintains the values of resilience and fairness: all nodes are treated equally. Thus, HALoN is not a neutral artifact. We have codified into the system a consideration of the circumstances of our potential users, the implications of their usage, and the ethics that surround it all, as documented here.

The common good, as described within the Markkula framework, is concerned with the conditions that allow communities to function in a way that benefits all members, especially those who are underserved by the status quo. Care ethics, in contrast, centers relationships, interdependence, and the obligation to respond to the needs of others with attentiveness, responsibility and compassion. The common good and care ethics lenses operate across the boundaries between layers because the values they articulate are not specific to any one technical function. They require that the system, taken as a whole, expands access to communication in a way that neither reproduces existing inequities nor abstracts away the human needs of those meant to be served. For this reason, while the analysis that follows moves

layer by layer, it does so with these two ethical lenses held constant.

The physical layer of a network may seem to be the layer most abstracted from ethical consequence. It is indeed the layer where the physicality of a technical artifact is developed–the transistors, electromagnetic waves, electronic chips, antennae, amongst a variety of other components– but this does not free it from the ethical considerations of the layers above, nor its own. The ethical lenses most applicable at the physical layer are those of the common good and of justice. The common good lens asks whether a design serves the interests of the community as a whole, with particular attention to the members of that community who are least served by the status quo. In a disaster context, this requires examining whether the physical layer design produces coverage precisely in the locations where communication infrastructure is most likely to have failed. Is it where the most vulnerable populations are most likely to be found? Or does it replicate, in its coverage pattern, the same hierarchies of access that characterize typical telecommunications? The justice lens asks whether the design's benefits and burdens are distributed equitably, whether the physical choices that make the system accessible to some users exclude others, and if so, if these decisions are ethically justifiable. This includes considerations of who is able to access the network and who bears the costs of its deployment (in terms of resource allocation, environmental impact, risk, etc.). It also requires questioning what tradeoffs are being made in the name of efficiency or performance, and whether those tradeoffs systematically disadvantage certain groups.

The high-altitude balloon deployment mechanism addresses one of the most consistent features of disaster geography: that the physical landscape disrupts communication in exactly the places where communication is often most urgently needed. Elevated nodes extend line-of-sight coverage over terrain features (mountains, dense vegetation, collapsed structures, flooded valleys, etc.) that ground-level deployment cannot overcome without fixed infrastructure. The populations most likely to be isolated by these terrain features are, with notable regularity, the populations that have been historically underserved by telecommunications infrastructure, due to this very geography. A design that achieves coverage specifically in these locations takes seriously the distributive dimension of the common good, asking not only how much coverage the system provides in aggregate, but for whom. For the purposes of testing HALoN, we did not receive permission to deploy our network nodes on balloons, but this is the ideal deployment mechanism, and the one that we have proposed.

The ethical lens of justice is the most directly applicable at the internet layer. Our scheduling protocol allocates a shared resource, the per-node transmission queue, across competing messages originating from users and relay nodes operating under energy scarcity and mesh constraints. Multiple scheduling strategies are technically viable for a given network configuration, and the selection among them reflects, explicitly or otherwise, a set of priorities about whose transmissions receive preferential access and under what conditions. In a network designed for post-disaster or infrastructure-denied communication, these priorities ought to best support equitable processing of communications.

HALoN's scheduling approach emphasizes time-slot and scheduled transmission fairness among nodes under the

specific constraints of a LoRa mesh. The design prioritizes the equitable distribution of transmission opportunities by structuring when nodes are able to transmit, reducing the likelihood that any single node can dominate access to the network simply by virtue of its location, processing speed, or any alternate factors. Rather than allocating the channel directly, the scheduling architecture coordinates access through per-node timing and queue management, using time-division multiplexing to reduce contention and collisions. In effect, the system enforces a shared timing logic that prevents repeated overlap, distributing the cost of waiting across participants in a more controlled and consistent way. In a mobile balloon-assisted topology, where link availability will fluctuate, this distributional property is especially important.

To maximize the common good, the algorithm penalizes packets with high retransmission counts (Rp), poor destination link qualities (Lp), and larger packet sizes (Zp). This prevents monopolization of the transmission channel by any one message, ensuring that the system serves all constituents fairly. However, to promote justice and protect individual rights, the algorithm offsets these efficiency penalties through Message Priority (Pp) and Message Age (Ap). The longer a message remains undelivered, the higher its score becomes, acting as a safeguard against starvation and reducing the risk that individuals with poor network links are persistently disadvantaged. HALoN's scheduler explicitly reacts to transmission failures by incorporating acknowledgment feedback, allowing nodes to detect unsuccessful deliveries and selectively retransmit messages: this improves the likelihood of successful communication under dynamic conditions.

Community review processes, in which representatives of the populations for whom the system is intended participate in evaluating whether it serves their needs and under what conditions, must be incorporated into future design iterations.

13.3 Political

HALoN operates in the 900 MHz ISM band, which is unlicensed for operation in North America under Part 15 of the FCC regulations. This choice is, perhaps, more accurately described as a design constraint that aligns with our emphasis on equitable access. A system designed to operate in licensed spectrum requires its operators to obtain regulatory approval, a process that can involve significant financial and administrative overhead. These requirements are not prohibitive for large institutions like telecommunications companies, government agencies, or well-resourced NGOs with established regulatory relationships, but they represent a meaningful barrier for the community organizations, volunteer networks, and local emergency response teams that are, in practice, most likely to be present and capable in the immediate aftermath of a natural disaster.

Equally important to address: obtaining a private spectrum is expensive, and thus, in practice, these expenses are passed down to our users. As students in a research lab, we did not have access to a unique band, but had no need

for one, as the use of an unlicensed band supports the distribution of deployment authority more broadly, making the act of establishing network coverage something that does not require private institutional permission as a precondition (like use of Starlink would, for example).

HALoN operates without reliance on centralized infrastructure, like cellular towers, existing service providers, or even the internet, and without need for a license. Indeed, we hope that HALoN will serve the needs of a civilian populace when these structures have failed, where existing systems are either damaged, inaccessible, or were never present to begin with, like in the context of natural disasters, remote regions, and environments lacking infrastructure. In removing these dependencies, the system attempts to redistribute the capacity to establish communication networks to smaller organizations, volunteer responders, and local communities.

Technologies deployed on shared infrastructure do not operate only in the hands of their intended users. The design choices that make a system accessible and deployable by legitimate actors may, and frequently do, make it accessible to actors whose purposes are at variance with the system's intended function. The most consequential adversarial use case for HALoN is passive surveillance. HALoN transmits over unlicensed radio spectrum in cleartext. Any actor in possession of a compatible LoRa receiver can receive, read, and log every message transmitted on the network, including sender identifiers, recipient identifiers, priority levels, free-text content, and GPS coordinates, without interacting with the network, without being known to its operators, and without taking any action detectable by network participants. In a politically stable environment with a low-risk user population, the practical consequences of this exposure may be limited. In a deployment context involving displaced populations, politically vulnerable communities, or individuals for whom location information is sensitive, the consequences are substantially more serious. A comprehensive log of who is communicating, where they are, and where they are moving represents information that could be used to harm the very populations the system is intended to serve.

The system was not designed to enable this use. It was designed in a way that makes this use trivially achievable, and once deployed, it will produce these capabilities for monitoring actors regardless of the designers' intent. Those choices pursue genuine values of access and equity and should not be abandoned without strong justification. The potential for passive surveillance brings with it an argument for implementing end-to-end encryption for message payloads. This measure is technically feasible on the Feather 32u4 platform, and something we hope to implement. Its absence from the current design represents an obligation whose discharge is a precondition of deployment in any context involving populations for whom location information or message content is sensitive.

Another potential adversarial use case is message injection. HALoN does not currently implement sender authentication, with the consequence that any actor with compatible hardware can transmit messages on the network claiming origin from any node. In a crisis coordination scenario, a false message claiming to originate from a legitimate network participant can divert response capacity from genuine needs, create confusion about the state of the affected area, or delay assistance to persons requiring it. The harm this enables is not a technical nuisance; in the scenarios for which

HALoN is designed, it could have direct consequences for the allocation of limited emergency response resources. A lightweight message authentication scheme, in which each message carries a cryptographic signature that receiving nodes can verify, is technically achievable and should be treated as a prerequisite for deployment rather than a deferred enhancement.

13.4 Economic

Furthermore, in optimizing for short, emergency messages, we were able to use lower-end hardware to implement the network, in pursuit of making the system more affordable and accessible to communities that might be more vulnerable in times of crises. The hardware platform, Adafruit Feather 32u4 boards with integrated LoRa radios, was selected on criteria that include, but are not reducible to, technical performance. The Feather 32u4 is available through consumer electronics channels, is compatible with the Arduino software ecosystem, and is supported by an extensive body of freely available documentation and community knowledge. Each board can be purchased for under $30, making the costs non-prohibitive compared to other solutions. The lower cost of the board also makes replacement of failed boards inexpensive and simplifies code migration to newer versions of hardware platforms during upgrades. These characteristics matter ethically because they bound the effective user base of the system. A platform available only through specialized suppliers, or requiring proprietary development tools, or lacking accessible documentation, concentrates the capacity to deploy and maintain the system in the hands of those with access to specialized supply chains and technical education. Again, the design consideration of decentralization we have prioritized with HALoN was chosen in part to resist dependencies and hierarchies like this.

The utilitarian ethics lens is also applicable at the internet layer in a way that complements the justice and rights analyses. A scheduling protocol that causes battery-constrained nodes to transmit during predictable periods of high contention, consuming energy without delivering messages, is a protocol that fails, in a specific technical sense, to serve the interests of the nodes and users it governs. There is an account of integrity in the design that requires that the protocol allocate resources fairly, and that it allocates them in ways that are genuinely useful to the nodes that depend on it.

13.5 Health and Safety

The ACM Code of Ethics' requirement that computing professionals contribute to society and human well-being, and the IEEE Code's requirement that engineers hold paramount the safety, health, and welfare of the public, apply at the physical layer with particular directness. A physical layer design that is technically sophisticated but practically inaccessible to the organizations best positioned to deploy it in a crisis would satisfy neither obligation in any meaningful sense.

The rights lens applies in a related way at the internet layer. At the very minimum, we argue that users have a right not to be arbitrarily interfered with in their attempt to communicate, meaning the scheduling protocol must not systematically block or suppress their messages for reasons unrelated to legitimate network constraints. In other words, all nodes must be treated alike, as must all messages of the same priority. Furthermore, given the crisis-centric context in which this system is designed to operate, equitable opportunity to transmit information engages with the rights lens, since this information may be tied in turn to a right to safety or security.

Within this structure, the queue learning message scheduling algorithm at each node plays a critical role in determining how individual messages are ordered for transmission. Because each node operates under partial and shifting knowledge of the network, the algorithm adapts to local conditions such as congestion, retransmission history, and message backlog (aka queue length). This allows the node to make more informed decisions about which messages to prioritize, rather than relying on a static or purely first-come-first-served policy. Importantly, this adaptive behavior helps prevent persistent starvation of certain messages while still responding to real-time constraints, reinforcing the broader goal of fairness at the system level.

Traditional scheduling approaches, such as First-Come, First-Served (FCFS) or Shortest Job First (SJF), offer procedural simplicity but fail to dynamically balance the competing ethical demands of a crisis network. FCFS respects strict procedural fairness but risks catastrophic channel congestion, while SJF maximizes throughput but systematically starves users attempting to send larger messages, regardless of intent. Furthermore, neither option is intelligent enough to adapt to the dynamicity inherent to many crisis environments. It is in resolution of this conflict that we implemented the aforementioned queue-learning scheduling algorithm. At every transmission window, the scheduler evaluates each packet based on a six-component state vector, where each vector factors into the final decision.

A large risk concerns the interaction between scheduling and the dynamic topology changes produced by balloon drift and ground-node mobility. Algorithms calibrated for a relatively stable topology may perform poorly when a high-altitude relay drifts out of range. During the period of topology adaptation, messages scheduled to transmit through the impacted nodes may experience delivery failures that the scheduling protocol is not yet fully positioned to recover from independently. Prolonged communication blackouts during mobility events undermine the system's purpose, which is to allow for connection during these very communication blackouts.

Furthermore, a significant risk concerns the inherent tension between network utility and individual crisis needs. The system deliberately penalizes packets larger than a particular threshold to keep the channel clear and ready for efficient use. However, in a disaster, complex messages (such as detailed medical supply requests or multi-point coordinate routes) are inherently larger. The system may systematically deprioritize complex coordination efforts in favor of simpler, shorter messages, perhaps going as far as influencing the type of disaster response the network can facilitate. We, however, believe that the decisions we made around accessibility, decentralization and fairness which result in this behavior, still stand, but acknowledge their potential impact.

The transport layer governs the end-to-end reliability of communication: whether a message reaches its destination across a multi-hop LoRa mesh; whether the sender is informed of delivery or failure; and whether the system's behavior under degraded conditions is transparent to the people depending on it. The ethical considerations at this layer center on the obligations the system incurs toward the individual user in the moment of transmission. These are obligations of honesty, accuracy, and what the Markkula Center Framework's rights lens identifies as the legitimate claim of persons to information that is material to their own decisions and their own safety. The rights lens is the primary applicable ethical lens at the transport layer because the transport layer's function is, fundamentally, to ensure that a message either has arrived or has not, and that the sender will be accurately informed of the outcome. In a general-purpose communication system, the failure to keep this promise is a matter of user experience. However, in a system designed for disaster response and life-safety contexts, it is even more so. A person who has transmitted an emergency message, attempting to communicate their location or need for assistance, has a right to know whether that message was received.

HALoN implements a lightweight acknowledgment scheme and bounded retransmission logic as the primary mechanisms to ensure users know whether a message was delivered. Upon successful receipt of a message across the mesh, the receiving node transmits a brief acknowledgment packet to the sender. The energy cost of this is non-negligible, but our decision to implement this rests on the rights analysis: the user's claim to accurate delivery feedback in a life-safety context has moral priority over the engineering convenience of a simpler or more power-efficient protocol that would leave the sender in silence. This mechanism ensures that the system does not offer the illusion of connectivity, but rather a verified account of it.

Retransmission logic operates through a bounded process. When an acknowledgment is not received within a specified window, the sending node retransmits the message, but after a defined number of attempts, the system reports delivery failure rather than continuing indefinitely. This bounding reflects a commitment to accuracy over false reassurance. An unbounded retransmission process would offer the user no reliable information about the state of their message, creating a prolonged period of uncertainty. A bounded process that reports failure clearly, however unwelcome that report, provides information that is accurate and that serves the user's capacity to make informed decisions. We are choosing to prioritize honesty over the appearance of persistence.

Message fragmentation handles payloads that exceed the maximum packet size of the LoRa physical layer. Fragments are transmitted sequentially and reassembled at the destination. In the event of partial delivery, the current design discards the incomplete message to avoid misleading recipients. This conservative choice reflects a judgment that, in a crisis context, a partial message may be potentially harmful. An emergency coordinator who acts on partial location data or truncated instructions may be worse off than one who received no message at all. Discarding incomplete fragments is therefore an act of protecting the recipient from misinformation. In this context, data integrity is a moral requirement for fairness and safety.

The acknowledgment scheme, while correctly designed in principle, introduces a failure mode where acknowledgment packets are themselves subject to loss. When an acknowledgment packet is lost, the sender receives a false negative delivery status: it is informed that its message was not received when in fact it was. A false negative delivery status is a less serious harm than a false positive, as it leads to redundant transmission rather than misplaced confidence in a message that did not arrive. It is, however, a harm nonetheless, psychologically or otherwise, to a user whose usage of the system impacts their views on their own safety.

The bounded retransmission threshold introduces a calibration question that has ethical weight. A threshold appropriate for moderate channel congestion may produce premature failure reports under heavy congestion for messages that might have been delivered with additional attempts. Conversely, a threshold set too high extends the period during which the sender remains uncertain about delivery status. Whether the current threshold is appropriately calibrated to the range of conditions the system will encounter, including balloon drift and relay intermittency, is a question that requires field testing to answer with accuracy, and is something we cannot concretely determine without an understanding of the crisis landscape.

The transport layer currently operates without end-to-end encryption. Message payloads, including free-text content and GPS coordinates, are transmitted in cleartext and are readable by any actor in possession of a compatible LoRa receiver. The harm this creates is most clearly visible at the application layer, but it originates at the transport layer, where payload handling is defined. The absence of encryption represents an outstanding design obligation whose discharge is a precondition of responsible deployment in any context involving populations for whom location information or emergency status is sensitive. Addressing this gap is a requirement for meeting our duty of care toward users in life-safety scenarios, and is a direction for future iterations of HALoN.

The most probable single mechanism of potential ethical failure for HALoN does not require malicious intent or technical incompetence. It requires only the well-documented tendency to represent a system's capabilities based on performance under favorable conditions while underweighting the degradation that is likely to occur under the conditions that will actually prevail in deployment. HALoN is a research prototype whose delivery success rates, acknowledgment reliability, and routing performance have been evaluated primarily under controlled laboratory conditions that do not replicate the spectrum congestion, node instability caused by balloon drift, energy management cycles, and accumulated small failures of real deployment in a real disaster environment. The gap between controlled-condition performance and realistic-deployment performance is a standard feature of prototype development; it becomes an ethical problem when controlled-condition performance is communicated to potential deploying organizations as though it were representative of what they should expect in practice.

13.6 Manufacturability

The system is built on a Long Range (LoRa) based mesh network, composed of low-cost, low-power nodes capable of transmitting short messages over long distances. These nodes may be deployed on the ground or, in an ideal scenario, elevated using high-altitude balloons to extend line-of-sight coverage across terrain that would otherwise obstruct communication. The system is able to achieve low power operation through low data-rate communication, lightweight protocols, and intermittent transmission patterns.

The hardware platform, Adafruit Feather 32u4 boards with integrated LoRa radios, was selected on criteria that include, but are not reducible to, technical performance. The Feather 32u4 is available through consumer electronics channels, is compatible with the Arduino software ecosystem, and is supported by an extensive body of freely available documentation and community knowledge. The lower cost of the board also makes replacement of failed boards inexpensive and simplifies code migration to newer versions of hardware platforms during upgrades. These characteristics matter ethically because they bound the effective user base of the system. A platform available only through specialized suppliers, or requiring proprietary development tools, or lacking accessible documentation, concentrates the capacity to deploy and maintain the system in the hands of those with access to specialized supply chains and technical education.

13.7 Sustainability

The system is able to achieve low power operation through low data-rate communication, lightweight protocols, and intermittent transmission patterns. HALoN is not designed to replicate the internet nor to support high-bandwidth applications and continuous connectivity (e.g., streaming). We have intentionally optimized for the reliable transmission of short text messages and positional data under severe resource limitations; this is, ultimately, a temporary system for emergency communications. Although this decision could be viewed as a constraint, it is the reflection of a set of core design principles. In a disaster context, the ability to send a short SOS message or share coordinates is often more valuable than the ability to transmit large amounts of data.

LoRa's low bandwidth is a constraint that we accepted deliberately, as it so happened to best align with our needs: reliable delivery of short text messages and positional data constitute the minimum viable communication for life-safety purposes. Optimizing for this use case serves a wider population more equitably than optimizing for richer communication features for a narrower one, so it was a feature that we were happy to work with.

The battery capacity of LoRa nodes, while favorable compared to many wireless technologies, is not unlimited, which is a tradeoff we were willing to work with. Nodes operating continuously in a sustained disaster response scenario will exhaust their batteries, but given that HALoN is a temporary system with low energy usage, we do not anticipate it to be an issue of note. That said, the design's current documentation does not provide deploying organiza-

tions with sufficiently specific guidance on expected operational lifetimes under various conditions. We acknowledge that this guidance is necessary for organizations to plan battery replacement or supplemental power arrangements, so this represents an outstanding documentation obligation.

13.8 Environmental Impact

The selection of an operating frequency is not simply a technical optimization due to signal propagation characteristics. It is simultaneously a determination of who may operate the system and under what regulatory conditions. The selection of a hardware platform determines not only computational capacity but the practical accessibility of the system to organizations that lack specialized procurement relationships or technical supply chains. The high-altitude deployment mechanism, whether nodes are mounted on towers, carried by vehicles, or elevated on balloons, etc., determines what geography the system serves and, by extension, which populations are within its reach.

The 900 MHz ISM band is shared. Under conditions of high device density, which are of high probability in the urban and semi-urban disaster scenarios where HALoN is most likely to be deployed, interference from consumer electronics, industrial equipment, and other emergency communication systems operating in the same band can degrade HALoN's reliability substantially. Likewise, there is a similar risk that HALoN may be the disruptor of other communications in the shared band. This is not a failure mode that testing in low-density environments will reveal, and therefore, it has not been one that we have been able to test thoroughly, but we acknowledge the possibility.

The battery capacity of LoRa nodes, while favorable compared to many wireless technologies, is not unlimited. The design's current documentation does not provide deploying organizations with sufficiently specific guidance on expected operational lifetimes under various conditions. We acknowledge that this guidance is necessary for organizations to plan battery replacement or supplemental power arrangements, so this represents an outstanding documentation obligation.

13.9 Usability

The system as currently designed requires some technical knowledge to deploy. The physical setup of nodes and the initialization of the network are not tasks that can be reliably performed by non-informed personnel. This does represent a genuine barrier to deployment by some of the community organizations the system is intended to serve, and it is a barrier that our user documentation addresses.

Perhaps the most probable source of real harm is the well-intentioned but inadequately informed deploying organization. A system designed for accessibility will in some cases be deployed by organizations that lack the expertise to understand its limitations, that have not prepared complementary communication strategies for when the system fails, and that have communicated its capabilities to affected populations in terms that exceed what the system can

reliably deliver. The harm produced by this failure mode requires no bad intentions; it requires only the combination of genuine desire to help with insufficient understanding of the conditions under which the system will and will not perform.

Documentation of HALoN's limitations must be included in any materials communicated to potential deployers with sufficient specificity that the conditions under which performance degrades are understandable for non-specialists.

Moving from a deterministic schedule to a reinforcement learning model introduces a distinct set of ethical risks. The primary risk is algorithmic opacity. Under extreme congestion, a user's life-safety message may be repeatedly delayed by the queue-learner for various issues (e.g., the local link quality is poor and the packet is penalized). While the Age (Ap) metric is designed to eventually force the message through, the sender has limited visibility into why their transmission is being delayed.

13.10 Lifelong learning

A structured training requirement for deploying organizations, including explicit evaluation of whether the organization understands the system's failure modes and has a plan for complementary communication strategies, must be treated as a precondition of deployment rather than an optional supplement. Community review processes, in which representatives of the populations for whom the system is intended participate in evaluating whether it serves their needs and under what conditions, must be incorporated into future design iterations.

The ACM Code of Ethics requires that computing professionals be honest about limitations and potential problems; the IEEE Code requires honesty and realism in stating claims based on available data. These limitations are the features of a research prototype, and we hope that this honest acknowledgment is an exercise of the professional and ethical obligations that the design process incurs.

Whether the current threshold is appropriately calibrated to the range of conditions the system will encounter, including balloon drift and relay intermittency, is a question that requires field testing to answer with accuracy, and is something we cannot concretely determine without an understanding of the crisis landscape.

13.11 Compassion

Care ethics centers relationships, interdependence, and the obligation to respond to the needs of others with attentiveness, responsibility and compassion. The notions of the common good and care ethics provide a foundation for understanding HALoN that extends beyond any single layer of its architecture. While other ethical lenses are applied more directly within specific technical contexts, these two lenses, in our view, are a precondition to the entire system.

The virtue ethics lens is applicable at the transport layer in a way that complements the rights analysis. A system that provides accurate feedback on its own performance, including accurate notification of failure, embodies a form of

integrity that is not reducible to utilitarian calculation. There is a consequentialist argument for accuracy: accurate failure notification enables users to seek alternative communication paths (such as a retransmission) and thereby improves user outcomes. But there is also an argument that is not consequentialist, that a system designed to serve people in extremity has an obligation to be honest with them about what it can and cannot do, independent of whether that honesty produces better outcomes in every case. The design of HALoN's transport layer proceeds from the determination that this obligation is real and binding.

In the Jesuit tradition of engineering education, the pursuit of technical excellence is inseparable from the responsibility to pursue what is good. As we have been taught, the head, the heart, and the hands must work in tandem: to separate engineering from ethics is to sever the head and the hands from the heart. Anchoring HALoN's architecture in human dignity, equity, and care is therefore a commitment to the Jesuit values of our education, to the duty of the engineer, both the professional and the personal.

The analysis presented in this document rests on the premise that the design of a technical system is inseparable from the values it embeds, that our obligation as engineers to examine those values thoroughly belongs within the design process. HALoN is a system designed in pursuit of these goals: access to communication infrastructure for populations underserved by existing systems, equity in the distribution of network reliability, and the capacity of communities in crisis to communicate their needs and coordinate their response. The analysis at each layer has attempted to examine how well the design's choices actually serve those values: where our design succeeds, where it falls short, and what our outstanding obligations are.

Our system design embodies a coherent and defensible set of value commitments but also carries limitations: its current absence of end-to-end encryption and its gap between simulated performance and field-validated performance under realistic conditions, among others. We are proud of the work that we have developed but know that more must be done before the system can be deployed responsibly in the contexts for which it is intended.

Our personal commitment through HALoN's design is to serve, with care, honesty, and intention, the people who need it most; as engineers educated in the Jesuit tradition, we also hold a commitment to the inseparability of excellence from responsibility. And it is, as our analysis has attempted to show, what must be addressed at the three implicated layers of the architecture, in every design decision, and in the system as a whole. We owe this diligence to future users of HALoN, who may ultimately depend deeply on what we have built.

Chapter 14

Conclusion

HALoN explores how a communication network can be built when you explicitly remove the assumptions that traditional networking relies on: stable infrastructure, global synchronization, and centralized control. Instead, the system is grounded in a set of tightly constrained but physically realizable components like low-power LoRa radios, microcontroller-based nodes, and locally computed routing and scheduling decisions.

At the core of the implementation is a link-state routing protocol adapted to severe resource constraints. Each node maintains a locally constructed view of network topology derived from RSSI-based link cost estimation and periodic neighbor updates. This information is used to compute shortest paths via a lightweight Dijkstra implementation, producing a next-hop table that is continuously updated as link conditions change.

On top of this routing layer, HALoN implements a per-node transmission scheduling system that directly shapes network performance under congestion. Baseline schedulers (FCFS and SJF) are used to characterize system behavior under simple deterministic policies, while a Q-learning-based scheduler introduces adaptive prioritization driven by observed latency, retransmissions, queue pressure, and link quality. The key design constraint here is that all learning and decision-making occurs online and locally, without global state aggregation. This forces the policy to operate under partial observability, mirroring the constraints of the physical deployment.

Because full-scale physical deployment is not feasible, a simulator was developed to reproduce the system at scale while preserving the same packet-level semantics as the firmware implementation. The simulator models node placement in Euclidean space, RSSI-dependent link formation, stochastic traffic injection, and queue-based transmission behavior. This allows evaluation across conditions that are otherwise inaccessible on hardware, including higher node densities, sustained congestion, and controlled variation in traffic patterns.

System performance is evaluated using packet delivery ratio, end-to-end latency, and queue waiting time per packet class. Each metric captures a different structural aspect of network behavior. Delivery ratio measures the integrity of the routing and forwarding layer under loss and contention. End-to-end latency captures the combined effect of routing path length, medium access delays, and retransmissions, making it a direct indicator of overall system responsiveness.

Queue waiting time, disaggregated by packet class, isolates the behavior of the scheduling layer and makes visible how prioritization policies distribute delay across heterogeneous traffic. Taken together, these metrics prevent a single aggregate statistic from masking trade-offs between efficiency, fairness, and responsiveness.

The results of this system are the result of constrained design space exploration: what is achievable when routing, scheduling, and medium access are all forced to operate under local information and limited computation. In this setting, performance is not defined solely by throughput or latency minimization, but by the system's ability to remain functional, adaptable, and interpretable under failure-prone and partially connected conditions.

Chapter 15

Acknowledgments

The following acknowledgments are made by Sahana Prasanna Chandramohan:

My work for this thesis is first and foremost dedicated to my loving parents, who are everything. You are the reason I love and pursue what I do: thank you, Amma and Appa. It is all for you, always.

No acknowledgment of mine would be complete without recognizing my dear younger brother, my light and soul– Aran, I look forward to reading your thesis in a few years' time!

Finally, to my professors, without whom I would not have even known to dream of engineering in the ways I do now: my most heartfelt thank you for your encouragement, guidance and care these past few years.

Chapter 16

Appendices

16.1 Network Simulation Code

```
import matplotlib
matplotlib.use('MacOSX')
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import networkx as nx
import matplotlib.animation as animation
from matplotlib.widgets import Button, RadioButtons, Slider
import numpy as np
import random
import copy
from collections import defaultdict
from enum import Enum

WHITE     = '#FFFFFF'
BLACK     = '#111111'
GREY_LT   = '#F4F4F4'
GREY_MED  = '#DDDDDD'
GREY_DRK  = '#888888'

SAT_RED    = '#E8190C'
SAT_YELLOW = '#F5CB00'
SAT_BLUE   = '#0055FF'
SAT_GREEN  = '#00C244'

LINK_CLR   = '#CCCCCC'
LINK_ACTIVE= '#0055FF'

plt.rcParams.update({
    'font.family'     : 'Helvetica',
    'axes.edgecolor'  : GREY_MED,
    'axes.linewidth'  : 0.7,
    'figure.facecolor': WHITE,
    'text.color'      : BLACK,
})

MAX_LINK_DIST = 3.2
PACKET_SPEED  = 0.15

PACKET_COLORS = [
    SAT_RED, SAT_BLUE, SAT_GREEN, SAT_YELLOW,
```

```
    '#FF6600', '#AA00FF', '#00BBCC', '#FF007F',
    '#007700', '#CC2200', '#0033CC', '#AACC00',
    '#FF4488', '#00AAFF', '#FF8800', '#5500AA',
]

class Retransmission(Enum):
    ZERO = 0; ONE = 1; TWO_PLUS = 2

class LinkQuality(Enum):
    BAD = 0; DECENT = 1; STRONG = 2

class QueueLevel(Enum):
    LOW = 0; MEDIUM = 1; HIGH = 2

class Priority(Enum):
    LOW = 0; MEDIUM = 1; HIGH = 2; CONTROL = 3

class Age(Enum):
    FRESH = 0; AGING = 1; OLD = 2

class Size(Enum):
    SHORT = 0; MEDIUM = 1; LONG = 2

def retransmission_bucket(r):
    if r == 0:  return Retransmission.ZERO
    if r == 1:  return Retransmission.ONE
    return Retransmission.TWO_PLUS

def link_quality_bucket(history):
    if not history: return LinkQuality.DECENT
    rate = sum(history[-10:]) / len(history[-10:])
    if rate < 0.6: return LinkQuality.BAD
    if rate < 0.9: return LinkQuality.DECENT
    return LinkQuality.STRONG

def queue_bucket(q):
    if q <= 2: return QueueLevel.LOW
    if q <= 5: return QueueLevel.MEDIUM
    return QueueLevel.HIGH

def age_bucket(age):
    if age < 2: return Age.FRESH
    if age < 6: return Age.AGING
    return Age.OLD

def size_bucket(s):
    if s <= 64:  return Size.SHORT
    if s <= 512: return Size.MEDIUM
    return Size.LONG

def encode_features(packet, queue_len, st):
    return {
        'retransmission': retransmission_bucket(packet.retransmissions),
        'link'          : link_quality_bucket(packet.success_history),
        'queue'         : queue_bucket(queue_len),
        'priority'      : packet.priority,
        'age'           : age_bucket(st - packet.injection_time),
        'size'          : size_bucket(packet.size),
    }
```

```
100
101 INITIAL_WEIGHTS = {
102     'retransmission': {Retransmission.ZERO: 0.0, Retransmission.ONE: -0.30,
            Retransmission.TWO_PLUS: -0.70},
103     'link'          : {LinkQuality.BAD: -0.50, LinkQuality.DECENT: 0.0, LinkQuality.
            STRONG: 0.50},
104     'queue'         : {QueueLevel.LOW: -0.70, QueueLevel.MEDIUM: -0.30, QueueLevel.
            HIGH: 0.0},
105     'priority'      : {Priority.LOW: 0.0, Priority.MEDIUM: 0.40, Priority.HIGH: 0.70,
            Priority.CONTROL: 1.0},
106     'age'           : {Age.FRESH: 0.0, Age.AGING: 0.30, Age.OLD: 0.70},
107     'size'          : {Size.SHORT: 0.0, Size.MEDIUM: -0.30, Size.LONG: -0.70},
108 }
109
110 class QLearner:
111     def __init__(self, alpha=0.10, gamma=0.90, epsilon=0.10):
112         self.alpha, self.gamma, self.epsilon = alpha, gamma, epsilon
113         self.weights = {k: dict(v) for k, v in INITIAL_WEIGHTS.items()}
114
115     def q_value(self, features):
116         return sum(self.weights[k][v] for k, v in features.items())
117
118     def select_packet(self, queue, queue_len, st):
119         if random.random() < self.epsilon:
120             return random.choice(queue)
121         return max(queue, key=lambda p: self.q_value(encode_features(p, queue_len, st)
                ))
122
123     def update(self, features, reward, next_queue, st):
124         next_q = 0.0
125         if next_queue:
126             next_q = max(
127                 self.q_value(encode_features(p, len(next_queue), st))
128                 for p in next_queue
129             )
130         td = reward + self.gamma * next_q - self.q_value(features)
131         for k, v in features.items():
132             self.weights[k][v] = max(-1.0, min(1.0,
133                 self.weights[k][v] + self.alpha * td))
134
135 nodes = {
136     0: [1.0, 1.0], 1: [2.0, 3.5], 2: [4.0, 0.8],
137     3: [5.5, 3.5], 4: [6.5, 1.5],
138 }
139 node_queues     = {n: [] for n in nodes}
140 node_busy_until = {n: 0  for n in nodes}
141 links           = []
142 routing_tables  = {}
143 active_messages = []
144 completed_stats = []
145 sim_time        = 0.0
146 running         = False
147 show_stats      = False
148 dragging_node   = None
149 scheduling_mode = 'FCFS'
150 ql_agent        = QLearner()
151 next_node_id    = max(nodes.keys()) + 1
152
153 num_packets     = 10
```

```
time_interval   = 5.0
message_data    = []
comparison_results          = {}
comparison_message_template = []

def generate_packets(count, interval):
    global comparison_message_template
    rng  = random.Random(random.randint(0, 999999))
    nids = list(nodes.keys())
    data = []
    for i in range(count):
        src = rng.choice(nids)
        dst = rng.choice([n for n in nids if n != src])
        data.append({
            'time'    : rng.uniform(0, interval),
            'src'     : src, 'dst': dst,
            'size'    : rng.choice([16,32,64,128,256,400,512,1024]),
            'color'   : PACKET_COLORS[i % len(PACKET_COLORS)],
            'sent'    : False,
            'priority': rng.choice(list(Priority)),
        })
    data.sort(key=lambda x: x['time'])
    message_data.clear(); message_data.extend(data)
    comparison_message_template = [copy.deepcopy(d) for d in data]

def regenerate_from_template():
    message_data.clear()
    message_data.extend([copy.deepcopy(d) for d in comparison_message_template])

def update_topology():
    global links
    G = nx.Graph()
    G.add_nodes_from(nodes.keys())
    links = []
    for i in nodes:
        for j in nodes:
            if i >= j: continue
            d = ((nodes[i][0]-nodes[j][0])**2 + (nodes[i][1]-nodes[j][1])**2)**0.5
            if d <= MAX_LINK_DIST:
                links.append((i, j))
                G.add_edge(i, j, weight=d)
    for n in nodes:
        try:
            paths = nx.single_source_shortest_path(G, n)
            routing_tables[n] = {
                d: (p[1] if len(p) > 1 else None)
                for d, p in paths.items()
            }
        except Exception:
            routing_tables[n] = {d: None for d in nodes}

class MessagePacket:
    def __init__(self, data, index, start_time):
        self.pid            = f'P{index}'
        self.index          = index
        self.src            = data['src']
        self.dest           = data['dst']
        self.color          = data['color']
        self.size           = data['size']
```

```
        self.priority       = data.get('priority', Priority.LOW)
        self.current_node   = self.src
        self.pos            = list(nodes[self.src])
        self.target_node    = None
        self.arrived        = False
        self.state          = 'QUEUED'
        self.injection_time = start_time
        self.wait_time      = 0.0
        self.total_trip_time= 0.0
        self.retransmissions= 0
        self.success_history= []
        self._set_next_hop()
        node_queues[self.src].append(self)

    def _set_next_hop(self):
        neighbours = {n for (a,b) in links for n in ([b] if a==self.current_node else
            ([a] if b==self.current_node else []))}
        if self.dest in neighbours:
            self.target_node = self.dest
        else:
            self.target_node = routing_tables.get(self.current_node, {}).get(self.dest
                )
        if self.target_node is None:
            self.arrived = True

    def move(self, dt):
        if self.state == 'QUEUED':
            self.wait_time += dt; return
        if self.target_node not in nodes:
            self.arrived = True; return
        tx, ty = nodes[self.target_node]
        dx, dy = tx - self.pos[0], ty - self.pos[1]
        dist = (dx**2 + dy**2)**0.5
        if dist < PACKET_SPEED:
            self.pos = [tx, ty]
            self.current_node = self.target_node
            if self.current_node == self.dest:
                self.arrived = True
                self.state   = 'ARRIVED'
                self.total_trip_time = sim_time - self.injection_time
                self.success_history.append(1)
                completed_stats.append(self)
            else:
                self.state = 'QUEUED'
                node_queues[self.current_node].append(self)
                self._set_next_hop()
        else:
            self.pos[0] += (dx/dist) * PACKET_SPEED
            self.pos[1] += (dy/dist) * PACKET_SPEED

def process_node_queues(frame):
    for nid in list(nodes.keys()):
        if node_busy_until.get(nid, 0) <= frame:
            q = node_queues.get(nid, [])
            if not q: continue
            if scheduling_mode == 'SJF':
                q.sort(key=lambda p: p.size)
                pkt = q.pop(0)
            elif scheduling_mode == 'Q-Learning':
```

```
                pkt      = ql_agent.select_packet(q, len(q), sim_time)
                features = encode_features(pkt, len(q), sim_time)
                q.remove(pkt)
                reward   = 1.0 - 0.05*(sim_time - pkt.injection_time)
                ql_agent.update(features, reward, q, sim_time)
            else:
                pkt = q.pop(0)
            pkt.state = 'MOVING'
            node_busy_until[nid] = frame + max(20, pkt.size // 10)

def run_headless(algo_name):
    local_queues = {n: [] for n in nodes}
    local_busy   = {n: 0  for n in nodes}
    local_active, local_done = [], []
    local_t, frame = 0.0, 0
    local_msgs = [copy.deepcopy(d) for d in comparison_message_template]
    local_ql   = QLearner()
    dt = 0.05

    class LP:
        def __init__(s, data, idx, t0):
            s.pid, s.index = f'P{idx}', idx
            s.src, s.dest  = data['src'], data['dst']
            s.color, s.size= data['color'], data['size']
            s.priority     = data.get('priority', Priority.LOW)
            s.current_node = s.src
            s.pos          = list(nodes[s.src])
            s.target_node  = None
            s.arrived      = False
            s.state        = 'QUEUED'
            s.injection_time= t0
            s.wait_time    = 0.0
            s.total_trip_time=0.0
            s.retransmissions=0
            s.success_history=[]
            nbrs = {n for (a,b) in links for n in ([b] if a==s.current_node else ([a]
                if b==s.current_node else []))}
            s.target_node = s.dest if s.dest in nbrs else routing_tables.get(s.
                current_node,{}).get(s.dest)
            if s.target_node is None: s.arrived = True
            local_queues[s.src].append(s)

        def move(s, dt, t):
            if s.state == 'QUEUED': s.wait_time += dt; return
            if s.target_node not in nodes: s.arrived=True; return
            tx,ty = nodes[s.target_node]
            dx,dy = tx-s.pos[0], ty-s.pos[1]
            dist  = (dx**2+dy**2)**0.5
            if dist < PACKET_SPEED:
                s.pos = [tx,ty]; s.current_node = s.target_node
                if s.current_node == s.dest:
                    s.arrived=True; s.state='ARRIVED'
                    s.total_trip_time = t-s.injection_time
                    s.success_history.append(1); local_done.append(s)
                else:
                    s.state='QUEUED'; local_queues[s.current_node].append(s)
                    nbrs={n for (a,b) in links for n in ([b] if a==s.current_node else
                         ([a] if b==s.current_node else []))}
```

```
                    s.target_node = s.dest if s.dest in nbrs else routing_tables.get(s
                        .current_node,{}).get(s.dest)
                    if s.target_node is None: s.arrived=True
            else:
                s.pos[0]+=(dx/dist)*PACKET_SPEED; s.pos[1]+=(dy/dist)*PACKET_SPEED

    while frame < 200000:
        for nid in nodes:
            if local_busy.get(nid,0) <= frame:
                q = local_queues.get(nid,[])
                if q:
                    if algo_name=='SJF':
                        q.sort(key=lambda p:p.size); pkt=q.pop(0)
                    elif algo_name=='Q-Learning':
                        pkt=local_ql.select_packet(q,len(q),local_t)
                        feat=encode_features(pkt,len(q),local_t); q.remove(pkt)
                        local_ql.update(feat,1.0-0.05*(local_t-pkt.injection_time),q,
                            local_t)
                    else:
                        pkt=q.pop(0)
                    pkt.state='MOVING'; local_busy[nid]=frame+max(20,pkt.size//10)
        for i,msg in enumerate(local_msgs):
            if not msg['sent'] and local_t>=msg['time']:
                if msg['src'] in nodes and msg['dst'] in nodes:
                    local_active.append(LP(msg,i,local_t))
                msg['sent']=True
        for m in local_active[:]:
            m.move(dt,local_t)
            if m.arrived: local_active.remove(m)
        local_t+=dt; frame+=1
        if all(m['sent'] for m in local_msgs) and not local_active: break

    n_inj = sum(1 for m in local_msgs if m['sent'])
    n_arr = len(local_done)
    pp    = [{'pid':p.pid,'size':p.size,'wait':p.wait_time,'latency':p.total_trip_time
        ,
              'src':p.src,'dest':p.dest,'priority':p.priority.name} for p in
                  local_done]
    return {
        'total_injected': n_inj, 'total_arrived': n_arr,
        'delivery_rate' : n_arr/n_inj if n_inj else 0,
        'avg_wait'      : sum(p.wait_time for p in local_done)/n_arr if n_arr else 0,
        'avg_latency'   : sum(p.total_trip_time for p in local_done)/n_arr if n_arr
            else 0,
        'max_latency'   : max((p.total_trip_time for p in local_done), default=0),
        'min_latency'   : min((p.total_trip_time for p in local_done), default=0),
        'throughput'    : n_arr/local_t if local_t>0 else 0,
        'sim_time'      : local_t, 'per_packet': pp,
    }

def show_comparison_graphs():
    algos = list(comparison_results.keys())
    clrs  = {'FCFS': SAT_BLUE, 'SJF': SAT_RED, 'Q-Learning': SAT_GREEN}

    fig2,axes = plt.subplots(2,3,figsize=(17,9),facecolor=WHITE)
    fig2.suptitle('Algorithm Comparison', fontsize=20, fontweight='bold', color=BLACK,
         y=0.97)
    fig2.subplots_adjust(top=0.88,bottom=0.08,hspace=0.50,wspace=0.35,left=0.06,right
        =0.97)
```

```
377
378     def sa(ax, title):
379         ax.set_facecolor(WHITE)
380         ax.tick_params(colors=GREY_DRK, labelsize=9)
381         ax.spines[['top','right']].set_visible(False)
382         ax.spines['bottom'].set_color(GREY_MED)
383         ax.spines['left'].set_color(GREY_MED)
384         ax.set_title(title, fontsize=12, fontweight='bold', color=BLACK, pad=8)
385
386     bkw = dict(width=0.45, edgecolor=WHITE, linewidth=2, zorder=3)
387
388     metrics = [
389         ('avg_latency', 'Avg Latency (s)',    '{:.2f}',  axes[0,0]),
390         ('avg_wait',    'Avg Queue Wait (s)', '{:.2f}',  axes[0,1]),
391         ('throughput',  'Throughput (pkt/s)', '{:.3f}',  axes[1,0]),
392     ]
393     for key, title, fmt, ax in metrics:
394         vals = [comparison_results[a][key] for a in algos]
395         bars = ax.bar(algos, vals, color=[clrs[a] for a in algos], **bkw)
396         top  = max(vals) if max(vals)>0 else 1
397         for b, v in zip(bars, vals):
398             ax.text(b.get_x()+b.get_width()/2, v+top*0.03,
399                     fmt.format(v), ha='center', fontsize=10, color=BLACK, fontweight='
                        bold')
400         sa(ax, title)
401
402     ax = axes[0,2]
403     vals = [comparison_results[a]['delivery_rate']*100 for a in algos]
404     bars = ax.bar(algos, vals, color=[clrs[a] for a in algos], **bkw)
405     ax.set_ylim(0, 120)
406     for b,v in zip(bars,vals):
407         ax.text(b.get_x()+b.get_width()/2, v+2, f'{v:.0f}%',
408                 ha='center', fontsize=10, color=BLACK, fontweight='bold')
409     sa(ax, 'Delivery Rate (%)')
410
411     ax = axes[1,1]
412     n_pkts = max(len(comparison_results[a]['per_packet']) for a in algos)
413     if 0 < n_pkts <= 30:
414         x = np.arange(n_pkts); w = 0.25
415         for i,a in enumerate(algos):
416             pp   = comparison_results[a]['per_packet']
417             lats = [p['latency'] for p in pp]+[0]*(n_pkts-len(pp))
418             ax.bar(x+i*w-w, lats[:n_pkts], w, label=a, color=clrs[a], edgecolor=WHITE,
                        linewidth=0.5)
419         ax.set_xticks(x)
420         ax.set_xticklabels([f'P{i}' for i in range(n_pkts)], fontsize=7, rotation=45)
421     else:
422         for a in algos:
423             lats=[p['latency'] for p in comparison_results[a]['per_packet']]
424             ax.plot(lats, label=a, color=clrs[a], lw=2.5, marker='o', markersize=3,
                        markeredgecolor=WHITE)
425     ax.legend(fontsize=8, framealpha=0.9, edgecolor=GREY_MED)
426     sa(ax, 'Per-Packet Latency')
427
428     ax = axes[1,2]
429     x  = np.arange(len(algos)); w=0.3
430     ax.bar(x-w/2,[comparison_results[a]['max_latency'] for a in algos],w,
431            label='Max',color=[clrs[a] for a in algos],edgecolor=WHITE,linewidth=2)
432     ax.bar(x+w/2,[comparison_results[a]['min_latency'] for a in algos],w,
```

```
            label='Min',color=[clrs[a] for a in algos],edgecolor=WHITE,linewidth=2,
                alpha=0.45)
    ax.set_xticks(x); ax.set_xticklabels(algos)
    ax.legend(fontsize=8, framealpha=0.9, edgecolor=GREY_MED)
    sa(ax, 'Latency Range (max / min)')

    plt.show()

def on_press(event):
    global dragging_node
    if event.inaxes is not ax_graph: return
    if event.xdata is None: return
    for nid,(x,y) in nodes.items():
        if (event.xdata-x)**2+(event.ydata-y)**2 < 0.20:
            dragging_node = nid; return

def on_motion(event):
    if dragging_node is None: return
    if event.inaxes is not ax_graph: return
    if event.xdata is None: return
    nodes[dragging_node] = [event.xdata, event.ydata]
    update_topology()

def on_release(event):
    global dragging_node
    dragging_node = None

def on_dblclick(event):
    global next_node_id
    if event.inaxes is not ax_graph: return
    if not event.dblclick or event.xdata is None: return
    for nid,(x,y) in nodes.items():
        if (event.xdata-x)**2+(event.ydata-y)**2 < 0.35: return
    nid = next_node_id; next_node_id+=1
    nodes[nid]=[event.xdata, event.ydata]
    node_queues[nid]=[]
    node_busy_until[nid]=0
    update_topology()

def click_start(event):
    global running, show_stats
    if not message_data: generate_packets(num_packets, time_interval)
    running=True; show_stats=False

def click_stop(event):
    global running, show_stats
    running=False; show_stats=True

def click_restart(event):
    global sim_time, running, show_stats, active_messages, completed_stats, ql_agent
    sim_time=0.0; running=False; show_stats=False
    active_messages.clear(); completed_stats.clear()
    for nid in nodes:
        node_queues[nid]=[]; node_busy_until[nid]=0
    for msg in message_data: msg['sent']=False
    ql_agent=QLearner()
    update_topology()

def click_generate(event):
```

```
    click_restart(event)
    generate_packets(num_packets, time_interval)

def click_compare(event):
    global comparison_results
    if not comparison_message_template: generate_packets(num_packets, time_interval)
    comparison_results={}
    for algo in ['FCFS','SJF','Q-Learning']:
        comparison_results[algo]=run_headless(algo)
    show_comparison_graphs()

fig = plt.figure(figsize=(20, 11), facecolor=WHITE)
try:
    fig.canvas.manager.set_window_title('HALoN Network Scheduler Simulator')
except Exception:
    pass

fig.text(0.50, 0.965, 'HALoN     Network Packet Scheduler Simulator',
         ha='center', fontsize=22, fontweight='bold', color=BLACK)
fig.text(0.50, 0.940, 'drag nodes to move      double-click empty space to add node
        topology & routes update live',
         ha='center', fontsize=9.5, color=GREY_DRK, style='italic')

gs = fig.add_gridspec(1, 3, width_ratios=[2.4, 0.85, 0.95],
                      left=0.02, right=0.98, top=0.925, bottom=0.14, wspace=0.18)

ax_graph  = fig.add_subplot(gs[0], facecolor=WHITE)
ax_table  = fig.add_subplot(gs[1], facecolor=GREY_LT)
ax_status = fig.add_subplot(gs[2], facecolor=GREY_LT)

for ax in (ax_table, ax_status):
    for sp in ax.spines.values():
        sp.set_edgecolor(GREY_MED); sp.set_linewidth(0.8)

_refs = []

def _btn(rect, label, cb, fc=GREY_LT, hc=GREY_MED, tc=BLACK):
    a = fig.add_axes(rect)
    b = Button(a, label, color=fc, hovercolor=hc)
    b.label.set_fontsize(9.5); b.label.set_fontweight('bold'); b.label.set_color(tc)
    b.on_clicked(cb)
    _refs.extend([a, b])
    return b

bx, by, bw, bh, gap = 0.02, 0.048, 0.092, 0.042, 0.006
_btn([bx,                by, bw, bh], 'Generate',     click_generate)
_btn([bx+1*(bw+gap),     by, bw, bh], 'Start',        click_start,   '#D6F5DF','#
    A8EBBB','#005C1A')
_btn([bx+2*(bw+gap),     by, bw, bh], 'Stop & Stats', click_stop,    '#FAE8D0','#
    F5C990',SAT_RED)
_btn([bx+3*(bw+gap),     by, bw, bh], 'Restart',      click_restart)
_btn([bx+4*(bw+gap),     by, bw, bh], 'Compare All',  click_compare, SAT_BLUE,'#0033BB
    ',WHITE)

ax_radio = fig.add_axes([0.54, 0.008, 0.15, 0.115], facecolor=WHITE, frameon=False)
radio = RadioButtons(ax_radio, ('FCFS', 'SJF', 'Q-Learning'), activecolor=SAT_BLUE)
for lbl in radio.labels:
    lbl.set_fontsize(10); lbl.set_color(BLACK); lbl.set_fontweight('bold')
def change_mode(label):
```

```
    global scheduling_mode, ql_agent
    scheduling_mode = label
    if label == 'Q-Learning': ql_agent = QLearner()
radio.on_clicked(change_mode)
_refs.extend([ax_radio, radio])

ax_sl1 = fig.add_axes([0.77, 0.096, 0.20, 0.022], facecolor=GREY_LT)
slider_pkts = Slider(ax_sl1, 'Packets', 2, 100, valinit=num_packets, valstep=1,
                     color=SAT_BLUE, track_color=GREY_MED)
slider_pkts.label.set_fontsize(9.5); slider_pkts.label.set_color(BLACK)
slider_pkts.label.set_fontweight('bold'); slider_pkts.valtext.set_fontsize(9.5)

ax_sl2 = fig.add_axes([0.77, 0.060, 0.20, 0.022], facecolor=GREY_LT)
slider_time = Slider(ax_sl2, 'Interval (s)', 1, 60, valinit=time_interval, valstep
    =0.5,
                     color=SAT_GREEN, track_color=GREY_MED)
slider_time.label.set_fontsize(9.5); slider_time.label.set_color(BLACK)
slider_time.label.set_fontweight('bold'); slider_time.valtext.set_fontsize(9.5)

ax_sl3 = fig.add_axes([0.77, 0.024, 0.20, 0.022], facecolor=GREY_LT)
slider_nodes = Slider(ax_sl3, 'Nodes', 2, 20, valinit=len(nodes), valstep=1,
                      color=SAT_RED, track_color=GREY_MED)
slider_nodes.label.set_fontsize(9.5); slider_nodes.label.set_color(BLACK)
slider_nodes.label.set_fontweight('bold'); slider_nodes.valtext.set_fontsize(9.5)

def on_pkts(v):
    global num_packets, sim_time, running, show_stats, active_messages,
        completed_stats, ql_agent
    num_packets = int(v)
    sim_time = 0.0; running = False; show_stats = False
    active_messages.clear(); completed_stats.clear()
    for nid in nodes:
        node_queues[nid] = []; node_busy_until[nid] = 0
    ql_agent = QLearner()
    generate_packets(num_packets, time_interval)

def on_intv(v):
    global time_interval, sim_time, running, show_stats, active_messages,
        completed_stats, ql_agent
    time_interval = v
    sim_time = 0.0; running = False; show_stats = False
    active_messages.clear(); completed_stats.clear()
    for nid in nodes:
        node_queues[nid] = []; node_busy_until[nid] = 0
    ql_agent = QLearner()
    generate_packets(num_packets, time_interval)

def on_node_count(v):
    global next_node_id, sim_time, running, show_stats, active_messages,
        completed_stats, ql_agent
    target  = int(v)
    current = len(nodes)

    if target > current:
        for _ in range(target - current):
            nid = next_node_id; next_node_id += 1
            nodes[nid] = [random.uniform(0.3, 7.8), random.uniform(0.3, 4.8)]
            node_queues[nid]     = []
            node_busy_until[nid] = 0
```

```
601
602     elif target < current:
603         for nid in sorted(nodes.keys(), reverse=True)[:current - target]:
604             del nodes[nid]
605             node_queues.pop(nid, None)
606             node_busy_until.pop(nid, None)
607
608     sim_time = 0.0; running = False; show_stats = False
609     active_messages.clear(); completed_stats.clear()
610     for nid in nodes:
611         node_queues[nid] = []; node_busy_until[nid] = 0
612     ql_agent = QLearner()
613     update_topology()
614     generate_packets(num_packets, time_interval)
615
616 slider_pkts.on_changed(on_pkts)
617 slider_time.on_changed(on_intv)
618 slider_nodes.on_changed(on_node_count)
619 _refs.extend([ax_sl1, slider_pkts, ax_sl2, slider_time, ax_sl3, slider_nodes])
620
621 def _draw_graph(frame):
622     ax = ax_graph
623     ax.clear()
624     ax.set_facecolor(WHITE)
625     ax.set_xlim(-0.5, 8.5); ax.set_ylim(-0.5, 5.5)
626     ax.set_xticks([]); ax.set_yticks([])
627     for sp in ax.spines.values(): sp.set_visible(False)
628     ax.grid(True, alpha=0.07, color=GREY_MED, linewidth=0.7)
629     ax.set_title('Network Topology', fontsize=13, fontweight='bold', color=BLACK, pad
        =6)
630
631     for n1,n2 in links:
632         x1,y1=nodes[n1]; x2,y2=nodes[n2]
633         ax.plot([x1,x2],[y1,y2], color=GREY_MED, lw=2.5,
634                 solid_capstyle='round', zorder=1)
635         mx,my = (x1+x2)/2,(y1+y2)/2
636         d = ((x1-x2)**2+(y1-y2)**2)**0.5
637         ax.text(mx,my+0.14, f'{d:.1f}', fontsize=6.5, ha='center',
638                 color=GREY_DRK, zorder=2)
639
640     for nid,(x,y) in nodes.items():
641         busy_left = max(0, node_busy_until.get(nid,0)-frame)
642         nc = SAT_RED if busy_left>0 else SAT_BLUE
643
644         ax.scatter(x,y, s=1100, c=nc, alpha=0.12, zorder=2, linewidths=0)
645         ax.scatter(x,y, s=600,  c=nc, edgecolors=WHITE, linewidths=3.0, zorder=3)
646         ax.text(x,y, str(nid), ha='center', va='center',
647                 color=WHITE, fontweight='bold', fontsize=14, zorder=4)
648
649         if busy_left>0:
650             ax.text(x, y+0.52, f'TX {int(busy_left)}f', fontsize=7, ha='center',
651                     color=SAT_RED, fontweight='bold',
652                     bbox=dict(boxstyle='round,pad=0.22', fc=WHITE, ec=SAT_RED, lw=0.9,
                            alpha=0.95))
653
654         q = node_queues.get(nid,[])
655         if q:
656             qs = ','.join(str(p.size) for p in q[:4])
657             if len(q)>4: qs+=f'+{len(q)-4}'
```

```
            ax.text(x, y-0.52, f'Q[{qs}]', fontsize=6.5, ha='center', color=BLACK,
                    bbox=dict(boxstyle='round,pad=0.22', fc=SAT_YELLOW, ec=BLACK, lw
                        =0.7, alpha=0.92))

    for m in active_messages:
        if not m.arrived:
            mkr = 's' if m.state=='MOVING' else 'o'
            ax.scatter(m.pos[0], m.pos[1], color=m.color, s=130, marker=mkr,
                       edgecolors=WHITE, linewidths=1.8, zorder=5)
            ax.text(m.pos[0], m.pos[1]+0.30, m.pid, fontsize=7, ha='center',
                    color=BLACK, fontweight='bold', zorder=5)

def _draw_routing():
    ax = ax_table
    ax.clear(); ax.axis('off'); ax.set_facecolor(GREY_LT)
    ax.set_title('Routing Table', fontsize=12, fontweight='bold', color=BLACK, pad=6)

    rt = ''
    for src in sorted(nodes.keys()):
        rt += f'Node {src}:\n'
        for dst in sorted(nodes.keys()):
            if src==dst: continue
            nxt = routing_tables.get(src,{}).get(dst)
            arrow = f'-> {nxt}' if nxt is not None else '  ---'
            rt += f'  dst{dst}: {arrow}\n'
        rt += '\n'

    ax.text(0.06, 0.97, rt, va='top', fontsize=8.5, color=BLACK,
            fontfamily='monospace', transform=ax.transAxes, linespacing=1.35)

def _draw_status():
    ax = ax_status
    ax.clear(); ax.axis('off'); ax.set_facecolor(GREY_LT)

    if show_stats:
        ax.set_title('Results', fontsize=12, fontweight='bold', color=BLACK, pad=6)
        n_inj = sum(1 for m in message_data if m['sent'])
        n_arr = len(completed_stats)
        aw = sum(p.wait_time for p in completed_stats)/n_arr if n_arr else 0
        al = sum(p.total_trip_time for p in completed_stats)/n_arr if n_arr else 0

        t  = f'Algorithm:  {scheduling_mode}\n'
        t += f'Delivered:  {n_arr} / {n_inj}\n'
        t += f'Avg Wait:   {aw:.2f}s\n'
        t += f'Avg Latency:{al:.2f}s\n'
        t += f'Duration:   {sim_time:.1f}s\n\n'
        t += '-- Packet Log --\n'
        for p in completed_stats[:20]:
            pri = p.priority.name[0]
            t += f'{p.pid}[{pri}]({p.size}B) {p.src}->{p.dest} lat:{p.total_trip_time
                :.1f}s\n'
        if len(completed_stats)>20:
            t += f'  +{len(completed_stats)-20} more\n'
        ax.text(0.06,0.97, t, va='top', fontsize=8.5, color=BLACK,
                fontfamily='monospace', transform=ax.transAxes, linespacing=1.35)
    else:
        ax.set_title('Status', fontsize=12, fontweight='bold', color=BLACK, pad=6)
        n_s = sum(1 for m in message_data if m['sent'])
        n_d = len(completed_stats)
```

```
        n_total = len(message_data)
        t  = f'Mode:      {scheduling_mode}\n'
        t += f'Time:      {sim_time:.1f}s\n'
        t += f'Injected:  {n_s}/{n_total}\n'
        t += f'In-flight: {len(active_messages)}\n'
        t += f'Delivered: {n_d}/{n_s if n_s else n_total}\n\n'
        if completed_stats:
            t += '-- Packet Log --\n'
            for p in completed_stats[-12:]:
                pri = p.priority.name[0]
                t += f'{p.pid}[{pri}]({p.size}B) {p.src}->{p.dest} {p.total_trip_time
                    :.1f}s\n'
            if n_d > 12:
                t += f'  +{n_d-12} more\n'
            t += '\n'
        if message_data:
            pending = [(idx, m) for idx, m in enumerate(message_data) if not m['sent
                ']]
            t += '-- Pending --\n'
            for idx, m in pending[:8]:
                pri = m['priority'].name[0]
                t += f'. P{idx}[{pri}] {m["src"]}->{m["dst"]} {m["size"]}B @{m["time
                    "]:.1f}s\n'
            if len(pending) > 8:
                t += f'  +{len(pending)-8} more\n'
        else:
            t += 'No packets.\nAdjust sliders     auto-updates.'
        ax.text(0.06,0.97, t, va='top', fontsize=8.5, color=BLACK,
                fontfamily='monospace', transform=ax.transAxes, linespacing=1.35)

def update_frame(frame):
    global sim_time
    dt = 0.05

    if running:
        process_node_queues(frame)
        for i,msg in enumerate(message_data):
            if not msg['sent'] and sim_time >= msg['time']:
                if msg['src'] in nodes and msg['dst'] in nodes:
                    active_messages.append(MessagePacket(msg, i, sim_time))
                msg['sent'] = True
        for m in active_messages[:]:
            m.move(dt)
            if m.arrived: active_messages.remove(m)
        sim_time += dt

    _draw_graph(frame)
    _draw_routing()
    _draw_status()

fig.canvas.mpl_connect('button_press_event',   on_press)
fig.canvas.mpl_connect('button_press_event',   on_dblclick)
fig.canvas.mpl_connect('motion_notify_event',  on_motion)
fig.canvas.mpl_connect('button_release_event', on_release)

update_topology()
generate_packets(num_packets, time_interval)

ani = animation.FuncAnimation(fig, update_frame, interval=30, cache_frame_data=False)
```

```
771 _refs.append(ani)
772 plt.show()
```

Bibliography

[1] P. St. John, J. Serna, and R.-G. Lin, "Must reads: Here's how paradise ignored warnings and became a deathtrap," Dec 2018. [Online]. Available: https://www.latimes.com/local/california/la-me-camp-fire-deathtrap-20181230-story.html

[2] California Department of Forestry and Fire Protection, "Camp fire incident information," CAL FIRE, Tech. Rep., 2018, accessed: May 2026. [Online]. Available: https://www.fire.ca.gov/incidents/2018/11/8/camp-fire/

[3] Y. Akbary and L. Chutel, "Taliban shut down the internet during anti-government protests," *The New York Times*, Sep. 2025, accessed: May 2026. [Online]. Available: https://www.nytimes.com/2025/09/04/world/asia/afghanistan-taliban-internet-blackout.html

[4] J. Weatherbed, "Starlink's satellite internet is still expensive for disaster connectivity," *The Verge*, Jan. 2024, accessed: May 2026. [Online]. Available: https://www.theverge.com/2024/1/3/24023895/starlink-satellite-internet-pricing-hardware-costs

[5] American Radio Relay League, "Amateur radio demographics and licensing trends," *ARRL News*, 2023, accessed: May 2026. [Online]. Available: https://www.arrl.org/news/amateur-radio-demographics-and-licensing-trends

[6] [Online]. Available: https://lora.readthedocs.io/en/latest/#range-vs-power

[7] LoRa Alliance, "LoRaWAN link layer specification v1.0.4 (TS1-1.0.4)," LoRa Alliance, Tech. Rep., Oct. 2020. [Online]. Available: https://lora-alliance.org/resource_hub/lorawan-104-specification-package/

[8] "Dijkstra's algorithm for adjacency list representation," accessed: May 2026. [Online]. Available: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php

[9] "Epsilon-greedy algorithm in reinforcement learning," Jan 2023, accessed: May 2026. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/epsilon-greedy-algorithm-in-reinforcement-learning/

[10] J. Zhao and R. Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*. New York, NY, USA: Association for Computing Machinery, 2003, pp. 1–13.

[11] J. Mo and J. Walrand, "Fair end-to-end window-based congestion control," *IEEE/ACM Transactions on Networking*, vol. 8, no. 5, pp. 556–567, Oct. 2000.

[12] Federal Emergency Management Agency, "Resilient communications for disaster response," FEMA, Tech. Rep., 2021, accessed: May 2026. [Online]. Available: https://www.fema.gov

[13] ——, "Resilient communications for disaster response and recovery," FEMA, Tech. Rep., 2021, accessed: May 2026. [Online]. Available: https://www.fema.gov/sites/default/files/documents/fema_resilient-communications.pdf

[14] Semtech Corporation, *SX1276/77/78/79 — 137 MHz to 1020 MHz Low Power Long Range Transceiver*, rev. 7 ed., May 2020. [Online]. Available: https://www.semtech.com/products/wireless-rf/lora-connect/sx1276

[15] M. C. Bor, U. Roedig, T. Voigt, and J. M. Alonso, "Do LoRa low-power wide-area networks scale?" in *Proceedings of the 19th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '16)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 59–67.

[16] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ: Wiley, 2018.

[17] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," 2018, accessed: May 2026. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[18] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[19] D. M. Pozar, *Microwave engineering*, 4th ed. Hoboken, NJ: Wiley, 2012.

[20] Adafruit Industries, "Adafruit Feather 32u4 with LoRa radio module," Adafruit Learning System, 2025, accessed: Apr. 2025. [Online]. Available: https://learn.adafruit.com/adafruit-feather-32u4-radio-with-lora-radio-module

[21] Federal Communications Commission, "47 CFR 15.247 — operation within the bands 902–928 MHz, 2400–2483.5 MHz, and 5725–5850 MHz," Electronic Code of Federal Regulations, Tech. Rep., 2025, accessed: Apr. 2025. [Online]. Available: https://www.ecfr.gov/current/title-47/part-15/section-15.247

[22] ——, "47 CFR 15.5 — general conditions of operation," Electronic Code of Federal Regulations, Tech. Rep., 2025, accessed: Apr. 2025. [Online]. Available: https://www.ecfr.gov/current/title-47/part-15/section-15.5

[23] Federal Aviation Administration, "14 CFR part 101, subpart D — unmanned free balloons," Electronic Code of Federal Regulations, Tech. Rep., 2025, accessed: Apr. 2025. [Online]. Available: https://www.ecfr.gov/current/title-14/chapter-I/subchapter-F/part-101/subpart-D

[24] J. Moy, "OSPF version 2," Internet Engineering Task Force, Request for Comments 2328, Apr. 1998. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2328

[25] IEEE, "IEEE standard for information technology—Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 10: Mesh networking," IEEE Std 802.11s-2011, Sep. 2011.

[26] Arduino, "Arduino language reference," Arduino Documentation, 2025, accessed: Apr. 2025. [Online]. Available: https://www.arduino.cc/reference/en/

[27] ISO/IEC, "ISO/IEC 14882:2011 — information technology — programming languages — C++," International Organization for Standardization, Tech. Rep., 2011. [Online]. Available: https://www.iso.org/standard/50372.html

[28] Python Software Foundation, "The python language reference, version 3," 2025, accessed: Apr. 2025. [Online]. Available: https://docs.python.org/3/reference/

[29] Ansys, Inc., "Ansys HFSS: High frequency structure simulator," Ansys Electronics Desktop, 2024. [Online]. Available: https://www.ansys.com/products/electronics/ansys-hfss

[30] The MathWorks, Inc., "MATLAB r2024b," 2024. [Online]. Available: https://www.mathworks.com/products/matlab.html