CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Wireless Monitoring Tool for BLE Mesh Networks

Master's thesis in Embedded Electronic System Design

TOMASS PULS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

# Wireless Monitoring Tool for BLE Mesh Networks

TOMASS PULS

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Wireless Monitoring Tool for BLE Mesh Networks

TOMASS PULS



Supervisor at CSE: Arne Linde, Computer Engineering
Supervisor at company: Henrik Persson
Examiner: Per Larsson-Edefors, VLSI Systems

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2024

Wireless Monitoring Tool for BLE Mesh Networks

TOMASS PULS
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

This thesis aims to address the limitations of existing wired monitoring solutions for Bluetooth Low Energy (BLE) mesh networks by developing a wireless monitoring tool. The tool is designed to simplify network management and provide performance insights. The thesis focuses on the development of a dedicated hardware tool using the nRF5340 DK, implementation of data analysis software, and creation of a user-friendly graphical user interface (GUI) to monitor and visualize key performance metrics such as Message Delivery Ratio (MDR), throughput, and latency in BLE mesh networks.

The methodology involved constructing a hardware tool for wireless data capture, developing Python-based software for data analysis, and designing a GUI for real-time data visualization. Extensive testing was conducted in various mesh network topologies to evaluate the tool's performance. The tool demonstrated robust performance in accurately measuring MDR and latency across different network setups. The wireless monitoring tool successfully provides a non-intrusive, user-friendly solution for real-time analysis of BLE mesh networks. It offers valuable insights into network performance and health, making it a practical tool for both academic research and industrial applications. Future work could extend the tool's capabilities to multiple measurement points and multi-channel analysis.

Keywords: BLE, Bluetooth-Low Energy, BLE mesh, RBC mesh, wireless monitoring tool, latency, message delivery ratio, mesh network topology

# Acknowledgements

I would like to express my sincere gratitude to Plejd for providing me with the opportunity to conduct this thesis at their company. Special thanks to my technical supervisor Henrik Persson for his invaluable time, help, and guidance throughout this project. I am also deeply appreciative of the other employees at Plejd who generously shared their ideas and provided guidance, contributing to the success of this thesis.

Tomass Puls , Gothenburg, 2024-06-16

# Contents


**1 Introduction** **1**
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 2
- 1.3 Objectives . . . 3
- 1.4 Delimitation . . . 3
- 1.5 Thesis Structure . . . 4

**2 Theoretical Background** **5**
- 2.1 Wireless Communication on the nRF5340 DK . . . 5
  - 2.1.1 nRF5340 DK . . . 5
  - 2.1.2 Softdevice . . . 6
    - 2.1.2.1 Timeslot API . . . 7
  - 2.1.3 The Generic Attribute Profile (GATT) . . . 7
    - 2.1.3.1 GATT in BLE and BLE mesh . . . 8
- 2.2 nRF OpenMesh . . . 9
  - 2.2.1 The nRF OpenMesh Concept . . . 9
- 2.3 Trickle Algorithm . . . 12
- 2.4 Software for Analysis . . . 13
  - 2.4.1 DearPyGUI . . . 13
  - 2.4.2 NetworkX . . . 13
  - 2.4.3 Pandas . . . 14
- 2.5 Performance Metrics . . . 14
  - 2.5.1 Message Delivery Ratio (MDR) . . . 15
  - 2.5.2 Latency . . . 15

**3 Methodology** **17**

**4 Design and Implementation** **19**
- 4.1 Embedded Software Design (nRF5340 DK) . . . 19
  - 4.1.1 Radio Driver . . . 20
  - 4.1.2 FIFO and shared memory . . . 22
  - 4.1.3 USB Driver . . . 23
- 4.2 GUI Software Design . . . 24
  - 4.2.1 USB Driver Module . . . 24

4.2.2 Mesh Communication Module . . . 25
4.2.3 Authentication Module . . . 25
4.2.4 Mesh Topology Analysis Module . . . 26
4.2.5 Message Delivery Ratio (MDR) Analysis . . . 27
4.2.6 Latency Analysis . . . 28
4.3 Data Analysis . . . 29
4.3.1 Mesh Topology . . . 29
4.3.2 Message Delivery Ratio (MDR) . . . 30
4.3.3 Latency . . . 32

**5 Experimental Setup** **35**
5.1 Mesh Topology Analysis . . . 35
5.2 Message Delivery Ratio (MDR) Analysis . . . 38
5.3 Latency Analysis . . . 39

**6 Results** **43**
6.1 Mesh Topology Analysis . . . 43
6.2 Message Delivery Ratio (MDR) Analysis . . . 45
6.3 Latency Analysis . . . 49

**7 Discussion** **55**
7.1 Mesh Topology Analysis . . . 55
7.1.1 Analysis of Findings . . . 56
7.1.2 Interpretation of Results . . . 57
7.1.3 Challenges Encountered . . . 57
7.2 Message Delivery Ratio (MDR) Analysis . . . 58
7.2.1 Analysis of Findings . . . 59
7.2.2 Interpretations of Results . . . 59
7.2.3 Challenges Encountered . . . 60
7.3 Latency Analysis . . . 61
7.3.1 Analysis of Findings . . . 61
7.3.2 Interpretation of Results . . . 62
7.3.3 Challenges Encountered . . . 63

**8 Conclusion** **65**

**Bibliography** **67**

# 1

# Introduction

In today's interconnected world, Bluetooth Low Energy (BLE) mesh networks are foundational to the Internet of Things (IoT), enabling devices to communicate efficiently with minimal power usage. As these networks become integral to a wide range of applications, from smart homes to industrial automation, the ability to monitor, spot weak links and optimize their performance is becoming more important.

This thesis introduces a wireless monitoring tool designed to simplify and enhance the management of BLE mesh networks using the nRF OpenMesh protocol [1]. This tool centers around the nRF5340 Development Kit (DK) [2] for its hardware capabilities. Unlike existing wired solutions, which can be cumbersome and complex to set up, this tool provides a wireless, user-friendly alternative that does not require alterations to the devices within the network. It focuses on measuring performance metrics, including Message Delivery Ratio (MDR), throughput, and latency. In addition, it analyzes, maps and displays the topology of the mesh network and offers near real-time visualization through a graphical user interface (GUI), simplifying network analysis and management.

The subsequent sections will outline the motivation for this research, a review of related work, the objectives it seeks to achieve, the study's delimitations, and an overview of the thesis's organization.

## 1.1 Motivation

Existing tools for monitoring and analyzing BLE mesh networks, as in [3] and [4], often rely on wired connections directly to network devices. This presents challenges in terms of real-world applicability, complexity, and the scope of analysis. Wired solutions can be difficult or impractical to deploy in real-world mesh network environments, especially those that are already operational or in hard-to-reach locations, hindering our understanding of network behavior in practical settings. Wired setups often introduce complexity, requiring modifications to existing network nodes. Additionally, many research studies concentrate on a single performance metric, such as Round-Trip-Time (RTT) [4], Packet Delivery Ratio (PDR) [5] and [6], or latency

[7]. This narrow focus fails to capture the holistic performance picture needed for real-world network management.

This thesis addresses these limitations by developing and assessing a wireless BLE mesh network monitoring tool. This tool enables non-intrusive monitoring of network behavior without the need for physical modifications to a network. It prioritizes ease of use, making network analysis accessible to a wider group of users. The tool is tailored to perform performance evaluation, with the ability to monitor multiple metrics, providing a more accurate representation of overall network health.

While the tool in this thesis utilizes a single measurement point, focuses on a specific channel and bitrate, and prioritizes application-level message delivery, it serves as a foundation for potential future expansions. These might include monitoring across multiple points, a broader range of channels and bitrates, as well as deeper packet-level analysis.

## 1.2 Related Work

The exploration of BLE mesh networks in current research offers insights into standards, performance metrics, and application-specific challenges. The Bluetooth Mesh standard is detailed in [4], emphasizing metrics such as RTT, important for evaluating network performance. The work in [7] expands on BLE mesh standards important for navigating proprietary network analyses. Investigations into sensor network performance within indoor settings [8] shed light on the importance of accurate performance metrics for site-specific analysis, aligning with the thesis's aim to monitor BLE mesh networks effectively.

Murillo et al. [5] offer insights into the impact of network architecture choices on performance metrics. This provides valuable context for analyzing and visualizing network topologies. Additionally, studies on smart lighting systems using BLE mesh [6] highlight the practical challenges in network performance monitoring. The development of wired monitoring/emulating solutions like Automated Physical Testbeds (APTB) [3] and experimental evaluations of BLE mesh [9] underscore the complexities and potential of wireless-network monitoring.

This thesis leverages a hardware device and a GUI for data analysis and visualization, aiming to bridge the gap between theoretical research and practical monitoring needs. While the works cited above provide valuable insights, this thesis aims to offer a solution made for simplified and user-friendly analysis of live BLE mesh environments. By synthesizing insights from existing studies and focusing on near real-time monitoring with wireless instrumentation, the thesis can provide contribution to the field, addressing both academic and practical aspects of BLE mesh network monitoring.

## 1.3 Objectives

This thesis aims to bridge the gap in BLE mesh network monitoring through the development of a wireless solution. The specific objectives to achieve this goal are:

1. **Develop a Dedicated Hardware Tool:** Construct a dedicated hardware tool using the nRF5340 DK that wirelessly captures data from BLE mesh networks, transmitting it to a PC for processing and analysis.

2. **Implement Data Analysis Software:** Develop Python-based software to process the gathered data and extract performance metrics, including MDR with throughput, latency and analyze network topology.

3. **Create a User-Friendly GUI:** Design a user-friendly GUI to visualize the analyzed data, providing clear insights into network performance and health.

4. **Conduct Real-World Testing and Evaluation:** Conduct testing of the hardware and software components in real-world network environments. This evaluation will validate the tool's effectiveness in enhancing BLE mesh network monitoring and management.

## 1.4 Delimitation

To ensure the feasibility and focus of this thesis, it's important to establish specific boundaries for the scope of this research:

- **Single Measurement Point:** The wireless monitoring tool developed in this project will utilize a single measurement point within the BLE mesh network. While the tool may be adaptable to multiple measurement points in the future, the current scope emphasizes demonstrating functionality with a single point.

- **Proprietary Mesh:** The development and testing of the tool will utilize the nRF OpenMesh protocol. While this limits the direct applicability of findings to other mesh implementations, it provides a well-defined and practical environment for demonstrating the tool's effectiveness..

- **Channel and Bitrate:** The evaluation of the wireless monitoring tool will be conducted using a single nRF OpenMesh channel operating at a 1 Mbit/s bitrate. This simplifies the analysis and allows focus on core tool functionality.

- **Message-Level Evaluation:** The primary focus will be on message delivery success rather than individual packet analysis. This aligns with the real-world concern of ensuring complete application-level message transmission within the mesh network, particularly relevant when using algorithms like the Trickle algorithm.

- **Packet Validation:** This work will not delve into the particulars of mesh packet validation (CRC checks, encryption, decryption, header validation, etc.). These mechanisms are assumed to be in place and functioning correctly.

- **Security Aspects:** Security considerations are not explicitly addressed within the scope of this thesis.

## 1.5 Thesis Structure

This thesis is organized into eight chapters to provide a clear and systematic presentation of the research conducted on wireless monitoring of BLE mesh networks. The structure is designed to guide through the progression from initial concepts to detailed analysis and conclusions.

Chapter 1 introduces the research, including the motivation, objectives, and the scope of the thesis. This foundational chapter sets the stage by contextualizing the importance of BLE mesh networks and the necessity for effective monitoring tools.

Chapter 2 delves into the theoretical background necessary for understanding BLE mesh networks. It covers technologies and protocols that underpin the functioning and monitoring of these networks, providing a basis for the subsequent application and development of the monitoring tool.

In Chapter 3, the methodology employed in the research is outlined. This includes the approaches taken for system design, data collection, and the analysis techniques used to evaluate the performance of the monitoring tool.

Chapter 4 details the system design of the monitoring tool, including both hardware and software components. This chapter explains the architectural decisions, the development process, and the integration of different modules that form the complete system.

Chapter 5 describes the experimental setup used to test and validate the monitoring tool. It specifies the configurations, the environments in which tests were conducted and the parameters that were measured.

Chapter 6 presents the results obtained from the experimental tests.

Chapter 7 contains a discussion of the findings from the experiments. It compares expected outcomes with actual results, discusses the implications of the findings, and examines potential improvements and future work.

Finally, Chapter 8 concludes the thesis by summarizing the research outcomes and their implications for the field of wireless communications. This chapter also suggests areas for further research and the potential impact of the study on future developments in wireless monitoring technologies.

# 2

# Theoretical Background

This project brings together wireless networking, data analysis, and user interface design for the purpose of analyzing and visualizing BLE mesh networks. Understanding the core concepts and technologies involved is important. This chapter provides the necessary background in several key areas. First, the exploration will cover the nRF5340 DK hardware, the underlying SoftDevice responsible for BLE communication, and the GATT protocol that defines data exchange in BLE environments. The chapter will also delve into the Timeslot API, which allows applications to manage radio resources when using a SoftDevice  essential for mesh network functions. Next, the focus will be on the mechanics of the nRF OpenMesh Mesh protocol. This rebroadcasting-based approach is fundamental to understanding how these networks operate and the type of data that will be captured for analysis. Finally, the discussion will focus on the suite of Python libraries used to transform raw data into meaningful insights, including DearPyGUI for GUI creation, NetworkX for network analysis, and Pandas for data manipulation.

## 2.1 Wireless Communication on the nRF5340 DK

Understanding the key components enabling BLE mesh communication on the nRF5340 DK is essential for analyzing network behavior. This section explores the hardware itself, the underlying SoftDevice, the Timeslot API for resource management, and the GATT protocol that defines BLE communication.

### 2.1.1 nRF5340 DK

The nRF5340 DK [2], powered by the nRF5340 System-on-Chip (SoC), is a development platform designed for high-performance applications such as BLE mesh networking. At its heart is a single ARM Cortex-M33 processor with two cores, a main application core operating at speeds up to 128 MHz, and a network core at speeds up to 64 MHz. This dual-core setup enables efficient parallel processing, enhancing the system's responsiveness and computational capabilities which are important for demanding network monitoring tasks. Importantly, only the network core has access to the radio peripheral, while only the application core has access to the USB peripheral. These cores can communicate with each other using a shared

memory area.

The nRF5340 SoC's RX sensitivity, reaching up to -104 dBm, is a significant advantage for BLE mesh monitoring. This sensitivity directly increases the DK's ability to detect and capture wireless packets, ensuring comprehensive data collection within the mesh network and minimizing the likelihood of missing critical information. Additionally, the nRF5340 DK comes equipped with 1 MB of Flash and 512 KB of RAM, providing sufficient storage and processing capacity, particularly for capturing and forwarding BLE mesh data. The DK also features rapid USB communication speeds, up to 12 Mbps, facilitating fast and reliable data transfer to a PC for analysis.

nRF microcontrollers feature a functionality named SHORTCUTS. The SHORTCUTS provide a direct way to link events happening in the microcontroller's hardware to tasks that need to be executed. This hardware-based linking, rather than relying on software interrupts, reduces latency and improves system responsiveness.

Development with the nRF5340 DK is described in [10]. It involves working with its two Arm Cortex M33 processors for various radio communication and application-level tasks. The development process includes setting up the nRF Connect SDK, understanding the architecture for different protocols, and managing multi-image builds for separate network and application cores. For specific development tasks, such as programming, debugging, and FOTA updates, detailed steps and examples are provided, tailored to whether the development is being done using Visual Studio Code or the command line.

### 2.1.2 Softdevice

The SoftDevice is a foundation of wireless development on the Nordic Semiconductor nRF5 Series devices [11]. It is a pre-compiled, pre-linked binary software stack encapsulating the complexities of wireless protocols like BLE. By providing a high-level programming interface, the SoftDevice simplifies development and ensures the stability and reliability of wireless applications.

Nordic Semiconductor SoftDevices are certified to support specific protocols. Within this project, mesh nodes utilize the SoftDevice due to its certification for BLE communication, allowing them to interact with BLE GATT-compatible devices such as smartphones.

The SoftDevice isolates its internal protocol logic from user applications, protecting these core functions and offering deterministic behavior important for real-time applications. This architecture streamlines development, allowing developers to focus on their application's unique features rather than the intricacies of wireless protocols.

#### 2.1.2.1 Timeslot API

With both the SoftDevice and mesh applications needing radio access, the Timeslot API provides an important mechanism for managing radio resources efficiently in applications that utilize the SoftDevice [12]. In a mesh network, where the application needs to perform its own wireless actions while the SoftDevice handles core BLE functions, the radio becomes a shared resource that needs careful scheduling [13].

To prevent conflicts between mesh-specific wireless actions and the SoftDevice's core BLE functions, the Timeslot API allows applications to request dedicated time slots for radio operations or other high-priority tasks. The SoftDevice manages these requests, ensuring that they do not conflict with its own protocol operations. This guarantees that mesh-specific communication needs are met while maintaining the overall integrity of the SoftDevice's Bluetooth functions.

As an example, each 152 ms, the SoftDevice needs to perform BLE advertisements. These advertisements are brief transmission bursts used to make the device discoverable by other BLE devices and establish connections. During each advertisement event, the SoftDevice takes control of the radio peripheral from the application for approximately 10 ms on average. This temporary takeover ensures that critical BLE advertisements have priority and between them the application is free to use the radio peripheral as needed.

### 2.1.3 The Generic Attribute Profile (GATT)

The Generic Attribute Profile (GATT) serves as the foundational framework for data exchange within BLE communication [14]. It leverages the Attribute Protocol (ATT) to establish a hierarchical structure for organizing and managing data, ensuring interoperability between BLE devices. This structure, as illustrated in Figure 2.1, is comprised of the following key elements:

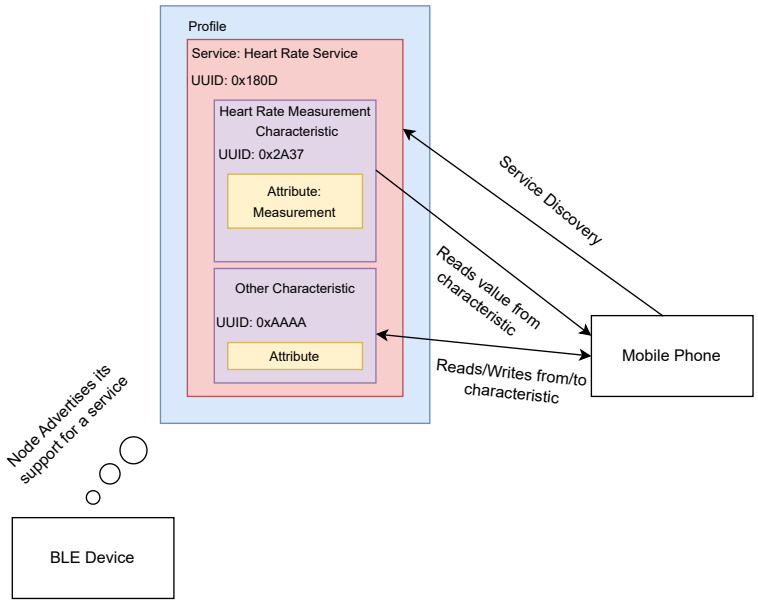


Figure 2.1: Simplified GATT visualization, showing communication flow with Services, Characteristics and Attributes

- **Services:** Services, represented by the red box in the figure above, encapsulate collections of related functionalities offered by a BLE device. For example, a heart rate monitor might expose a "Heart Rate Service," a standardized service with a known UUID, to provide functionalities pertaining to heart rate measurement.

- **Characteristics:** Characteristics, represented by the purple boxes in the figure above, reside within services and represent specific data points or behaviors. The "Heart Rate Service" could include characteristics such as "Heart Rate Measurement," containing the actual heart rate value, and "Body Sensor Location," specifying the sensor's placement.

- **Attributes:** Attributes, represented by the yellow boxes in the figure above, form the basic units of data within GATT. Each attribute possesses a unique handle for identification and properties that dictate permitted operations such as read, write, and notify. The "Heart Rate Measurement" characteristic would likely have an attribute storing the heart rate as an integer value.

GATT's significance in BLE applications lies in facilitating energy-efficient data exchange. By defining how devices structure and expose their capabilities and data, GATT streamlines BLE development and ensures seamless communication between devices within diverse IoT applications.

#### 2.1.3.1 GATT in BLE and BLE mesh

It's important to emphasize that GATT plays a distinct role within the context of BLE mesh networks. Standard BLE communication using GATT typically involves

point-to-point interactions between devices [14]. In contrast, BLE mesh networks utilize GATT services for:

- **Provisioning and Configuration:** Adding nodes to a BLE mesh and setting up their initial parameters often involves GATT interactions.

- **External Communication:** BLE mesh networks can bridge with non-mesh BLE devices (e.g., phones, tablets) using specialized GATT-based services. This allows external devices to monitor and control mesh devices.

## 2.2 nRF OpenMesh

The nRF OpenMesh, formerly Rebroadcast (RBC) Mesh, is an open-source framework designed for creating robust and self-forming mesh networks [1]. Its core principle lies in message rebroadcasting, where nodes automatically forward messages to ensure network-wide distribution. The nRF OpenMesh Mesh employs a decentralized architecture, eliminating the need for central controllers or predetermined routing paths.

The nRF OpenMesh integrates with technologies discussed in previous chapters. To coordinate radio access within the SoftDevice environment, it utilizes the Timeslot API, ensuring coexistence with BLE operations. Additionally, the framework employs the Trickle algorithm to intelligently manage rebroadcast frequency, optimizing network traffic by minimizing redundancy while ensuring rapid propagation of updates. The nRF OpenMeshh leverages GATT concepts for interaction between the mesh network and external BLE devices, incorporating a Mesh GATT service to promote interoperability.

The nRF OpenMesh distinguishes itself from BLE Mesh, described in [4], by utilizing dedicated channels (41, 10, 22) and varying data rates (2Mbit/s, 1Mbit/s, 250kbit/s). This strategic choice minimizes interference with BLE advertising and common WiFi channels. The nRF OpenMesh mesh nodes operate without predefined scan intervals or windows, maximizing responsiveness.

### 2.2.1 The nRF OpenMesh Concept

The nRF OpenMesh protocol, upon which this project builds, employs a system of indices as its primary mechanism for managing data within the network. Think of these indices as numbered slots, as shown in Figure 2.2, where information can be stored and retrieved. Nodes are granted specific access levels (read, write, or both) to various indices, which controls their ability to interact with different elements of the mesh.

Devices within the nRF OpenMesh Mesh, such as smart light bulbs, are often assigned dedicated indices. These indices might represent properties like the bulb's on/off state or its brightness level, or both. By having both read and write access

to these indices, nodes within the mesh, as well as external BLE devices (via the Mesh GATT service), can directly monitor and control these devices.

Input devices, like buttons, don't have their own indices. Instead, they control outputs (such as light bulbs) by having write access to the corresponding output's index. To stay synchronized, an input device should also have read access to the output index it controls. This ensures that if the output's state is changed from another source (e.g., through a mobile app), the button's internal state will update accordingly.

Some devices might consist of multiple logical units that act as a single entity, a group. These devices often utilize a TX index (Transmit Index) for outgoing data and an RX index (Receive Index) for incoming control information. The entire group of units shares read access to the index where they receive steering messages. In addition, they transmit their acknowledgments and responses on the RX index, which is monitored by the controlling devices. This separation of indices is important to prevent situations where multiple devices might simultaneously attempt to reply on the same index, potentially overwriting the original steering command and preventing nodes that missed the original message from receiving it.

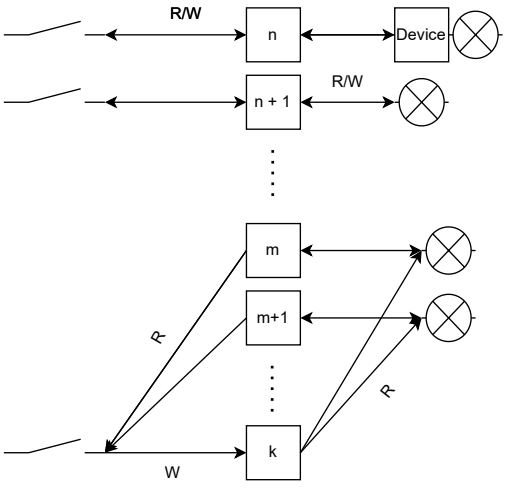


Figure 2.2: Representation of Index Management in the nRF OpenMesh

To ensure consistency, each node keeps a cache of current versions associated with each index. When a node sends a message that updates an index, the version number for that index is incremented. There's an important distinction in how messages are treated:

- **SET/GET Commands:** When a node sends a SET or GET command to another index, the receiving node will answer with a RESP (response) or ACK (acknowledgement) message containing an incremented version.

- **State Announcements:** When a node itself changes its state or level, it will

announce this change on its own index with a DR (Don't Respond) flag. All other nodes in the network will rebroadcast that message. Since these messages are not meant to generate any response, other nodes do not increment the version, but simply rebroadcast it.

Upon receiving an updated message, nodes first determine its relevance. If it is relevant, they immediately begin rebroadcasting it. The Trickle algorithm manages the timing of these rebroadcasts. If a node receives the same updated message from at least four other nodes before its next scheduled rebroadcast, it will stop its own rebroadcast. This collaborative approach ensures that all nodes maintain a consistent view of the data within the mesh. Additionally, when a mesh node communicates with a BLE device, it broadcasts any messages from that device across the mesh to ensure network-wide synchronization.

## 2.3 Trickle Algorithm

The Trickle algorithm is a solution to optimize communication within wireless networks [15], especially in resource-constrained environments where energy conservation and network bandwidth are crucial. The nRF OpenMesh Mesh relies on Trickle for efficient re-broadcasts minimizing message collision probability. The Trickle addresses two primary objectives. Firstly, Trickle aims to minimize redundant retransmissions of information, thus preserving precious network resources. Secondly, it strategically adjusts transmission intervals to maximize the likelihood of message delivery even under lossy network conditions. An example of the Trickle algorithm is visualized in Figure 2.3 below.

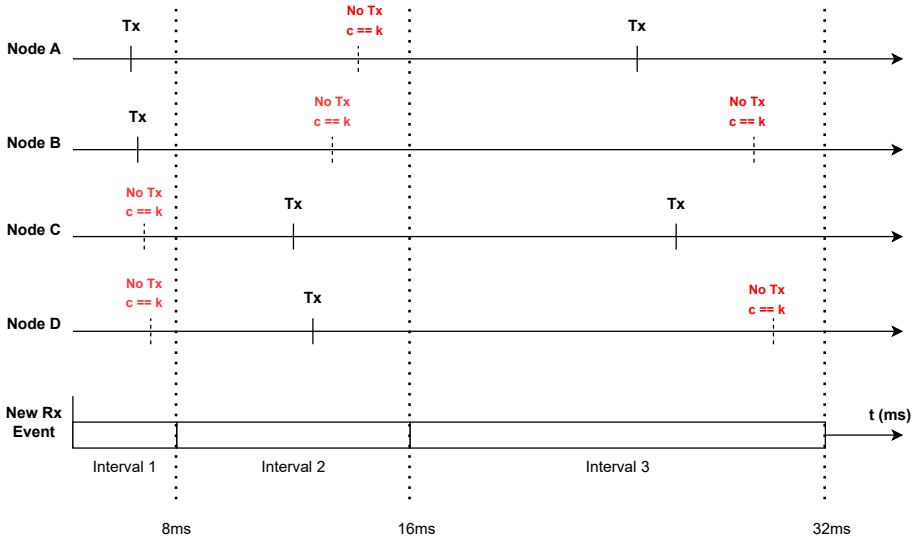


Figure 2.3: Visualization of Trickle algorithm. $I_{min} = 8$ ms, $I_{max} = 32$ ms, $k = 2$

The Trickle algorithm relies on four main parameters to control its behavior: minimal interval size ($I_{min}$), maximum interval size ($I_{max}$), redundancy constant ($k$), and a counter ($c$). Time is divided into intervals. At a random point within each interval, a node decides whether or not to broadcast its data. This decision hinges on the relationship between the $c$ and the $k$. If the counter is less than the $k$, the node broadcasts. If a node receives a broadcast containing the same information it already has, it increments its counter, message is consistent. If a node hears new or updated information, it resets its interval to the minimum value to hasten the spread of that information. However, if a node hears an old, inconsistent information, it resets its interval of current rebroadcast to the minimum value to hasten the spread of consistent information. At the end of each interval, the interval size doubles, until a maximum is reached, to reduce communication overhead when information remains stable within the network.

The figure above illustrates a communication session with the Trickle algorithm

configured to be $I_{min} = 8$ ms, $I_{max} = 32$ ms, $k = 2$. At time zero, a new message is introduced into the network. Nodes begin their first rebroadcast interval, randomly selecting a transmission time within that period. Nodes A and B happen to select times earlier than C and D. Since the redundancy constant ($k$) is 2, nodes C and D suppress their rebroadcasts after hearing the message twice. In the second period, C and D select earlier transmission times, leading to A and B suppressing their rebroadcasts, and so on. This process continues, ensuring the message spreads quickly while minimizing redundant transmissions.

Due to hardware resource limitations, nodes in this project completely suppress rebroadcasts if they hear at least k transmissions of the same message within their current interval. This means they stop rebroadcasting entirely, preventing unnecessary repetition.

## 2.4 Software for Analysis

Having explored the technologies that enable BLE mesh communication, let's shift our focus to the software tools used to analyze and visualize the captured network data.

### 2.4.1 DearPyGUI

DearPyGUI is a Python library designed for the creation of graphical user interfaces (GUIs) [16]. It is well-suited for this project due to its ease of use, performance, and the need for a responsive interface to visualize and control a BLE mesh network. DearPyGUI utilizes an Immediate Mode GUI paradigm, which simplifies development and offers enhanced responsiveness compared to traditional GUI libraries.

DearPyGUI's C++ backend promotes efficiency, while its Python API maintains accessibility for developers. The library provides a diverse set of widgets and customization options, enabling the creation of interfaces suitable for a variety of purposes. In contrast to retained mode libraries like Tkinter, DearPyGUI's approach grants more direct control over the GUI's state, potentially simplifying the development of complex, interactive interfaces.

DearPyGUI utilizes a main loop to continuously redraw the GUI, process events, and update the interface. This model ensures responsiveness and adaptability to real-time changes, making it appropriate for applications like mesh network visualization.

### 2.4.2 NetworkX

The NetworkX Python library offers a powerful suite of tools for analyzing the topology and performance of complex networks, including BLE mesh networks [17]. Its flexibility in representing networks as graphs aligns perfectly with the structure of mesh networks. In this project, mesh devices, such as smart lights or sensors, are modeled as nodes within the graph. Direct connections between these devices are

represented as edges. This graph-based representation enables the application of NetworkX's rich set of algorithms to gain insights into the mesh network's characteristics.

NetworkX provides functions for creating, manipulating, and analyzing graphs. It includes algorithms for calculating layout positions, which is essential for visualizing the mesh network's topology in an informative and comprehensible manner. Furthermore, determining efficient routes for data transmission is crucial within mesh networks. NetworkX's shortest path algorithms can identify the optimal routes for packets to travel, potentially revealing the routing choices made within the mesh.

Beyond these core functionalities, NetworkX provides a vast array of advanced algorithms. Centrality measures can pinpoint the most influential nodes within the mesh, highlighting critical junctions or potential bottlenecks. Additionally, the identification of connected components can reveal isolated segments within the mesh, indicating potential areas with connectivity issues.

### 2.4.3 Pandas

The Pandas Python library is a cornerstone of data manipulation and analysis, providing powerful tools to work with datasets [18]. It introduces efficient data structures, namely DataFrames and Series, that streamline the handling and analysis of large datasets. Pandas supports a wide range of data operations, including data cleaning, filtering, aggregation, and merging  essential tasks for data scientists, analysts, and researchers.

One of Pandas' primary advantages is its ability to work with various data formats (CSV, Excel, SQL databases, JSON). This versatility aligns with diverse data sources encountered in real-world projects. Pandas integrates with other Python scientific computing libraries like NumPy and Matplotlib, enhancing its capabilities within comprehensive data analysis workflows.

Within this project, Pandas plays a crucial role in processing and analyzing the data collected from the nRF53 DK hardware. After decryption, Pandas will structure the data into DataFrames for easier manipulation. This will enable filtering data based on specific criteria, calculating statistical metrics, like MDR with throughput, latency and merging datasets to gain insights into the BLE mesh network's topology and performance.

## 2.5 Performance Metrics

Performance metrics are essential for evaluating and optimizing BLE mesh networks. These metrics help us understand the network's ability to deliver data reliably, efficiently, and with appropriate responsiveness.

### 2.5.1 Message Delivery Ratio (MDR)

MDR is a performance metric that reflects the proportion of successfully delivered messages relative to the total number of messages sent. MDR is important in evaluating the reliability and efficiency of communication within mesh networks. It evaluates the network's reliability and efficiency, reflecting its ability to maintain data integrity and accurate delivery across multiple nodes.

MDR is calculated as:

$$MDR = \frac{\text{Number of messages successfully delivered}}{\text{Total number of messages sent}} * 100\% \tag{2.1}$$

MDR is closely connected with the PDR (Packet-Delivery-Ratio). While MDR measures the success of complete message deliveries, PDR focuses on individual packet deliveries. According to the article in [19], PDR is significantly influenced by the topology of the network and its ability to handle varying traffic loads effectively. This influence extends to MDR, as a higher PDR generally leads to a higher MDR.

As throughput increases, maintaining high MDR becomes challenging due to congestion and collisions. Effective throughput management involves adaptive strategies such as dynamic routing adjustments, prioritizing critical messages and handling traffic bursts to preserve data integrity.

### 2.5.2 Latency

Latency in [7] is discussed as an important performance metric, particularly relevant for applications where a human expects immediate response, such as smart home lights where user interactions demand quick system reactions.

Latency refers to the time delay experienced from when a packet is sent to when it is received at another nodes in the network. It can be affected by various factors. Transmission time, which encompasses the duration it takes for a packet to travel across the network, is significantly shaped by the network's setup and the nature of the transmission medium. Additionally, the hop count, or the number of intermediary nodes a packet traverses, contributes to overall delays, as each node potentially adds its own processing and forwarding time. Network topology and configuration also play critical roles; for instance, systems like Bluetooth Mesh or 6BLEMesh manage data transfers differently, thus affecting latency.

Managing latency effectively in BLE mesh networks is important for ensuring that user interactions are seamless and that the network can support real-time applications efficiently. This is particularly important in environments like smart homes, where delay-sensitive operations (like turning lights on/off) are common.

# 3

# Methodology

This thesis focuses on the development of a wireless BLE mesh monitoring tool, seeking to provide insights into mesh performance and topology without requiring modifications to existing network devices. To achieve this goal, a structured methodology was employed, encompassing literature review, hardware/software development, data analysis, and comprehensive evaluation.

1. **Literature Analysis:**
   A thorough review of scholarly articles, standards, and prior research on BLE mesh technologies, network performance metrics, and monitoring tools formed the project's foundation. This analysis illuminated existing solutions, identified gaps (such as the need for a wireless, user-friendly monitoring tool), and established a theoretical basis for evaluating performance metrics.

2. **Hardware and Software Requirement Definition:**
   The selection of hardware and software technologies was guided by insights from the literature review and knowledge about existing network. In the selection of hardware for the mesh network, the nRF53 was chosen due to its compatibility with the existing network comprised of nRF51 and nRF52 devices. The nRF53's radio architecture aligns with the nRF51 and nRF52, ensuring seamless communication within the mesh. Additionally, the nRF53's speed and RX sensitivity were key factors in its selection. The increased speed enhances the device's ability to capture mesh packets, minimizing the risk of data loss, while the improved RX sensitivity allows for reliable communication even in challenging network conditions. Python was selected as the programming language for its versatility, ease of use, and powerful libraries. In particular, the Pandas library offers robust data manipulation and analysis capabilities, while DearPyGUI provides a user-friendly framework for rapid GUI development.

3. **Firmware Implementation:**
   Development of a custom radio peripheral driver tailored to the nRF5340 architecture, leveraging the Nordic SDK, was crucial for mesh packet capture. This driver was complemented by the establishment of inter-core communication for data transfer between the network core and the application core.

Additionally, USB communication drivers were used to facilitate data transfer to a PC.

4. **GUI Implementation:**
   The DearPyGUI library was utilized to create an intuitive interface for user interaction. Key components include authentication mechanisms, mesh site selection, and data visualization. Graphing elements were carefully chosen to effectively display network topology and performance metrics as MDR with throughput and latency.

5. **Mesh data analysis:**
   Mesh topology analysis was implemented leveraging logic of the Trickle algorithm, allowing for the inference of direct communication links between nodes through analysis of rebroadcast patterns. Methods for calculating MDR, throughput and latency were developed, employing Python's Pandas library for data manipulation and statistical analysis. The ability to process and analyze mesh data is essential for the core functionality of the monitoring tool.

6. **Testing and Evaluation:**
   The evaluation phase of the monitoring tool is designed to test its functionality and accuracy in real-world settings, focusing on its ability to correctly monitor and report on performance metrics of a mesh network and discover its topology. This phase involves a series of systematic tests to assess the reliability and effectiveness of the tool in a controlled environment that simulates operational mesh network conditions.

   Mesh topology analysis was tested by creating predefined topologies in which nodes were configured to communicate only with specific neighbors. This allowed for controlled assessment of the tool's ability to correctly infer these direct links, verifying its accuracy in mapping network connections.

   Next, the tool's capability to accurately monitor and record the MDR alongside throughput rates across the network is evaluated. Testing involves sending a predefined number of messages through the network at varying throughputs to determine if the tool can correctly quantify the percentage of successfully delivered messages and accurately measure the data throughput. This test helps in assessing the robustness of the monitoring tool in maintaining performance under different network loads.

   Latency tests measure the time taken for messages to travel from a source to a destination within the network. This involves transmitting packets between various pairs of nodes and measuring the time until these packets are acknowledged. The monitoring tool's accuracy in recording these times is verified against a theoretical model to ensure that latency is captured correctly.

# 4

# Design and Implementation

This system design prioritizes a wireless, user-friendly, and comprehensive solution for BLE mesh network monitoring. The foundation lies in the nRF5340 DK, selected for its dual-core architecture. This allows the network core to handle dedicated network monitoring while the application core focuses on data analysis and GUI interactions, ensuring efficiency and responsiveness. To capture traffic from legacy mesh nodes without requiring modifications, a custom radio driver was developed specifically for this project. FIFO buffers are employed for synchronized and reliable data transfer between the cores, maintaining data integrity during the process. Once captured data reaches the PC via USB, it undergoes decryption and validation. Finally, the DearPyGUI library facilitates the creation of a user-friendly graphical interface, offering semi-real-time network insights, topology visualization, and in-depth analysis of performance metrics like MDR and latency. Figure 4.1 provides a visual representation of this architecture, highlighting the integration of hardware and software components.

## 4.1 Embedded Software Design (nRF5340 DK)

The embedded software developed for the nRF5340 DK plays an important role in enabling wireless BLE mesh network monitoring. It consists of a custom radio driver, robust inter-core communication mechanisms, and USB data transfer, enabling capture of mesh traffic and its transmission to a PC for in-depth analysis [20]. Apart from mesh packet payload, the monitoring tool also records metadata associated with each packet, including the timestamp of capture relative to system radio start-up time and the channel on which the packet was transmitted.

Development began with the creation of a dedicated radio driver tailored to the nRF5340's dual-core architecture and its hardware. This involved careful configuration of the radio peripheral's state transitions (Disabled, Ramp-up, Idle, and RX/TX), as depicted in Figure 4.2. To optimize performance and conserve power, SHORTCUTS, described in Section 2.1.1, was used.

To facilitate communication between the application and network cores, a shared memory area was established, as illustrated in Figure 4.3. This area contains

two FIFO buffers, one for incoming (RX) data and one for outgoing (TX) data. Semaphore-based locks protect these buffers to prevent data corruption during concurrent access.

Finally, using drivers provided by nRF, USB communication was implemented to bridge the nRF5340 and the PC. Using Python and the Pyserial library, a script was developed to enable reliable two-way communication, facilitating the transfer of captured data packets for analysis and visualization.

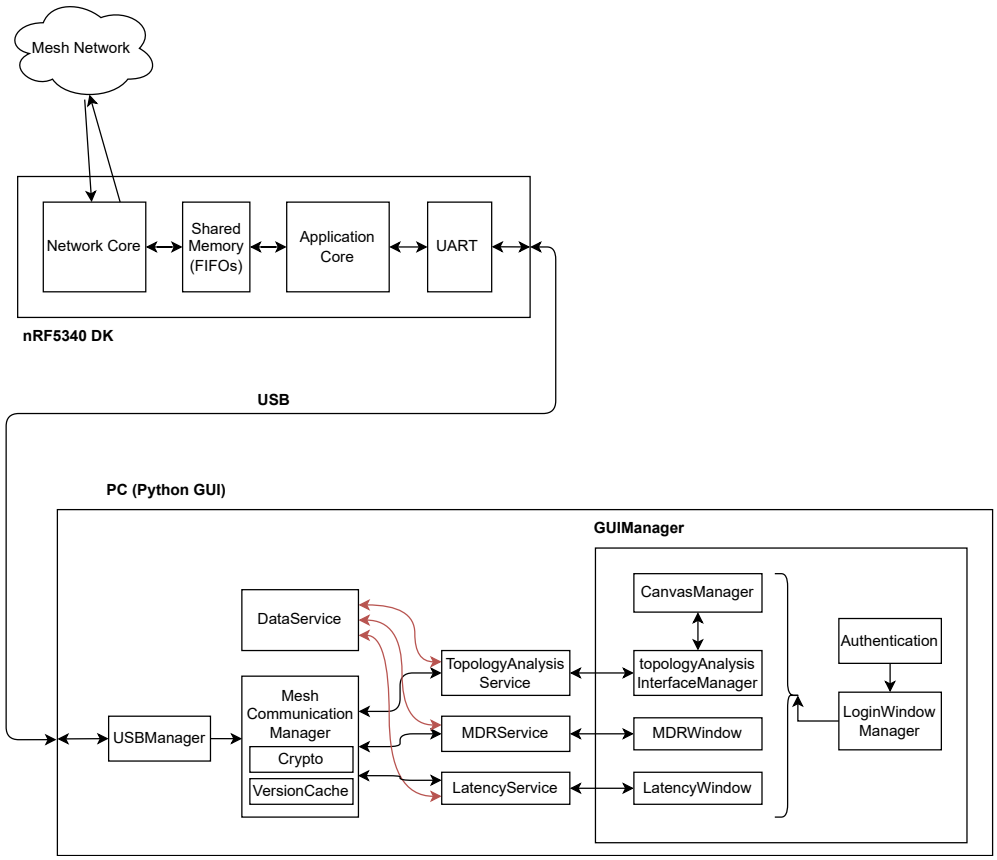


Figure 4.1: High-level architecture of the system

### 4.1.1 Radio Driver

The radio driver for the nRF5340 DK resides in `remote/src/driver/radio.c` [20]. It initiates its operations by initializing the radio hardware through the `radio_init_hardware()` function. This function configures the essential registers required for the radio's proper operation. During initialization, frequency and access address settings are specified using `NRF_RADIO->FREQUENCY` and `NRF_RADIO->PREFIX0`. Packet configurations are also established to define how data packets are structured and processed. CRC settings are configured via `NRF_RADIO->CRCPOLY`, `NRF_RADIO->CRCCNF`, and `NRF_RADIO->CRCINIT` to ensure the integrity of data during radio operations. To prevent any unintended triggers during setup, radio interrupts are

cleared and disabled initially using `NRF_RADIO->INTENCLR` and `NVIC_DisableIRQ(RADIO_IRQn)`.

Following the setup, the radio is prepared to transition into either transmission (TX) or reception (RX) modes depending on network requirements. This transition is managed by engaging specific tasks (`TASKS_TXEN` or `TASKS_RXEN`) and employing SHORTCUTS to facilitate seamless state transitions between `READY` and `START`, and `END` and `DISABLE` events, see Figure 4.2.

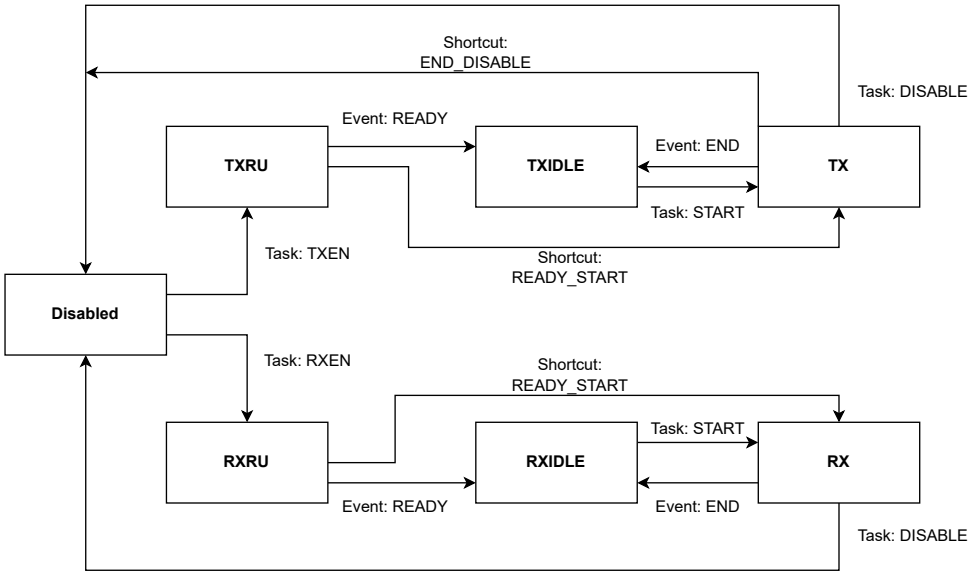


Figure 4.2: State Machine of Radio Peripheral

In RX Mode, the `radio_start_listening()` function is called to prepare the radio for receiving data. It sets the `TASKS_RXEN` to activate the receiver and configures the `PACKETPTR` to point to the location designated for incoming data, typically a buffer for incoming packets. Reception is further facilitated by enabling SHORTCUTS between `READY` and `START` events, and `END` and `DISABLE` events, ensuring smooth transitions through the radio states without software intervention.

When the device needs to switch from RX to TX mode, such as when data needs to be sent while the radio is in RX mode, the transition is handled by the `on_event_disabled()` function, triggered by a `DISABLED` event. This event occurs once the radio finishes processing a received packet and the `END` event is handled. If there are packets ready to be sent, detected through the inspection of the outgoing FIFO buffer, the `radio_start_sending()` function is called. This function prepares the transmission buffer and sets the `TASKS_TXEN`, switching the radio to TX mode.

In TX Mode, the data to be transmitted is loaded into the radios `PACKETPTR`. The transmission process is automatically managed by SHORTCUTS that link `READY` to `START` and `END` to `DISABLE`. This configuration ensures that the radio

starts transmitting as soon as it is ready and returns to a disabled state once the transmission is complete.

After a packet is transmitted and the radio transitions back to a disabled state, indicated by the `DISABLED` event, the `on_event_disabled()` function assesses whether more data needs to be sent. If no additional data is queued for transmission, the `radio_start_listening()` function is re-invoked to switch the radio back to RX mode, preparing the device to receive more incoming packets.

### 4.1.2 FIFO and shared memory

The radio driver utilizes a shared memory area to facilitate communication between the application and network cores. This shared area includes two FIFO buffers: one for incoming (RX) data and another for outgoing (TX) data, situated at memory addresses `0x20070000` for RX and `0x20075000` for TX, see Figure 4.3. The source code is located in `src/libs/compact_fifo_mutex.c` and `src/libs/compact_fifo_mutex.c` for both application and network cores respectively [20]. These buffers are important for the efficient and synchronized transfer of data between the cores and their subsequent processing or transmission.

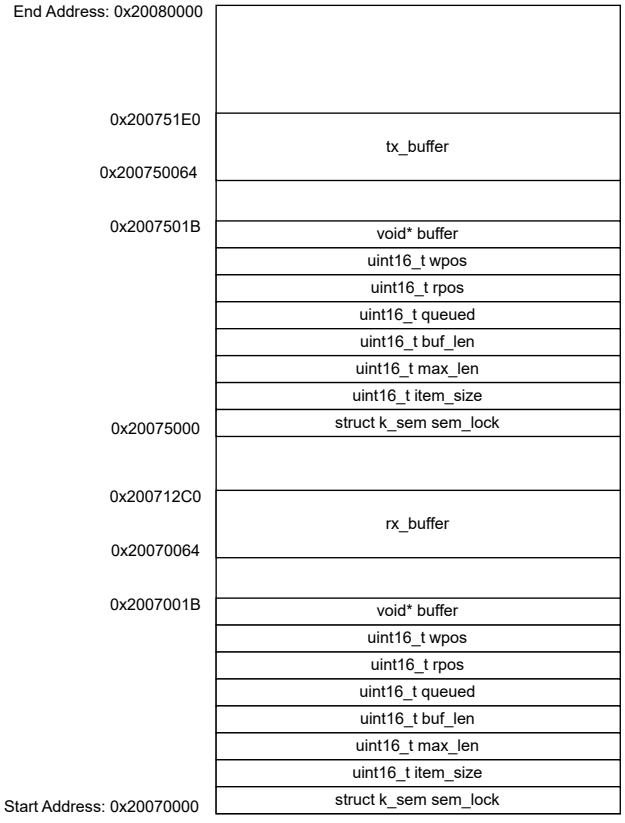


Figure 4.3: Shared Memory Structure with FIFOs

The RX FIFO buffer, initialized at the address `0x20070000`, is primarily used

by the network core to place incoming packets from the mesh network. These packets are then consumed by the application core, which processes them and sends them to the PC via USB. This buffer is designed to handle a large number of incoming packets, as indicated by its capacity to store up to 50 elements, defined by `RADIO_RX_BUFFER_NUM_ELEMENTS`.

Conversely, the TX FIFO buffer, located at `0x20075000`, is used for outgoing data. Packets coming from the USB, typically sent by a host PC, are placed into this buffer by the application core. The network core then retrieves these packets from the buffer to transmit them into the mesh network. This buffer is slightly smaller, configured to hold up to 10 elements , defined by `RADIO_TX_BUFFER_NUM_ELEMENTS`, reflecting its role in handling typically less frequent transmissions compared to receptions.

Both FIFO buffers are managed using semaphore-based locks, ensuring thread safety and preventing data corruption during concurrent access. This is critical in a multi-core environment where both cores may need to access the buffers simultaneously for reading and writing operations.

The implementation of these FIFOs involves several functions within the `compact_fifo_mutes.c` file, which handle initialization, clearing, pushing, popping, peeking, and querying the queued elements. Functions like `compact_fifo_mutex_init` initialize the FIFO with specified parameters including buffer pointers, item size, and buffer length. The `compact_fifo_mutex_clear` function resets the FIFO, clearing all its contents, which is particularly useful for reinitializing the FIFO state during error recovery or system resets.

Push and pop operations are managed by `compact_fifo_mutex_push` and `compact_fifo_mutex_pop`, respectively. These functions handle adding new items to the FIFO and removing items from it, all while managing the read and write positions within the buffer to maintain the FIFO order. Semaphore locking mechanisms within these functions ensure that these operations are atomic, preventing race conditions and ensuring that data integrity is maintained even under high throughput conditions.

### 4.1.3 USB Driver

The USB driver in `src/driver/usb.c` handles both incoming and outgoing data transfers through the use of dedicated buffers and routines designed to ensure reliable communication. It is initialized through the `usb_init` function, which sets up the necessary configurations for the USB device and checks its readiness.

The `usb_send` function manages the packaging and sending of data over USB. It constructs a `usb_packet_t` structure, which includes start and end delimiters as well as the actual radio packet data to be sent. This data is then pushed into the transmission buffer using `compact_fifo_push`. Transmission readiness is checked by

`uart_irq_tx_complete`, and if the device is ready, `uart_irq_tx_enable` is invoked to enable the transmission interrupt, which handles the actual data sending.

Data reception is managed by the `interrupt_handler` function, which is set as the callback for UART interrupts via `uart_irq_callback_set`. This handler checks for updates and pending interrupts with `uart_irq_update` and `uart_irq_is_pending`. It handles received data when `uart_irq_rx_ready` is true, reading the data into a buffer from the UART FIFO and then using a callback function `cb` to send this data to the network core.

The main thread of the application core is responsible for pushing and popping data to and from the USB FIFO. It is also responsible to transfer that USB data between shared memory FIFO and USB FIFO.

## 4.2 GUI Software Design

To unlock the insights captured by the hardware, the monitoring tool features a user-friendly GUI. The GUI was built using DearPyGUI and provides visualization and analysis of the mesh network data. This section describes design of each module build within the GUI. Software that handles the GUI is located in `gui/` [21].

### 4.2.1 USB Driver Module

As depicted in Figure 4.1, the Python GUI is connected with the monitoring tool through the `USBManager` module. The USB Driver module, located in `drivers/usb.py` [21], was designed with careful consideration for robustness and efficiency. It uses the well-established `serial` library in Python to interface with the monitoring hardware over a USB connection.

A main principle of the design of this module is the implementation of the Singleton design pattern, ensuring that only a single instance of the `USBManager` class exists within the entire application. This centralized management prevents potential conflicts with the USB serial connection, ensuring communication stability.

The connection process is handled by the `initiate_connection` function. It continuously scans available serial ports, comparing their identifiers against the hardware's predefined description. Once a match is found, the function initializes the serial connection with appropriate parameters as baud rate and timeout settings.

Maintaining data integrity was a key focus within the USB module. With the `clear_buffer` function it is possible to remove any leftover data from previous communication cycles, ensuring a clean channel for accurate data exchange.

The module provides streamlined mechanisms for both sending and receiving data over USB. The `send_data` function simplifies the process of transmitting messages to the hardware, while the `receive_data` function carefully detects specific delimiters

to accurately frame individual messages. This message structure enables reliable interpretation and processing.

### 4.2.2 Mesh Communication Module

Next step in the flow is the `MeshCommunicationService` located in `network/mesh_communication.py`, that simplifies the process of interacting with the BLE mesh network over a USB connection. Its core purpose is to handle all necessary validation and packet translation tasks, shielding the user from the underlying complexities. The user's primary interaction involves providing the payload to be sent, and in return, the module delivers the payload and metadata of received packets. It integrates tightly with other system components, including the `USBManager`, a cryptographic subsystem, and a `VersionCache`, see Figure 4.1. Parts of the software code that are confidential are not shown.

This service too follows the Singleton design pattern. Upon initialization, it uses the `initiate_connection()` function from the `USBManager` to establish a link with the hardware. The `initialize_crypto_keys()` function sets up encryption/decryption and a `VersionCache` object is created for packet integrity management.

Key functions within the module manage mesh radio control. Functions `enable_radio()` and `disable_radio()` send the appropriate commands over USB, allowing the tool to manage the device's operating mode.

The `MeshCommunicationService` handles both packet transmission and reception. When sending data, it utilizes `construct_mesh_packet()` to assemble packets with headers and payloads, applying encryption if necessary. Packets are then sent via the `USBManager's send_data()` function. Incoming packets are received using the `USBManager's receive_data()` function. Timestamps are extracted from the packet's metadata, headers are validated using `verify_headers()`, payloads are decrypted and a `check_version_cache()` updates local version cache and ensures the packet is unique. Functions `verify_ble_phy_header()` and `verify_rbc_header()` check the structure of received packets against mesh protocols, while `crc_check()` confirms the integrity of decrypted data.

### 4.2.3 Authentication Module

From the user perspective, everything starts with the log-in window handled by the `LogInWindowManager`, as illustrated in Figure 4.1. The `LogInWindowManager` is located in `gui/auth_tab` [21], where a mesh network site to be monitored can be selected. If no such site is available in the drop-down menu, then authentication must be performed. The Authentication Module is responsible for securely establishing user access to the mesh network and retrieving essential data for monitoring and analysis. It safeguards the integrity of the mesh network by ensuring that only authorized users can observe network traffic and interact with mesh devices.

The authentication flow consists of two core components. The `GuiManager` object

initializes the user interface, including a dedicated authentication tab. It manages the visual aspects of the authentication process, such as displaying the input fields and providing feedback to the user. Core authentication logic is delegated to the Authentication Module.

Within the authentication tab, the user selects a server environment and enters their login credentials. Upon submission, the Authentication Module takes over, securely communicating with the backend service `PlejdParse` via `AuthenticationService` in `network/authentication.py`. The credentials are transmitted in encrypted format, and `PlejdParse` verifies them against the cloud server's user database. If authentication is successful, `PlejdParse` retrieves the following:

- **Cryptographic Key:** Used to decrypt encrypted mesh network traffic.
- **Mesh Access Key:** Grants the monitoring tool the ability to interact with the mesh network.
- **Device and Index List:** A detailed list of devices within a specific mesh site and their associated indices.

After a successful login, the Authentication Module sends mesh access key to the monitoring tool through the `MeshCommunicationService` and provides the acquired data back to the GUI Manager. The GUI Manager then presents the user with a drop-down menu to select a specific mesh site. This enables targeted monitoring of a particular mesh site. Data about the selected site is parsed and stored locally.

### 4.2.4 Mesh Topology Analysis Module

The Mesh Topology Analysis module in `gui/topology_analysis_interface.py` [21] is designed to analyze and visualize the network topology of a BLE mesh network. This module works in close collaboration with the GUI, `gui/gui_manager.py`, and a back-end logic layer, `gui/canvas_manager.py`, to provide users with an interactive experience for managing the topology analysis process.

The analysis begins within `GuiManager` in `gui/gui_manager.py`. After successful log-in, the `GuiManager` shows a window where it enables the topology analysis functionality within the GUI. The window displays start/stop button managed and canvas (managed by `CanvasManager` class, where the topology is drawn and a sidebar next to the canvas). The sidebar displays information on the number of nodes and edges, as well as node names and their corresponding MAC addresses.

User interactions with the GUI are handled by the `TopologyAnalysisInterfaceManager`. This manager efficiently translates user inputs and commands, such as analysis start/stop and node or path selections, into instructions for the back-end logic. In response, the back-end starts or stops the analysis or calculates the shortest paths between the specified source and destination nodes, supporting in-

depth analysis of the mesh network's connectivity.

To initiate the analysis, the `TopologyAnalysisInterfaceManager` calls upon `TopologyAnalysisService` in `network/topology_analysis_service.py`. The `TopologyAnalysisService` handles the process of discovering the structure of the mesh network, using the `MeshCommunicationService` for network interactions. This involves carefully stimulating individual mesh nodes by sending a `GET` command to each node index and analyzing responses to reveal connections within the network. The `TopologyAnalysisService` employs concurrent threads to manage the transmission of stimulation commands and the reception of responses.

The data collected from the mesh nodes, including timestamps, MAC addresses, and other relevant packet information, is saved in a CSV file and processed for analysis by the `DataService` class. The analysis itself is described in Section 4.3.1 below. The module periodically extracts and interprets this data to identify unique connections between nodes. The `NetworkX` Python library, described in Section 2.4.2, is used to construct a graph representation of the mesh network topology.

The processed topology data, managed within the `NetworkX` graph, is then rendered within the GUI's canvas in `gui/gui_manager.py`. Node and edge coordinates from `NetworkX` are carefully mapped to fit within the GUI's display, ensuring a clear and informative visualization. Users can directly engage with the visualized topology, selecting nodes to view shortest paths that are subsequently highlighted on the canvas. The visualization also serves as a foundation for further analysis, such as calculating MDR and latency between nodes.

### 4.2.5 Message Delivery Ratio (MDR) Analysis

The MDR analysis module is a component of the BLE mesh monitoring tool that is designed to continuously assess the reliability of message transmissions within the network. The software code for this module resides in `gui/mdr_tab.py` [21].

The MDR analysis process begins within the `GuiManager` class in `gui/gui_manager.py`, which establishes the main window of the application and organizes its layout. The `MDRTab` class in `gui/mdr_tab.py` is responsible for creating a dedicated "MDR Analysis" option within the application's menu bar, providing a clear entry point for users to initiate measurements. This analysis can only be performed after the network topology has been discovered.

The `MDRTab` class is specifically made for MDR analysis, setting up a dedicated tab within the analysis menu. When interacted with, it triggers the MDR analysis window managed by the `MDRWindow` class. This window enables users to select source and destination nodes through drop-down menus populated based on MAC to label mappings obtained from mesh topology analysis results. Error handling and validation are important for the user interaction within the MDR window. The system checks to ensure that both nodes are selected, distinct, and linked by a

viable path before initiating the MDR analysis. Should these conditions be met, the `MDRService` class in `network/mdr_service.py` proceeds with the analysis; if not, the system displays error messages to guide the user.

The `MDRService` class interacts directly with the network via the `MeshCommunicationService`. This service begins a background thread for collecting packet data, vital for the MDR analysis. The class periodically processes the collected data, calculates the MDR and throughput, and updates the GUI with these results through a callback mechanism linked to `self.mdr_callback`. This mechanism ensures that results are dynamically plotted in two separate plots within the MDR window using Dear PyGui's bar plot functionalities. One plot displays the MDR percentages while the other shows throughput metrics, thus providing interactive and real-time data visualization that enhances the user interface. The system allows for multiple runs of the analysis for the same link between source and destination, with varying throughputs, which can be visualized distinctly in these plots. This feature is particularly useful for assessing network performance under different conditions and is further detailed in Section 4.3.2.

Additionally, the GUI includes robust session management features, enabling users to start and stop the analysis as needed.

### 4.2.6 Latency Analysis

The latency analysis module within the BLE mesh monitoring tool is designed to measure communication latency between mesh nodes, offering valuable insights into network performance. The software code for this module resides in `gui/latency_tab.py` [21].

The latency analysis process is initialized by the `GuiManager` in `gui/gui_manager.py`, that establishes the application's main window. Within the `GuiManager`, the `LatencyTab` class adds a dedicated "Latency Analysis" option to the application's menu bar, providing users with a clear entry point to initiate measurements.

Upon selecting this option, the `LatencyTab` class launches the `LatencyWindow`. This window offers a user-friendly interface for configuring latency measurements, including dropdown menus for selecting source and destination nodes. When the user clicks the "Start Measurement" button, the `LatencyWindow` coordinates with the `LatencyService` in `network/latency_service.py` to begin data collection and analysis.

The `LatencyService`, leveraging the `MeshCommunicationService` for network interactions, handles the core latency measurement process. Latency quantifies the duration required for a packet to travel from the source node to the destination node. Specifically, it measures the time from when the packet is sent from the source node until it is rebroadcast by any neighboring node of the destination node. All incoming traffic is saved in a CSV file. The service continuously extracts timestamps associ-

ated with packet transmissions and receptions, enabling the accurate calculation of latency. The analysis itself is described in Section 4.3.3 below.

For a user to gain deeper understanding of the network behavior, the `LatencyService` calculates both empirical and theoretical latency values. Empirical latency directly reflects the observed latency. Theoretical latency is calculated using a latency estimation formula from work in [4] and adopting it to the current network. The way in which it is adopted is described in Section 5.3. The formula considers factors such as the number of hops between the nodes and known network topology information. This comparison helps identify potential bottlenecks or deviations from ideal network performance.

Upon completion of data collection, the `LatencyWindow` dynamically creates a histogram to visualize latency distribution in one subplot. It also displays theoretical latency, average latency, and maximum recorded latency in another subplot. The values for theoretical, average, and maximum latency are preserved between each measurement session to facilitate comparison. This graphical representation helps users in understanding the latency characteristics of the network. The interface allows users to start, stop, and restart latency measurements, offering flexibility for network analysis.

## 4.3 Data Analysis

The software code for data analysis is available in [21] under `data/data_service.py`. For data analysis `DataService` class is made. It is also a Singleton within the system and is responsible for processing acquired mesh network data in order to discover mesh topology and calculate MDR for different throughputs, and latency. Internally, the `DataService` maintains several data structures to manage the analysis process.

### 4.3.1 Mesh Topology

The topology analysis algorithm within the system is responsible for discovering the topology of the mesh network. It analyzes collected data to figure out how nodes connect with each other and their overall relationships within the network. The algorithm starts by loading the discovery data from a CSV file. This data includes details of all the packets that have traveled across the network, including their timestamps, contents, and the MAC addresses of the nodes that sent them. It converts the timestamps into a structured format for easier analysis and then identifies all the unique MAC addresses present in the network, each MAC address representing a single node.

Next, the algorithm focuses on pinpointing 'RESPONSE' messages. These messages are important because they are triggered by GET requests from the analysis tool itself and rebroadcast by nodes throughout the network. For each unique MAC address, the algorithm finds messages that represent fresh information (new version numbers), which helps narrow down the search for messages likely initiated by a

particular node. To avoid incorrectly identifying rebroadcasts as original messages, the algorithm uses site data that indicates the correct index that should be associated with messages originating from each node.

The key part of the algorithm involves discovering connections using the Trickle algorithm's configuration. In this project, the Trickle algorithm is configured with the following parameters: $I_{min} = 32$ ms, $I_{max} = 4096$ ms, and $k = 4$. The data analysis algorithm looks for direct connections by trying to match the identified source messages with same messages sent by other nodes within a short time window. If another node rebroadcasts the same message (with matching index, payload, and version) within 32 milliseconds, it signals a strong probability of a direct connection between the two nodes. The time window aligns with the network's Trickle algorithm configuration, which manages rebroadcast behavior. The algorithm includes safeguards to minimize inaccurate detections. For instance, it ignores instances where the time difference is less than 16 milliseconds to avoid accidentally interpreting missed original messages as connections. It also requires a minimum number of rebroadcasts to confidently determine a true connection.

The Python code implements the algorithm as follows. The `data_processing_find_connections()` function in `data/data_service.py` is the primary function, and it begins by loading data with the help of the `load_data()` and `_load_site_data()` functions. Timestamps are formatted consistently using `get_packet_timestamp()`. The algorithm then filters messages, separating SET/GET messages bound for the destination and other message types sent on the source node's index. It determines the neighbors of the destination node using the `node_neighbor_map`. Source messages and neighbor rebroadcasts are matched using `pd.merge()` and the time difference between these pairs is calculated by `calculate_latency()`. Various filtering criteria are applied: duplicates are dropped with `drop_duplicates()`, time windows (16ms  32ms) are used for validation, and a connection threshold, defined by `NETWORK_TOPOLOGY_THRESHOLD`, determines if a connection is sufficiently likely.

### 4.3.2 Message Delivery Ratio (MDR)

The MDR analysis algorithm employs a two-step process to determine MDR for a specific throughput for messages sent between a source node and a destination node. Step 1 focuses on identifying acknowledgements for SET and GET messages that are sent from source to destination. The algorithm begins by isolating all SET and GET messages originating from the source and directed towards the destination node's index. For each of these SET or GET messages, the algorithm anticipates a response from the destination node containing an incremented version number. All observed acknowledgement packets (RESP and ACK) from any node in the network are compared against the original source messages. It's possible that the tool might miss the initial response directly from the destination, but other nodes rebroadcasting that same response still act as confirmation that the destination node has received the source message. A successful match based on index and version number confirms an acknowledgement.

Step 2 aims to identify acknowledgements for other types of messages that are sent by the source node on its own index and should be rebroadcasted by the destination node. The algorithm isolates messages from the source node that are not SET or GET and are sent on the source's own index. Next, all messages sent by the destination are identified, and a pairing attempt is made between the isolated source messages and the destination packets. A successful match indicates a direct acknowledgement. Unacknowledged messages from the previous steps are noted. Following this, the destination node's neighbors are determined, and packets sent by those neighbors are analyzed. The algorithm searches for neighbor rebroadcasts that match the contents of the unacknowledged messages.

A single moving window of 32 ms (matching the Trickle algorithm's initial rebroadcast interval) is applied to all potential rebroadcasts. If a destination node doesn't rebroadcast a message, it implies either the message wasn't received, or it was received but dropped due to Trickle redundancy. If fewer than four neighbors rebroadcast the message within the initial 32ms interval, the destination node would be expected to rebroadcast it at least one time. Therefore, if at least four rebroadcasts occur within the window at any point of time, the algorithm infers that the destination node received the original message and, due to Trickle behavior, suppressed its own rebroadcast.

The `calculate_mdr()` function in `data/data_service.py` implements the MDR calculation logic. Step 1 begins by filtering for relevant SET/GET messages (`source_set_and_get_messages`) as described in the algorithm above. It anticipates incremented-version responses, `'Next_Version'`, and merges observed acknowledgements using `pd.merge()` to ensure even potentially missed responses are considered. Duplicate matches are removed with `drop_duplicates()` and the data is split into acknowledged, `acknowledged_merge_set_and_get_messages`, and unacknowledged messages, `unacknowledged_merge_set_and_get_messages`.

Step 2 in the code parallels the algorithm's logic for identifying direct acknowledgements for other message types. It isolates relevant source messages, called `other_messages`, and attempts to pair them with destination messages (`destination_messages`) using `pd.merge()`. Matches represent direct acknowledgements, stored in `acknowledged_other_messages`.

For indirect acknowledgements, if there are no neighbors, the MDR is calculated using `add_or_update_mdr_pair()`. Otherwise, neighbor packets are retrieved using `neighbor_packets`, missing payloads are addressed, and they are merged with unacknowledged messages to align with Step 2 of the algorithm. The 32ms rolling window, implemented with `pd.Timedelta()`, `rolling()`, `count()`, and the threshold of four rebroadcasts directly implement the Trickle-based inference described in the algorithm.

Finally, the code combines direct and indirect acknowledgements (`acks`). MDR and throughput are calculated and passed to `add_or_update_mdr_pair()` for storage

and updates.

### 4.3.3 Latency

Latency Data Analysis involves quantifying the delay between message transmission from a source node and its subsequent rebroadcast by neighbors of the destination node. The algorithm utilizes a structured sequence to accurately calculate and evaluate the latency for each message within a specified network environment.

Step 1 of the algorithm starts by identifying all messages originating from the source node. It categorizes messages into two groups: SET and GET messages directed at the destination node's index, and other types of messages that are sent on the source node's own index but are not SET or GET. This distinction is critical as it defines the scope of messages relevant for further analysis, focusing only on those which the destination node is expected to acknowledge or rebroadcast.

In Step 2, the algorithm locates the neighbors of the destination node, crucial for tracking the rebroadcast of messages. The neighbors are acquired from the discovered Network Topology using built-in NetworkX functions. This step ensures that subsequent analysis considers only the rebroadcasts from nodes that are directly related to the destination node, thereby filtering out irrelevant data.

Following this, in Step 3, the algorithm retrieves packets from these neighbors and filters them to include only those packets that match the source messages identified earlier. This is achieved by merging datasets based on common message attributes such as version, index, and payload, ensuring that only corresponding pairs of original messages and rebroadcasts are considered.

In Step 4, for each matched pair of source message and neighbor packet, the time difference is calculated. This measurement is pivotal as it represents the latency of the message from the moment it leaves the source node to when it is rebroadcast by any of the destination node neighbor. The algorithm then filters out duplicate entries based on MAC address to consider only the first occurrence of each message, which represents the earliest rebroadcast time. The final steps involve calculating both average and maximum latencies. These statistics are important for understanding the range of latency behaviors in the network.

The `calculate_latency()` function in `data/data_service.py` is at the heart of the latency analysis process. It starts by loading message data using `load_data()`, retrieving site and topology information via `_load_site_data()`, and ensuring timestamps are properly formatted with `get_packet_timestamp()`.

The code extracts the source and destination indexes from the loaded `site_data`. To streamline the analysis, `get_first_occurrences()` isolates the first instance of each unique message sent by the source node. These messages are then separated into SET/GET messages (`source_node_set_get_messages`) directed at the destination

and other message types (`source_node_other_messages`) sent on the source's own index.

The `node_neighbor_map` is essential in determining the neighbors of the destination node. The code filters all packets sent by these neighbors, retaining those that match the source messages in terms of version, index, and payload. The `fillna()` function handles missing payload cases, and the `pd.merge()` function performs the matching task.

With the paired source-neighbor packets, `calculate_latency()` determines the time difference (`TimeDiff`) between the original transmission in the `all_source_node_messages` dataset and the rebroadcast timestamp found in `neighbor_packets`. The `drop_duplicates()` function ensures only the earliest (and likely most accurate) rebroadcast is considered for each distinct message.

For in-depth statistics, the code uses `groupby()` to form groups based on the 'Version', 'Index', and 'Payload' attributes within the `merged_df_mac_filtered` dataset. The aggregate function `agg()` then finds the minimum and maximum latency for each message group. Finally, the average and maximum latency values are calculated across all messages, providing a comprehensive overview of the network's latency characteristics.

# 5

# Experimental Setup

This chapter details the experimental setup used to evaluate the capabilities of the developed monitoring tool within a controlled mesh network environment. The experiments are designed to validate the tool's accuracy and responsiveness in identifying mesh network topologies, MDR with throughput and latency under various conditions. Table 5.1 provides the simulation parameters across all experiments.

| **Parameter** | **Value** |
|---|---|
| Transmission Power | 3 dBm |
| Trickle I_min | 32 ms |
| Trickle I_max | 4096 ms |
| Trickle k | 4 messages |
| Radio Throughput | 1 Mbps |
| GET Packet Size | 23 Bytes |
| RESP Packet Size | 29 Bytes |
| DR Packet Size | 26 Bytes |

Table 5.1: Simulation Parameters

The following sections will detail the setup and procedures for assessing each of these functionalities, highlighting the custom-designed experiments and the specific metrics used to gauge the tool's effectiveness and accuracy.

## 5.1 Mesh Topology Analysis

Validating the mesh topology analysis capabilities of the developed monitoring tool requires a carefully designed experimental environment. Relying on real-world mesh network deployments presents several challenges that hinder accurate evaluation. Existing mesh networks often evolve organically, resulting in complex and potentially unknown topologies. This makes it difficult to establish a definitive ground truth against which to evaluate the tool's output. Additionally, achieving a specific topology in real-life mesh networks can be difficult due to factors like signal interference or the physical range of nodes. For instance, ensuring that certain nodes have absolutely no link between them is often impractical. Real-world mesh networks may also experience nodes joining or leaving unexpectedly, potentially making it

difficult to assess the accuracy of the tool's topology analysis in the face of those changes.

A custom-designed mesh network overcomes these challenges. By intentionally designing the network topology, the experiment creates a reference point against which the tool's output can be compared. This enables precise evaluation of the tool's accuracy in detecting both the presence and absence of mesh connections. Manipulating the placement and programming of nodes in the custom mesh allows for the creation of specific network scenarios. This includes testing the tool's response to situations where clusters of nodes have no connection or where only one relaying node connects these clusters.

The custom mesh network features a topology consisting of two distinct clusters, as illustrated in Figure 5.1. Cluster 1 is fully connected, with each node having a direct link to all other nodes in that cluster. Cluster 2 takes the form of a tree structure, with varying degrees of hops between the nodes. These two clusters are then connected by a single link between one node from each cluster.

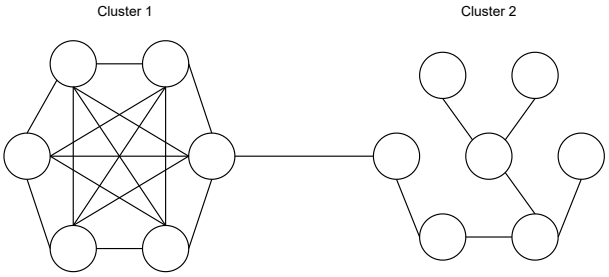


Figure 5.1: Experimental Mesh Topology setup for testing Mesh Topology Analysis functionality

This design offers several advantages for validation. The defined topology provides a clear reference point for evaluating the tool's accuracy in detecting both present and absent connections. Additionally, the design tests the tool's ability to handle fully connected clusters, hierarchical tree structures, and critical single-link inter-cluster connections. The setup also provides a starting point to assess how the tool handles varying network sizes.

To evaluate the tool's ability to detect the network topology and its responsiveness to changes, the experiment is conducted in four phases. Each topology discovery round lasts for 3 minutes.

- **Cluster 1 Discovery:** Only Cluster 1 is active. This establishes a baseline discovery time and confirms accurate detection of the initial topology.
- **Cluster 2 Discovery:** Only Cluster 2 is active, testing the tool's ability to discover a separate, independent cluster.

- **Cluster Addition:** With Cluster 1 discovered and active, Cluster 2 is turned-on (and other way around). This measures the tool's responsiveness in detecting the network expansion.

- **Simultaneous Discovery:** Both clusters are activated from the beginning, assessing how the tool performs when handling the entire network concurrently.

To achieve the desired topology, node firmware is modified to listen only to predefined MAC addresses. This selective listening enables precise control over the connections formed within the mesh. Upon pressing the "START" button within the Mesh Topology Analysis window in the GUI, two threads are created: one for transmitting GET messages to specific node indices, and another dedicated to listening. GET messages from the GUI are relayed to the monitoring tool hardware, which broadcasts them into the mesh. Responses (RESP) and rebroadcasts are captured by the monitoring tool and transmitted back to the GUI, where they undergo verification, decryption, and writing to a CSV file.

Finally, a data processing thread is executed every two seconds. This thread analyzes the data in the CSV file according to methods outlined in the 4.3.1 section. Based on this analysis, the Mesh Topology Window is dynamically updated to reflect the current state of the mesh network.

To evaluate the tool's performance against the established ground truth, the following metrics will be used:

- **Accuracy:** The percentage of correctly detected connections present in the actual network topology.

- **False Positives:** The percentage of connections reported by the tool that do not actually exist in the network.

- **False Negatives:** The percentage of existing connections that the tool fails to detect.

- **Responsiveness:** The time taken for the tool to correctly detect and display the updated topology after the second cluster is activated.

## 5.2 Message Delivery Ratio (MDR) Analysis

The evaluation of the tool's MDR analysis capabilities relies on three custom-designed mesh networks, as illustrated in Figures 5.2 and 5.3 below.

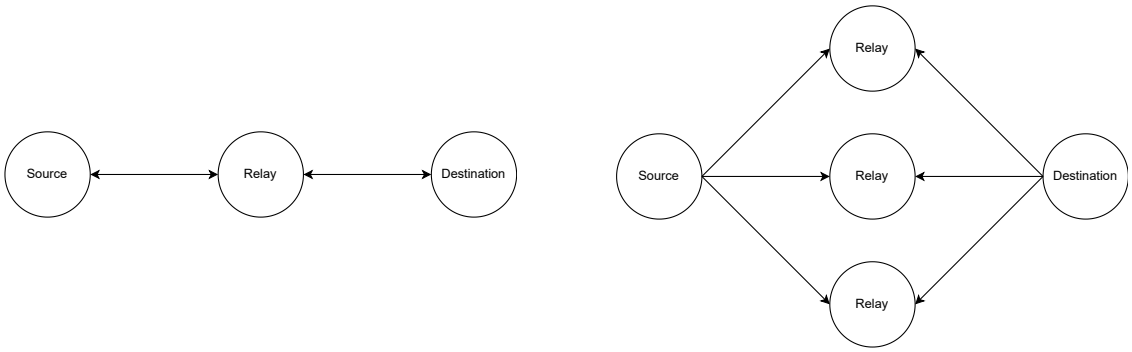


Figure 5.2: Single-Hop MDR Test Topologies A and B. All relay nodes has a link with each other.

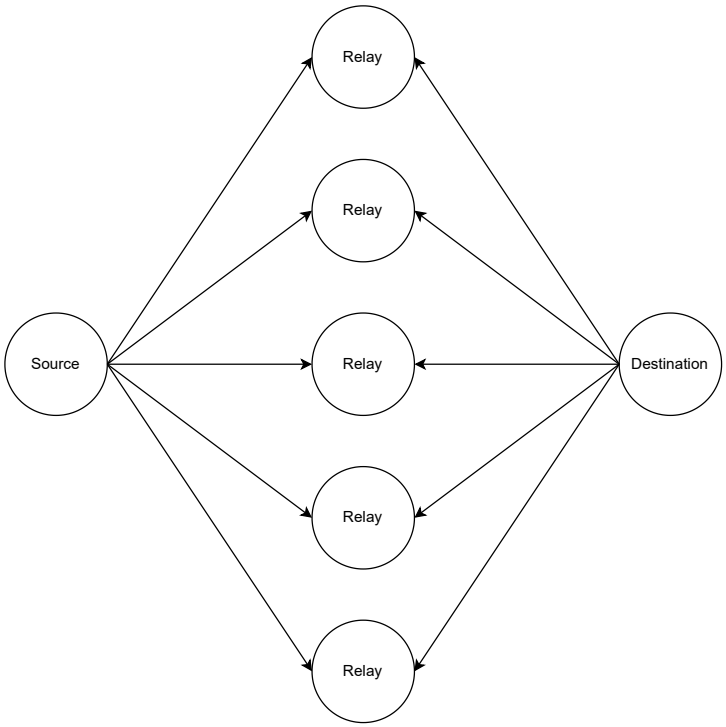


Figure 5.3: MDR Test Topology C: Single-Hop, Five Relay Nodes. All relay nodes has a link with each other.

Topology A, one the left side in Figure 5.2, consists of a source node, a destination node, and a single relay node. The source and destination nodes are configured not to communicate directly, requiring messages to pass through the relay node, thus creating a single hop. Topology B, one the right side in Figure 5.2, is similar to Topology A but includes three relay nodes to facilitate multiple routing paths and potential for packet rebroadcasting, increasing the network complexity. Topology C in Figure 5.3 expands further with five relay nodes, providing the most complex

scenario intended to test the tools robustness in a denser network environment and its capabilities of detecting indirect acknowledgements, as described in Section 4.3.2.

Each topology is tested under eight different message throughput rates to assess the tool's performance across a range of network loads. The specific rates tested are 1, 2, 4, 8, 10, 12.5, 16.66, and 25 messages per second (Msg/s) for 100 DR messages.

Firmware modifications were made to the source and destination nodes to log the number of messages transmitted and received and to ignore each other messages. This data serves as a basis for calculating the actual MDR, which is then compared against the MDR recorded by the tool to evaluate its accuracy. Upon pressing the "START" button within the MDR Window in the GUI an RX thread is created where messages captured by the monitoring tool undergo verification, decryption, and writing to a CSV file.

For each topology, the experimental procedure unfolds in several phases. Initially, the setup and initialization phase involves setting communication parameters and configuring and discovering each topology. This is followed by the data collection phase where both the tool and the nodesmodified to log transmission datarecord the number of messages transmitted and received. Post-test, the actual MDR is calculated by analyzing this logged data, focusing on the correspondence between the number of messages sent by the source node and those received by the destination node. Finally, the comparison and analysis phase involves comparing the MDR values recorded by the tool against the manually calculated values to pinpoint discrepancies and validate the tool's accuracy.

## 5.3 Latency Analysis

To verify accuracy of the latency analysis functionality, a theoretical model of message forwarding times is constructed. In a BLE mesh network, devices communicate by re-broadcasting messages to extend the range of the network. To avoid collisions, nodes introduce random back-off delays before each rebroadcast. Objective is to calculate the average time it takes for a message to be forwarded, considering these rebroadcast delays and the possibility of multiple rebroadcasts if earlier attempts fail.

**Assumptions:**

- Any neighbor is equally likely to be the forwarder.
- Collisions with other transmissions are neglected, since Trickle algorithm should minimize that probability.

As a starting point, the equation for Average Forwarding Time from [4] is used. Assuming that all nodes are listening on the same channel and the sending node has neighbors $n$, the average forwarding time can be described with the equation

below, where $t_{maximum}$ is the back-off time between 0 and $t_{maximum}$ calculated by the rebroadcasting node.

$$E[T] = \frac{t_{maximum}}{n+1} \tag{5.1}$$

The forwarding time in this work is controlled by the configuration of the Trickle Algorithm (2.3). $I_{min}$ is set to $32ms$, which means that the first re-broadcast will be calculated to be after a fixed time of $I_{min}/2 + random(0, I_{min}/2)$. For the first re-broadcast period, the equation can be rewritten as:

$$E[T] = \frac{I_{min}}{2} + \frac{I_{min}/2}{n+1} \tag{5.2}$$

Messages are re-broadcasted multiple times, up to a maximum limit, which is configured to be seven in this work. Each rebroadcast period has a fixed duration that doubles with each attempt, plus a random back-off time (0 up to the fixed duration). This means that the second re-broadcast will be calculated to be after $2^1 * I_{min}/2 + random(0, 2^1 * I_{min})$. The third will be calculated to be after $2^2 * I_{min}/2 + random(0, 2^2 * I_{min})$ and so on, up to the seventh re-broadcast. By calling the rebroadcast $r$ yields:

$$E[T] = 2^{r-1} * \frac{I_{min}}{2} + \frac{2^{r-1} * I_{min}/2}{n+1} = \frac{2^{r-1} I_{min}}{2}(1 + \frac{1}{n+1}) \tag{5.3}$$

The next rebroadcast is calculated only when the maximum time of that period expires. If the period of the first rebroadcast is $I_{min} = 32ms$ then the second re-broadcast will occur at $32 + 2^1 * I_{min}/2 + random(0, 2^1 * I_{min})$ ms. That must be taken into account in the equation.

$$E[T] = \frac{2^{r-1} I_{min}}{2}(1 + \frac{1}{n+1}) + \sum_{z=1}^{r-1} 2^z I_{min} \tag{5.4}$$

Each rebroadcast has a probability of success reaching the destination. If the destination fails to receive the rebroadcast, it will listen for the next one. The idea is to multiply the average time calculated for each period by the probability of needing that period (i.e., all previous attempts failing). For example, probability needing second rebroadcast period is probability that first period fails and the second one succeeds. The overall expected average forwarding time is then the sum of the expected times for each rebroadcast period to get the overall average forwarding time, taking into account success probabilities.

$$E[T] = \sum_{x=0}^{r-1} p(x)_{success}(\frac{2^x I_{min}}{2}(1 + \frac{1}{n+1}) + \sum_{z=1}^{x} 2^z I_{min}) \prod_{i=1}^{x} p(i-1)_{failure} \quad (5.5)$$

The nodes in the network are using Timeslot API in order to share radio peripheral time between SoftDevice, that handles core BLE functions, and application that performs its own wireless actions. The application gets interrupted by the SoftDevice each 152ms (which is the period of BLE Advertisments) for 10 ms. When that happens, the SoftDevice takes control of the radio peripheral and the application is blocked from performing any radio related actions for 10 ms. This means that the application is prevented from using the radio (receiving and transmitting) $10/152 = 0.0658 = 6.58\%$ of the time. The hardware of nRF micro-controllers does a good job to minimize interference and maximize success of message receiving. So roughly one can say that probability of failing to receive a message in each rebroadcast period is 6.58% and probability of success is 93.42%, which leads to:

$$E[T] = \sum_{x=0}^{r-1} 0.9342(\frac{2^x I_{min}}{2}(1 + \frac{1}{n+1}) + \sum_{z=1}^{x} 2^z I_{min}) \prod_{i=1}^{x} 0.0658^i \quad (5.6)$$

Equation 5.6 is applied for each hop individually. There is therefore need to be able to acquire number of neighbors for each hop to get accurate estimation of Average Latency from source node to destination node in question. That information is acquired from the results of Mesh Topology Analysis.

The measurements will be compared against the theoretical model using a setup similar to the one used in [4]. Each setup is run five times with 1000 messages to provide reliable average measurements.

- **Variable Hops:** Tests with 1 to 4 hops as in Figure 5.4.
- **Variable Neighbors**: Tests with 1 to 10 neighbors for a single hop, in Figure 5.3.

Firmware modification were made to the mesh network nodes only to respond to some specific MAC addresses in order to achieve desried mesh topologies. Upon pressing the "START" button within the Latency Window in the GUI an RX thread is created where messages captured by the monitoring tool undergo verification, decryption, and writing to a CSV file.

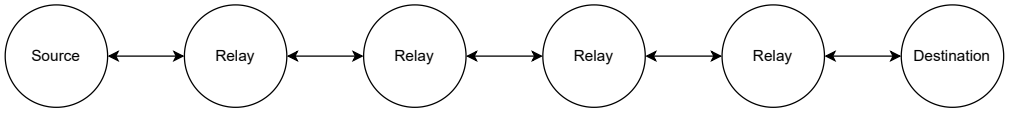


Figure 5.4: The topology used to conduct the multi-hop tests.

# 6

# Results

This chapter presents the experimental results obtained from the evaluation of the mesh network analysis tool. The experiments were designed to assess the tool's performance in three key areas: mesh topology discovery, MDR and latency.

## 6.1 Mesh Topology Analysis

The experiment described in the previous section evaluated the tool's ability to discover the network topology and its responsiveness to dynamic changes. The tool was tested using a custom mesh network featuring two distinct clusters, as illustrated in Figure 5.1. One cluster was fully connected, while the other had a tree structure. The clusters were connected by a single link.

The experiment was conducted in four phases, each lasting three minutes. Table 6.1 summarizes the results for discovery time, accuracy, false positives, and false negatives for each phase.

| | Cluster 1 | Cluster 2 | Cluster 2 after Cluster 1 | Cluster 1 after Cluster 2 | Cluster 1 + Cluster 2 |
|---|---|---|---|---|---|
| Average Discovery Time (s) | 11.03945 | 13.898 | 22.3535 | 26.946 | 31.2045 |
| Max. Time (s) | 17.15 | 25.15 | 36.26 | 37.24 | 49.38 |
| Min. Time (s) | 8.06 | 6.08 | 12.09 | 18.13 | 21.2 |
| False Positives | 0 | 0 | 0 | 0 | 0 |
| False Negatives | 0 | 0 | 0 | 0 | 0 |
| Accuracy (%) | 100 | 100 | 100 | 100 | 100 |

Table 6.1: Mesh Topology Analysis Results Across Discovery Phases

Figures 6.1 through 6.3 visually depict the tool's output for each experimental phase. The discovery time for Cluster 1 averaged 11.04 seconds, ranging from a minimum of 8.06 seconds to a maximum of 17.15 seconds. Cluster 2 had a slightly longer average discovery time of 13.89 seconds, with a minimum of 6.08 seconds and a maximum of 25.15 seconds. When Cluster 2 was discovered after Cluster 1, the average discovery time increased to 22.35 seconds, with a maximum of 36.26 seconds and a minimum of 12.09 seconds. Discovering Cluster 1 after Cluster 2 resulted in an even longer average discovery time of 26.95 seconds, a maximum of 37.24 seconds, and a minimum of 18.13 seconds. Simultaneous discovery of both clusters led to the

longest average discovery time of 31.2 seconds, with a maximum of 49.38 seconds and a minimum of 21.2 seconds.

Importantly, the tool maintained excellent accuracy throughout the experiment. There were no instances of false positives or false negatives, resulting in a consistent accuracy rate of 100% across all phases.

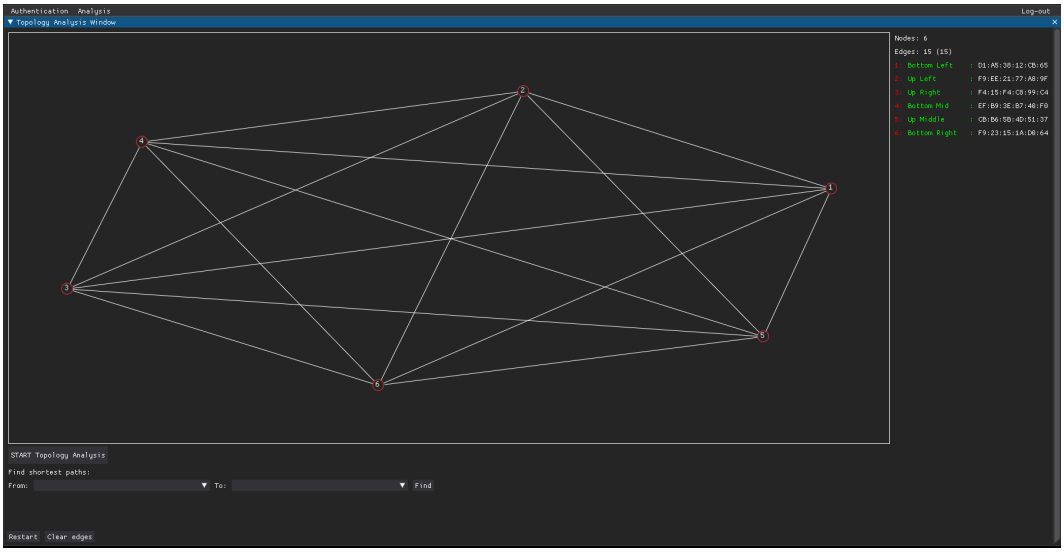


Figure 6.1: Mesh Topology Discovered by the Tool: Cluster 1 Isolated

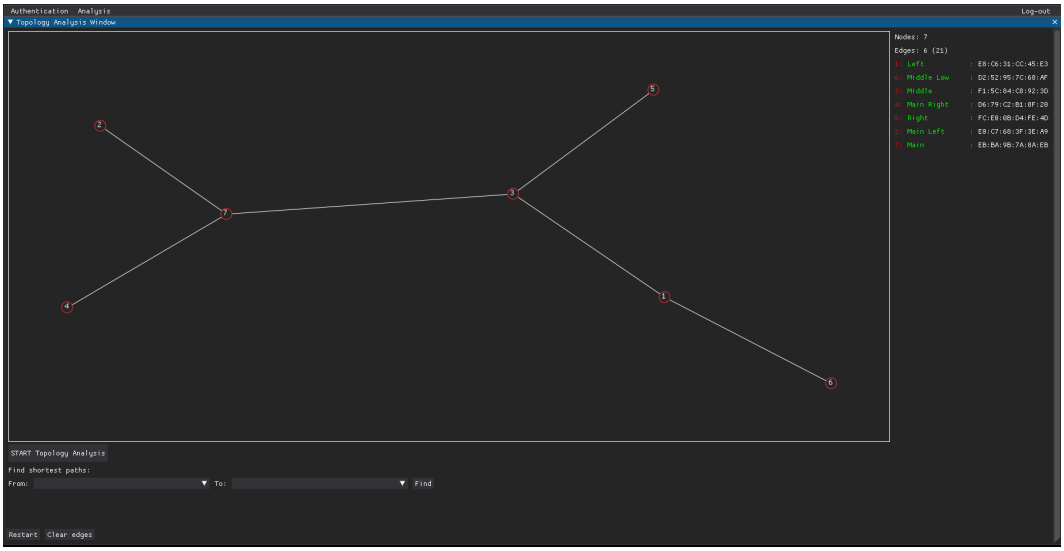


Figure 6.2: Mesh Topology Discovered by the Tool: Cluster 2 Isolated

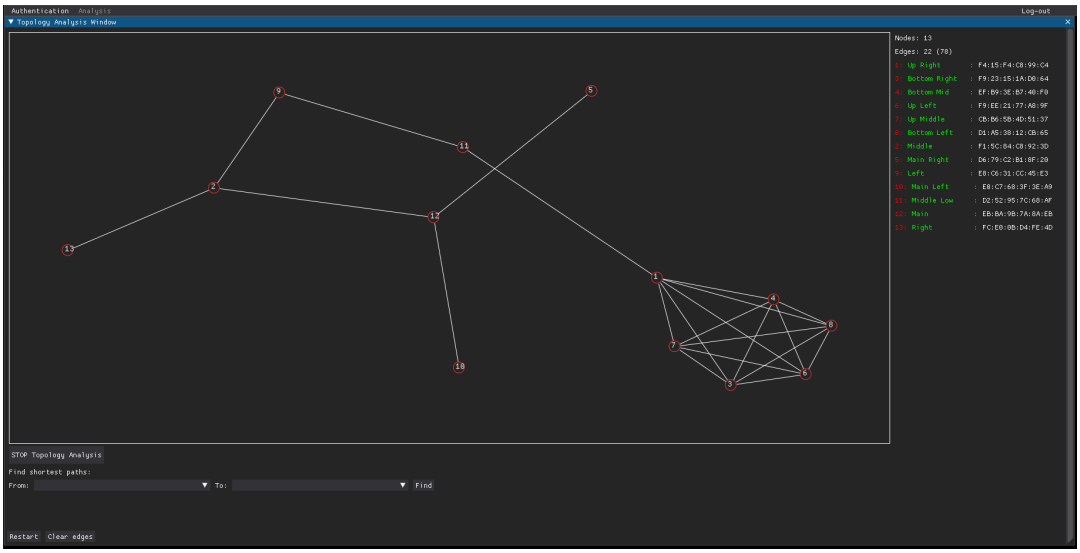


Figure 6.3: Mesh Topology Discovered by the Tool: Both Clusters Active

## 6.2 Message Delivery Ratio (MDR) Analysis

The experimental design incorporated three distinct mesh topologies to assess the tools capability in different network structures. These topologies, labeled as Topology A, Topology B, and Topology C, increased in complexity from one relay node to five relay nodes respectively. The evaluation was conducted across a series of message throughput rates, ranging from 1 Msg/s to 25 Msg/s. In evaluating the differences between the recorded and actual MDRs from the experimental setups across the three mesh topologies, several key observations can be noted. These discrepancies are particularly pronounced as network complexities increase and at higher throughput rates. Discovered network topologies and MDR results are shown in Figures 6.4 to 6.9.

For topology A in Figures 6.4 and 6.5 it can be observed that at lower throughputs (1 Msg/s to 4 Msg/s), the recorded MDR matches the actual MDR perfectly at 100%. As throughput increases, the discrepancies between the recorded and actual MDRs become more pronounced. For instance, at 25 Msg/s, the tool recorded an MDR of 61.9%, whereas the actual MDR was slightly higher at 65.71%.

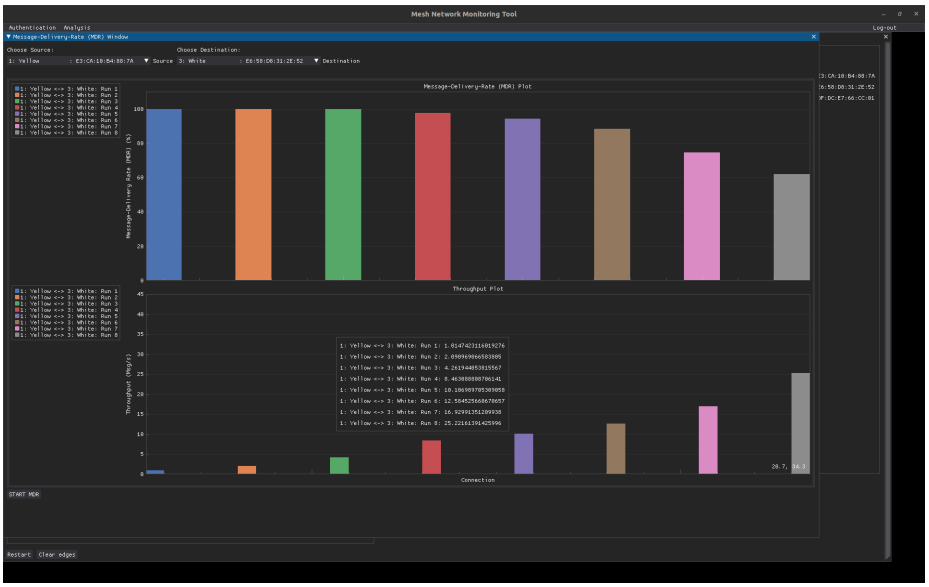


Figure 6.4: Recorded MDR Analysis results for Topology A displayed in the GUI.

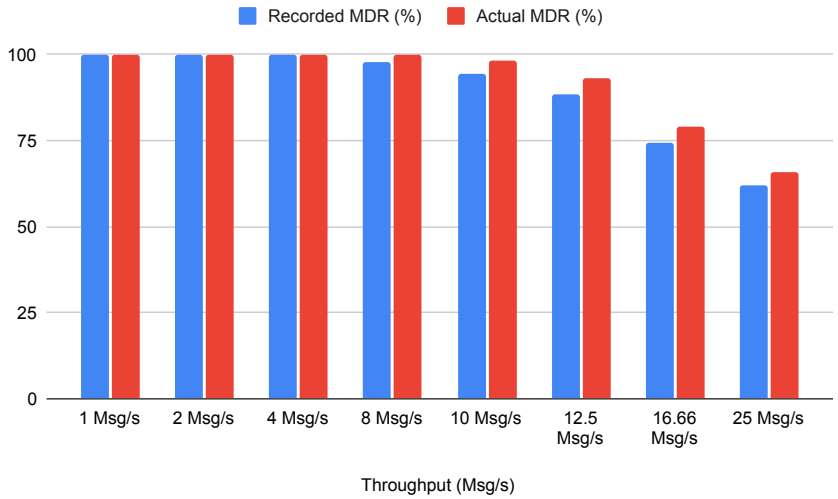


Figure 6.5: Recorded and Actual MDR results for Topology A.

For topology B in Figures 6.6 and 6.7, the tool records 100% MDR effectively up to 8 Msg/s. The first discrepancy occurs at 12.5 Msg/s, where the recorded MDR of 99.23% is slightly less than the actual of 100%. At higher throughputs, especially 25 Msg/s, the tool records 73.33% MDR compared to the actual 84,76%.

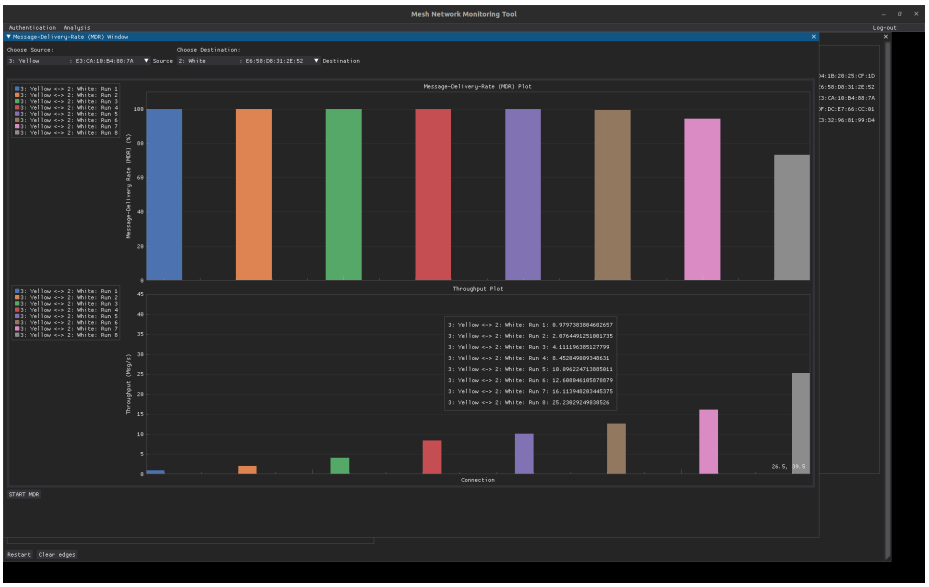


Figure 6.6: Recorded MDR Analysis results for Topology B displayed in the GUI.

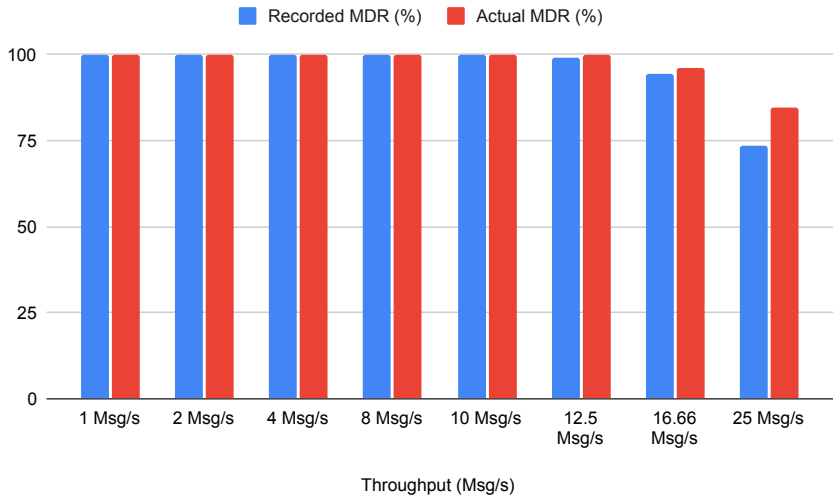


Figure 6.7: Recorded and Actual MDR results for Topology B.

For topology C in Figures 6.8 and 6.9, at 1 Msg/s and 2 Msg/s, there is no discrepancy between the recorded and actual MDRs. Both are perfect at 100%. Starting from 4 Msg/s, the tool begins to underestimate the MDR more significantly, with the widest gap at 25 Msg/s (recorded MDR at 79.04% versus actual MDR at 92.38%).

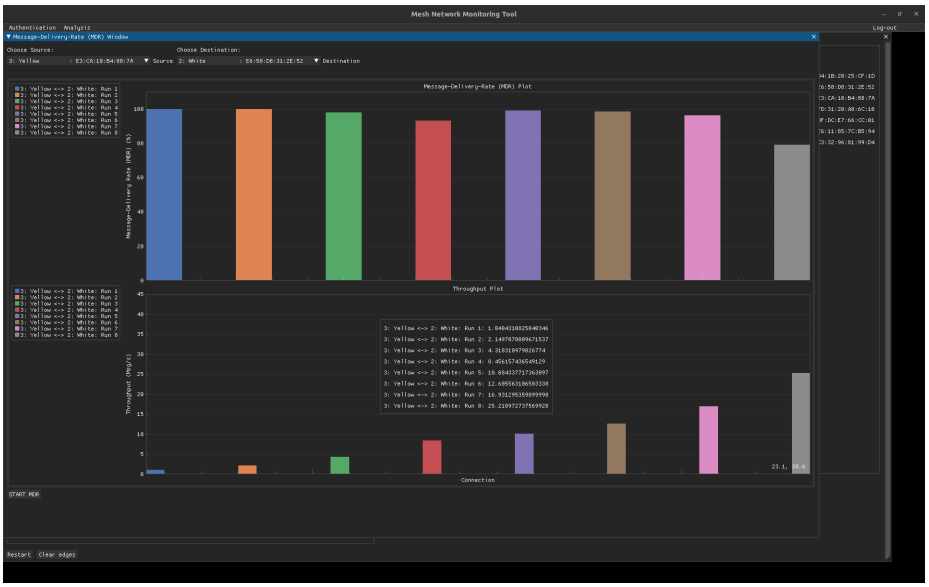


Figure 6.8: Recorded MDR Analysis results for Topology C displayed in the GUI.

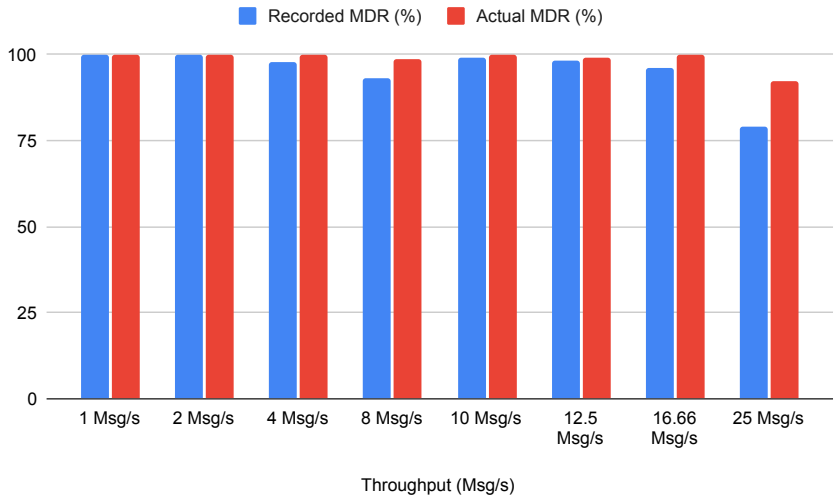


Figure 6.9: Recorded and Actual MDR results for Topology C.

## 6.3 Latency Analysis

This chapter presents the results of the latency analysis experiments. The experiments were designed to evaluate the latency characteristics of message forwarding in a BLE mesh network under various conditions. Specifically, these experiments examined how latency is affected by:

- **Number of Hops:** The number of intermediate nodes a message traverses between source and destination.
- **Number of Neighbors:** The number of relay nodes within communication range of a given node.

The following will provide detailed findings on the effect of hops, neighbors, and show combined results of these factors on the network's latency performance.

Figure 6.10 shows results of the recorded latency between source node and destination node with one hop in between. Using equation 5.6 the theoretical average for this setup is calculated to be 28.42 ms. The measured average forwarding time is relatively consistent across each test run, averaging 27.89 ms, with slight variations between 27.46 ms and 28.06 ms. The maximum forwarding times show more fluctuation, ranging from 65.69 ms to 104.60 ms, with an average maximum forwarding time of 76.96 ms.

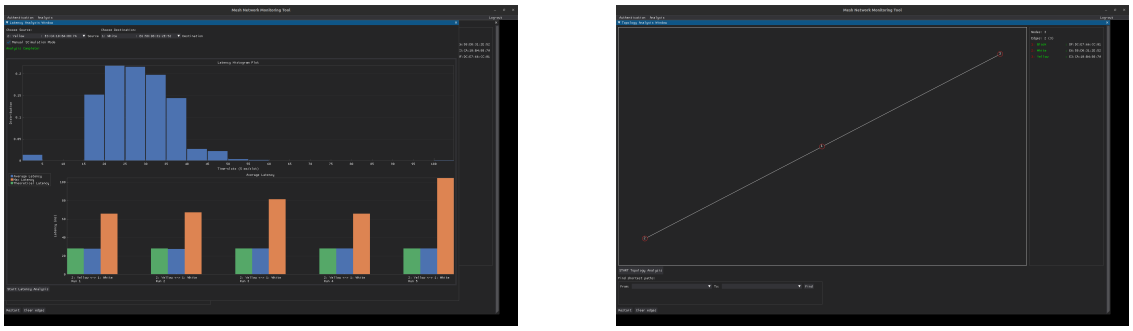

Figure 6.10: Results of recorded latency between source node and destination node with one hop in between.

Figure 6.11 shows results of recorded latency between the source and destination nodes with two hops in between. Using equation 5.6, the theoretical average forwarding time for this setup is calculated to be 56.48 ms. The measured average forwarding time remains relatively close to the theoretical value, averaging 54.16 ms. Individual measurements range from 53.05 ms to 54.71 ms. Maximum forwarding times in the two-hop setup display variations between 118.40 ms and 139.56 ms, with an average maximum forwarding time of 132.72 ms.

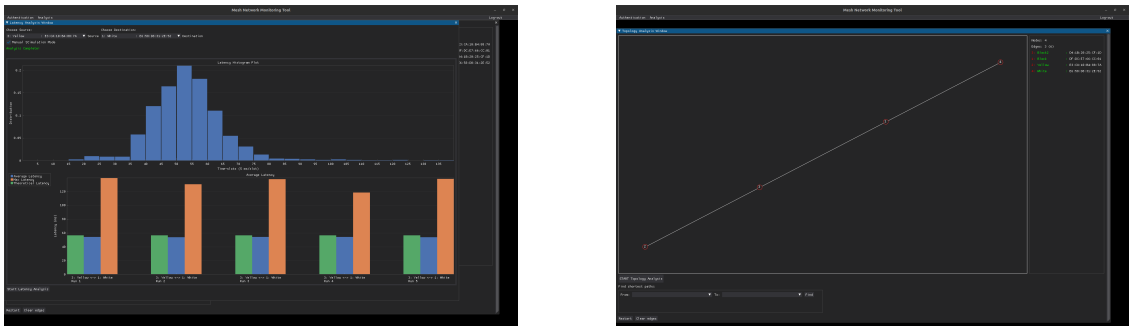

Figure 6.11: Results of recorded latency between source node and destination node with two hops in between.

Figure 6.12 shows results of recorded latency between the source and destination nodes with three hops in between. Using equation 5.6, the theoretical average forwarding time for this setup is calculated to be 84.73 ms. The measured average forwarding time is close to the theoretical value, averaging 82.01 ms with individual measurements ranging from 80.70 ms to 84.03 ms. Maximum forwarding times in the three-hop setup show variability between 160 ms and 189.12 ms, with an average maximum forwarding time of 172.64 ms.

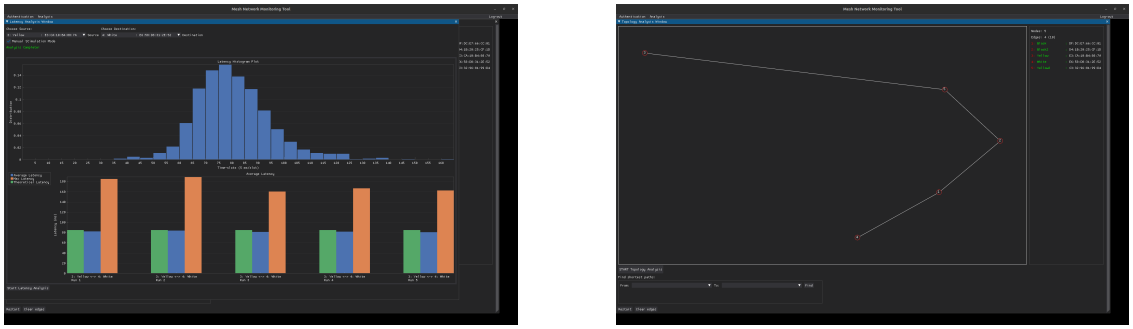

Figure 6.12: Results of recorded latency between source node and destination node with three hops in between.

Figure 6.13 shows results of recorded latency between the source and destination nodes with four hops in between. Using equation 5.6, the theoretical average forwarding time for this setup is calculated to be 112.97 ms. The measured average forwarding time remains consistent with the theoretical value, averaging 111.04 ms. Individual measurements range from 110.03 ms to 111.96 ms. Maximum forwarding times in the four-hop setup display variations between 207.07 ms and 265.16 ms, with an average maximum forwarding time of 233.83 ms.

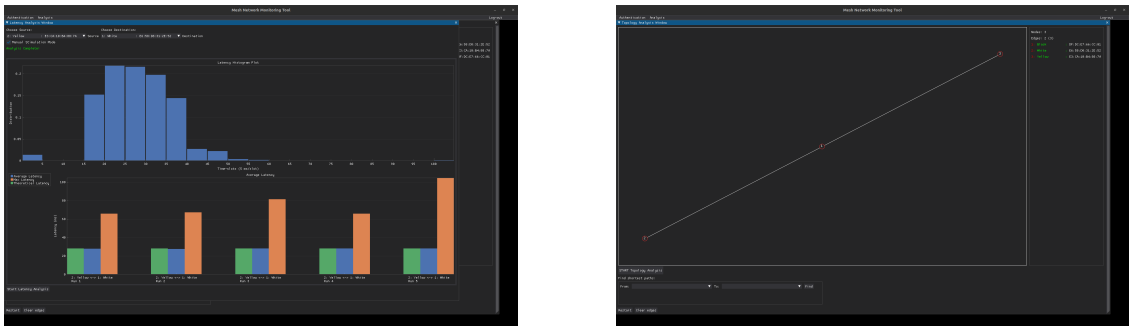

Figure 6.13: Results of recorded latency between source node and destination node with four hops in between.

Figure 6.14 shows results of recorded latency between the source and destination nodes with one hop and three relaying neighbors. Using equation 5.6, the theoretical average forwarding time for this setup is calculated to be 23.94 ms. The measured average forwarding time is slightly higher at 25.51 ms, with individual measurements ranging from 25.39 ms to 25.68 ms. Maximum forwarding times with three neighbors show variation between 67.33 ms and 101.36 ms, with an average maximum forwarding time of 90.36 ms.

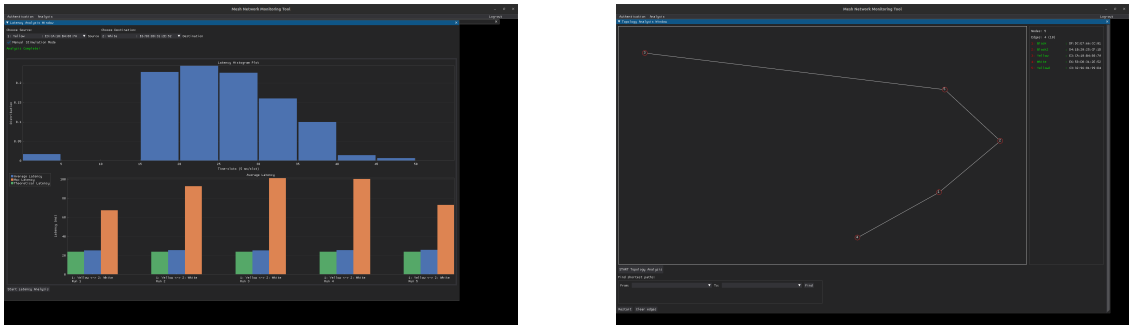

Figure 6.14: Results of recorded latency between source node and destination node with one hop and three neighbours

Figure 6.15 shows results of recorded latency between the source and destination nodes with one hop and five potential relaying neighbors. Using equation 5.6, the theoretical average forwarding time for this setup is calculated to be 22.50 ms. The measured average forwarding time is again slightly higher at 23.63 ms, with individual measurements ranging from 23.48 ms and 23.73 ms. Maximum forwarding times in the five-neighbor setup fluctuations between 65.15 ms and 104.71 ms, with an average maximum forwarding time of 84.25 ms

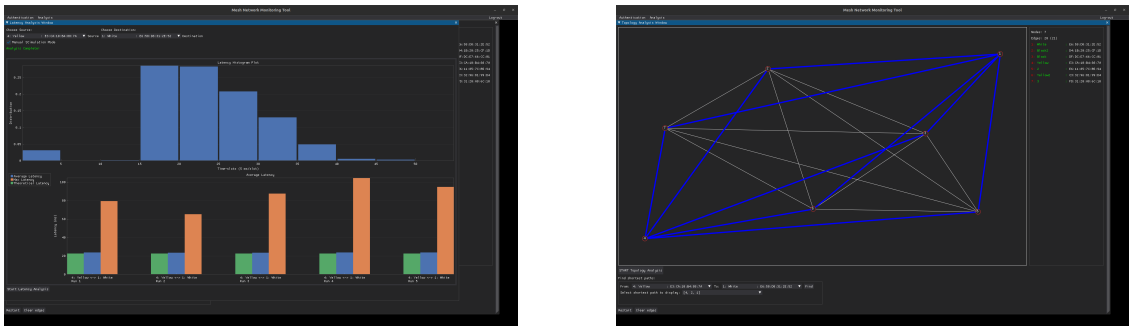

Figure 6.15: Results of recorded latency between source node and destination node with one hop and five neighbours

Figure 6.16 shows results of recorded latency between the source and destination nodes with one hop and eight potential relaying neighbors. The theoretical average forwarding time, using equation 5.6, is calculated to be 21.54 ms. The measured average forwarding time is very close at 21.69 ms, with individual measurements ranging from 21.51 ms to 21.90 ms. Maximum forwarding times with eight neighbors show fluctuation between 58.56 ms and 92.50 ms, with an average maximum forwarding time of 79.97 ms.

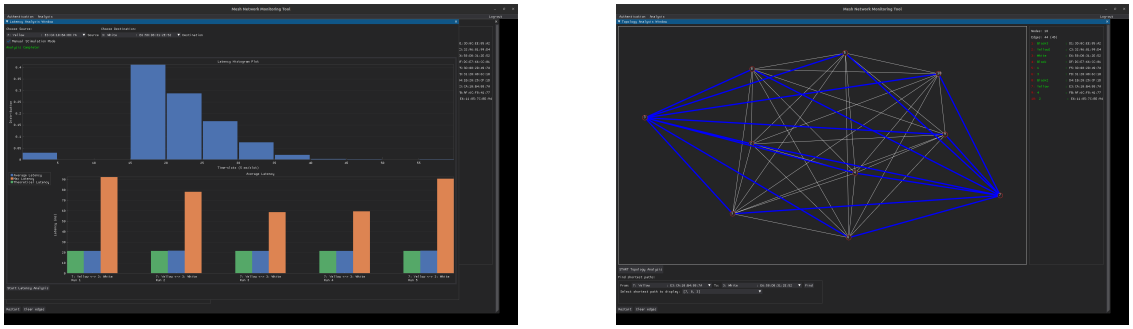

Figure 6.16: Results of recorded latency between source node and destination node with one hop and eight neighbours

Figure 6.17 shows results of recorded latency between the source and destination nodes with one hop and ten potential relaying neighbors. Using equation 5.6, the theoretical average forwarding time is calculated to be 21.2 ms. The measured average forwarding time is slightly lower at 20.36 ms, with individual measurements ranging from 20.10 ms to 20.60 ms. Maximum forwarding times in the ten-neighbor setup display variation, between 58.20 ms and 106.35 ms, with an average maximum forwarding time of 83.96 ms.

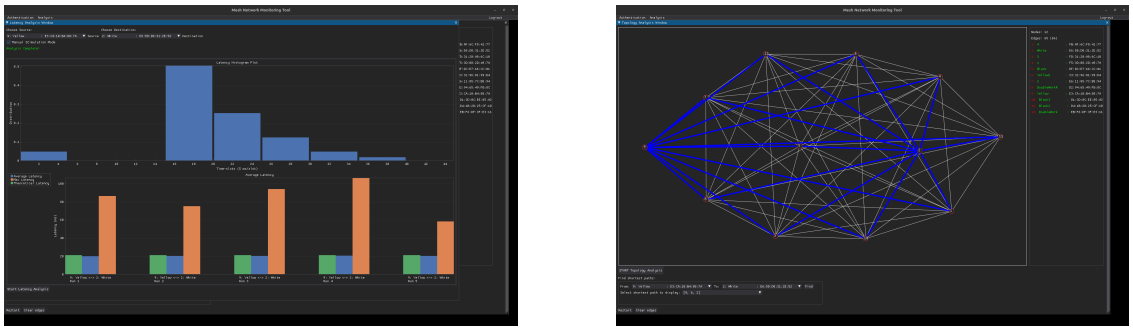

Figure 6.17: Results of recorded latency between source node and destination node with one hop and ten neighbours

Combined results of the experiment are presented in figures 6.18 and 6.19. The theoretical average forwarding time increases linearly as the number of hops between source and destination grows. The measured average forwarding time closely follows this linear trend. Notably, the measured maximum forwarding time shows the most significant increase and variability as the number of hops increases.

Increasing the number of potential relaying neighbors has a slight decreasing effect on the theoretical average forwarding time. The measured average forwarding time generally follows this downward trend, but displays some fluctuations, particularly between setups with 3 and 5 neighbors. The measured maximum forwarding time also shows variability, but overall suggests a slight decrease as the number of neighbors increases.

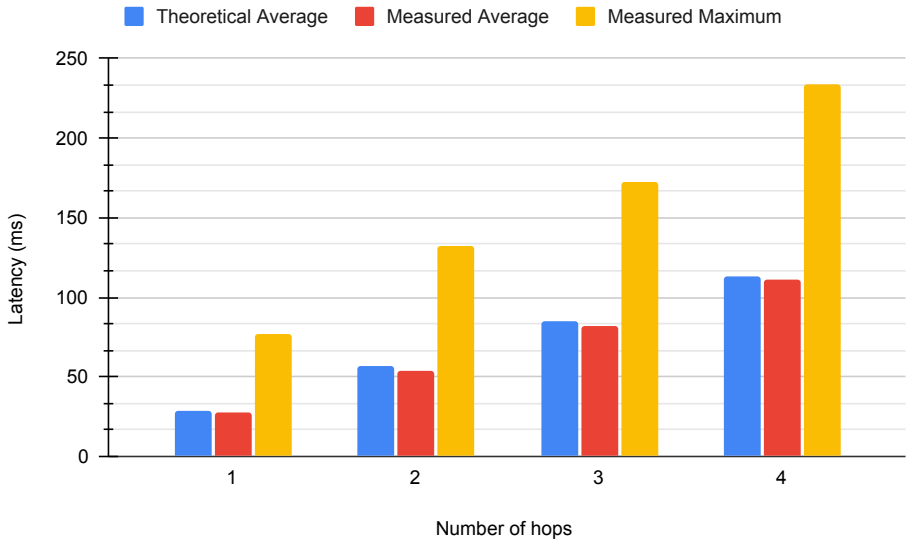


Figure 6.18: Multi-hop communication measurements with a varying amount of hops.

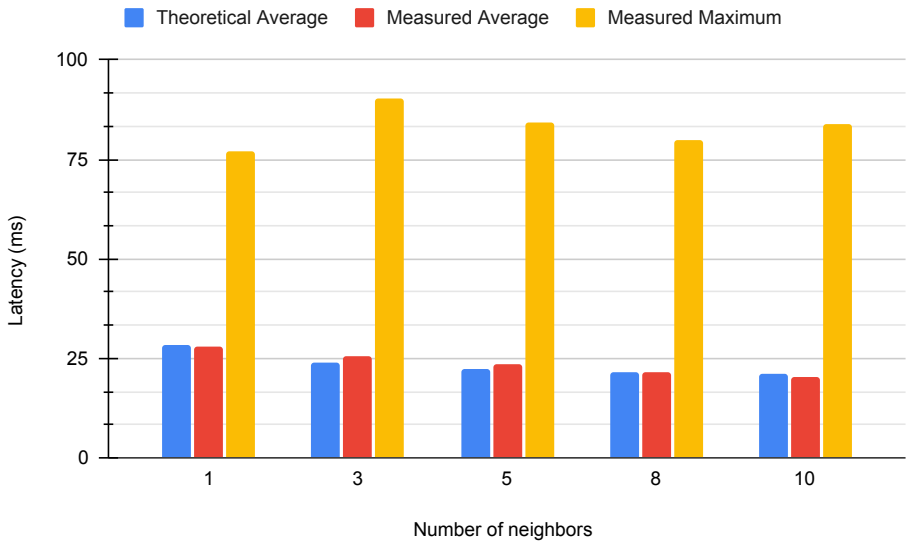


Figure 6.19: One hop communication measurements with a varying amount of neighbors.

# 7

# Discussion

This chapter consolidates and interprets findings from the mesh topology, MDR, throughput and latency analyses conducted in this research. Each analysis aimed to evaluate the performance and accuracy of a custom-developed wireless monitoring tool. Results highlight the tool's effectiveness under varying conditions of network topology, message throughput, and latency, while also revealing its limitations in scenarios of high network congestion and complexity.

## 7.1 Mesh Topology Analysis

The primary objective of the mesh topology analysis was to identify the topology within the mesh network environment, using a dedicated wireless monitoring tool developed as part of this research, alongside the designed GUI.

Understanding network topology is important as it influences performance metrics such as latency, MDR and overall network robustness. The topology analysis combined with analysis of performance metrics helps in spotting potential weak points in the network, such as nodes with suboptimal connectivity that might become bottlenecks or single points of failure. This analytical insight is important for both theoretical exploration and practical application, enabling optimized design and improved troubleshooting of BLE mesh networks.

The system design and methodology for analyzing the network topology was detailed in the Sections 4.2.4 and 4.3.1 of this thesis. The experimental setup was described in Section 5.1. This analysis was conducted using a combination of hardware and software tools specifically developed for this project. The nRF5340 DK served as the cornerstone of the hardware setup, tasked with capturing mesh network traffic and sending it to the PC. The custom-built software on this kit was designed to interact with the nRF OpenMesh, facilitating data collection without altering the existing network structure.

Software tools played an important role in analyzing the collected data. The system used a combination of Python scripts and libraries such as Pandas for data manipulation, NetworkX to visualize the network topology, and custom algorithms that

leverage the Trickle algorithm to infer communication links between nodes. The topology analysis was performed through a GUI interface that allowed for real-time interaction and visualization of the network structure.

Data for topology analysis was primarily gathered through active scans where the nRF5340 DK transmitted stimulation messages and received responses across the network. The packets were processed to identify unique node identifiers and to map the communication links based on packet transmission and reception patterns. The nodes and interconnections between them were then visualized within the GUI, providing a graphical representation of the network's topology.

The experimental setup was methodically structured to validate the monitoring tools capabilities in a controlled environment before applying it to more complex, real-world scenarios. This included configuring specific network topologies by altering node firmware and observing the tools ability to accurately map and report the network structure. This structured approach allowed for a detailed examination of the tool's effectiveness in real-time network monitoring and topology analysis.

### 7.1.1 Analysis of Findings

The results of the topology analysis are presented in Table 6.1. Cluster 1, which was fully connected, showed relatively quick discovery times, averaging 11.04 seconds. This suggests a robust inter-node communication where each node can efficiently communicate with its neighbors and time their response messages in a way that avoid collisions.

For Cluster 2, characterized by its hierarchical tree structure, the average discovery time was slightly longer at 13.89 seconds. This increase was influenced by the cluster design where each message lived longer time in the network as it traveled through multiple hops, increasing chances of message collision. Another factor was the node's selective message processing and the operational dynamics of the Trickle algorithm. Nodes programmed to ignore messages from non-targeted MAC addresses could potentially miss crucial communications, impacting the effectiveness of the Trickle algorithm in triggering necessary rebroadcasts.

When clusters were discovered sequentially, the average discovery times increased significantly, suggesting an interaction between network load and discovery efficiency. Discovering Cluster 2 after Cluster 1 resulted in an average discovery time of 22.35 seconds, whereas discovering Cluster 1 after Cluster 2 took longer, averaging 26.95 seconds. These extended times could be attributed to the residual network activity and congestion from the initially discovered cluster, which might have compounded the selective message processing delays in the subsequent cluster.

Simultaneous discovery of both clusters led to the longest average discovery time of 31.2 seconds. This result highlights the challenges posed by network congestion and increased complexity when the system attempts to map the topology of a signifi-

cantly larger and combined set of nodes. The presence of simultaneous transmissions likely worsened the tools and nodes' limitations in managing and processing all incoming messages effectively.

### 7.1.2 Interpretation of Results

The observed discovery times and network behavior align with different theoretical models predicting faster connectivity in fully connected networks versus hierarchical ones. However, these results must be contextualized within the limitations of the monitoring tool. The tool's inability to consistently receive all packets due to collisions, the need to manage its reception queue, or its non-reception mode while processing data potentially skews the observed discovery times, particularly under high network congestion. Additionally, the tool's role in sending TX messages to stimulate nodes further reduces its capacity to receive incoming packets.

These findings highlight the importance of robust network design that anticipates practical challenges such as packet loss and tool reception limitations. Design strategies that minimize potential collision points and optimize packet flow can significantly enhance network reliability and performance. The necessity for advanced monitoring capabilities that can adapt to high congestion and effectively manage data throughput without losing critical information is also underscored, illustrating a key area for future development in BLE mesh network tools.

### 7.1.3 Challenges Encountered

The monitoring tool faced reception challenges, as previously discussed, which were compounded by the experimental modifications made to the node firmware. Nodes were programmed to process messages only from certain MAC addresses, though they continued to receive all transmitted messages. This selective processing meant that nodes could accidentally miss important messages, adding another layer of complexity to the network's functional dynamics and potentially obscuring true network performance.

This selective message processing by nodes may have led to critical communications being overlooked, particularly in scenarios of high network activity where many nodes were transmitting simultaneously. As a result, nodes might not only miss responses intended for them but also necessary rebroadcasts that could have impacted the tool's ability to accurately map the network topology. This, combined with the tool's own limitations in handling received packets, likely contributed to the increased and variable discovery times observed during the experiments.

The experimental setup's increased complexity, with nodes selectively processing communications, likely worsened network congestion issues. This setup increased the likelihood of packet collisions and loss, as nodes still received but did not process messages from non-targeted MAC addresses, occupying airtime and potentially causing further disruptions in network communications.

These adjustments underscore the nuanced challenges that arise from experimental modifications and highlight the need for careful planning and consideration when designing network experiments. They also suggest areas for further research and development, particularly in optimizing node firmware to better manage selective message processing without compromising the overall network integrity and performance.

Due to the fact that the monitoring tool itself could miss original response messages from the nodes, thereby inferring an incorrect connection, a redundancy constant was introduced. This constant determined how many times a complete communication sequence must be received from two nodes to infer a connection between them. This measure was crucial in ensuring the reliability of the inferred network topology, especially in complex network environments where the probability of message loss or misinterpretation is high.

## 7.2 Message Delivery Ratio (MDR) Analysis

The primary objective of the MDR analysis was to assess how accurately a monitoring tool captures the correct MDR. MDR, the percentage of messages successfully reaching their destinations, is an important metric for evaluating network reliability and efficiency. Three custom-designed mesh networks (Figures 5.2 and 5.3) were used to study how the monitoring tool's MDR calculations align with the actual message delivery rates under varying network conditions. Specifically, the analysis investigated the impact of network topology (node arrangement and connectivity) and message throughput (varied message rates) on the tool's MDR measurement accuracy.

The system design and methodology for analyzing MDR was detailed in the Sections 4.2.5 and 4.3.2 of this thesis. The experimental setup was described in Section 5.2. MDR analysis builds upon the foundation laid in the previous section, utilizing the same hardware and software tools but with a specific focus on evaluating message delivery rates across different network configurations and loads. This analysis involved a series of controlled experiments using the three custom-designed mesh networks. Here's a summary of the experimental approach:

- **Network Configurations:** The experiments varied the number of relay nodes and their placement to understand how topology impacts the monitoring tool's MDR calculations.

- **Message Throughput:** Eight different message rates were tested to assess how the tool's MDR measurements hold up under varying network loads.

- **Data Collection and Analysis:** Messages were actively transmitted at specified rates. Delivery success rates were recorded both by the monitoring tool and through direct logging mechanisms to calculate the actual MDR. This allowed for a direct comparison to evaluate the tool's accuracy.

- **Real-Time Interaction and Visualization**: A custom GUI interface provided several key features for analyzing MDR in real-time. Users could select the specific source and destination nodes to focus on the MDR of their logical connection. Two graphs displayed the tool's recorded MDR and network throughput with a slight delay, while also saving results from previous experimental runs and plotting them alongside new data for easy visual comparison.

### 7.2.1 Analysis of Findings

For Topology A, observed MDR remained perfect (100%) at lower throughputs from 1 Msg/s to 4 Msg/s, as seen in Figure 5.2. However, as the throughput increased, discrepancies became apparent; notably at 25 Msg/s, the recorded MDR fell to 61.9%, while the actual MDR was higher at 65.71%. This indicates that the monitoring tool struggles with higher traffic volumes in simpler network setups, but is still accurate.

In Topology B, as detailed in Figure 5.2, the tool maintained a 100% MDR up to a higher threshold of 8 Msg/s. The first discrepancy appeared at 12.5 Msg/s with a minimal deviation (recorded MDR 99.23% versus actual 100%). At the highest tested rate of 25 Msg/s, a larger gap was observed with recorded MDR at 73.33% compared to an actual MDR of 84.76%. This shows a better performance in moderately complex setups but still highlights challenges at peak loads.

Topology C, shown in Figure 5.3, demonstrated perfect MDR at very low rates (1 Msg/s and 2 Msg/s). Discrepancies began to emerge from 4 Msg/s onward, with the most significant gap noted at 25 Msg/s where the recorded MDR was 79.04% against an actual MDR of 92.38%. This suggests that even in complex topologies with more relay nodes, the tool's performance degrades under high load, underestimating the actual delivery success.

The trend across all topologies indicates that as the network complexity and message throughput increase, the discrepancies between recorded and actual MDRs become more pronounced. This is particularly noticeable at the highest message rate of 25 Msg/s in each topology, where the monitoring tool consistently records lower MDRs than actual, pointing to potential issues in handling congestion and packet collisions effectively.

### 7.2.2 Interpretations of Results

The results from the MDR analysis resonate with those from the network topology analysis, confirming the impact of network structural complexity and load on performance. As seen in both analyses, the monitoring tool's limitations became apparent under higher network loads and more complex configurations.

The observed discrepancies in MDR, especially at higher message rates, are partly due to the limitations of the monitoring tool as well as the nodes themselves. The nodes' inability to capture all messages, even though the actual MDR is better than

recorded by the tool, points to limitations in the nodes' message handling capabilities. This situation suggests that operational thresholds need careful consideration. Specifically, a message rate interval should not be shorter than 60ms (approximately 16.66 Msg/s) to maintain reasonable network performance. However, a more optimal message interval from both the monitoring tool and nodes' perspective would be 80ms (12.5 Msg/s), balancing the need for high throughput with system reliability.

The findings underscore the importance of strategic network planning that anticipates potential issues such as packet loss and throughput limitations. Understanding the specific points at which network performance begins to degrade allows network engineers to design systems that optimize packet flow and minimize collision points, thereby enhancing overall network reliability and efficiency. Such strategic planning is crucial in networks where maintaining a high MDR is essential for the application's success.

### 7.2.3 Challenges Encountered

One of the primary challenges encountered during this analysis was network congestion, which became increasingly problematic as the message rate and number of nodes in the network increased. Network congestion not only affects the latency and throughput of the network but also complicates the process of accurately monitoring and analyzing MDR due to the increased likelihood of packet loss and collisions.

The MDR data analysis methodology employed in this study includes the use of both direct and indirect acknowledgments to infer message delivery. Indirect acknowledgments are particularly important in scenarios where direct message rebroadcast by the destination node might not be detectable due to network congestion or the monitoring tool's operational limitations, such as not being in receive (RX) mode.

The Trickle algorithm, as used in this thesis, is slightly modified compared to original specification. Traditionally, the Trickle algorithm allows nodes to refrain from rebroadcasting a message if it has been heard several times within the same period. In this modified version, nodes permanently drop a message if they detect it at least four times in the same period, which adds to the complexity of accurately determining whether a message was received by the destination node.

The monitoring tool's ability to classify messages as acknowledged or unacknowledged heavily relies on detecting rebroadcasts from the neighbors of the destination node. If many nodes are active, and especially if the destination node has rebroadcasted a message, there is a chance that the tool does not receive this due to either network congestion or its non-reception mode. Consequently, the tool may incorrectly classify a successfully received message as unacknowledged if it fails to detect enough rebroadcasts from neighboring nodes, leading to potential inaccuracies in the reported MDR.

## 7.3 Latency Analysis

This study's latency analysis aimed to assess the accuracy of a monitoring tool in measuring delays within BLE mesh networks. The goal was to validate its reliability in providing latency measurements and displaying them in the GUI, essential for optimizing BLE mesh network deployments and performance. The system design and methodology for analyzing the network topology of BLE mesh networks was detailed in the Sections 4.2.6 and 4.3.3 of this thesis. The experimental setup was described in Section 5.3

A systematic approach was used, similar to work in [4], to test the tool's measurements against known network configurations and controlled message transmissions. The hardware and software setup from the Mesh Topology and MDR Analysis sections was reused, with the nRF5340 Development Kit playing a key role in capturing network data. Python scripts and libraries like Pandas facilitated data manipulation and analysis.

Latency was measured from message transmission at the source to its rebroadcast by a neighbor of the destination node. This indirect approach to confirming message receipt was necessary due to constraints in directly observing the destination node's behavior without disrupting the network's normal traffic.

Data was gathered through active network scanning, with messages sent at intervals and neighbor rebroadcasts monitored. The tool logged time stamps of relevant events, used later for latency calculation. Validation tests compared these measurements with theoretically expected values under controlled conditions, allowing for discrepancy identification and correction.

### 7.3.1 Analysis of Findings

The results clearly indicate that latency increases with the number of hops between the source and destination nodes. For instance:

- In a one-hop setup, the measured average latency was 27.896 ms, closely aligning with the theoretical average of 28.42 ms.
- With two hops, the average measured latency was 54.156 ms versus the theoretical average of 56.48 ms.
- A three-hop configuration recorded an average latency of 82.008 ms, near the theoretical prediction of 84.73 ms.
- The most extended four-hop scenario showed a measured average latency of 111.042 ms, closely matching the theoretical 112.97 ms.

These measurements validate the theoretical model's accuracy in predicting that

each additional hop contributes significantly to overall latency, with a near-linear increase observed in the experimental data.

The analysis also explored how increasing the number of neighboring nodes affects latency, revealing a general trend of reduced latency with more neighbors, due to more opportunities for faster message rebroadcast:

- With one hop and three neighbors, the measured average latency was slightly higher at 25.5125 ms compared to the theoretical 23.94 ms.
- Increasing to five neighbors, the measured average latency slightly increased to 23.635 ms against a theoretical 22.5 ms.
- With eight neighbors, the latency further improved to a measured 21.69 ms, very close to the theoretical 21.54 ms.
- Ten neighbors yielded the best performance with a measured average latency of 20.358 ms, outperforming the theoretical expectation of 21.2 ms.

### 7.3.2 Interpretation of Results

The analysis of multi-hop topologies revealed a consistent trend in the message time distribution across different setups, with each increment in hop count resulting in a corresponding shift towards higher latency times. This shift is a clear indicator of the additive latency introduced by each additional hop, validating the theoretical model that predicts increased delay due to processing and transmission times at each intermediate node.

In contrast to the multi-hop findings, the multi-neighbor setups demonstrated a progressive shift in message distribution towards the fastest possible rebroadcast time of 16ms. This shift suggests that increasing the number of neighbors at each node can significantly enhance the probability of earlier message rebroadcast, effectively reducing overall network latency. The data showed fewer messages reaching the maximum rebroadcast interval of 32ms or later, highlighting the benefit of dense neighbor configurations in facilitating quicker data dissemination.

The monitoring tool demonstrated robust performance in both multi-hop and multi-neighbor scenarios, providing empirical data that effectively mirrored the theoretical models. However, the greater accuracy and reliability in environments with dense neighbor setups suggest that the tool's performance is optimized in scenarios where network traffic is more evenly distributed among multiple nodes. This could be due to the increased chances of capturing rebroadcasts from any one of several neighbors, which reduces the likelihood of missing critical rebroadcast events that confirm message receipt.

The findings from this analysis not only underscore the tools capability to accurately

measure and report latency in BLE mesh networks but also highlight potential areas for its application. For instance, in designing networks for critical real-time applications, such as in healthcare monitoring or industrial automation, the tool can provide valuable insights into how network topology impacts latency. Moreover, the demonstrated accuracy in dense networks suggests that the tool could be particularly useful in optimizing network designs in urban or complex industrial settings where node density is naturally high.

### 7.3.3 Challenges Encountered

Throughout the latency analysis, several challenges emerged, mirroring issues encountered in previous network topology and MDR analyses. These challenges predominantly stemmed from the inherent complexities of BLE mesh networks and the limitations of the monitoring tool.

One significant challenge was network congestion, which became increasingly problematic as the number of nodes increased in multi-neighbour setup. In densely populated networks collisions occurred more frequently. These collisions sometimes caused the monitoring tool to miss the original message sent by the source node. Instead, a rebroadcast by a source node could be misidentified as the original message. This misidentification led to the recording of anomalously low latency measurements, occasionally as low as 0-5ms, as observed in the latency distribution from the results section. These challanges are also the reason why the measurements do not follow exactly the same trend as in [4].

# 8

# Conclusion

This thesis presents the development and evaluation of a wireless BLE mesh network monitoring tool, leveraging the nRF5340 DK, with a focus on performance metrics such as MDR, latency, and network topology visualization. The study aims to address the challenges posed by existing wired monitoring solutions, offering a non-intrusive and user-friendly alternative for real-time network analysis.

The primary objective was to create a dedicated hardware tool capable of wirelessly capturing data from BLE mesh networks and transmitting it to a PC for analysis. This was achieved using the nRF5340 DK, coupled with Python-based software for data processing and a GUI for visualization. The tool demonstrated robust performance in various experimental setups, effectively capturing and analyzing network metrics.

The tool's ability to accurately measure MDR was validated through controlled experiments with custom-designed mesh networks. Results indicated that while the tool performed well at lower message rates, discrepancies between recorded and actual MDRs became more pronounced at higher rates and in more complex topologies. These findings suggest potential limitations in handling network congestion and packet collisions.

Latency measurements across different network configurations revealed a near-linear increase in latency with each additional hop, consistent with theoretical predictions. The analysis also showed that increasing the number of neighboring nodes generally reduced latency, highlighting the benefits of dense network setups for quicker message dissemination. However, challenges such as network congestion and misidentification of messages were noted.

The tool effectively identified network topologies and direct communication links between nodes. This capability is important for understanding network behavior and optimizing design, especially in real-world applications where network reliability is crucial.

The tool can be used for real-time network analysis. Its ability to wirelessly capture data without requiring physical modifications to network nodes makes it particularly

suitable for monitoring and optimizing networks in practical settings, such as smart homes. This non-intrusive approach allows for easier deployment and management of BLE mesh networks. It can be used for monitoring, and from the results, network performance can be assessed, compared, improved, and optimized.

Compared to other tools, this monitoring tool stands out because it is wireless, eliminating the need for cumbersome wired connections. This user-friendly alternative simplifies network setup and monitoring, beneficial for both academic research and industrial applications. The tool's GUI provides near real-time visualization of network metrics, making it easier to identify and address issues promptly. It maintains good accuracy even in dense network setups. By capturing multiple performance metrics, the tool offers a holistic view of network health, aiding in informed decision-making for network optimization.

Future research could expand the tool's capabilities by enabling GATT communication between the tool and network nodes, analyzing data across multiple channels, and adapting the tool for use with multiple measurement points. These advancements would further enhance the tool's applicability and accuracy in various network scenarios.

Social, ethical, and ecological considerations are important for implementing BLE mesh networks. The social impacts of this tool include the potential to improve public services, healthcare, and daily life by making networks more reliable and efficient. It is important to make sure that these technological benefits are available to everyone, regardless of their social and economic status.

Ethically, the tool must focus on protecting data privacy and security. It is important to design the tool with strong privacy features and robust encryption to keep sensitive information safe, especially in homes and healthcare settings. Being transparent about the tool's encryption methods, conducting thorough testing, and clearly communicating its limitations will encourage responsible use and build trust among users. Following regulations like GDPR and getting informed consent from users when collecting personal data are key ethical requirements.

Ecologically, the tool aims to improve the energy efficiency of BLE mesh networks, helping to protect the environment. By reducing the energy use of IoT systems, the tool helps lessen the environmental impact of technology, supporting the creation of greener IoT solutions and reducing carbon emissions.

In conclusion, this thesis demonstrates the successful development of a wireless BLE mesh network monitoring tool that addresses the limitations of existing wired solutions. By providing comprehensive and real-time analysis of network performance metrics, the tool offers significant potential for improving the management and reliability of BLE mesh networks in various practical applications. Future enhancements will continue to build on this foundation, aiming to provide even greater accuracy and versatility in network monitoring.

# Bibliography


[1] Nordic Semiconductor, *Nrf51-ble-bcast-mesh*, GitHub repository, 2023. [Online]. Available: https://github.com/NordicPlayground/nRF51-ble-bcast-mesh.

[2] Nordic Semiconductor. "Nrf5340 product specification." Accessed: 2024-03-05. (2024), [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fstruct_nrf53%2Fstruct%2Fnrf5340.html (visited on 03/05/2024).

[3] A. Sikora, J. Sebastian E, A. Yushev, E. Schmitt, and M. Schappacher, "Automated Physical Testbeds for Emulation of Wireless Networks," *MATEC Web of Conferences*, vol. 75, p. 06 006, Jan. 2016. DOI: 10.1051/matecconf/20167506006.

[4] M. Baert, J. Rossey, A. Shahid, and J. Hoebeke, "The bluetooth mesh standard: An overview and experimental evaluation," *Sensors*, vol. 18, no. 8, 2018, ISSN: 1424-8220. DOI: 10.3390/s18082409. [Online]. Available: https://www.mdpi.com/1424-8220/18/8/2409.

[5] Y. Murillo, B. Reynders, A. Chiumento, S. Malik, P. Crombez, and S. Pollin, "Bluetooth now or low energy: Should BLE mesh become a flooding or connection oriented network?" In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, 2017, pp. 1–6. DOI: 10.1109/PIMRC.2017.8292705.

[6] S. Ye, X. Huang, Z. Zhong, *et al.*, "Research on Networking Technology based on BLE Mesh," in *2022 IEEE 5th International Conference on Electronic Information and Communication Technology (ICEICT)*, 2022, pp. 540–542. DOI: 10.1109/ICEICT55736.2022.9908725.

[7] S. M. Darroudi, C. Gomez, and J. Crowcroft, "Bluetooth low energy mesh networks: A standards perspective," *IEEE Communications Magazine*, vol. 58, no. 4, pp. 95–101, 2020. DOI: 10.1109/MCOM.001.1900523.

[8] G. Ferrari, P. Medagliani, S. Piazza, and M. Martalo, "Wireless sensor networks: Performance analysis in indoor scenarios," *EURASIP Journal on Wireless Communications and Networking*, vol. 2007, pp. 41–41, Dec. 2007. DOI: 10.1155/2007/81864.

[9] E. D. Leon and M. Nabi, "An experimental performance evaluation of bluetooth mesh technology for monitoring applications," in *2020 IEEE Wireless Communications and Networking Conference (WCNC)*, 2020, pp. 1–6. DOI: 10.1109/WCNC45663.2020.9120762.

[10] Nordic Semiconductor. "Developing with nRF5340 DK  nRF Connect SDK documentation." Accessed on: Mar. 27, 2024. (2023), [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/device_guides/working_with_nrf/nrf53/nrf5340.html.

[11] Nordic Semiconductor. "Nrf52 softdevices." Accessed: 2024-03-05. (2024), [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fstruct_nrf52%2Fstruct%2Fnrf52_softdevices.html (visited on 03/05/2024).

[12] Nordic Semiconductor. "Multiprotocol service layer timeslot." Accessed: 2024-03-05. (2024), [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrfxlib/mpsl/doc/timeslot.html (visited on 03/05/2024).

[13] Nordic Semiconductor. "Softdevice controller scheduling." Accessed: 2024-03-05. (2024), [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrfxlib/softdevice_controller/doc/scheduling.html (visited on 03/05/2024).

[14] Bluetooth SIG, *The bluetooth low energy primer*, https://www.bluetooth.com/bluetooth-resources/the-bluetooth-low-energy-primer/, Accessed: 15-Mar-2024, Accessed 2024.

[15] P. Levis, T. H. Clausen, O. Gnawali, J. Hui, and J. Ko, *The Trickle Algorithm*, RFC 6206, Mar. 2011. DOI: 10.17487/RFC6206. [Online]. Available: https://www.rfc-editor.org/info/rfc6206.

[16] Dear PyGui. "Dear pygui documentation." Accessed: 2024-03-05. (2024), [Online]. Available: https://dearpygui.readthedocs.io/en/latest/index.html (visited on 03/05/2024).

[17] NetworkX Developers. "Introduction to networkx." Accessed: 2024-03-05. (2024), [Online]. Available: https://networkx.org/documentation/stable/reference/introduction.html (visited on 03/05/2024).

[18] pandas development team. "Pandas documentation." Accessed: 2024-03-05. (2024), [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/ (visited on 03/05/2024).

[19] M. F. Khan, E. A. Felemban, S. Qaisar, and S. Ali, "Performance Analysis on Packet Delivery Ratio and End-to-End Delay of Different Network Topologies in Wireless Sensor Networks (WSNs)," in *2013 IEEE 9th International Conference on Mobile Ad-hoc and Sensor Networks*, 2013, pp. 324–329. DOI: 10.1109/MSN.2013.74.

[20] Tomass Puls, *Mesh Network Monitoring Tool nRF5340 Firmware*, 2024. [Online]. Available: https://github.com/rynze1997/thesis_fw_monitoring_tool.

[21] Tomass Puls, *Mesh Network Monitoring Tool GUI*, 2024. [Online]. Available: https://github.com/rynze1997/mesh-monitoring-tool-gui.