MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Information and Communication Technologies (ICT)

Impact of Wi-Fi 6 Target Wake Time (TWT)

Suported by Nordic Semiconductors ASA
in Stockholm
Under the supervision of Ioannis Glaropoulos

Author:
Sylvestre van Kappel

Under the direction of
Prof. Medard Rieder
HES-SO Valais Wallis

External expert:
Ioannis Glaropoulos

Stockholm, Nordic Semiconductor ASA, February 7, 2024

Information about this report

Contact information

Author: Sylvestre van Kappel
MSE Student
HES-SO//Master
Switzerland
Email: *sylvestrevk@gmail.com*

Declaration of honor

I, undersigned, Sylvestre van Kappel, hereby declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: _____

Signature: _____

Validation

Accepted by the HES-SO//Master (Switzerland, Lausanne) on a proposal from:

Prof. Medard Rieder, Thesis project advisor

Ioannis Glaropoulos, Nordic Semiconductors ASA, Main expert

Stockholm, Nordic Semiconductors ASA, March 7, 2024

Prof. Medard Rieder	Prof. Nabil Abdennadher
Advisor	Program director

Abstract

TODO

Key words: keyword1, keyword2, keyword3

Contents

Abstract (English/Français) **v**

Contents **vii**

List of Figures **ix**

List of Tables **xiii**

List of Listings **xv**

1 Introduction **1**
1.1 State-of-the-Art . . . 2
1.2 Problem Statement . . . 5

2 Wi-Fi and Networking Essentials **7**
2.1 OSI Model . . . 9
2.2 Wi-Fi Essentials and Data Link Layer . . . 12
2.3 Wi-Fi Association Procedure . . . 17
2.4 Wi-Fi 6 . . . 21
2.5 Wi-Fi Power Save Modes . . . 25
2.6 Target Wake Time . . . 30
2.7 Network Layer . . . 32
2.8 Transport Layer . . . 39
2.9 Application Layer Protocols . . . 42

3 Testbed Design **45**
3.1 Testbed . . . 46
3.2 Server . . . 47
3.3 Tests . . . 48
3.4 Monitoring . . . 54
3.5 Network Topology . . . 55

4 Testbed Client Implementation **57**
4.1 System Architecture . . . 58
4.2 Testbed Configuration . . . 59
4.3 Software Components . . . 68

5 Testbed Server Implementation **91**
5.1 Testbed Server . . . 92
5.2 Class Diagram . . . 92

5.3 Resources 93
5.4 Benefits of the Private Server 95
5.5 Drawbacks of the Private Server 95

6 Analysis **97**
6.1 Limitations of employing TWT in practical applications 98
6.2 ARP Analysis 101
6.3 NDP Analysis 104
6.4 Test Results Analysis 106

7 Conclusions **133**
7.1 Project summary 133
7.2 Comparison with the initial objectives 133
7.3 Encountered difficulties 133
7.4 Future perspectives 133

A Source Code Repositories **135**

B Wi-Fi Monitor Mode on Linux **137**

C ARP Tests **139**

D NDP Test **143**

E Sensor Tests Results **149**

F Large Packet Tests Results **165**

G Multi Packet Tests Results **177**

H Actuator Tests Results **189**

References **201**

Glossary **205**

List of Figures

2.1 OSI model layers . . . 9
2.2 OSI model and TCP/IP model comparison . . . 10
2.3 Data encapsulation [8] . . . 11
2.4 An example of a Wi-Fi network setup . . . 12
2.5 Unicast, multicast, and broadcast traffic . . . 14
2.6 MAC frame structure . . . 15
2.7 Wi-Fi connection sequence [9] . . . 17
2.8 Beacon frame structure . . . 18
2.9 Probe request/response . . . 18
2.10 Authentication frame . . . 19
2.11 Association request frame . . . 20
2.12 Association response frame . . . 20
2.13 OFDMA [9] . . . 22
2.14 Wi-Fi router with and without beamforming [9] . . . 23
2.15 Overlapping BSS [9] . . . 24
2.16 TIM element structure [9] . . . 26
2.17 TIM beacon and broadcast/multicast traffic [23] . . . 27
2.18 Legacy Power Save mode [23] . . . 27
2.19 WMM Power Save mode [23] . . . 28
2.20 Change Power Save wake-up mode form DTIM to Listen Interval [23] . . . 29
2.21 TWT scheduling [23] . . . 30
2.22 TWT Intervals . . . 31
2.23 IPv4 packet structure [25] . . . 34
2.24 IPv6 packet structure [25] . . . 36
2.25 UDP packet structure [25] . . . 39
2.26 DTLS Connection ID [32] . . . 40
2.27 MQTT publish-subscribe model [36] . . . 42
2.28 CoAP packet structure [36] . . . 43

3.1 Testbed Protocol Stack . . . 46
3.2 Sensor TWT Test . . . 48
3.3 CoAP Blockwise Transfer . . . 49
3.4 Multi Packet Test . . . 50
3.5 Actuator Public Server Test . . . 51
3.6 Actuator Private Server Test . . . 52
3.7 Actuator Private Server Test with Echo . . . 52
3.8 Actuator Public Server Test with Emergency Transmission . . . 53
3.9 Actuator Private Server Test with Echo and Emergency Transmission . . . 53
3.10 Network Topology . . . 55

List of Figures

4.1 Testbed Component Diagram . . . 58
4.2 Testbed Global Configurations Overview . . . 59
4.3 Testbed CoAP Configurations . . . 60
4.4 Testbed IP Configurations . . . 61
4.5 Testbed Wi-Fi Configurations . . . 61
4.6 Testbed Tests Configurations . . . 62
4.7 Testbed Sensor Tests Configurations . . . 63
4.8 Testbed Large Packet Tests Configurations . . . 64
4.9 Testbed Multi Packet Tests Configurations . . . 65
4.10 Testbed Actuator Tests Configurations . . . 66
4.11 Testbed Logging and Profiling Configurations . . . 67
4.12 wifi_sta component . . . 68
4.13 Wi-Fi Connection Sequence Diagram . . . 69
4.14 wifi_ps component . . . 70
4.15 wifi_twt component . . . 71
4.16 Wi-Fi TWT Setup Sequence Diagram . . . 72
4.17 Wake-Ahead Timer . . . 73
4.18 wifi_utils component . . . 73
4.19 profiler component . . . 74
4.20 CoAP component data flow . . . 76
4.21 CoAP component . . . 77
4.22 CoAP request transmission sequence . . . 78
4.23 CoAP request reception sequence . . . 79
4.24 CoAP Utils component . . . 80
4.25 CoAP Security component . . . 81
4.26 Sensor Test Component . . . 82
4.27 Test Execution Sequence . . . 83
4.28 Sensor Test Sequence Diagram . . . 84
4.29 Actuator Test Sequence Diagram . . . 86

5.1 Testbed Server Class Diagram . . . 92

6.1 Typical protocol stack of an IoT device . . . 98
6.2 Packet loss for the sensor use case . . . 107
6.3 Response time histogram for a sensor use case test performed on the public server, with a 5 second TWT interval and 16 ms TWT session duration 108
6.4 Average response time for the sensor use case . . . 109
6.5 Standard deviation of the average response time for the sensor use case . 110
6.6 Current consumption during a TWT session . . . 111
6.7 Average current for sensor use case . . . 111
6.8 Average current for sensor use case . . . 112
6.9 Response time histogram for a sensor use case test using recovery mode (threshold = 2), with a 5 second TWT interval and 8 ms TWT session duration . . . 114
6.10 Recovery mode current consumption comparison . . . 115
6.11 Recovery mode current trace . . . 117
6.12 Packet loss for large packet test case . . . 119
6.13 Proportion of lost requests and responses . . . 120

6.14 Average response time for large packet test case 121
6.15 Packet loss for multi packet use case 123
6.16 Average response time for multi packet use case 124
6.17 Packet loss for the actuator use case 126
6.18 Echo time histogram for an actuator use case test using a 5 second TWT interval and 16 ms TWT session duration 127
6.19 Response time for the actuator use case 128
6.20 Modified test case with double echo 129
6.21 Response time for actuator use case with double echo 130
6.22 Response time histogram for an actuator use case test with double echo, using a 5 second TWT interval and 57 ms TWT session duration 131
6.23 Average current consumption with and without emergency uplink 132

List of Tables

2.1 Wi-Fi generations [19] 21

4.1 Sensor Use Case Test Outputs 85
4.2 Actuator Use Case Test Outputs 86

6.1 ARP and CoAP frames exchanged after the connection 101
6.2 ARP and CoAP frames exchanged after the traffic resumed 102
6.3 NDP and CoAP frames exchanged after the connection 104
6.4 AP asks for STA's local IP address 105
6.5 AP asks for STA's global IP address 105
6.6 STA asks for servers's global IP address 105
6.7 STA asks for AP's local IP address 105

E.1 Power measurements for the sensor use case 164

F.1 Power measurements for the large packet test case 175

G.1 Power measurements for the multi packet use case 187

H.1 Power measurements for the actuator use case 199

Listings

4.1 Sensor TWT test invocation 82
4.2 Test Result Example 87
4.3 Test Invocation 88
6.1 Multi Packet Test Logs 122
B.1 Wi-Fi Monitor Mode on Linux 137

1 Introduction

The evolution of wireless communication technologies has significantly reshaped how devices connect and interact with their environment. Wi-Fi has become one of the most important technologies of this transformation, supporting billions of devices across a wide range of applications—from home networks to industrial IoT systems. However, the increasing number of connected devices and the wide range of applications bring challenges to performance, scalability, and energy efficiency. The recent advancements in Wi-Fi technology focus on addressing these issues. Among these, Target Wake Time (TWT) is a feature introduced in Wi-Fi 6 specifically designed to optimize energy consumption. This makes it particularly valuable for battery-powered IoT devices.

This thesis aims to evaluate the impact of TWT in practical IoT scenarios. TWT is a feature that minimizes power usage by enabling devices to schedule extended sleep times, reducing unnecessary active periods. However, this benefit comes at the cost of increased constraints on compatibility with existing protocols. Understanding these trade-offs is critical for the deployment of TWT in practical systems.

The report begins by providing a foundation of Wi-Fi and networking essentials. This includes an overview of the OSI model, the evolution of Wi-Fi standards, and the power-saving mechanisms that have been integrated over time. These fundamentals are crucial to understanding how TWT operates and its implications for IoT devices.

Following this, the design and implementation of a dedicated testbed are presented. The testbed replicates realistic IoT use cases and measures key metrics like latency, packet loss, and energy consumption. These tests aim to provide practical insights into TWT's strengths and limitations.

The key aspect explored in this thesis is the interaction between TWT and upper-layer protocols. Traditional communication protocols based on TCP are not well-suited for TWT's extended sleep intervals, as they rely on continuous communication. In contrast, protocols operating over UDP are better aligned with TWT's constraints. This thesis examines these challenges and proposes alternative approaches to ensure reliable communication.

Through detailed experiments and analysis, this work evaluates TWT's benefits and limitations. The results highlight the importance of aligning application requirements with TWT's operational constraints to maximize its potential. This thesis contributes to a deeper understanding of its role in enabling energy-efficient networks by addressing the challenges associated with integrating TWT into real-world IoT systems.

1.1 State-of-the-Art

1.1.1 Wi-Fi

Wi-Fi, standardized under the IEEE 802.11 family, is one of the most widely used wireless communication technologies. Its primary goal is to provide internet connectivity in environments ranging from home networks to large-scale enterprises. Since its inception, Wi-Fi standards have evolved a lot, and the advancements have introduced different features to increase throughput, coverage, and efficiency. These advancements also include mechanisms specifically introduced to decrease power consumption, which is a critical concern for battery-operated devices. Traditional Wi-Fi operation requires STAs to continuously monitor the channel for incoming packets, which consumes significant power. As Wi-Fi becomes a key enabler for IoT applications, minimizing energy consumption while maintaining reliable communication is essential.

1.1.2 Wi-Fi Power Saving Mechanisms

To address the critical challenge of reducing the power consumption of the STAs in Wi-Fi networks, several power-saving mechanisms have been integrated into the IEEE 802.11 standards over time. The two most notable approaches are:

- **Power Save Mode (PS Mode)**: Introduced in early Wi-Fi standards, PS mode allows STAs to reduce their energy consumption by entering a low-power state during periods of inactivity. Although effective, PS mode requires STAs to wake up periodically to listen for beacon frames, even when no data is transmitted. This frequent wake-up behavior reduces its efficiency for devices with infrequent communication needs. Chapter 2 provides a detailed description of Wi-Fi PS mode in Section 2.5.

- **Target Wake Time (TWT)**: Introduced in Wi-Fi 6 (IEEE 802.11ax), TWT allows STAs to negotiate specific wake and sleep schedules with the AP. By staying inactive for long intervals, TWT allows for excellent energy savings. However, this comes at the cost of increased link layer latency and reduced flexibility. TWT is best suited for devices with low data rates and high tolerance for latency, such as IoT devices. Chapter 2 provides a detailed description of Wi-Fi TWT in Section 2.6.

PS mode introduces minimal constraints on communication performance and aligns closely with traditional Wi-Fi behavior from a higher-layer perspective. However, frequent wake-ups limit its efficiency for devices with sparse transmissions. In contrast, TWT offers superior energy efficiency. However, extended sleep periods impose significant constraints on communication. This trade-off makes it unsuitable for latency-sensitive applications but has much potential for devices with predictable data transmission patterns.

Please note that this comparison between TWT and PS mode considers the PS mode using "common" settings (i.e., Legacy mode and DTIM wake-up mode, with a network using typical DTIM periods). Using the listen interval wake-up mode or changing the

DTIM periods would effectively reduce the wake-up frequency. However, this would impose similar constraints as TWT does but would not benefit from the advantages that the TWT negotiation brings. Therefore, these configurations are not considered in this comparison. The PS mode situation considers a "common" PS configuration (i.e., legacy mode, DTIM wake-up mode, AP in DTIM1, and a DTIM period of 100ms).

1.1.3 Challenges with Higher-Layer Protocols and TWT

While TWT offers considerable energy savings, its constraints significantly affect the compatibility of higher-layer communication protocols commonly used in IoT applications.

- **TCP and TWT Incompatibility**: Many application layer protocols, including MQTT and HTTP, use TCP as transport layer protocol. A lot of IoT device implementations rely on these protocols. TCP is a connection-oriented protocol requiring persistent connections and continuous communication to maintain state information and ensure reliable delivery. This behavior conflicts with TWT's long sleep intervals, where the device remains inactive for extended periods, causing connection drops. The inherent need to maintain a persistent connection makes TCP-based protocols poorly suited for applications that leverage TWT's extended sleep times.

- **CoAP as a Solution**: In contrast, the Constrained Application Protocol (CoAP) is better aligned with TWT. CoAP uses UDP, which is connectionless and does not require a persistent connection or continuous state maintenance. This allows CoAP to support long inactive intervals. CoAP's design enables efficient, stateless communication, which makes it a perfect fit for IoT devices utilizing TWT. Its lightweight nature ensures that the power-saving benefits of TWT are preserved while maintaining reliable communication.

Chapter 2 further explores CoAP and its underlying protocols.

1.1.4 Related Works

The key benefit of TWT is its potential to significantly reduce power consumption, as explored in *On the Joint Usage of Target Wake Time and 802.11ba Wake-Up Radio* [1]. This study evaluates the TWT approach to achieve energy savings in wireless networks. The results highlight substantial reductions in power usage compared to scenarios without TWT. However, the study does not address how these energy savings translate to performance and reliability in commercial IoT applications, particularly those characterized by intermittent communication.

The *IEEE 802.11ax Target Wake Time: Design and Performance Analysis in ns-3* study [2] further demonstrates TWT's ability to reduce energy consumption. It proves that TWT significantly reduces energy consumption by aligning station activity with scheduled wake times, achieving energy savings compared to continuously active modes. The paper explores trade-offs between energy costs, network performance, latency, and throughput. However, its focus on short intervals limits its relevance to IoT scenarios,

which often involve very low duty-cycle operations with intervals of many seconds. The impact on communication reliability and higher-layer protocols in these cases remains unexplored.

The *Clock Drift Impact on Target Wake Time in IEEE 802.11ax/ah Networks* study [3] analyzes a fundamental limitation of the TWT mechanism. The clock drift has a direct impact on TWT by causing STAs to deviate from their scheduled times. This study examines the effects of this limitation and proposes a solution: having the STA wake up slightly before the TWT session. By doing so, the STA ensures it doesn't miss the scheduled session. Although this approach results in the STA being awake for a longer period, which increases power consumption, the overall power usage remains significantly reduced. Unlike the two previously analyzed studies, this study is more practical and close to real-world problems, as it addresses issues that are often ignored in theoretical studies. However, like the first two studies, only TWT itself and its technical details are analyzed, and the impact it has on the upper layers remains unexplored.

The *Performance Evaluation of Video Streaming Applications with Target Wake Time in Wi-Fi 6* study [4] investigates the effectiveness of TWT in reducing contention in high-density networks. The findings show that TWT scheduling can significantly decrease contention overhead, leading to improved global throughput. However, the study focuses on scenarios with relatively high duty-cycle operations, which are not representative of the low-duty-cycle use cases typical of IoT applications. In this study, TWT is not used for power-saving purposes but to maximize global throughput. As a result, the implications for IoT communication performance remain unexplored.

While the reviewed studies provide valuable insights into the energy-saving capabilities, analytical modeling, and contention management benefits of TWT, none address its impact on communication, particularly on the upper layers of the protocol stack. IoT applications often involve long intervals, with devices staying inactive for many seconds, where TWT's influence on communication latency, reliability, and protocol behavior becomes more significant. Moreover, the reliance on idealized scenarios and simulated traffic patterns limits the applicability of these findings to real-world use cases.

1.1.5 Project objective

This thesis seeks to bridge this gap by analyzing the impact of TWT on communication performance in practical IoT deployments. By focusing on representative use cases with low-duty-cycle operations and leveraging realistic traffic patterns, this work aims to provide a comprehensive evaluation of TWT's implications for commercial applications. The findings will contribute to a better understanding of TWT's potential and limitations, enabling its optimization for real-world IoT scenarios.

The absence of public studies on the impact of TWT on communication highlights the limited current knowledge about this feature. TWT is still new and very experimental, which explains why it hasn't been widely adopted yet. Several factors contribute to this. Firstly, TWT introduces significant constraints, making it inherently unsuitable for many use cases. Secondly, it is not compatible with current standard protocols widely used in IoT. Additionally, as Wi-Fi 6 is still relatively new, many existing networks rely

on older-generation access points that do not support TWT. Despite these challenges, this study aims to address these issues and propose solutions to enable broader adoption and effective use of TWT in the future.

1.2 Problem Statement

TWT is a key feature of Wi-Fi 6 that significantly enhances power consumption efficiency. TWT can considerably reduce the power consumption of the Wi-Fi module compared to the traditional Power Save (PS) mode (DTIM/Legacy). This substantial reduction makes TWT an essential feature for Nordic Semiconductor, as they produce low-power Wi-Fi modules.

TWT is part of Wi-Fi 6, adopted in 2021. It is very recent, implemented in very few stacks, and few certifications are available. While TWT shows great promise for reducing power consumption, it introduces several major constraints to the network and is not suitable for all use cases. TWT is primarily usable in devices that can tolerate significant latency and have low data transmission rates. These constraints make it incompatible with the standard protocol stacks typically used in IoT applications. In summary, TWT is still considered experimental and does not yet meet customers' needs effectively.

This project aims to evaluate the impact of using TWT in Wi-Fi IoT applications by developing a program that implements typical commercial application use cases, utilizing alternative protocol stacks that align with TWT constraints. The program evaluates the TWT feature and provides metrics such as packet loss and latency. The program is designed to identify potential issues that TWT may cause and propose solutions for these problems.

There are two primary use cases for IoT devices. The first is the sensor use case, which involves uplink transmissions. In this scenario, the client (Wi-Fi station) initiates the transmission. A commercial example of this would be a connected temperature sensor that periodically transmits its measurements. The second primary use case is the actuator use case, which involves downlink transmissions. In this use case, the server initiates the transmission, and the client (Wi-Fi station) receives the data. An example of this application would be a connected door lock that can be opened or closed remotely. The testbed application developed for this project includes various tests for both configurations. Details about the use cases and tests can be found in Chapter 3.

The initial expectations regarding the impact of using TWT are that the end-to-end latency of a unidirectional transmission (uplink or downlink) may increase by up to one TWT interval. In terms of packet loss, no significant increase is anticipated, as the maximum throughput of the medium is not reached in the use cases considered. Protocols that rely on broadcast and multicast traffic are expected to face limitations, while those requiring a continuous stream, such as TCP, are deemed incompatible with TWT due to its extended sleep periods.

2 | Wi-Fi and Networking Essentials

This chapter offers an overview of the essential concepts in Wi-Fi communication, focusing on low-power aspects and IoT applications. It introduces fundamental concepts necessary to understand the thesis's results, analysis, and findings.

The chapter begins with an introduction to Wi-Fi and the OSI model. The subsequent sections focus on the data link layer and specific aspects of Wi-Fi, including Wi-Fi 6 features and power-saving mechanisms. Following this, the network layer is discussed, focusing on the Internet Protocol (IP). Finally, the last sections explore the transport and application layers, covering protocols such as UDP, DTLS, and CoAP.

This chapter establishes the foundation for analyzing Wi-Fi's role in supporting low-power IoT. The explanations are tailored to the level of detail necessary to understand the thesis outcomes.

Contents

2.1 OSI Model . . . 9
2.1.1 Encapsulation . . . 11
2.2 Wi-Fi Essentials and Data Link Layer . . . 12
2.2.1 Wi-Fi Essentials . . . 12
2.2.2 Wi-Fi Frames . . . 13
2.2.3 Handling Frame Loss and Acknowledgments . . . 13
2.2.4 Unicast, Multicast, and Broadcast Traffic . . . 14
2.2.5 MAC Frame Structure . . . 15
2.2.6 Channel access and collision avoidance . . . 16
2.3 Wi-Fi Association Procedure . . . 17
2.3.1 Discovery Phase . . . 18
2.3.2 Authentication Phase . . . 19
2.3.3 Association Phase . . . 20
2.3.4 Security Handshake . . . 20
2.4 Wi-Fi 6 . . . 21
2.4.1 OFDMA . . . 22
2.4.2 MU-MIMO and Beamforming . . . 23
2.4.3 TWT . . . 23
2.4.4 1024-QAM . . . 23
2.4.5 WPA3 security . . . 23
2.4.6 BSS Coloring . . . 24
2.5 Wi-Fi Power Save Modes . . . 25
2.5.1 Power States . . . 25
2.5.2 Wi-Fi Beacon frames . . . 26
2.5.3 DTIM Power Save . . . 26
2.5.4 Extended Sleep . . . 29

2.6 Target Wake Time 30
2.7 Network Layer 32
2.7.1 Internet Protocol (IP) 32
2.7.2 Address Resolution Protocol (ARP) 37
2.7.3 Neighbor Discovery Protocol (NDP) 37
2.8 Transport Layer 39
2.8.1 User Datagram Protocol 39
2.8.2 Transmission Control Protocol 41
2.9 Application Layer Protocols 42
2.9.1 HTTP 42
2.9.2 MQTT 42
2.9.3 CoAP 43

2.1 OSI Model

The Open Systems Interconnection (OSI) model is a framework used for implementing network communications. It divides the tasks into seven distinct layers, each responsible for a different role in the communication.

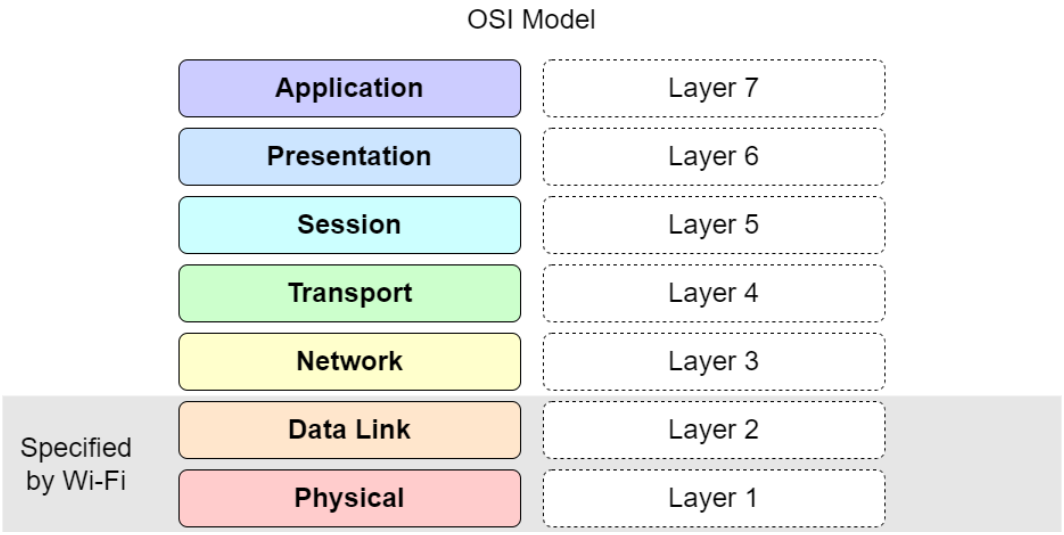


Figure 2.1: OSI model layers

1. **Physical Layer:** The physical layer transmits the raw binary data over a physical medium. It defines the hardware specifications. Examples of physical layers are cables and radio signals.

2. **Data Link Layer:** The Data link layer is divided into two sublayers: the Logical Link Control (LLC) and the Media Access Control (MAC). Data Link Layer is responsible for the direct communication between two connected devices. Network interfaces have a Layer 2 or MAC address used for this purpose. Examples of data link layers are Wi-Fi and Ethernet.

3. **Network Layer:** Layer 3 provides an end-to-end addressing system for transferring data packets across several Layer 2 networks. The most common example of a Layer 3 protocol is the Internet Protocol (IP). The Internet uses IP protocol to interconnect millions of networks around the world.

4. **Transport Layer:** Layer 4 offers end-to-end communication between systems (applications) through a network. It either offers best-effort connectionless or reliable connection-oriented communication. The most common transport protocols used in IP networking are TCP and UDP.

5. **Session Layer:** The session layer manages sessions between application processes. Application environments using remote procedure calls (RPCs) commonly use session-layer services.

6. **Presentation Layer:** The presentation layer defines how an application formats the data to be sent. It can also implement security by encrypting messages. Examples of presentation layers are SSL/ TLS, JSON, XML, HTML, or ASCII.

7. **Application Layer:** The application layer defines how the end-user application uses the layers lying under to communicate with other devices. Examples of Application Layer protocols are HTTP, MQTT, or CoAP.

However, it is important to remember that the OSI model is only a theoretical representation of a system. Some protocols (such as security on high levels) are sometimes difficult to position in and limit to a particular layer.

The TCP/IP model, which condenses the seven layers of OSI into four layers, is sometimes also used to address this problem in practical, real-world implementations.

The following figure shows how the layers are condensed.

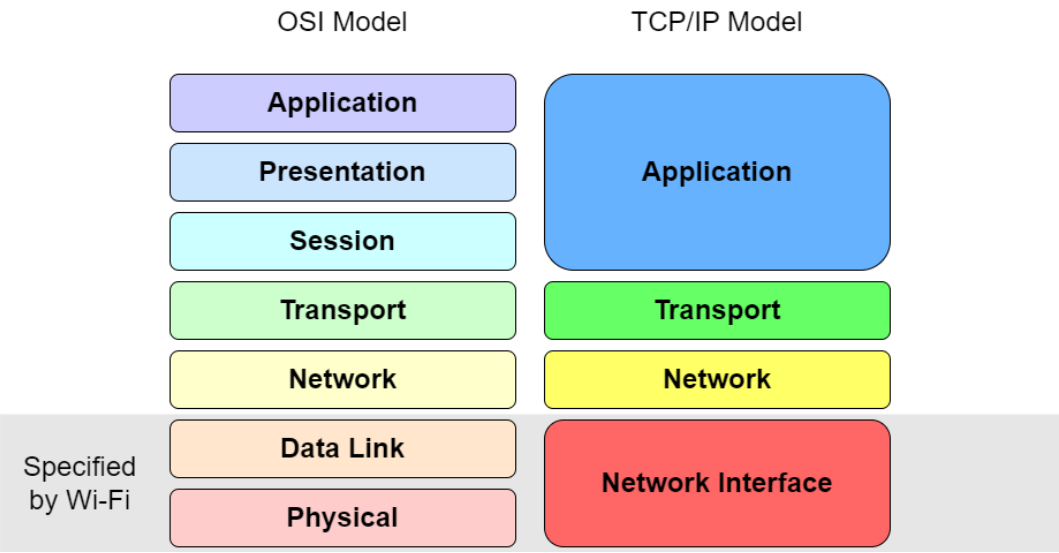


Figure 2.2: OSI model and TCP/IP model comparison

[5][6][7]

2.1.1 Encapsulation

Multiple protocols and layers are involved when data is transmitted over a network. The application payload data is encapsulated within packets, with each layer adding its header information. This encapsulation supplies all the management required for transmitting the actual payload data, such as addressing and routing the packets to their destination, adding security, or managing the data stream.

The following figure shows this principle.

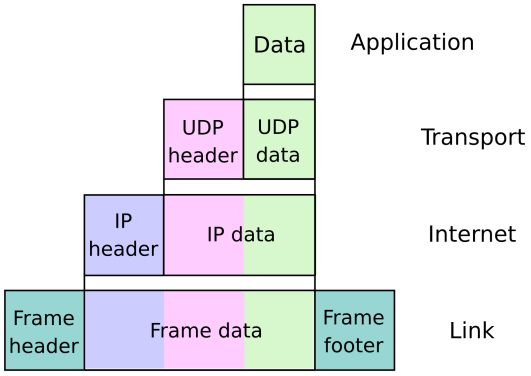


Figure 2.3: Data encapsulation [8]

2.2 Wi-Fi Essentials and Data Link Layer

2.2.1 Wi-Fi Essentials

Wi-Fi is a suite of wireless network protocols based on the IEEE 802.11 standards. Its purpose is to connect devices to each other by forming an IP-based Wireless Local Area Network (WLAN). IEEE 802.11 specifies the media access control (MAC) layer and physical (PHY) layer protocols for implementing WLAN computer communication. Wi-Fi is the world's most widely used wireless networking standard. It is used in most networks to allow laptops, smartphones, printers, and other devices to communicate and access the Internet. Wi-Fi has also become attractive for IoT applications because many networks are already deployed worldwide. [9] [10]

Wi-Fi uses the 2.4 GHz and 5 GHz radio bands, most commonly but not limited to. Newer versions of the standard also include support for the 6 GHz band. These bands are subdivided into multiple channels, which are used to transmit the frames. As a network can be in the range of another, Wi-Fi manages multiple access using CSMA/CA. The communication is best-effort delivery, meaning that it does not guarantee the delivery of packets. [10]

Wi-Fi is based on two main device types. Access Points (APs) and Stations (STAs). The central node of a Wi-Fi network is the AP. It broadcasts a wireless signal that allows the STAs to connect to the network. The STAs are devices such as laptops, smartphones, or IoT devices. [9]

A BSS (Basic Service set) is a network setup where one or more STAs connect to an AP, forming a Wi-Fi network. It is the most common type of Wi-Fi network configuration. A BSS is identified by a name called the Service Set Identifier (SSID), which STAs use to connect. However, the SSID is not a secure identifier, as it is user-defined and so not unique. The BSS also uses the AP's MAC address as another identifier, called the Basic Service Set Identifier (BSSID), which helps devices connect to the correct AP when multiple networks have the same SSID. [9]

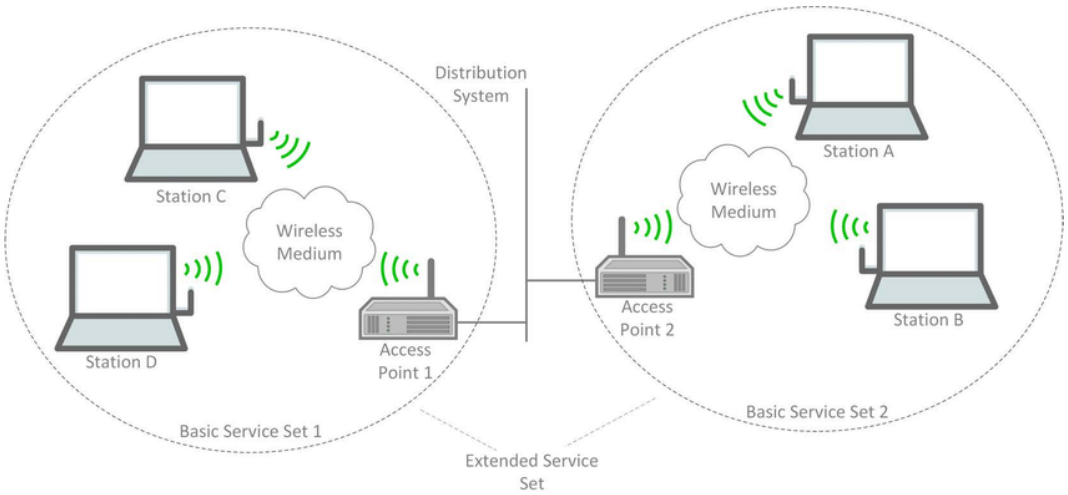


Figure 2.4: An example of a Wi-Fi network setup

2.2.2 Wi-Fi Frames

Wi-Fi communications allow devices to communicate wirelessly via electromagnetic waves. In Wi-Fi, data is transmitted in packets known as frames. While 802.11 (Wi-Fi) frames are similar to 802.3 (Ethernet) frames, each standard includes specific fields tailored to the unique requirements of its respective physical layer.

Frame Types

Wi-Fi frames include metadata and the payload (the actual data) and are categorized into three types:

- **Management frames:** These are utilized to establish and manage connections, such as association and authentication. (e.g., beacon frames, probe requests)
- **Control frames:** These control the channel access and the delivery of the other frames. (e.g., CTS, RTS, ACK, PS-Poll).
- **Data frames:** These carry the actual user data and contain higher-level data packets in their payload (e.g., IP packet)

[11][12]

2.2.3 Handling Frame Loss and Acknowledgments

In 802.11, the receiver acknowledges frames with an acknowledgment (ACK) message. If the sender does not receive the ACK, it retries the transmission until the frame is successfully delivered or a timeout occurs.

This mechanism is critical in wireless environments where signal interference and attenuation is likely. Frame loss can be considered a random occurrence, influenced by various factors related to the physical layer. These factors include, for example, other devices transmitting on the same channel (whether they are 802.11-compliant or not), significant physical distance, or obstacles between the AP and the STA.

Not all frames are acknowledged. Whether a frame is acknowledged depends on the traffic type (see 2.2.4) and frame type (see 2.2.2). Only unicast frames are acknowledged; specifically, all unicast data frames require acknowledgments. Acknowledgment behavior varies for management and control frames. For instance, PS-Poll frames are unicast and acknowledged. However, RTS, CTS, and ACK frames, while also unicast, are not acknowledged due to logical reasons inherent to their role in the protocol.

[11]

2.2.4 Unicast, Multicast, and Broadcast Traffic

Wi-Fi supports different traffic types for sending data over the network:

- **Unicast:** A unicast frame is sent by one device and addressed to a specific device (one-to-one). A significant portion of the internet traffic (e.g., loading a webpage) is unicast because each client device has unique data requests.
- **Multicast:** A multicast frame is sent by one device and addressed to multiple devices (one-to-many). Only one frame is sent, but multiple devices receive it. It is typically used for group communication, such as streaming media to multiple users.
- **Broadcast:** A broadcast frame is sent by one device and addressed to all the devices on the network (one-to-many). Only one frame is sent, but all the devices receive it. Broadcast frames are mainly used by discovery protocols, like ARP, when a packet is destined to all other devices. Since broadcast frames occupy shared bandwidth and are received by all devices, excessive broadcast traffic can slow down network performance.

Multicast and broadcast traffic lack reliability because the frames are not acknowledged.

It is important to note that this classification pertains to logical communication. Wi-Fi operates as a shared medium at the physical layer, meaning all communication is broadcast over the air. For example, it is possible to observe unicast frames using a Wi-Fi sniffer, even if these frames are not addressed to the sniffer.

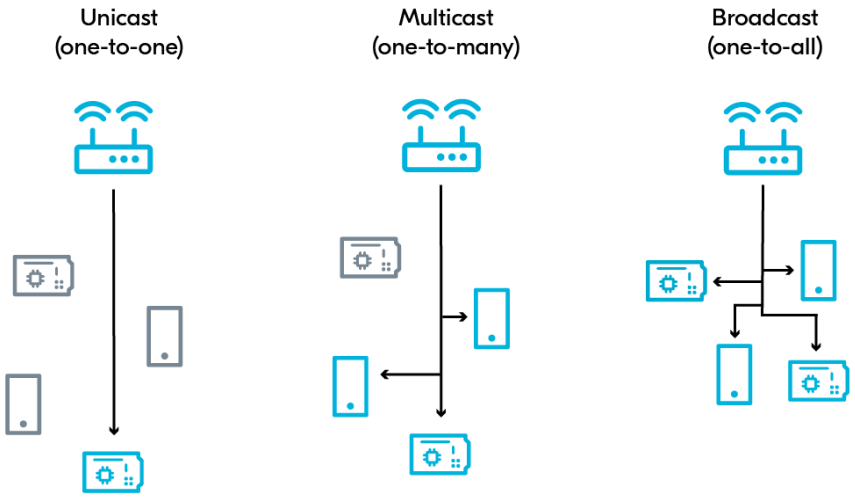


Figure 2.5: Unicast, multicast, and broadcast traffic

[9][11]

2.2.5 MAC Frame Structure

At the data link layer, 802.11 relies on the MAC protocol, which manages device access to the network. Each Wi-Fi frame has a MAC header, which includes the source and destination MAC addresses and a BSSID, identifying the AP.

Figure 2.6: MAC frame structure

- **Frame Control:** Indicates the type and settings of the frame (e.g., protocol version, type).
- **Duration:** Specifies the expected duration of the frame's medium occupancy in microseconds.
- **Addresses:** These may include up to four MAC addresses, helping determine the source, destination, and the AP. The fields are numbered because they can be used for different purposes depending on the frame type.
- **Sequence Control Field:** This field is used for defragmentation and to discard duplicate frames. It contains a fragment number and a sequence number. The sequence number is a counter of frames transmitted. If a higher-level packet is fragmented, all fragments will have the same sequence number but a different fragment number. Retransmitted frames keep the sequence number unchanged.
- **Frame body:** The frame body (data field) contains the higher-level packet (e.g., an IP packet).
- **Frame Check Sequence (FCS):** The frame closes with an FCS field. This field checks the integrity of the message and is comparable to a cyclic redundancy check (CRC).

Note that two more optional fields can be added at the end of the header: the QoS control field, which enables traffic prioritization, and the HT control field, which supports high-throughput features.

[11]

2.2.6 Channel access and collision avoidance

In Wi-Fi networks, multiple devices use the same channels. To avoid interference, the following mechanisms are used:

- **CSMA/CA:** Carrier Sense Multiple Access with Collision Avoidance is a mechanism where devices listen to the channel to check that it is free before transmitting. If this is not the case, the device defers the transmission.
- **RTS/CTS:** A sender device can send an RTS frame to check if the channel is clear. If the receiver device responds with a CTS frame, the channel is reserved for that device for a short time.

Collusion avoidance mechanisms can increase transmission efficiency in noisy environments by reducing the number of frames that would have needed retransmission. However, when the transmission is deferred due to channel occupancy, it adds extra delays in the transmission, which is called contention.

[11]

2.3 Wi-Fi Association Procedure

The STA initiates the procedure to connect to an AP. It has four main steps:

- Discovery
- Authentication
- Association
- Security procedure (handshake)

The association procedure sequence is shown below.

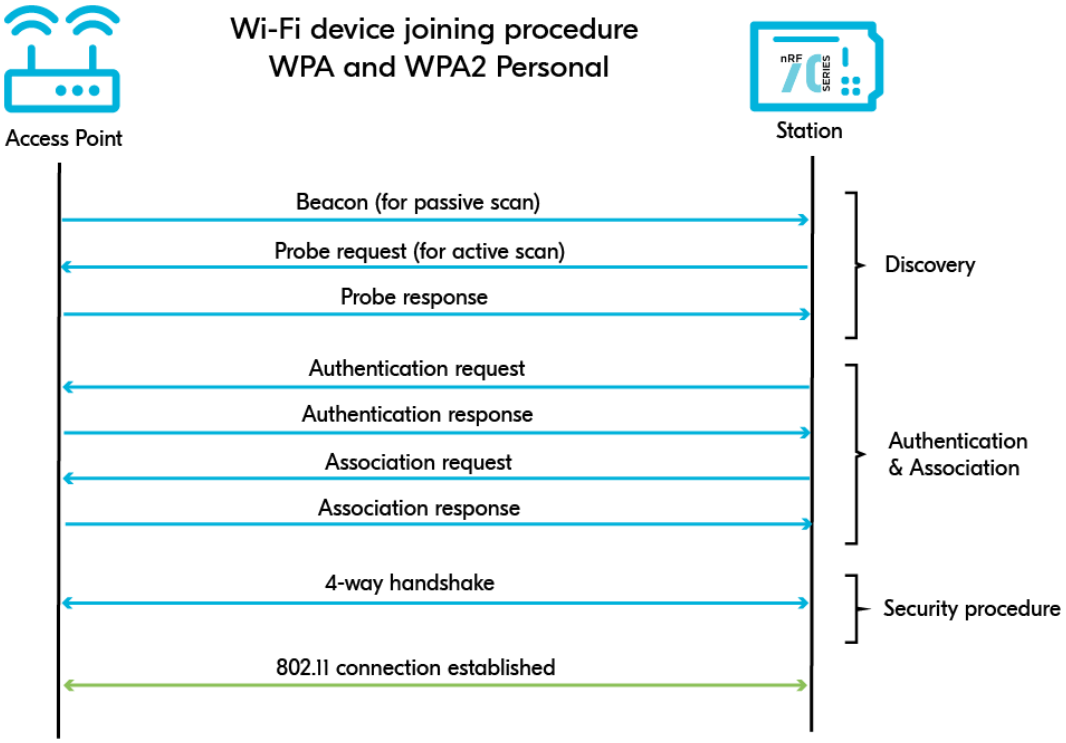


Figure 2.7: Wi-Fi connection sequence [9]

2.3.1 Discovery Phase

The first step in connecting to a Wi-Fi network is discovering the available APs. There are two methods to do this:

Passive Scanning

In this method, the STA listens on each channel for beacon frames periodically broadcasted by APs. These frames contain essential details such as the network's SSID/BSSID, supported data rates, and security protocols. With passive scanning, the STA receives this information and processes it to determine which networks are available.

The beacon frame is shown below.

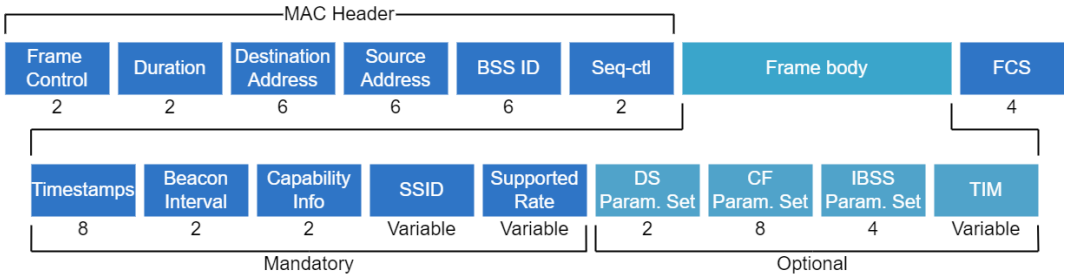


Figure 2.8: Beacon frame structure

Active Scanning

In this method, the STA sends a probe request to ask about available networks, either directed at a specific SSID or as a broadcast, asking any nearby APs to respond. The APs send probe responses with the same information found in beacon frames. This method speeds up network discovery, especially if the STA wants to connect to a specific network.

The probe frame is shown below.

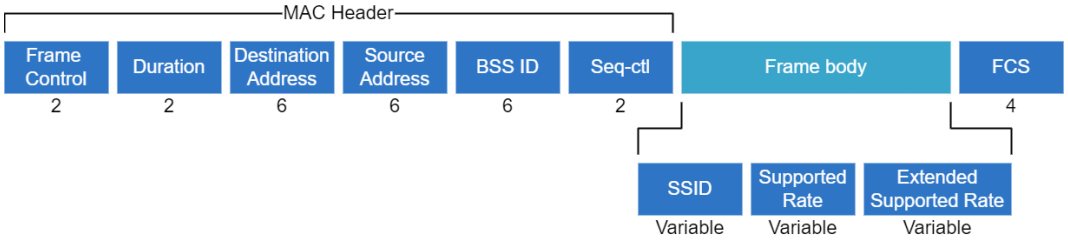


Figure 2.9: Probe request/response

Once the STA has discovered the AP it wants to connect to, it will proceed to the authentication phase. [13][14][15]

2.3.2 Authentication Phase

The STA sends an authentication request to the AP, which responds by accepting or rejecting the request. Authentication aims to verify that the requesting STA has the proper 802.11 capabilities to join the network.

IEEE 802.11-1997 included a WEP shared key exchange authentication mechanism. This exchange adds two more frames to the default open system authentication, resulting in a four-frame exchange. This shared key authentication method requires WEP encryption and is not recommended today as it is no longer secure.

When using WPA/WPA2/WPA3, a second authentication phase occurs after association (i.e., handshake).

The authentication frame is shown below.

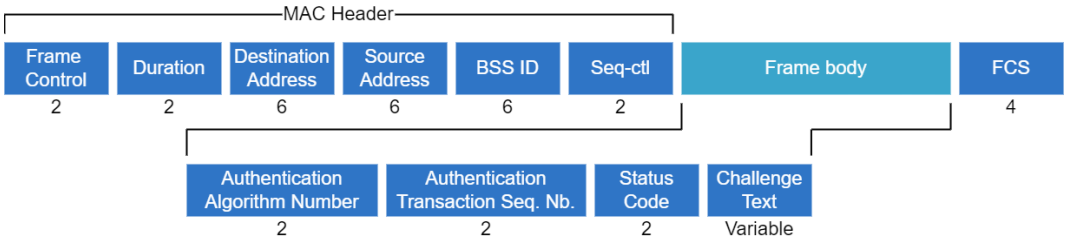


Figure 2.10: Authentication frame

The Authentication Frame consists of the following fields:

- **Algorithm Number:** 0 for Open System and 1 for Shared Key.
- **Transaction Sequence Number:** Indicate the current state of progress.
- **Status Code:** 0 for Success and 1 for Unspecified failures.
- **Challenge Text:** Used in Shared Key Authentication frame.

[13][15]

2.3.3 Association Phase

After successful authentication, the STA formally requests to join the network through the association process. The STA sends an association request frame, including its supported data rates and any other network capabilities it wants to use (e.g., power-saving features).

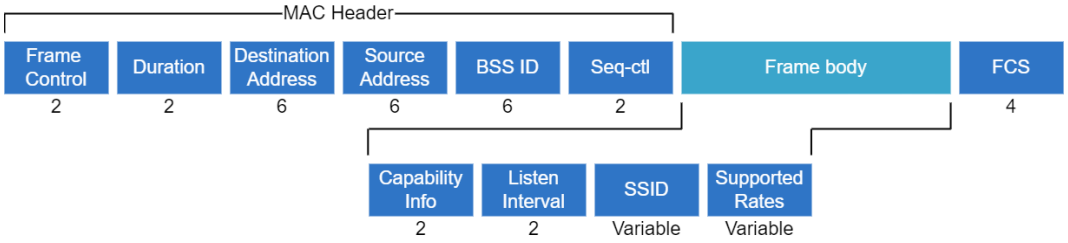


Figure 2.11: Association request frame

In response, the AP sends an association response frame. This frame either approves or denies the STA's request to join the network. If successful, the AP assigns the STA an Association ID (AID), a unique identifier used to track the STA on the network. The association response frame is shown below.

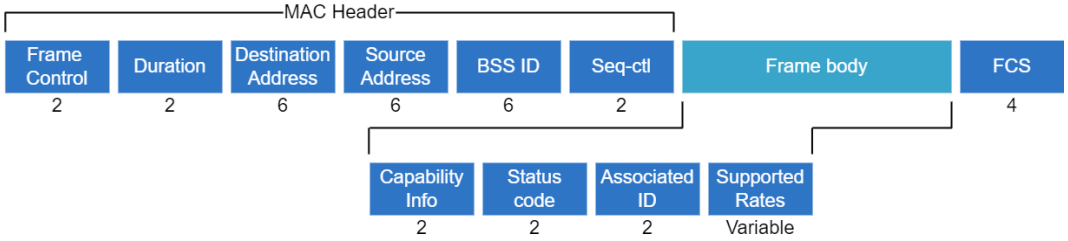


Figure 2.12: Association response frame

At this point, the STA is connected to the AP at the basic 802.11 level, but if the network is secured (WPA2 or WPA3), further steps are necessary to ensure that the data transferred between the STA and the AP is encrypted. [13][15]

2.3.4 Security Handshake

When using WPA, WPA2, or WPA3 to secure a Wi-Fi network, a security phase, called a handshake, occurs after the association stage with the AP. It is a post-association step that establishes a secure and encrypted connection between the STA and the AP. The handshake process ensures that both devices share the same encryption key, allowing secure communication.

WPA and WPA2 use a 4-way handshake. However, it is vulnerable to attacks such as KRACK [16], which enables data decryption without knowing the Pre-Shared Key (PSK). WPA3 improves WPA2 by employing the Simultaneous Authentication of Equals (SAE) protocol, which replaces the PSK method. SAE introduces a handshake called "Dragonfly" that significantly strengthens security against offline attacks. [17][15][18]

2.4 Wi-Fi 6

The development of Wi-Fi has seen significant progress since it has been introduced. The original IEEE 802.11 standards delivered fundamental wireless connectivity in the 2.4 GHz frequency band. Following developments, IEEE 802.11a and 802.11b focused on improving data transmission speeds within the 2.4 GHz and 5 GHz frequency bands, respectively. IEEE 802.11g standard combined the two approaches, working in the 2.4 GHz band while achieving speeds up to 54 Mbps.

IEEE 802.11n (Wi-Fi 4) represented a major advancement in Wi-Fi technology, providing speeds up to 600 Mbps in both the 2.4 and 5 GHz bands. IEEE 802.11ac (Wi-Fi 5) further enhanced performance, surpassing 1 Gbps in the 5 GHz band, making Wi-Fi suitable for streaming 4K videos.

The latest standard, IEEE 802.11ax (Wi-Fi 6), added significant advancements in radio technology, antenna design, and energy efficiency. The 6 GHz band have been added in the extended version, Wi-Fi 6E. While increasing data speed in crowded environments is the main objective of Wi-Fi 6, keeping low power consumption is also an important goal, which makes it an excellent choice for the growing use of low-power Internet of Things (IoT) devices. [9][19]

<table>
<tr><th>Generation</th><th>IEEE Standard</th><th>Adopted</th><th>Maximum link rate [Mb/s]</th><th>Radio frequency [GHz]</th></tr>
<tr><td>(Wi-Fi 0*)</td><td>802.11</td><td>1997</td><td>1–2</td><td>2.4</td></tr>
<tr><td>Wi-Fi 1*</td><td>802.11b</td><td>1999</td><td>1–11</td><td>2.4</td></tr>
<tr><td>Wi-Fi 2*</td><td>802.11a</td><td>1999</td><td rowspan="2">6–54</td><td>5</td></tr>
<tr><td>Wi-Fi 3*</td><td>802.11g</td><td>2003</td><td>2.4</td></tr>
<tr><td>Wi-Fi 4</td><td>802.11n</td><td>2009</td><td>6.5–600</td><td>2.4, 5</td></tr>
<tr><td>Wi-Fi 5</td><td>802.11ac</td><td>2013</td><td>6.5–6933</td><td>5</td></tr>
<tr><td>Wi-Fi 6</td><td rowspan="2">802.11ax</td><td rowspan="2">2021</td><td rowspan="2">0.4–9608</td><td>2.4, 5</td></tr>
<tr><td>Wi-Fi 6E</td><td>2.4, 5, 6</td></tr>
<tr><td>Wi-Fi 7</td><td>802.11be</td><td>expected 2024</td><td>0.4–23,059</td><td>2.4, 5, 6</td></tr>
<tr><td>Wi-Fi 8</td><td>802.11bn</td><td>expected 2028</td><td>100,000</td><td>2.4, 5, 6</td></tr>
<tr><td colspan="5">*Wi-Fi 0, 1, 2, and 3 are named by retroactive inference.
They do not exist in the official nomenclature.</td></tr>
</table>

Table 2.1: Wi-Fi generations [19]

The main features Wi-Fi 6 brings to the table are:

- Orthogonal Frequency Division Multiple Access (OFDMA)
- Multi-User Multiple Input Multiple Output (MU-MIMO)
- Target Wake Time (TWT)
- 1024-QAM
- BSS Coloring
- Beamforming
- WPA3 security

2.4.1 OFDMA

Wi-Fi uses Orthogonal Frequency Division Multiplexing (OFDM), which divides the available frequency spectrum into multiple subcarriers. The orthogonality of these subcarriers allows them to overlap without causing interference because the peak of one subcarrier aligns with the zero of the adjacent one, preventing interference. This optimizes the use of the available spectrum.

Before Wi-Fi 6, a single user occupied all available subcarriers in a channel during a transmission. Wi-Fi 6 introduces Orthogonal Frequency-Division Multiple Access (OFDMA), which groups subcarriers into smaller resource units (RUs) that can be assigned to multiple users simultaneously. This allows more efficient use of the increased number of subcarriers, improving spectrum and channel bandwidth utilization. The impact of OFDMA is a reduction in concurrent access to RUs, which decreases contention and enhances overall network performance. [9][19]

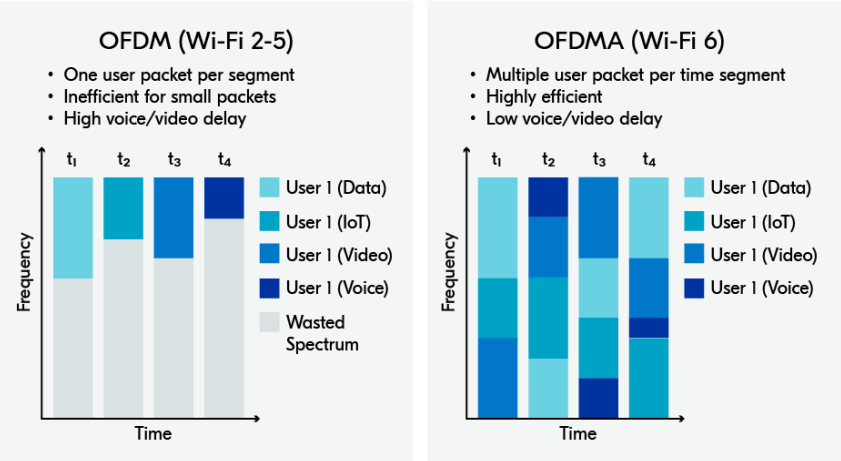


Figure 2.13: OFDMA [9]

2.4.2 MU-MIMO and Beamforming

In Wi-Fi 6, MU-MIMO has been extended to operate in both the uplink and downlink directions. This enables the AP to communicate with multiple STAs at the same time. Beamforming is used to separate devices in different spatial streams. It directs the wireless signal towards the intended device, which improves the signal strength and quality and avoids interferences in the other directions. Moreover, in Wi-Fi 6, OFDMA and MU-MIMO can operate concurrently, allowing for even more efficient utilization of the wireless spectrum by dividing frequencies and allowing more devices to share the same channel. [19][20] [21]

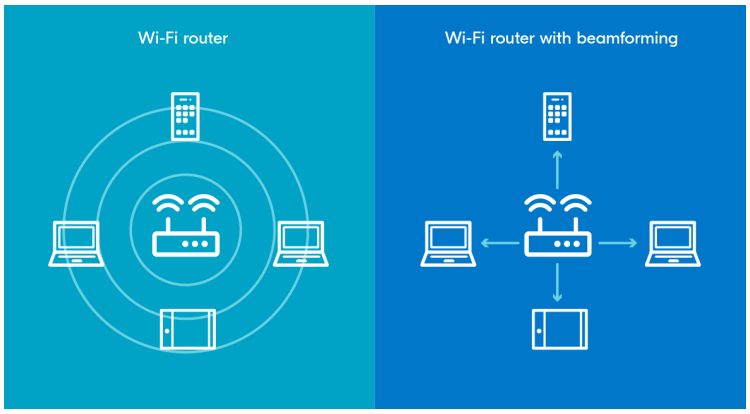


Figure 2.14: Wi-Fi router with and without beamforming [9]

2.4.3 TWT

Target Wake Time (TWT) is the most important newly introduced feature of Wi-Fi 6 for IoT and low-power applications. It allows an STA to negotiate sleep and wake-up times with the AP, permitting the device to activate its radio only when needed, thus saving on power consumption. As TWT is the Wi-Fi 6 feature covered in this thesis, it will be detailed later in the report. [9]

2.4.4 1024-QAM

Wi-Fi 6 includes 1024-QAM, a higher-order modulation scheme that increases data throughput by encoding more bits per symbol. However, the effective use of 1024-QAM requires the STA to be near the AP to maintain a strong signal, as the modulation is highly sensitive to signal quality. Additionally, this modulation is only available when the client uses an entire 20 MHz channel or wider. It does not apply to smaller RUs in OFDMA. [10]

2.4.5 WPA3 security

Wi-Fi 6 improves security through WPA3, the latest security protocol for Wi-Fi networks. WPA3 provides more robust encryption and security mechanisms than its predecessor, WPA2, particularly with features like Simultaneous Authentication of Equals (SAE), which enhances protection against brute-force attacks. WPA3-Enterprise offers 192-bit encryption. [9][17]

2.4.6 BSS Coloring

BSS coloring reduces interferences when multiple APs are situated in the range of each other,leading to spatial overlap. A unique 6-bit identifier is assigned to each BSS, referred to as a "color". It is included in the physical (PHY) header of the transmitted frames.

Without BSS coloring, when a device needs to transmit data, it employs CSMA/CA to defer its transmission if another device already uses the channel.

With BSS coloring, if a device detects a frame with the same color, CSMA/CA operates as usual, and the transmission is deferred. However, if the detected frame has a different color, indicating it belongs to an overlapping BSS, the device may proceed with its transmission if the signal strength is below a specified threshold.

This capability enables multiple BSSs to operate more efficiently on the same frequency, reducing co-channel interference and improving overall throughput. The 6-bit color field can distinguish up to 63 different BSSs. [9][22]

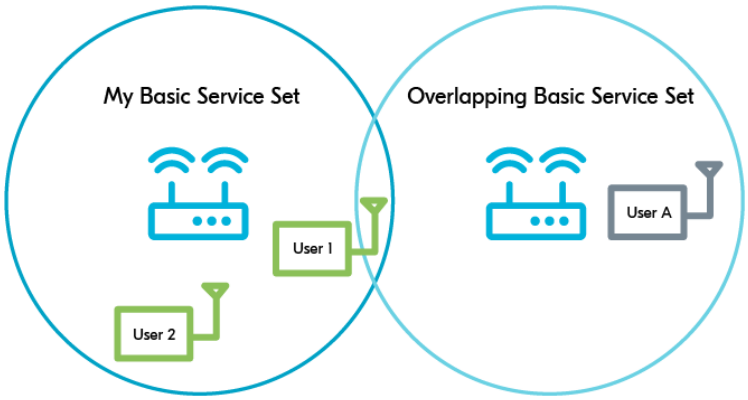


Figure 2.15: Overlapping BSS [9]

2.5 Wi-Fi Power Save Modes

Minimizing consumption on a battery-powered device is crucial to extending the battery's lifetime. The way to save power is to shut down digital logic and RF circuits when unnecessary during operation without disconnecting. In our Wi-Fi case, the focus is on turning ON and OFF the RF circuit of the Wi-Fi device.

Power saving is straightforward for a transmitting device. RF only needs to be ON when transmitting, and the device always knows when it is transmitting data. Receiving data is the challenging part of power saving with communication devices because the device does not always know when data is transmitted. As Wi-Fi uses bidirectional transmissions, this problem impacts it.

[9][23][24]

2.5.1 Power States

A Wi-Fi STA can be in active or sleep mode. In active mode, the RF circuit is ON, and the device can transmit and receive data anytime. In sleep state, the RF circuit is OFF, and the device cannot transmit or receive data. Nevertheless, the Wi-Fi state information is retained. If the device was connected before entering sleep mode, it stays connected as long as it does not need to communicate. When the device is in sleep state, it consumes very little power.

When power saving is not activated, the STA is always in active state mode. When power saving is active, the STA dynamically switches between the sleep and active states, waking up when frames are transmitted and sleeping when there is no traffic.

When using power-saving mode[1], the upper layers of the stack are not impacted. The method for receiving and transmitting frames stays the same. The only difference is the latency of the downlink traffic, which may be increased.

[23]

[1]This description refers specifically to the standard PS mechanism (DTIM) and omits TWT and Listen Interval

2.5.2 Wi-Fi Beacon frames

As seen in the Data Link section, the beacon is a frame the AP sends periodically (usually every 100ms). It announces the AP's SSID, the network-supported rates, operational channel numbers, security protocols, and other global network management information.

Beacon TIM Element

The beacon frame also includes a Traffic Indication Map (TIM) element. The TIM element is essential in Wi-Fi power-saving mode. It informs the STAs about the state of buffered data at the AP. The figure below shows the structure of a TIM element.

Figure 2.16: TIM element structure [9]

The Partial Virtual Bitmap contains the status of the buffered downlink traffic. If the AP has a unicast message for an STA, it will indicate it in the Partial Virtual Bitmap by adding the STA's AID. One bit (bit 0) of the Bitmap Control Field indicates if the AP has multicast/broadcast messages for the STAs.

Delivery Traffic Indication Message (DTIM) Beacon

A DTIM beacon is a special type of beacon frame that occurs at a specific interval, called a DTIM period. For example, if using a DTIM 1 period, every beacon frame will be a DTIM beacon. If using a DTIM 3 period, one beacon out of three will be a DTIM. The DTIM count is a real-time counter that indicates how many regular beacons (including the current one) will appear before the DTIM beacon. DTIM period and count are both part of the TIM element. The DTIM beacon announces the broadcast and multicast traffic pending transmissions. Broadcast and multicast frames are sent just after the DTIM beacon. These frames are not buffered for multiple DTIM periods and are not acknowledged, so an STA in sleep state will never receive them.

[9][11][24]

2.5.3 DTIM Power Save

The DTIM Power Save is the standard method. It uses the TIM information and DTIM beacon to manage the downlink traffic. DTIM Power Save supports two modes for retrieving the downlink traffic: Legacy and WMM.

In DTIM Power Save, the STA can wake up anytime to transmit uplink traffic. For downlink traffic, the STA wakes up at every DTIM beacon. It inspects the TIM element that informs of the incoming downlink traffic. If the AP has broadcast or multicast traffic, the STA stays awake after the beacon to receive the broadcast/multicast traffic.

The figure below shows how the STA receives DTIM beacons and broadcast/multicast traffic. In that example, the DTIM period is 3.

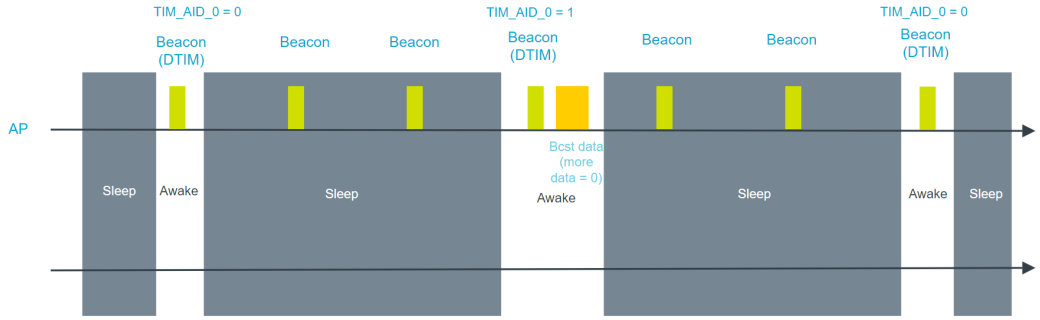


Figure 2.17: TIM beacon and broadcast/multicast traffic [23]

The TIM element also informs the STA about the unicast buffered frames at the AP. When this happens, the STA polls the AP to retrieve the downlink unicast frames. There are two modes used to retrieve the unicast frames: Legacy and Wireless Multimedia (WMM).

Legacy Power Save Mode

The legacy Power Save mode is based on the PS-Poll frame. After an STA has read its AID in the TIM element of a beacon frame, it sends a PS-Poll frame to the AP. Then, the AP responds to the PS-Poll frame with an ACK and sends its buffered frame. The STA responds to the data frame with an ACK. The data frame contains a subfield in the Frame Control Field of the MAC header called "More Data." If this subfield is set to 0, it means there are no more buffered frames at the AP. In this case, the STA goes back into the sleep state. If the "More Data" subfield is set to 1, the STA resends a PS-Poll frame to ask for the next frame.

The figure below shows how unicast downlink traffic is transmitted using Legacy Power Save mode.

Figure 2.18: Legacy Power Save mode [23]

WMM Power Save Mode

The Wireless Multimedia (WMM) Power Save mode works with a similar principle to the Legacy but saves a few frames. An STA in the sleep state starts a Service Period (SP) to retrieve the buffered unicast frames by sending a trigger frame that is a QoS Data/QoS Null frame. The AP then acknowledges this trigger frame and starts the transmission of the data frames. The QoS field in the MAC header of the data frames contains an End of Service Period (EOSP) bit. If the bit is set to 1, the STA acknowledges the frame and returns to sleep. If set to 0, the STA acknowledges the frame but stays awake to receive the next one(s). [9][23][24]

The figure below shows how unicast downlink traffic is transmitted using WMM Power Save mode.

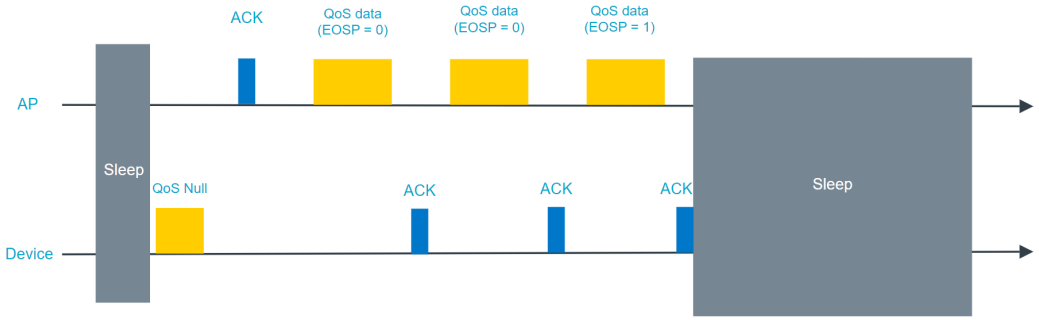


Figure 2.19: WMM Power Save mode [23]

2.5.4 Extended Sleep

The DTIM Power Save mode permits the saving of much energy by allowing the STA to sleep between the DTIM beacons. However, the STA still needs to wake up frequently, and if the device is, for example, an IoT device that requires very low traffic, it will wake up often for no reason. The Extended Sleep mode addresses this problem by adding another wake-up mode: the listen interval. It allows the STA to not wake up for all DTIM beacons. For example, if the listen interval is 10, the STA will wake up every 10 DTIM beacons. This method enables the STA to sleep longer, resulting in greater energy savings. However, the STA will miss some broadcast traffic, which could be problematic for the upper networking layers. The implementation of the complete system must take this into account and make sure that missing broadcast frames will not impact it.

The figure below shows an STA switching between DTIM and Listen Interval wake-up modes.

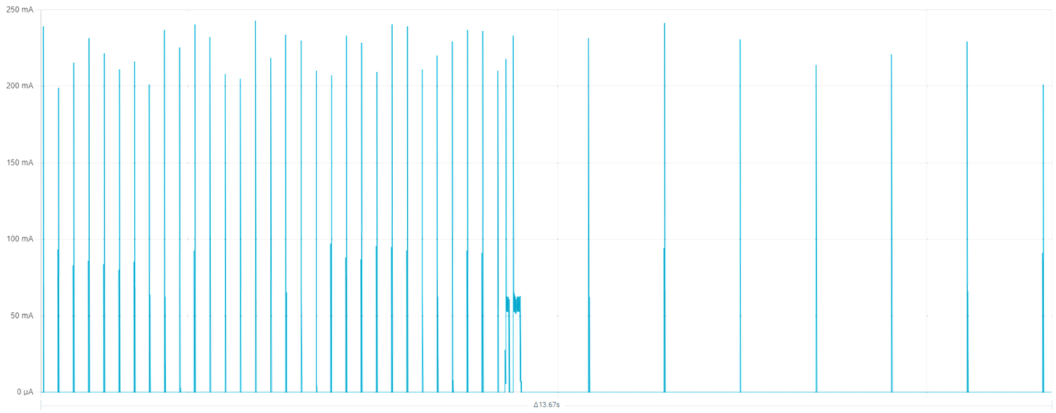


Figure 2.20: Change Power Save wake-up mode form DTIM to Listen Interval [23]

[23][9][24]

2.6 Target Wake Time

Target Wake Time (TWT) is a new power-saving mechanism introduced with Wi-Fi 6. It aims to address the problem of excessive wake-up by deterministically scheduling service periods. TWT allows the STA to sleep for long periods and to use intervals tailored to the needs of each application.

When using TWT, STAs can individually negotiate with the AP when and at which schedule frequency to wake up. Wi-Fi STAs can either negotiate an individual TWT agreement or participate in a TWT broadcast session scheduled by the AP. In addition to the power-save benefits, TWT also helps to reduce contention by waking up the STAs at different times.

The figure below shows the scheduling of two independent TWT flows.

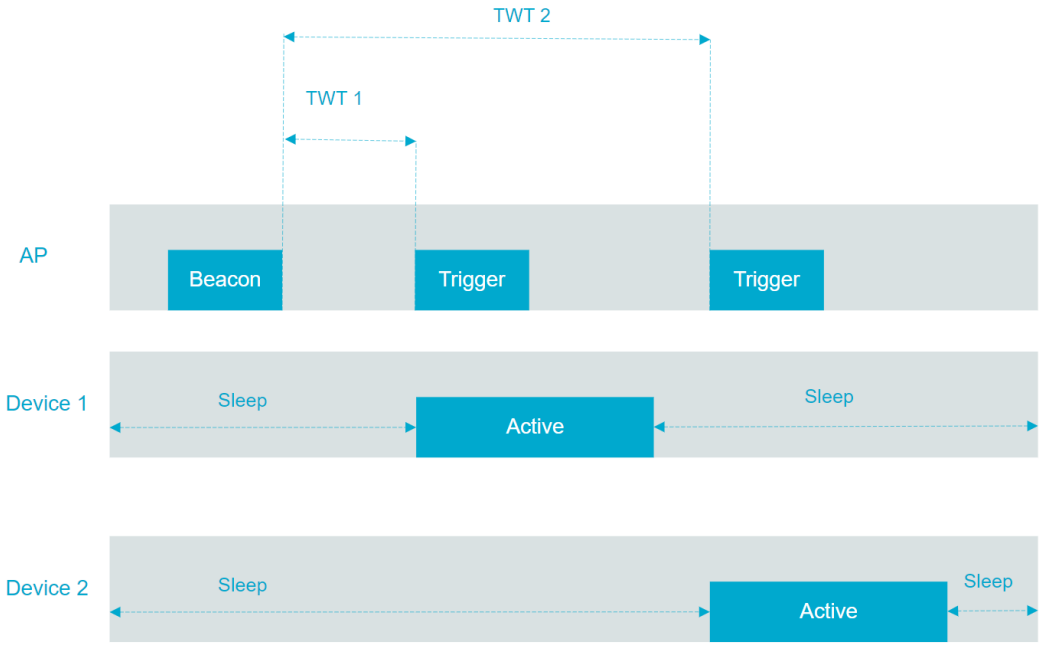


Figure 2.21: TWT scheduling [23]

With TWT, the device can theoretically be in sleep mode for hours. In practice, sleeping for hours causes problems on the upper layers of the stack, so the typical TWT interval is around 10 seconds, which adds significant power-saving improvements compared to PS mode.

TWT is an excellent feature for saving power. However, when using TWT, the STA does not wake up for the DTIM beacon. Thus, TWT cannot be used if the application relies on timely broadcast/multicast traffic reception. Moreover, sleeping for long periods adds significant latency in the system, which could cause problems in the upper layers.

A TWT session is characterized by the TWT interval, the total period (active + sleeping time), and the TWT Wake interval or service period (active time). The figure below shows the scheduling of TWT.

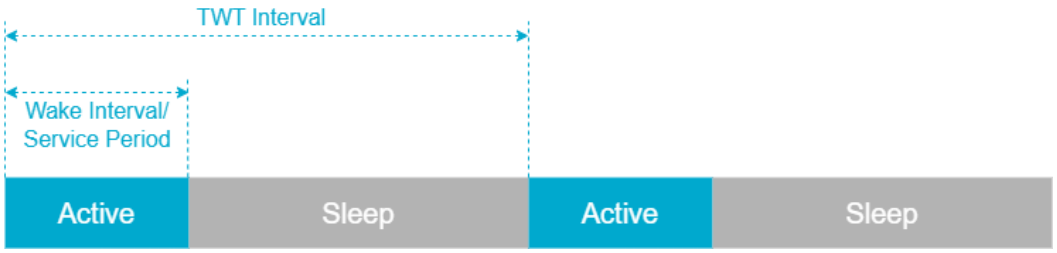


Figure 2.22: TWT Intervals

When using TWT, the STA needs first to connect to the AP. Then, it has to negotiate a TWT agreement with the AP. For this, it makes a TWT setup request. In this request, the following parameters are set:

- **Negotiation Type:** Individual or Broadcast
- **Flow ID:** This parameter maps the setup with the teardown.
- **Dialog Token:** This parameter maps the requests with the responses.
- **Trigger:** When enabled, the AP sends a trigger frame when the TWT service period starts.
- **Implicit:** When using implicit TWT, the STA determines when to wake up based on the TWT interval. With explicit TWT, the AP informs at each service period when the next one will be.
- **Announce:** When using announced TWT, the STA sends a PS-Poll at the beginning of every service period to inform the AP it is ready.
- **TWT Wake Interval:** This setting is a parameter of the wifi_twt_setup function. The TWT wake interval is the duration of the service period.
- **TWT Interval:** This setting is a parameter of the wifi_twt_setup function. The TWT interval is the total period of TWT (session duration + sleeping duration).

The AP replies to the TWT request with a response that contains the TWT negotiated parameters. The AP can modify the TWT parameters requested. Thus, the STA has to configure itself with the negotiated parameters sent in the response. After the negotiation, both AP and STA assume that TWT is in place.

To stop the TWT operation, the STA can send a teardown request. When the AP receives the teardown, it replies and returns to normal operation mode.

[9][23]

2.7 Network Layer

The network layer sits on top of the link layer. It is the layer 3 of the OSI model. The network layer is responsible for transmitting packets across many link layers.

2.7.1 Internet Protocol (IP)

The Internet Protocol (IP) is the most commonly used protocol to transfer data on top of 802.11. It addresses and routes packets between devices across different networks.

- **IP Addressing:** Every network interface of an STA or AP is assigned a unique IP address, which identifies it within the network. When a device needs to send a message to another device over the Internet, it uses this IP address to specify its destination. The MAC address, which operates at Layer 2, is only relevant within the local network segment.

- **Encapsulation:** When data is sent over an 802.11 network, the IP packet is encapsulated within the MAC frame. The IP header (part of the IP packet) contains the source and destination IP addresses, while the MAC header contains the corresponding MAC addresses.

- **Routing:** When an STA sends an IP packet, it is first directed to the AP, which routes it to other networks (e.g., the Internet). Layer 2 represents the communication between two directly connected devices (e.g., STA and AP), while Layer 3 represents the communication between the end devices (e.g., STA and Cloud Server). The AP (also acting as a router) uses the IP information to route packets correctly. If a packet is meant for a device on another network, the access point forwards it to the appropriate router for further transmission.

IP packets consist of a header and a payload data field. The header includes all information related to the IP protocol, such as source and destination addresses, while the data field contains the upper-layer packet, such as UDP or TCP.

Two versions of the IP protocol are used on the Internet: IPv4 and IPv6.

[25]

IPv4

IPv4 was the first IP protocol to be widely deployed. It utilizes 32-bit addresses, allowing for approximately 4.3 billion unique addresses. IPv4 addresses are represented in dotted decimal notation (e.g., 192.168.1.1) and consist of four 8-bit octets. Internet communications have used IPv4 for decades. However, its limited number of different addresses has required the development of various solutions, such as Network Address Translation (NAT) and subnetting, to maximize address usage.

IPv4 uses unicast traffic by default but also supports multicast and broadcast communication within local networks.

Dynamic Host Configuration Protocol v4 (DHCPv4) is the standard method for dynamically and automatically assigning IPv4 addresses to devices on a network. When a device connects to the network, it sends a DHCPv4 request to locate a DHCPv4 server. The DHCPv4 server assigns the device an available IPv4 address and provides other configuration settings, such as DNS server addresses and default gateway information. This process allows devices to join a network automatically.

IPv4 addresses are classified as either public or private. Public IPv4 addresses are globally unique addresses that identify devices on the Internet. These addresses are routable over the Internet, meaning any device with a public IPv4 address can communicate with other public IPv4 addresses across networks. Private IPv4 addresses are reserved for use inside private networks and are not routable over the Internet. These addresses help to preserve the IPv4 address space and are commonly used in home and business networks, where devices are only accessible within the local network.

Private IPv4 addresses include:

- 10.0.0.0 to 10.255.255.255
- 172.16.0.0 to 172.31.255.255
- 192.168.0.0 to 192.168.255.255

Routers use NAT to allow the devices on the private network to access the Internet. NAT translates private IPs to public IPs as they pass between the local network and the Internet.

[25][26][27]

IPv4 packet

The IPv4 packet header is 20 bytes long without options and can be extended to 60 bytes when options are used.

The first field of the IP packet is the version field, 4 for IPv4. Then, IHL is the header length. The DS field is used for QoS and helps classify packets based on their type of service. The total length of the packet is also included in the header. Identification and Fragment Offset fields work together to facilitate fragmentation and reassembly of IP packets. The TTL field prevents packets from looping indefinitely in the network in case of routing issues. When the packet is sent, it is set to an initial value (typically 64 or 128), and each router that forwards the packet decrements the TTL by 1. If it reaches 0, the packet is discarded. The protocol field specifies the type of upper-layer packet carried by the IP packet (e.g., TCP, UDP). The last field is the header checksum. It permits checking the integrity of the packet. The source and destination addresses are the most important fields of the IPv4 header. IPv4 also allows options fields in its header. Option examples are Record Routes or Time stamps used for traceroute. [25]

The figure below shows the structure of an IPv4 packet.

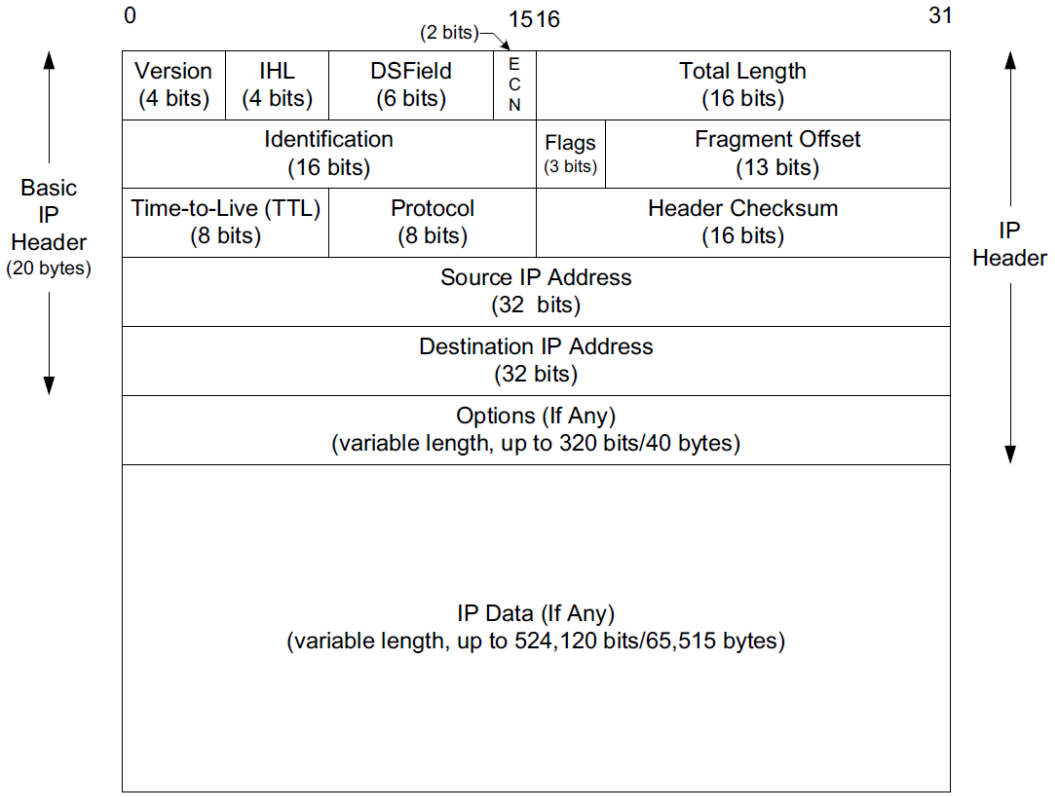


Figure 2.23: IPv4 packet structure [25]

IPv6

IPv6 was developed to handle the limitations of IPv4, particularly the shortage of IP addresses. IPv6 uses a 128-bit address space, allowing for 3.4e+38 unique addresses. This expansion accommodates the growing number of Internet-connected devices and simplifies address management and routing. IPv6 addresses are written in hexadecimal and separated by colons (e.g., 2001:0000:130F:0000:0000:09C0:876A:130B). Features such as auto-configuration (i.e., SLAAC), built-in security, and improved multicast support are directly included in IPv6, unlike in IPv4, where these features are implemented by other protocols (e.g., DHCP). IPv6 is increasingly being adopted as the standard, although IPv4 and IPv6 often coexist within networks, with many devices supporting both protocols to ensure compatibility during the transition.

DHCPv6 allows for automatically and dynamically assigning IPv6 addresses to devices on a network. It works like DHCP in IPv4. When a device connects, it requests a server to obtain its address.

Stateless Address Autoconfiguration (SLAAC) is a feature of IPv6 that allows devices to self-configure their IP addresses without needing a DHCP server. When a device connects to a network, it uses information from IPv6 Router Advertisements (RA) to create its own address by combining a prefix provided by the router with its unique interface identifier. SLAAC simplifies the configuration process in networks with basic connectivity needs, though DHCPv6 is often used in parallel to provide additional network details.

IPv6 addresses are categorized as either global or local. Global IPv6 addresses are globally routable and can be accessed over the Internet, much like public IPv4 addresses. They typically begin with the prefix 2000::/3 (e.g., 2001:db8::1) and are unique across the Internet, enabling direct communication between IPv6 devices without requiring NAT.

Local IPv6 addresses include Link-Local and Unique Local Addresses (ULA):

- **Link-Local Addresses:** These addresses are automatically assigned and only valid within a single network link. They always begin with fe80::/10 and are used for device-to-device communication on the same local network. Link-local addresses are essential for SLAAC and device discovery functions but are not routable beyond the local segment.

- **Unique Local Addresses (ULA):** These addresses are intended for private communications within an organization or between devices in a specific region. ULA addresses begin with fc00::/7 and are analogous to private IPv4 addresses. Although not routable over the Internet, they allow private networks to communicate internally without conflicting with global IPv6 addresses.

[25][28]

IPv6 packet

The IPv6 header is much simpler than the IPv4 one. It does not support options and has a fixed size of 40 bytes.

Like with IPv4, the IPv6 header has the same Version and DS fields. Packets that require special handling by routers use the Flow Label field (e.g., providing a particular QoS). It is helpful for applications that require real-time data transmission (e.g., streaming). Then, a field indicates the payload length. The Next Header field indicates the header type in the payload (e.g., UDP, TCP). The Hop Limit field is similar to the TTL in IPv4. It limits the lifespan of a packet to prevent it from circulating indefinitely in the network. Finally, the source and destination addresses are the last fields of the IPv6 header.

The figure below shows the structure of an IPv6 header.

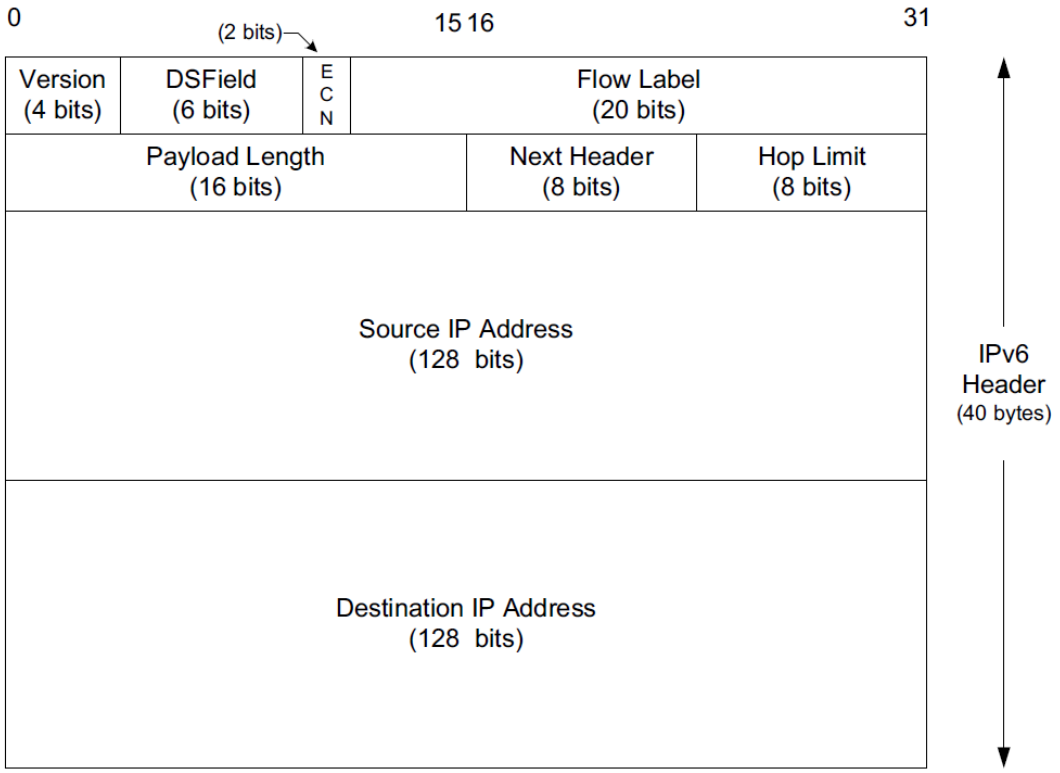


Figure 2.24: IPv6 packet structure [25]

2.7.2 Address Resolution Protocol (ARP)

ARP is the protocol that links IP and MAC addresses in IPv4 networks. When a device communicates with another, and the two devices are not on the same local network (e.g., via the Internet), the devices are identified by their IP address. However, the 802.11 frames are sent to a MAC address on the link layer. Thus, a router must know which IP address corresponds to which MAC address.

The devices use ARP to request the MAC address of the other devices on the network. An ARP request is a broadcast frame asking, "Who has this IP address?" All devices on the network see the request, but only the device with the requested IP responds with an ARP reply containing its MAC address. Then, data can be explicitly directed to the device's MAC address using unicast.

A router (wireless AP or wired) has an ARP cache containing the IP addresses and corresponding MAC addresses of all the STAs connected to its local network. Each STA in the cache has a timeout. When it expires, a new ARP request is broadcast to retrieve the STA's MAC address. The timeout duration is not standard and can vary between minutes and half an hour, depending on the AP's manufacturer.

An STAs use an ARP cache, which implements the same principle but in a more limited manner. After receiving an ARP reply, an STA stores the MAC address and corresponding IP in its ARP cache to reuse them without redoing an ARP request. The ARP cache is deleted after a small timeout and can only hold a limited number of addresses.

[25][26][29]

2.7.3 Neighbor Discovery Protocol (NDP)

ARP is replaced by the Neighbor Discovery Protocol (NDP) in IPv6. NDP is responsible for managing device interactions on the same local network link. NDP implements address resolution, router discovery, and duplicate address detection through ICMPv6 message types.

Key Functions of NDP:

- **Router and Prefix Discovery:** The devices of an IPv6 network use NDP to discover routers on the local network. Routers periodically send Router Advertisement (RA) messages to announce themselves and their configuration parameters. The devices can also send a Router Solicitation (RS) message to ask routers to respond immediately with an RA. The RA messages also contain the network prefix, which permits the devices to generate their IP addresses using SLAAC.

- **Address Resolution:** ARP protocol is replaced by NDP's Neighbor Solicitation (NS) and Neighbor Advertisement (NA) messages. When a device needs to find the MAC address associated with an IPv6 address, it sends an NS message. The device with the target IPv6 address responds with an NA message containing its MAC address, allowing direct communication between devices.

- **Duplicate Address Detection (DAD):** As with SLAAC, devices generate their IP addresses. Although unlikely, it is still possible that two devices generate the same address. To ensure that every device on the network has a unique IPv6 address, NDP performs a DAD. When a device generates a new IP, it sends an NS message to check if another device on the network is already using it. If no NA is received, the address is assigned to the device.

- **Redirect Function:** Routers can use the Redirect function to inform devices of a better route for reaching a particular destination. Suppose a router detects a more efficient path to the destination for a packet. In that case, it can send a Redirect message to the device, suggesting an alternative route to improve packet delivery within the network.

NDP is essential in IPv6 because it groups many functions that require different protocols in IPv4. Self-configuration and dynamic network discovery enhance the flexibility and scalability of IPv6 networks. Additionally, NDP can be secured by implementing the Secure Neighbor Discovery (SEND) extension.

[25][28][30]

2.8 Transport Layer

The transport layer sits above the network layer. IP communications have two primary transport layer protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These protocols enable messages to be directed to specific programs running on a machine. The IP address identifies the machine within the network, while the UDP or TCP port identifies the specific program or service using that port. The transport layer ensures proper data transmission between devices and applications by providing essential services such as error detection, data segmentation, and flow control.

2.8.1 User Datagram Protocol

The User Datagram Protocol (UDP) is a simple, datagram-oriented transport-layer protocol. It provides minimal functionality so applications can control how packets are sent and controlled themselves. UDP does not establish a connection before data transfer and does not ensure reliable delivery. It sends packets, known as datagrams, directly to the destination without waiting for acknowledgment and managing retransmissions.

The simplicity of UDP makes it very lightweight and fast, with low overheads. However, this comes at the cost of reliability. If datagrams are lost or truncated, the protocol can provide error detection, as it includes a checksum but does not implement any error correction. It is the responsibility of the application using UDP to handle the error corrections if needed.

The figure below shows the structure of a UDP packet.

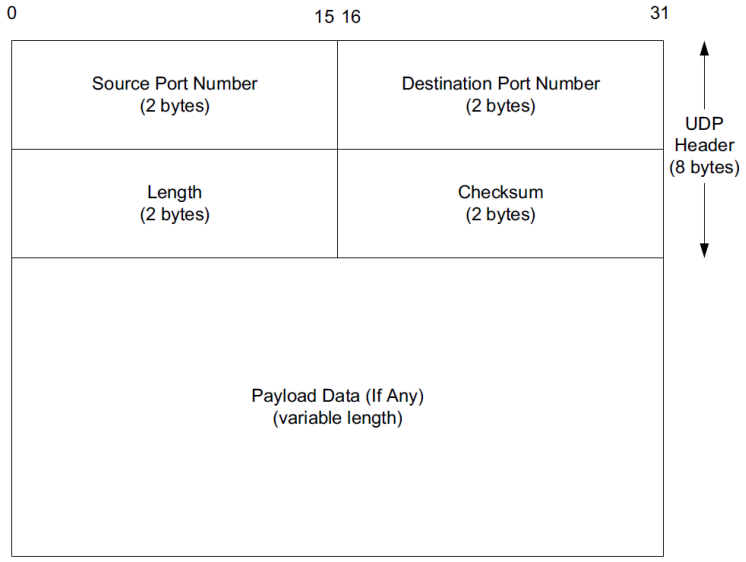


Figure 2.25: UDP packet structure [25]

The UDP header is 8 bytes long and contains only four fields. The two first fields are the source and destination ports. The source port is optional and may be set to 0 if the datagram never requires a response. The length field is the total length of the datagram in bytes, including header and payload data. The checksum allows for error detection. [25][31]

Security in UDP

Datagram Transport Layer Security (DTLS) is the protocol used to secure connections over UDP. It is derived from TLS and adapted to handle packet loss and reordering, making it suitable for unreliable communications.

The DTLS handshake occurs at the beginning of a session to establish a secure connection. It includes key exchanges, peer verification, and cipher suite negotiation. The handshake adds overhead, so mechanisms like Connection ID aim to avoid repeating it unnecessarily.

Peer verification ensures both parties are authenticated, protecting against man-in-the-middle attacks. DTLS supports multiple authentication methods:

- Certificates: X.509 certificates verify identities.
- Pre-Shared Keys (PSKs): Lightweight and suitable for constrained devices.
- Raw Public Keys: A simple alternative to certificates.

Connection ID (CID) is a feature introduced in DTLS 1.3 but is also present in some DTLS 1.2 implementations as an extension. It maintains session continuity and avoids the need to redo the handshake when IP changes occur. It identifies sessions using a unique ID instead of relying on the IP/Port tuple. This feature is particularly beneficial for IoT and mobile scenarios. The figure below shows how CID works.

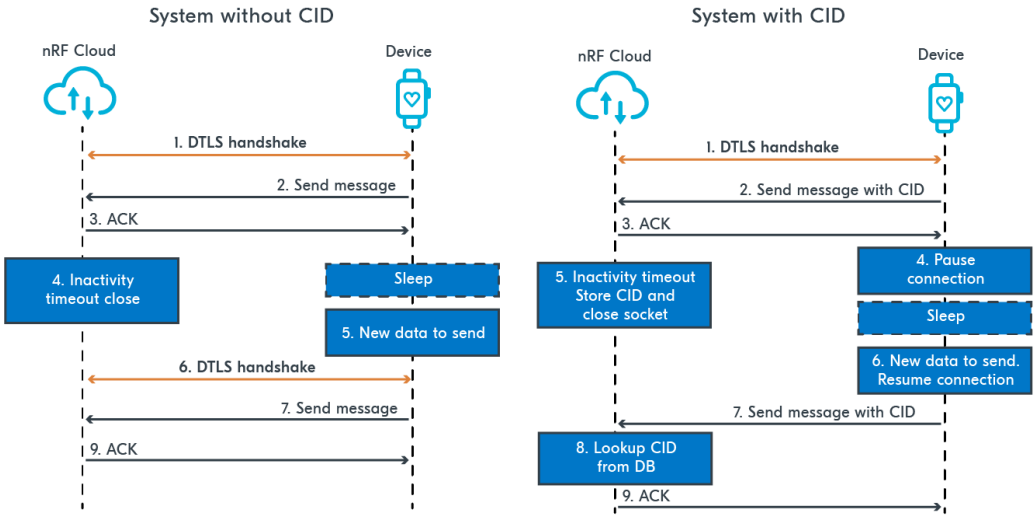


Figure 2.26: DTLS Connection ID [32]

Cipher suites include multiple ciphering algorithms used to provide confidentiality, authentication, and data integrity. Different algorithms provide different levels of security. ECDHE algorithms offer forward secrecy by generating unique keys for each session. This means that even if a private key is compromised, the past sessions remain secure. [25][32][33]

2.8.2 Transmission Control Protocol

The Transmission Control Protocol (TCP) provides connection-oriented and reliable communication. Unlike UDP, which only detects errors, TCP corrects them and ensures that data arrives accurately and in the correct order. As discussed in the link layer section, reliability is partially implemented at the link layer, where packets are retransmitted if they are not acknowledged. While most errors are corrected at this level, more is needed to guarantee reliable communication. Packets may still be dropped at the link layer if the maximum number of retries is reached or at intermediate routers in a multi-hop network, such as the Internet. TCP provides an end-to-end error correction, ensuring reliable communication across the entire data path.

TCP is connection-oriented, which means a connection is established between communicating devices before any data is exchanged. The connection ensures that the sender and the receiver are synchronized and agree on the communication parameters. Once the connection is established, both devices can exchange data. TCP monitors the data flow using ACK to ensure the packets are correctly received and sequence numbers to ensure the packets arrive in the correct order. If any problem is detected, TCP can request a retransmission. TCP also implements flow and congestion control mechanisms that adjust the rate of data transmission in order to react to the network conditions and to prevent overwhelming the receiver.

While TCP ensures reliability and robustness, it has increased overhead due to the need for connection establishment, maintenance, and termination. As a result, it is generally slower than connectionless protocols like UDP, which prioritize speed and efficiency over reliability.

[25][34]

2.9 Application Layer Protocols

The application layer is the top layer in the OSI and TCP/IP networking models. It defines the rules and conventions for exchanging data between systems. Application layer protocols facilitate various services, including file transfers, email communication, web browsing, and real-time data exchange.

This section describes some of the most important application layer protocols commonly used in the IoT context.

2.9.1 HTTP

The Hypertext Transfer Protocol (HTTP) is a TCP-based protocol that uses a client-server architecture. It employs a request-response model, where the client sends requests, and the server provides corresponding responses.

HTTP is predominantly utilized on the World Wide Web, where the client, typically a web browser, initiates an HTTP request to a web server. The server then replies with an HTTP response, which may include content such as a webpage formatted in HTML.

HTTP messages contain a start-line, an optional header section, and an optional body. In an HTTP request, the start-line specifies the request method (e.g., GET, POST, PUT, DELETE) and the target resource, identified by a Uniform Resource Identifier (URI), to which the method is applied. In an HTTP response, the start-line includes a status code indicating the result of the request (e.g., 200 for success, 404 for not found, 403 for forbidden).

[9][35]

2.9.2 MQTT

The Message Queuing Telemetry Transport (MQTT) is a TCP-based protocol widely utilized in IoT applications. It operates using a publish-subscribe communication model and involves two primary entities: a message broker and clients. Information is categorized into topics, to which clients can either publish data or subscribe to receive updates. The figure below illustrates this principle.

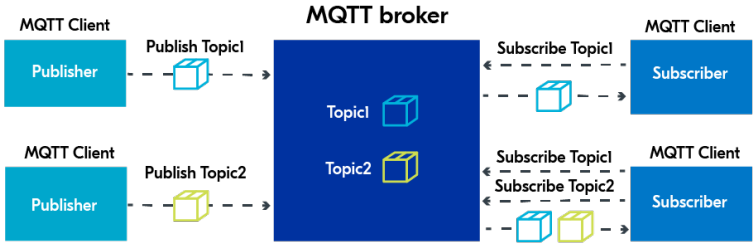


Figure 2.27: MQTT publish-subscribe model [36]

[36][37]

2.9.3 CoAP

The Constrained Application Protocol (CoAP) is similar to HTTP but is UDP-based and specifically designed for devices with limited memory, battery power, and bandwidth. Like HTTP, CoAP employs a client-server architecture and a request-response model. It operates on resources identified by URIs, and requests use methods such as GET, PUT, POST, and DELETE.

The use of UDP allows for lower overhead, eliminating TCP's built-in reliability, flow control, and congestion control mechanisms, which can be problematic in constrained networks. Instead, reliability in CoAP is handled at the application layer and can be selectively applied to specific requests. Furthermore, unlike TCP, CoAP provides configurable retransmission parameters, such as timeouts and backoff intervals, offering greater adaptability to diverse network conditions.

The figure below illustrates the structure of a CoAP packet.

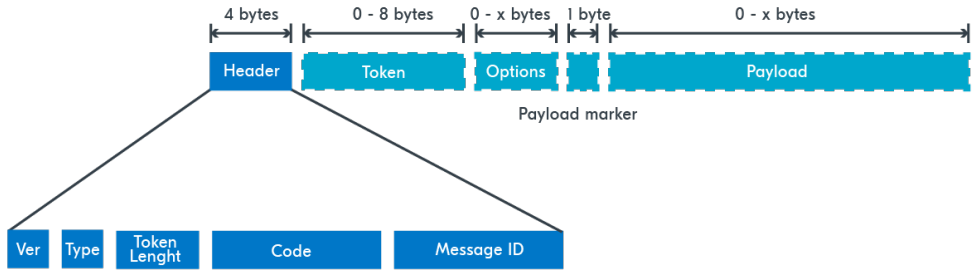


Figure 2.28: CoAP packet structure [36]

The following fields are noteworthy:

- **Type:** A subfield in the header that indicates whether the request is confirmable or non-confirmable. For confirmable messages, the client will retransmit the packet if it does not receive a response.
- **Code:** A subfield in the header that specifies the method (e.g., GET, PUT) in a request or the response code in a reply (e.g., 2.xx for success, 4.xx for client errors, 5.xx for server errors).
- **Token:** A random sequence of 0 to 8 bytes used to correlate requests and responses.
- **Options:** Various options can be added to a CoAP packet. The URI is included as an option. While typically required in CoAP requests, the URI is optional because some scenarios (e.g., retrieving all available resources on the server) do not involve a specific resource.
- **Payload:** The optional field that contains the actual payload data.

[36][38]

3 Testbed Design

This chapter introduces the test environment and methodology used to evaluate the impact of TWT and PS mechanisms on the wireless applications running on a Wi-Fi STA. It details the testbed application design, private and public servers, and the design of the tests. Each use case is examined with specific configurations to assess performance metrics such as packet loss, latency, and system behavior under different conditions.

This chapter focuses on the overall setup and design. The implementation of the testbed, including code description, is described in Chapters 4 and 5.

Contents

3.1 Testbed 46
3.1.1 Protocol Stack 46
3.2 Server 47
3.2.1 Private Server 47
3.2.2 Public Server 47
3.3 Tests 48
3.3.1 Sensor Use Case 48
3.3.2 Large Packet Test Case 49
3.3.3 Multi Packet Use Case 49
3.3.4 Actuator Use Case 51
3.4 Monitoring 54
3.4.1 Power Profiling 54
3.4.2 Network Monitoring 54
3.4.3 Test Monitoring 54
3.5 Network Topology 55

3.1 Testbed

The testbed is implemented on a Nordic nRF7002-DK [39]. This development kit integrates the nRF7002 Wi-Fi IC, and its host SoC is the nRF5340. The nRF5340 is a high-performance SoC from Nordic Semiconductor. It integrates two Arm Cortex-M33 processors, Bluetooth, high-speed SPI/QSPI, and other features. The application processor can be clocked at 64 or 128 MHz. It has 1 Mbyte Flash and 512 Kbyte RAM.

The testbed application is implemented on the application processor of the nRF5340 and uses the Zephyr RTOS [40], a compact real-time operating system designed for devices with limited resources. Nordic's SDK comes with Zephyr integrated by default.

The testbed application implements a complete wireless communication stack that could be used in a commercial application using TWT. The tests use this stack to measure the communication's performance in terms of lost packets, latency, and power consumption.

The detailed implementation of the application is described in the Testbed Client Implementation chapter (4).

3.1.1 Protocol Stack

The figure below shows the protocol stack implemented by the testbed application.

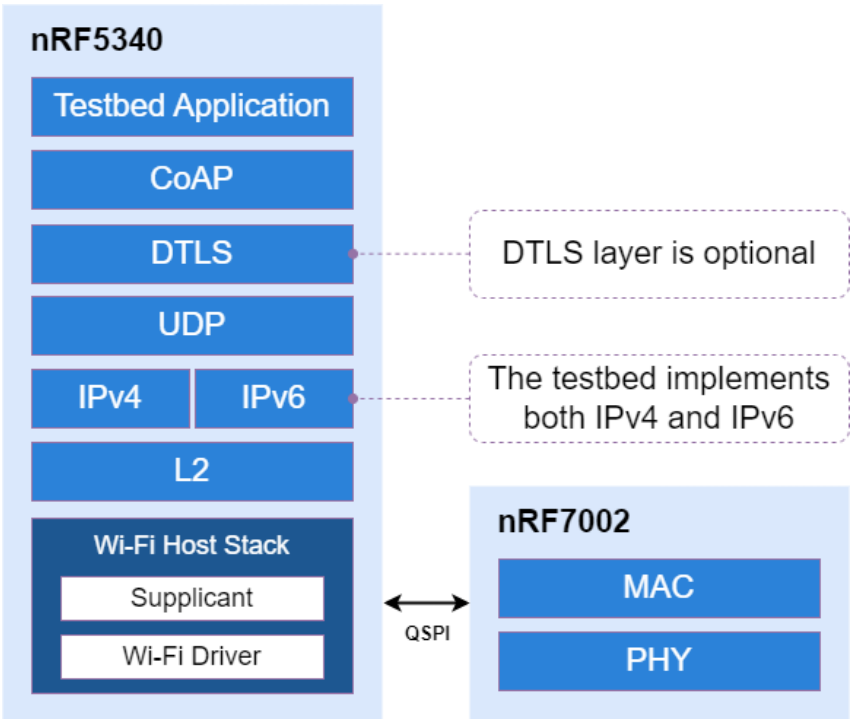


Figure 3.1: Testbed Protocol Stack

The testbed application exchanges packets with a server using the CoAP protocol. CoAP uses UDP as the transport protocol and can be secured with DTLS. The testbed can use both IPv4 and IPv6.

3.2 Server

The testbed can be used with two servers: a private server designed for the testbed and a public server. The private server implements more advanced tests and gives more information. The public server is hosted on the Internet, allowing the tests to be performed in a scenario that resembles a commercial application.

3.2.1 Private Server

The private server is developed using Java and the Californium library [41]. For the tests, the server is hosted on a Linux machine. The private server implements resources that are used for the tests. It also computes statistics and provides resources to retrieve these statistics.

The server is accessible on ports 5683 for CoAP/UDP and 5684 for CoAP/DTLS.

The source code and a pre-built JAR executable are available in the TWT-Testbed-Server repository, as referenced in Appendix A. The DTLS credentials and instructions on how to build and run the server are provided in the README file.

3.2.2 Public Server

The testbed can also be utilized with the Eclipse Californium server [41], a CoAP sandbox server available on the Internet for testing purposes. It is accessible at *californium.eclipseprojects.io* on port 5683 for CoAP/UDP and port 5684 for CoAP/DTLS. The server is accessible using both IPv4 and IPv6.

The DTLS credentials are provided in the README of the ecliplse-californium Github repository [42].

3.3 Tests

The tests implement the four use cases using TWT or PS on the private or a public server. Thus, there are four tests for each use case, each with multiple configurations and options.

In all TWT tests, one first request is exchanged with the server before setting up TWT to ensure everything is configured correctly.

3.3.1 Sensor Use Case

The sensor use case sends a CoAP request at every TWT session and typically expects a response in the next TWT session. The figure below shows the expected behavior of this test. The request is sent in the TWT session, and the response is buffered and transmitted within the next one.

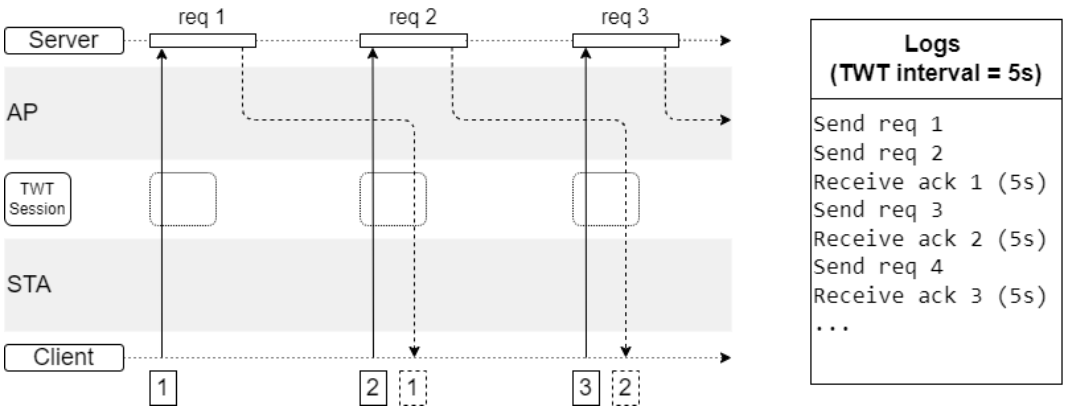


Figure 3.2: Sensor TWT Test

The PS version of the test also periodically sends a CoAP request. However, the response is expected before the subsequent request is sent. The requests are sent as non-confirmable requests, and no retransmissions are used because the goal of the test is to measure the proportion of requests successfully transmitted and responded to.

The test behavior is the same on the private and public servers. The difference is that on the private server, the requests are counted on the server side, and the count is transmitted to the client at the end of the test. This statistic is used to calculate the number of uplink and downlink packets lost. When using the public server, only the unresponded requests are counted without knowing if it is the request or the response that failed. In addition to tracking the number of lost packets, the test also monitors the time between a request and its corresponding response.

The sensor TWT test also implements a recovery mechanism. When activated, it detects if the number of pending requests exceed a configured threshold. If this is the case, TWT is teardown until all pending responses are received or a timeout of 2 seconds occurs. Then TWT is set up again. The minimum value for the threshold is 2.

3.3.2 Large Packet Test Case

The large packet test case is a variant of the sensor use case that allows the modification of the payload size. The payload size can be configured up to a maximum of 1 KB, resulting in a MAC frame of approximately 1200 bytes.

The public server version of the test sends a request with the configured size. However, the server responds with an ACK. The private server version allows control over the response size. The test can be configured to send a large request and receive a small response, send a small request and receive a large response, or send a large request and receive a large response.

The 1 KB limitation is specified in the CoAP standard [38] to avoid IP fragmentation. To send larger payloads, CoAP implements blockwise transfer, which splits the payload and sends it into multiple requests. However, this approach is incompatible with TWT, as a response is required for each block before sending the next one. Using blockwise transfer in a TWT scenario would require the teardown and re-setup of TWT. The figure below shows a CoAP blockwise transfer.

```
CON, MID:40925, GET, TKN:5c bf 6b 31 1a b1 d1 73, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40925, Empty Message
CON, MID:10556, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #0, coap://californium.eclipseprojects.io/large-separate
ACK, MID:10556, Empty Message
CON, MID:40926, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #1, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40926, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #1, coap://californium.eclipseprojects.io/large-separate
CON, MID:40927, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #2, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40927, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #2, coap://californium.eclipseprojects.io/large-separate
CON, MID:40928, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #3, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40928, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #3, coap://californium.eclipseprojects.io/large-separate
CON, MID:40929, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #4, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40929, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #4, coap://californium.eclipseprojects.io/large-separate
CON, MID:40930, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #5, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40930, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #5, coap://californium.eclipseprojects.io/large-separate
CON, MID:40931, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #6, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40931, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #6, coap://californium.eclipseprojects.io/large-separate
CON, MID:40932, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #7, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40932, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #7, coap://californium.eclipseprojects.io/large-separate
CON, MID:40933, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #8, coap://californium.eclipseprojects.io/large-separate
ACK, MID:40933, 2.05 Content, TKN:5c bf 6b 31 1a b1 d1 73, Block #8, coap://californium.eclipseprojects.io/large-separate
CON, MID:40934, GET, TKN:5c bf 6b 31 1a b1 d1 73, Block #9, coap://californium.eclipseprojects.io/large-separate
```

Figure 3.3: CoAP Blockwise Transfer

3.3.3 Multi Packet Use Case

This use case aims to analyze the behavior of communication when multiple packets are sent concurrently. In each iteration, the client sends a predefined number of CoAP PUT requests and then waits for the responses. A response is expected for each request. Once all responses are received—or a timeout occurs if some responses are lost—the test proceeds to the next iteration. As in the previous use cases, the number of lost packets and the latency are monitored. Additionally, the private version allows server statistics to be retrieved.

Unlike the previous use cases, which focus on analyzing communication performance regarding lost requests and latency, this test examines how the STA and AP handle multiple packets. It is intended to be performed while monitoring the Wi-Fi communication with Wireshark.

No flow or congestion control is implemented, as the objective is to analyze the handling of numerous concurrent requests and not to try to reach the medium's limits. Therefore, it is essential to note that packets may be lost if too many requests are sent concurrently. The following limitations may impact this test:

- **STA buffer:** The STA buffers the packets to send during the TWT session. If there are too many packets, some will be lost. This limitation is significant with TWT because packets are not sent immediately and are buffered for an extended period. To ensure packets are not lost at this stage, it is essential to monitor the Wi-Fi communication.
- **Server capacity:** Sending many concurrent requests risks overwhelming the server. To ensure packets are not lost due to server overload, the PS version of the test can be run to determine the maximum number of concurrent requests the server can handle. A better approach would be to monitor the network on the server side, but this is only possible with the private server.
- **AP buffer:** The AP buffers the responses to send during the TWT session. When the AP needs to send many packets, sending them may require multiple TWT sessions. Packets may also be dropped if the buffer is full. However, this is unlikely to happen in this use case because APs typically have large buffers, and the STA buffer limitation should be more restrictive.

These limitations should not be exceeded when utilizing fifteen concurrent requests on the private server and ten concurrent requests on the Californium public server. Nevertheless, exceeding these thresholds may provide valuable insights for analysis purposes.

The figure below shows the expected behavior of this test.

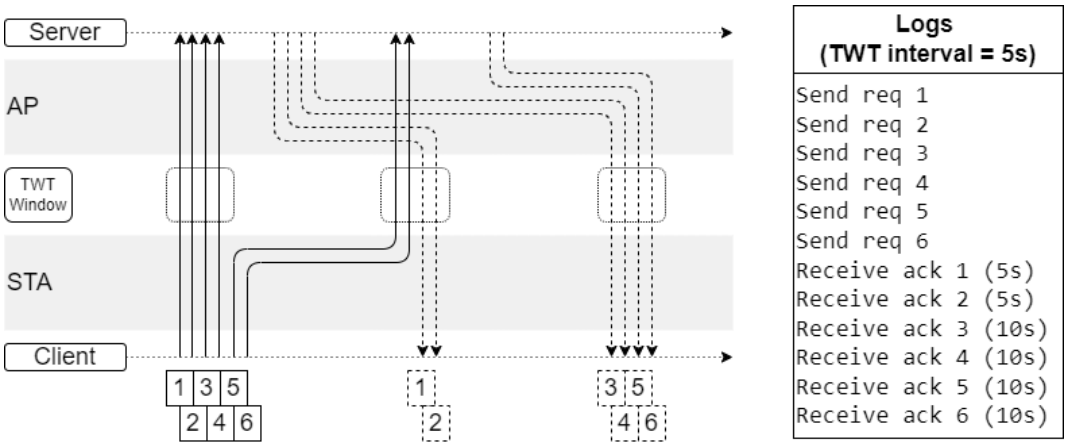


Figure 3.4: Multi Packet Test

3.3.4 Actuator Use Case

The actuator use case evaluates the communication behavior when the server initiates traffic exchanges. This is achieved using the CoAP observe mechanism. With this mechanism, the client sends a GET request containing an option to start observation. The server can then send responses to this request, which constitute the notifications. If the responses are confirmable CoAP messages, the client must acknowledge them. To terminate the observation, the client sends another GET request with an option to stop the observation. All messages involved—the initial GET request to start the observation, the responses, the response acknowledgments (if applicable), and the GET request to stop the observation—share the same token.

Unlike previous tests, the actuator tests are time-based rather than iteration-based. This means that the test finishes after a predefined time. With an iteration-based test, the client could not detect the end of the test, as the test messages could be lost. Consequently, minor inconsistencies may arise in the results. For instance, if the test ends while a message is in transmission, it will be recorded as sent but not received, even though it could have been received.

Public Server Tests

The public server version of the test uses the "obs" resource provided by the Californium server. This resource is observable and sends a response every five seconds. Since these notifications are confirmable CoAP messages, the client sends an ACK. If the server does not receive the ACK, it refrains from transmitting the following message and instead retransmits the unacknowledged message. When using TWT, if the session duration is short, the ACK will likely be sent during the next TWT session. Therefore, the TWT interval must be smaller than 2.5 seconds for the results to be consistent when doing the actuator test on the Californium public server (or use the emergency feature - see Section 3.3.4). This limitation means that the public server cannot be used for advanced testing. The test measures the number of notifications received during the test duration. With the public server, the latency of the ACK cannot be monitored.

The figure below illustrates the expected behavior of the actuator test using the public server version.

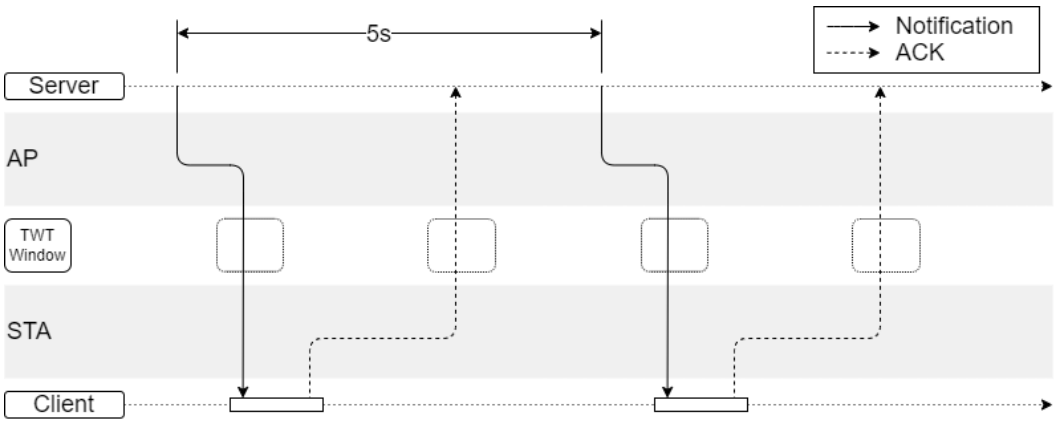


Figure 3.5: Actuator Public Server Test

Private Server Tests

The private server version of the actuator test uses random intervals between observation responses, with configurable minimum and maximum intervals. The server uses a uniform random distribution between the minimum and maximum. Unlike the public server test, the private server uses non-confirmable CoAP messages to avoid inconsistencies caused by lost packets and retransmissions. The test counts the number of responses received. At the end of the test, the number of responses sent is transmitted to the client, enabling a comparison between sent and received responses.

The figure below shows the expected behavior of the actuator test.

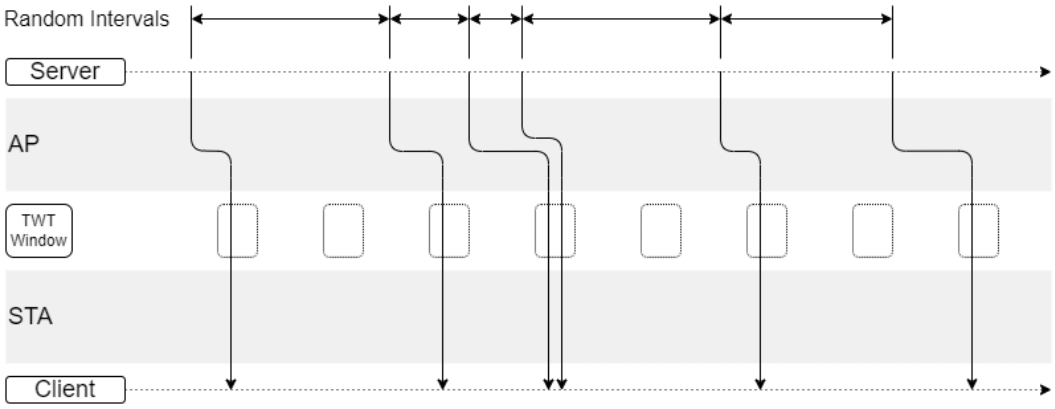


Figure 3.6: Actuator Private Server Test

The actuator test also includes an optional feature that echoes the observation messages with a request to another resource. This enables monitoring of the actuator message latency on the server side. The latency statistics are transmitted to the client at the end of the test. The echo request is not responded to.

The figure below illustrates the expected behavior of the actuator test when the echo option is activated.

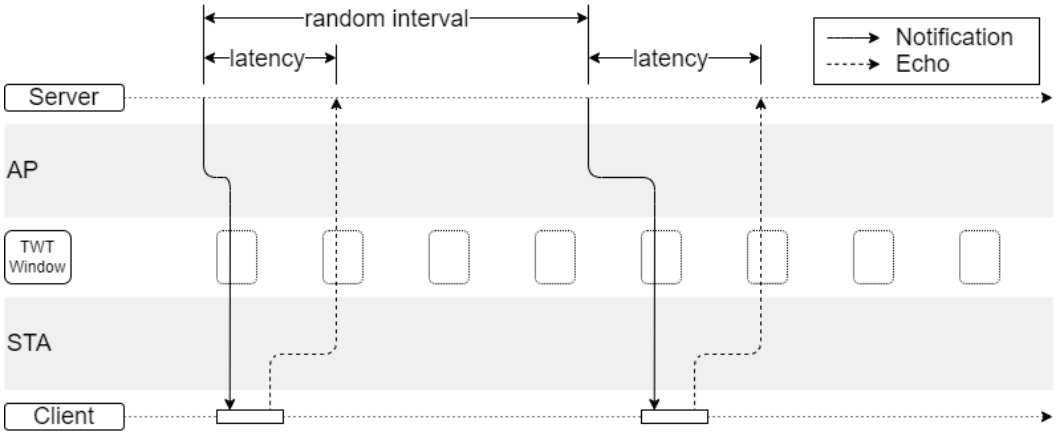


Figure 3.7: Actuator Private Server Test with Echo

Emergency Transmission

Emergency transmission is an option for the nRF70 Wi-Fi stack. It allows the transmission of uplink frames outside of a TWT session. The actuator use case includes an option to send the uplink frames with an emergency priority. When activated, the echo request (private server test) and the ACK (public server test) are immediately sent. This feature reduces the total round-trip time of an actuator message.

The figure below shows the emergency feature on the actuator public server test.

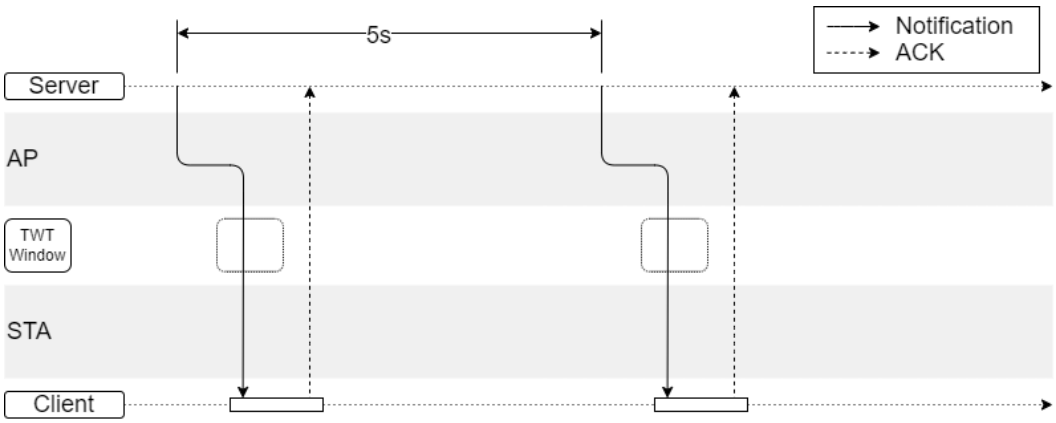


Figure 3.8: Actuator Public Server Test with Emergency Transmission

The figure below shows the emergency feature on the actuator private server test when the echo feature is activated.

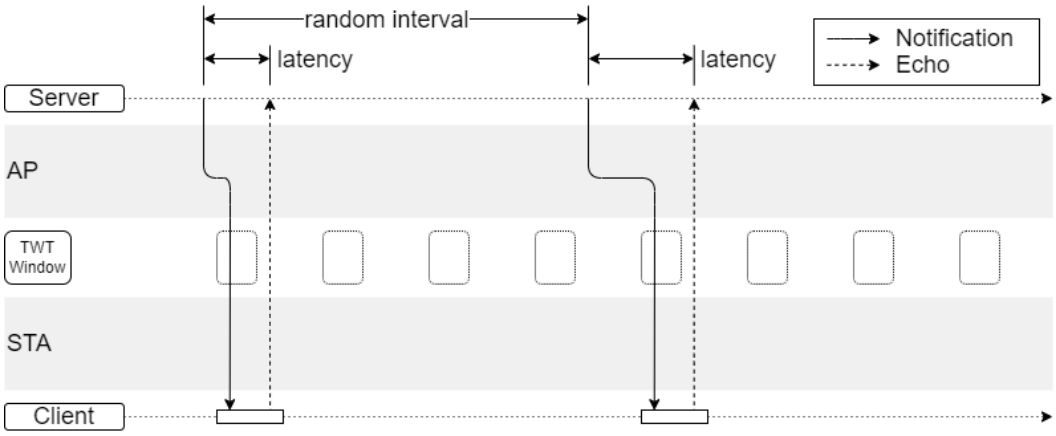


Figure 3.9: Actuator Private Server Test with Echo and Emergency Transmission

3.4 Monitoring

3.4.1 Power Profiling

The nRF7002's power consumption is monitored using the Power Profiler Kit 2 (PPK) [43]. The PPK can be connected to a computer and shows the current consumption of the Wi-Fi chip in real time. The power profiler also has digital inputs. The reading of these inputs is printed below the current curve. The digital inputs can be connected to the development kit's GPIOs to correlate the current consumption with the program state.

3.4.2 Network Monitoring

The Wi-Fi traffic can be monitored using Wireshark and by setting the Wi-Fi interface in monitor mode. However, monitor mode is only supported on Linux. The Wi-Fi interface must be set in monitor mode to monitor the traffic and select a specific Wi-Fi channel. The traffic can only be monitored on one channel. With this setup, Wireshark can capture the Wi-Fi frames. However, the content of the data frames is encrypted and can not be read. Nevertheless, Wireshark can decrypt the frames if it captures the WPA security handshake procedure and if the SSID and password are specified in the Wireshark parameters.

A script that sets the Wi-Fi interface in monitor mode is available in the appendix (B).

3.4.3 Test Monitoring

The testbed logs are printed on the serial port. Any serial monitor can show them. The logs give information about the Wi-Fi config, the sent and received messages, and the test results.

3.5 Network Topology

The figure below shows the network topology used for the tests.

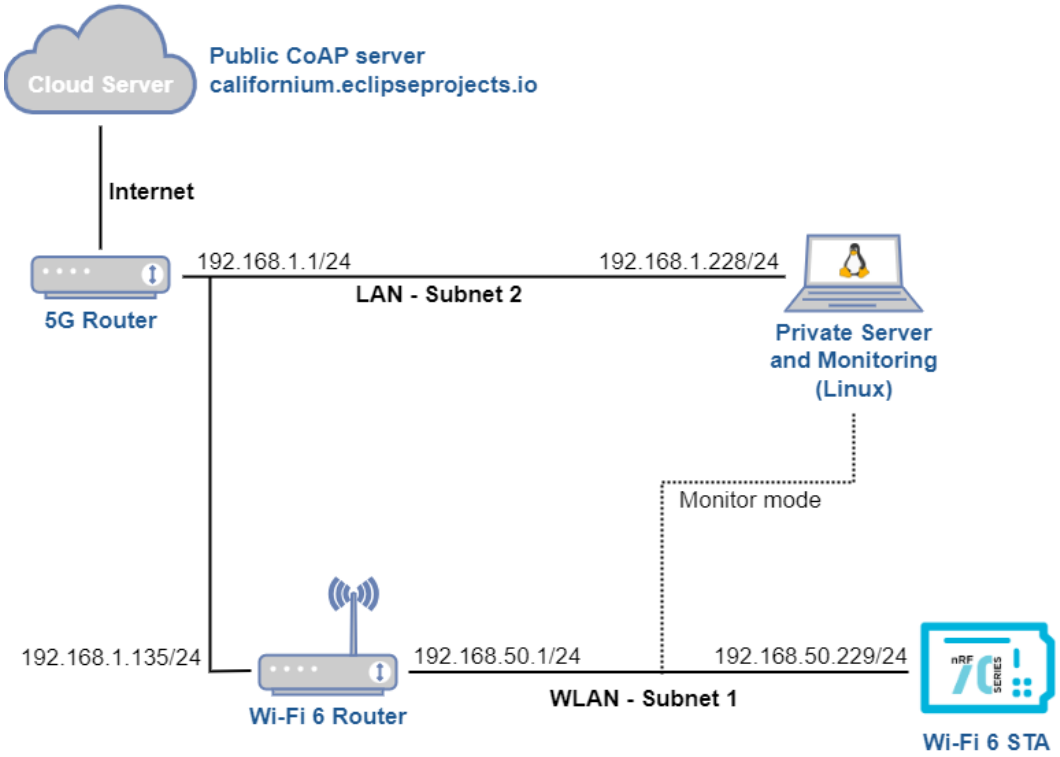


Figure 3.10: Network Topology

The test setup uses a Wi-Fi 6 Router AP. The STA is connected to its local subnetwork (WLAN - Subnet 1). The WAN interface of the AP is connected to the local subnetwork of a 5G router (LAN - Subnet 2). The 5G router allows the testbed to access the Internet and the public server.

The private server also connects to this local subnetwork (LAN - Subnet 2). The STA and the private server must be in different subnetworks to ensure the message flow and forwarding is the same as with a public server. If the private server and STA are in the same network, the STA and the server can exchange messages directly using the final destination MAC address without needing the packets to be routed by the AP.

As the computer running the private server is connected to the subnetwork of the 5G router with its ethernet interface, the Wi-Fi interface can also be used to monitor the Wi-Fi network between the Wi-Fi 6 AP and the STA. Note that when monitoring, the interface is not connected to the Wi-Fi network; it only sniffs the packets and does not impact the Wi-Fi network.

4 Testbed Client Implementation

This chapter describes the implementation of the testbed. The focus is on the client side; the application running on the nRF7002 DK (STA). The server implementation is described in Chapter 5.

This chapter includes a description of the general architecture, including a general component diagram. The following section describes the configuration of the testbed. Every setting of the testbed is described in this section. The last section describes the code of each component of the testbed application.

Contents

4.1 System Architecture . . . 58
4.2 Testbed Configuration . . . 59
4.2.1 CoAP Settings . . . 60
4.2.2 IP Settings . . . 61
4.2.3 Wi-Fi Settings . . . 61
4.2.4 Tests Settings . . . 62
4.2.5 Logging and Profiling . . . 67
4.3 Software Components . . . 68
4.3.1 Wi-Fi Station . . . 68
4.3.2 Wi-Fi Power Save . . . 70
4.3.3 Wi-Fi TWT . . . 71
4.3.4 Wi-Fi Utils . . . 73
4.3.5 Profiler . . . 74
4.3.6 CoAP . . . 75
4.3.7 CoAP Utils . . . 80
4.3.8 CoAP Security . . . 81
4.3.9 Tests . . . 82
4.3.10 Test Report . . . 87
4.3.11 Test Global . . . 88
4.3.12 Test Runner . . . 88
4.3.13 Main . . . 89

4.1 System Architecture

The application is split into multiple packages, each containing multiple components. A component is implemented by a source file and an associated header file, except for the tests with two source files, one for the public server version and one for the private server version. However, only one of these files is activated in a build, depending on whether the application is configured to use the private or public server.

The testbed includes the following packages:

- **wifi:** This package contains the components that allow the control of the Wi-Fi connection and power-saving mechanisms.
- **coap:** The CoAP package contains the components that implement a CoAP client tailored for the testbed needs.
- **profiler:** The profiler package only contains the profiler component. It offers tools for linking power measurements with the program state.
- **tests:** The test package contains all the tests and a test runner component. The test runner runs the tests according to the configuration.

The source code repository is available in Appendix A, and each component is described in section 4.3. The figure below shows the application architecture:

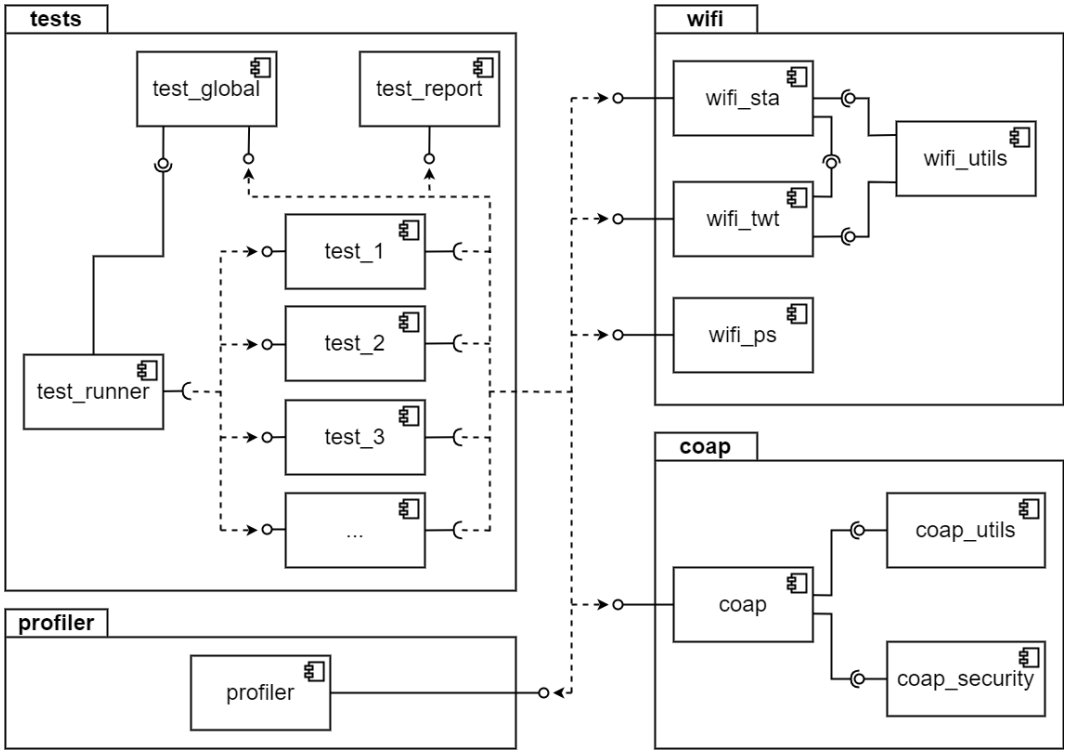


Figure 4.1: Testbed Component Diagram

4.2 Testbed Configuration

The testbed is entirely configurable using *kconfigs*. The custom *kconfigs* are defined in the *Kconfig* file. The baseline configuration (Zephyr kernel configurations) is in the *prj.conf* file, and the test specific configurations are in overlay files. The configuration is organized into sections and subsections and can be easily edited using the *kconfig* GUI tool of the nRF Connect vscode extension.

The testbed repository is available in Appendix A. The configuration files are in the *app* folder.

The figure below shows the main sections of the configuration.

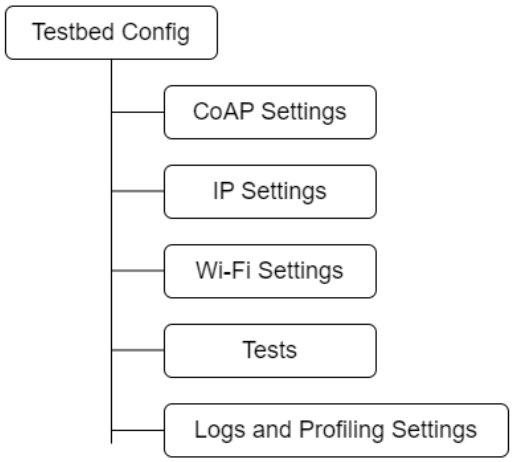


Figure 4.2: Testbed Global Configurations Overview

4.2.1 CoAP Settings

The figure below shows the CoAP configurations.

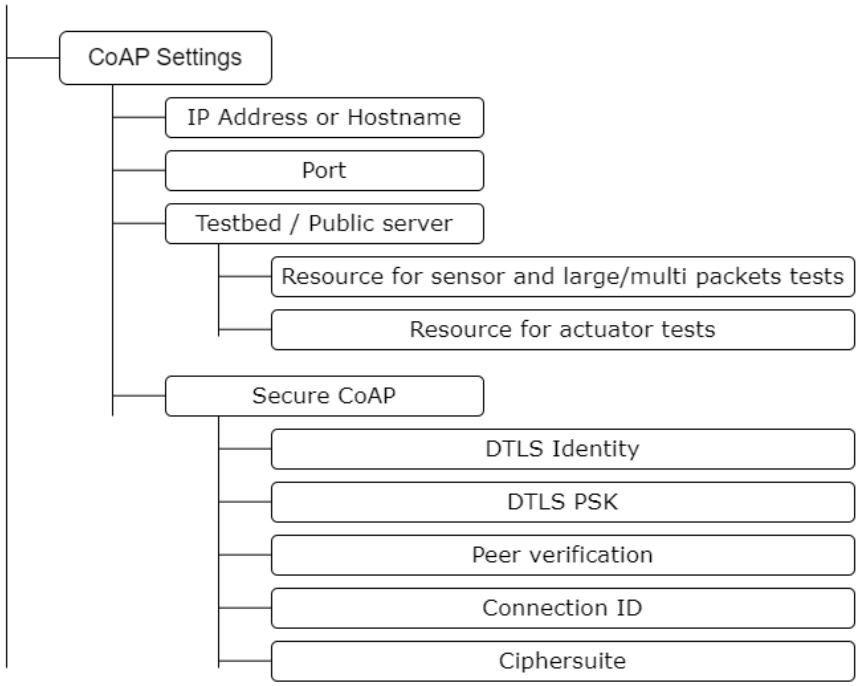


Figure 4.3: Testbed CoAP Configurations

The first field permits setting the server's IP address or hostname. If setting a hostname, the system will perform a DNS selecting if the testbed uses the private server (see 3.2) or a public server such as californium.eclipseprojects.io. The private server enables more advanced testing. If the public server is used, two more configurations are required to specify the resources used for the tests. On the californium server, the "validate" resource can be used for the sensor, large- and multi-packet tests, and the "obs" resource can be used for the actuator tests.

The "Secure CoAP" option enables a DTLS security layer. The testbed uses a PSK-based DTLS authentication. It requires an Identity and PSK. Peer verification and connection ID can be enabled or disabled. The testbed configuration also permits the choice of a cipher suite. The following cipher suites are available.

- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256

Both work with the private and the public server.

4.2.2 IP Settings

The IP section only contains one parameter to select IPv4 or IPv6. This parameter is global for all the tests.

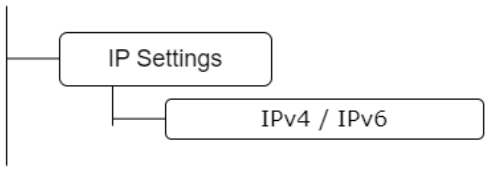


Figure 4.4: Testbed IP Configurations

4.2.3 Wi-Fi Settings

The figure below shows the Wi-Fi configurations.

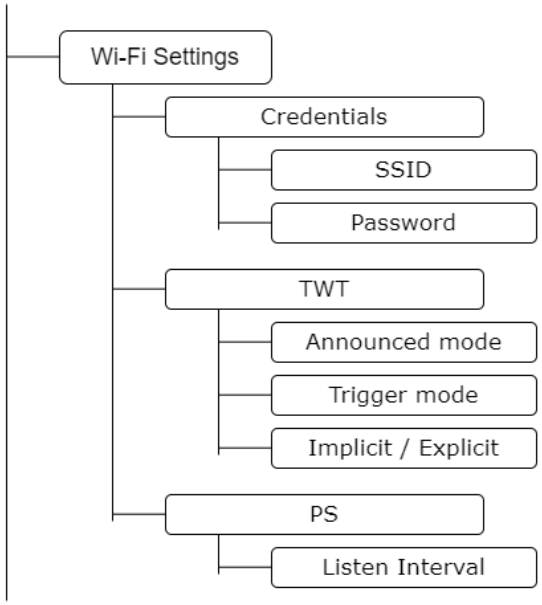


Figure 4.5: Testbed Wi-Fi Configurations

The first subsection permits setting the AP's SSID and password. The second allows for enabling and disabling TWT announced, trigger, and explicit/implicit modes. The PS section permits the setting of the listen interval value used when PS mode is activated and the wake-up mode is set to listen interval. These settings are global and used in all tests. The other TWT and PS settings, such as TWT interval, PS mode, etc., are specific for each test and set in the test configuration.

4.2.4 Tests Settings

The figure below shows how the tests are organized.

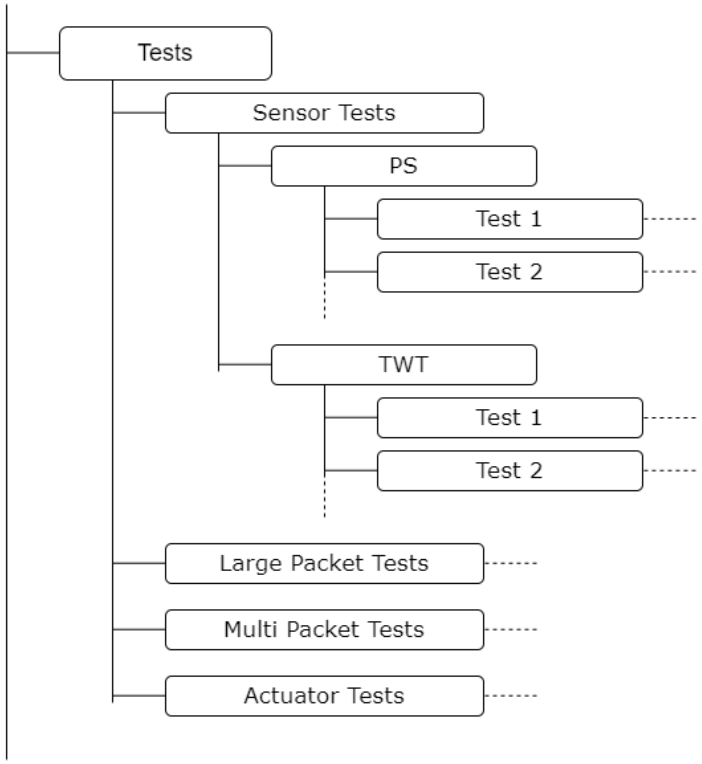


Figure 4.6: Testbed Tests Configurations

The testbed implements four use cases that can be used with different configurations. Each use case has a PS and a TWT version, and multiple configurations can be tested for each version.

Sensor Tests

The sensor use case is described in section 3.3.1.

The figure below shows the configurations for the sensor use case PS and TWT tests. Up to four PS and four TWT tests can be registered.

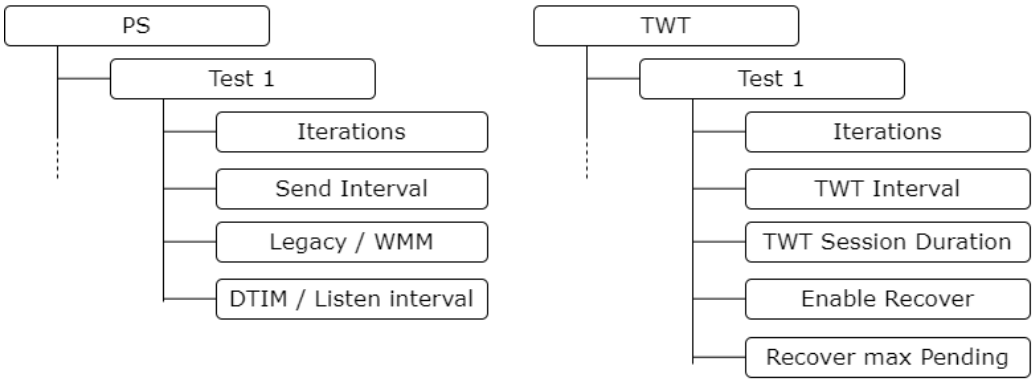


Figure 4.7: Testbed Sensor Tests Configurations

The PS tests enable the following settings:

- **Iterations:** The number of iterations to run. At every iteration, one CoAP PUT request is sent.
- **Send Interval:** The interval between each iteration in milliseconds.
- **Legacy/WMM:** The PS mode. Legacy and WMM modes are available.
- **DTIM/Listen Interval:** The PS wake-up mode. DTIM and Listen interval mode are available.

The TWT tests enable the following settings:

- **Iterations:** The number of iterations to run. At every iteration, one CoAP PUT request is sent. The request is sent at every TWT session.
- **TWT Interval:** The requested TWT interval in milliseconds.
- **TWT Session duration:** The requested duration of the TWT session.
- **Enable Recover:** When enabled, the testbed teardown and re-setup TWT if the responses are not received.
- **Recover max Pending:** This config is only available if Recover mode is enabled. It sets the maximum number of pending requests before teardown and re-setup TWT. Minimum value is 2.

Large Packet Tests

The large packet test case is a variant of the sensor test. It is described in section 3.3.2.

The figure below shows the configurations for the large packet test case PS and TWT tests. Up to four PS and four TWT tests can be registered.

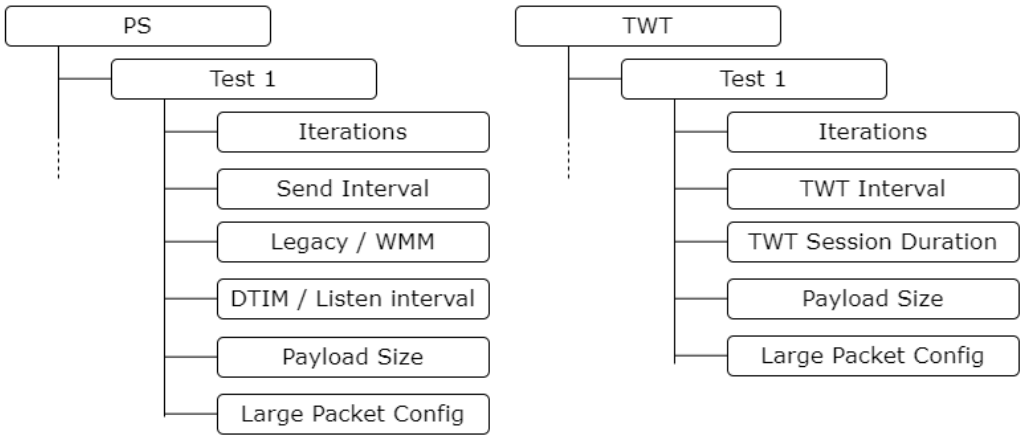


Figure 4.8: Testbed Large Packet Tests Configurations

The first settings are the same as for the sensor use case. The large packet test adds the following parameters:

- **Payload Size:** The size of the request's payload in bytes. The maximum value is 1024.
- **Large Packet Config:** This setting is only available on the private server version of the test. On the public server, the request is large, and the response depends on the server (small on Californium). There are three options:
  - Large Request, Small Response
  - Small Request, Large Response
  - Large Request, Large Response

Multi Packet Tests

The multi packet use case is described in section 3.3.3.

The figure below shows the configurations for the multi packet use case PS and TWT tests. Up to two PS and two TWT tests can be registered.

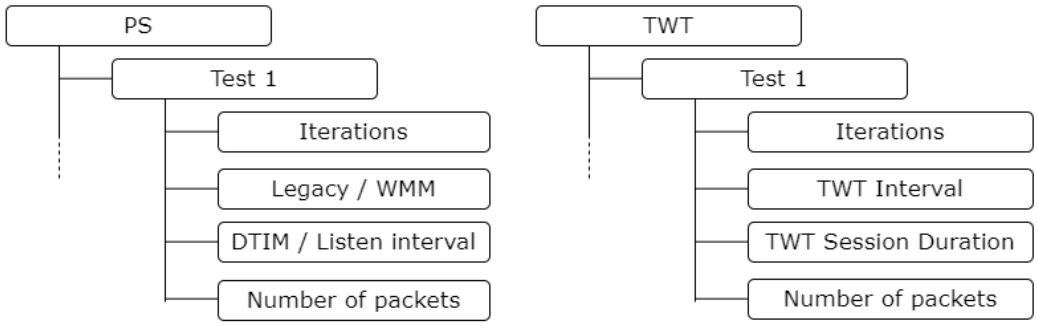


Figure 4.9: Testbed Multi Packet Tests Configurations

The following settings differ from the previous tests:

- **Iterations:** The number of iterations to run. At every iteration, multiple CoAP PUT requests are sent. After sending the requests, the system waits for all the responses before proceeding to the next iteration. A timeout prevents the system from blocking if some packets are lost.
- **Number of packets:** The number of packets sent at each iteration.

Actuator Tests

The actuator use case is described in section 3.3.4.

The figure below shows the configurations for the actuator use case PS and TWT tests. Up to four PS and four TWT tests can be registered.

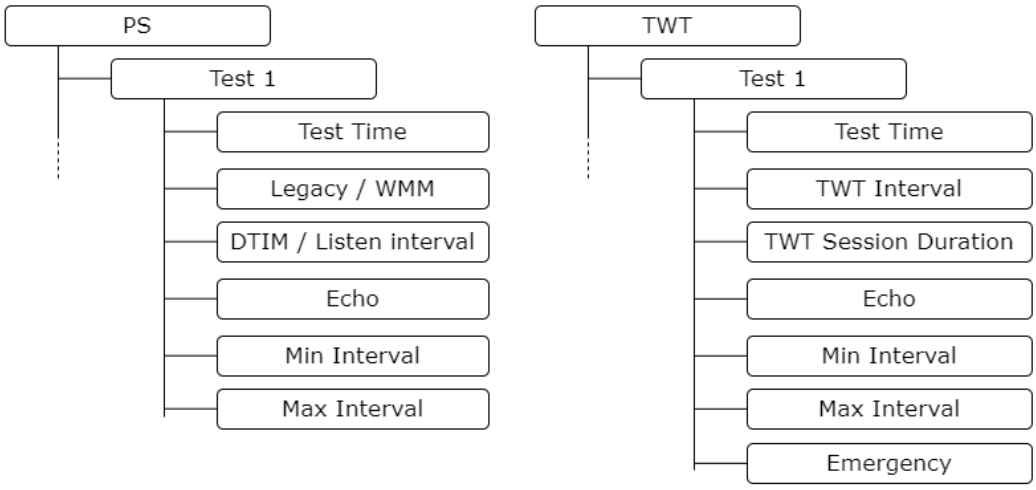


Figure 4.10: Testbed Actuator Tests Configurations

The following settings differ from the previous tests:

- **Test time:** The time of the test. Unlike previous tests, actuator tests are specified by time instead of iterations.
- **Echo:** This parameter is only available for the private server version of the tests. If enabled, the client responds to the server actuation messages on another resource. When activated, the latency is calculated.
- **Min Interval:** This parameter is only available for the private server version of the tests. This setting sets the minimum interval between two actuator messages in seconds.
- **Max Interval:** This parameter is only available for the private server version of the tests. This setting sets the maximum interval between two actuator messages in seconds.
- **Emergency:** When enabled, the uplink traffic packets (Echo for the private version and ACK for the public version) are sent outside the TWT session.

4.2.5 Logging and Profiling

The figure below shows the logging and profiling configurations:

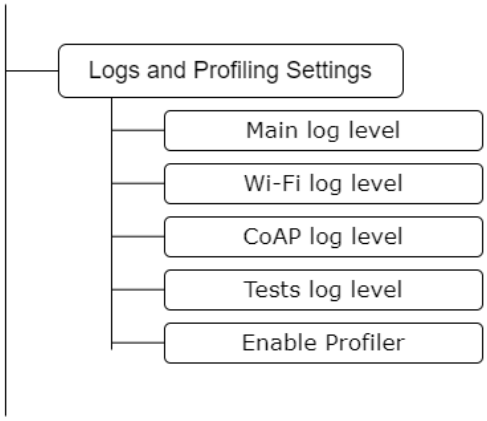


Figure 4.11: Testbed Logging and Profiling Configurations

The log-level fields allow changing the log level of each software component of the testbed. The default level is 3.

The profiler component is optional and can be turned on and off with the *Enable Profiler* parameter. If enabled, the project must be built with the devicetree overlay file (*nrf7002dk_nrf5340_cpuapp.overlay*).

4.3 Software Components

This section describes each component. The testbed repository is available in Appendix A, and the source code is in the *app/src* folder.

4.3.1 Wi-Fi Station

The Wi-Fi Station is the component that manages the connection with the AP. The figure below shows the functions it implements.

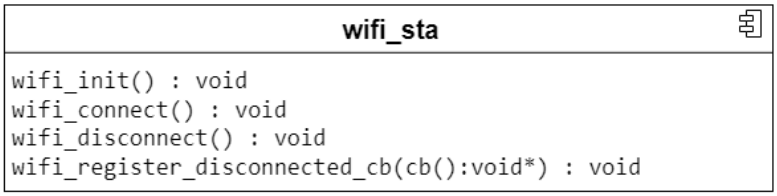


Figure 4.12: wifi_sta component

The Wi-Fi Station uses the Zephyr Network Management API [44]. This API allows the application to call routines and receive events to manage any network interface (the Wi-Fi interface, in this case).

The *wifi_init* function must be called before calling any other function of the Wi-Fi components. This function initializes two event callbacks of the Wi-Fi Station component. The first is called when the Wi-Fi STA is connected and disconnected from the AP, and the second is when the STA receives a new DHCP IP address. The *wifi_init* function also initializes the *wifi_twt* component.

The *wifi_connect* and *wifi_disconnect* functions work similarly. They send a connect/disconnect command with the *net_mgmt* function. Then, a semaphore blocks the function until the expected event arrives (connected or disconnected). The event releases the semaphore, and the function returns. For the connect function, both connected and DHCP-bound events are expected. If the connection or disconnection fails, the function returns an error code. The connect and disconnect functions only run into completion once the expected *net_mgmt* event is received.

The figure below shows the connection sequence. The disconnection process employs the same principle.

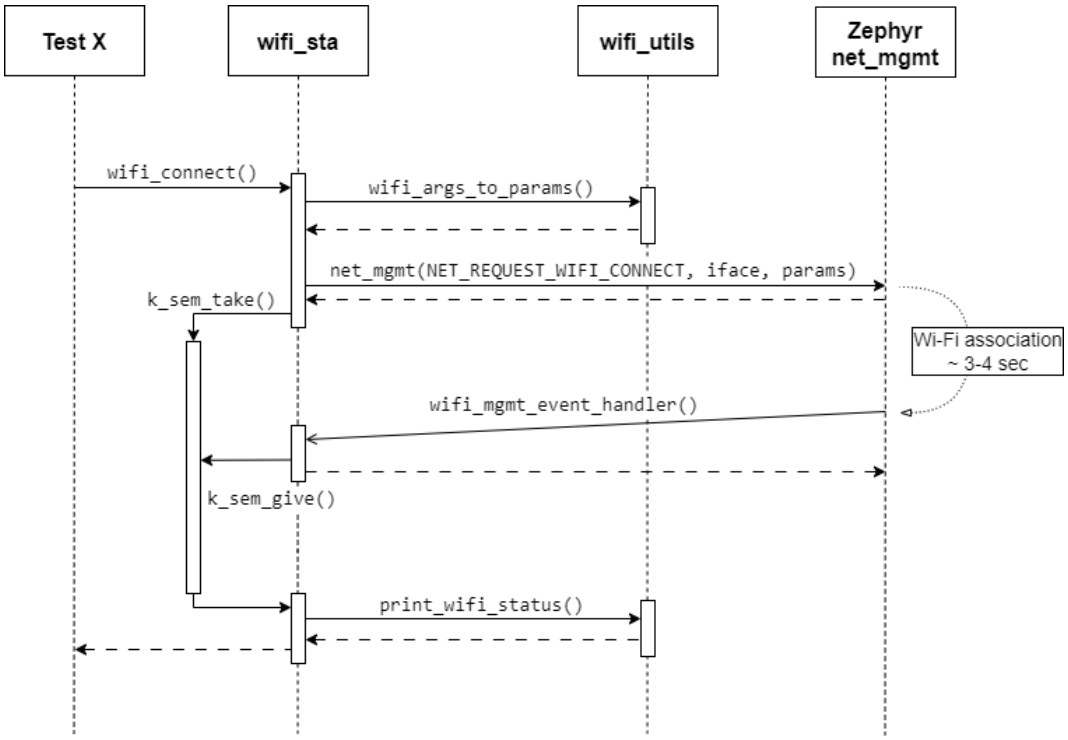


Figure 4.13: Wi-Fi Connection Sequence Diagram

Note that only the connect event is shown in this figure for better readability. The code implements two semaphores; both are taken when the *net_mgmt* connect command is sent, and each callback releases one.

This implementation waits for the connection to succeed before returning from the function. This choice has been made because the testbed needs to wait for the connection to succeed in every case. This kind of implementation should be avoided in a regular application.

The last function of the Wi-Fi Station component permits the registration of a callback function to handle unexpected disconnections. This callback allows the application to react to a disconnection, for example, by retrying the connection or generating an error.

4.3.2 Wi-Fi Power Save

The Wi-Fi Power Save component provides functions to activate and configure the Power Saving. The figure below shows the functions it implements.

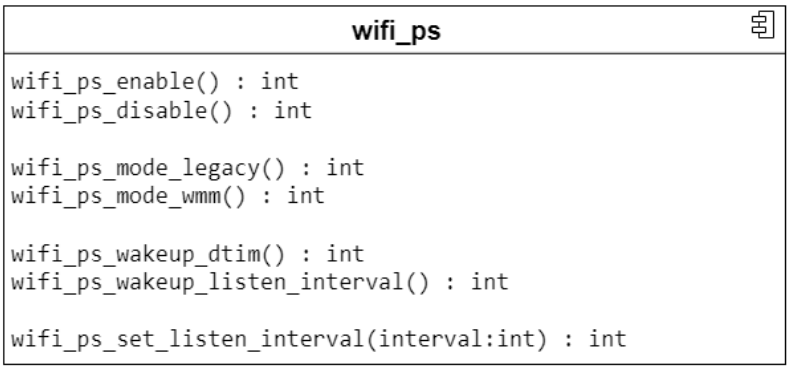


Figure 4.14: wifi_ps component

All the functions work similarly. The *net_mgmt* function from the Net Management API [44] is called with the *NET_REQUEST_WIFI_PS* event and a struct containing the requested parameters.

The listen interval can only be set before initializing the Wi-Fi. Its default value is 10, which means that the device wakes up for 1 beacon out of 10 when the wakeup mode is set to listen interval. The Power Save mode can not be changed when Wi-Fi is connected. The wake-up mode and enable/disable functions can be called anytime.

4.3.3 Wi-Fi TWT

The Wi-Fi TWT component provides all the functions needed to manage TWT. The figure below shows the functions it implements.

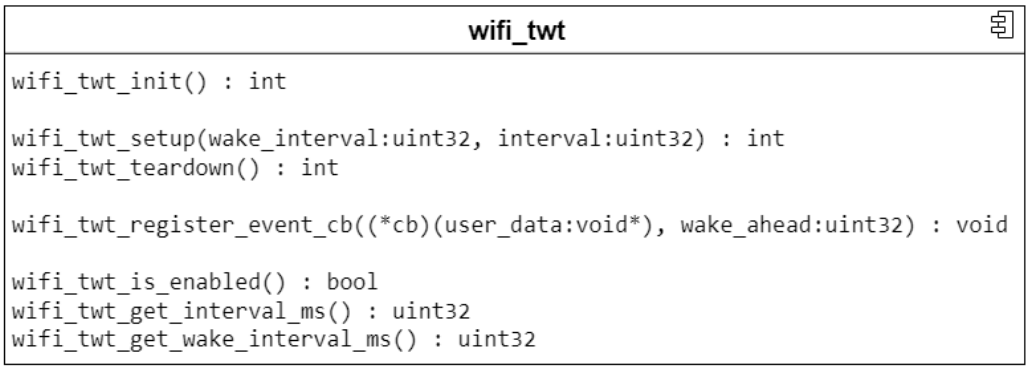


Figure 4.15: wifi_twt component

The initialization function from the Wi-Fi Station component calls the *wifi_twt_init* function. TWT is supported in the Net Management API. The *wifi_twt_init* function configures the callback called when the TWT session starts and ends. Setup and teardown functions manage TWT, and *wifi_twt_register_event_cb* allows registering a callback function for the TWT events. The three last functions implemented by the Wi-Fi TWT components permit retrieving the TWT state and negotiated intervals.

TWT Setup and Teardown

When calling the TWT setup function, a *net_mgmt* request is submitted. A struct containing the TWT parameters is passed with the request. The following settings are set:

- **Negotiation Type**: This setting is set to individual negotiation. This thesis focuses solely on tests involving one STA and one AP.
- **Flow ID**: Flow ID is incremented after each teardown.
- **Dialog Token**: Dialog token is incremented after each request.
- **Trigger**: Trigger-based session can be set in the testbed configuration.
- **Implicit**: Implicit wake-up can be set with in the testbed configuration.
- **Announce**: Announced TWT can be set with a in the testbed configuration.
- **TWT Wake Interval**: This setting is a parameter of the *wifi_twt_setup* function.
- **TWT Interval**: This setting is a parameter of the *wifi_twt_setup* function.

The figure below shows the TWT setup sequence.

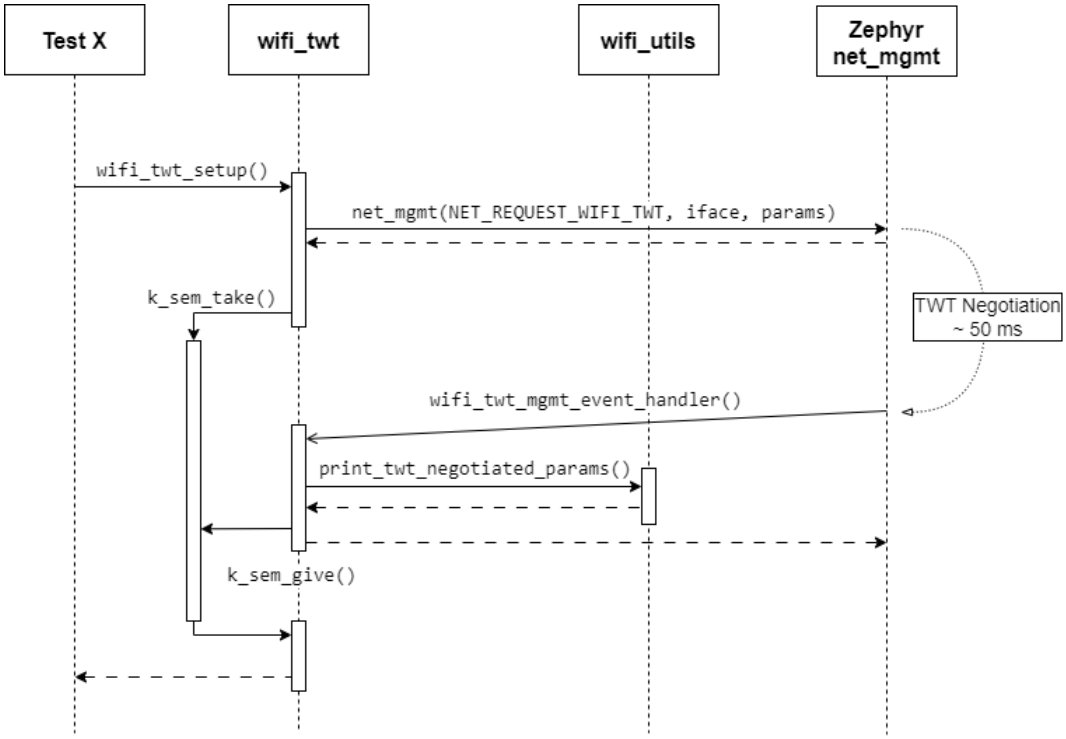


Figure 4.16: Wi-Fi TWT Setup Sequence Diagram

As for the connection, when the *net_mgmt* request is submitted, the function execution is blocked by a semaphore and resumes once the response is received. When the response is received, the TWT negotiated parameters are printed on the console. Warnings are printed if the negotiated parameters do not correspond to the requested ones. The semaphore uses a timeout that expires if the TWT response is never received. If the timeout expires, the *net_mgmt* request is retried a few times. Nevertheless, the TWT response is almost always received. An unresponded TWT request has only been observed once in thousands of tests.

The TWT teardown works like the setup, but the request is teardown instead of setup.

Wake-Ahead Mechanism

The *wifi_twt_register_event_cb* function permits the registration of a function called on the TWT session. The *wake_ahead* parameter allows the function to be called ahead of the TWT session, enabling the data to be prepared just before the session. This functionality is very interesting for a sensor use case. It allows the synchronization of the measurement and the transmission with the TWT session, enabling very short periods of active power state of both host and companion MCUs.

This mechanism is implemented in the TWT API. It uses the timer that manages the active and sleeping periods of the nRF7002 companion IC. The limitation of this implementation is that the timer is on the companion IC, so it has to wake up to send the wake-ahead event to the host MCU (nRF5340) and be in an active power state during the wake-ahead period. To avoid this limitation, the wake-ahead mechanism

of the TWT API is not used. Instead, it is simulated with a timer on the host MCU. A small compromise is made on power consumption because it uses one more timer. Nevertheless, the increased consumption due to the timer remains minimal.

The figure below shows the timer wake-ahead mechanism.

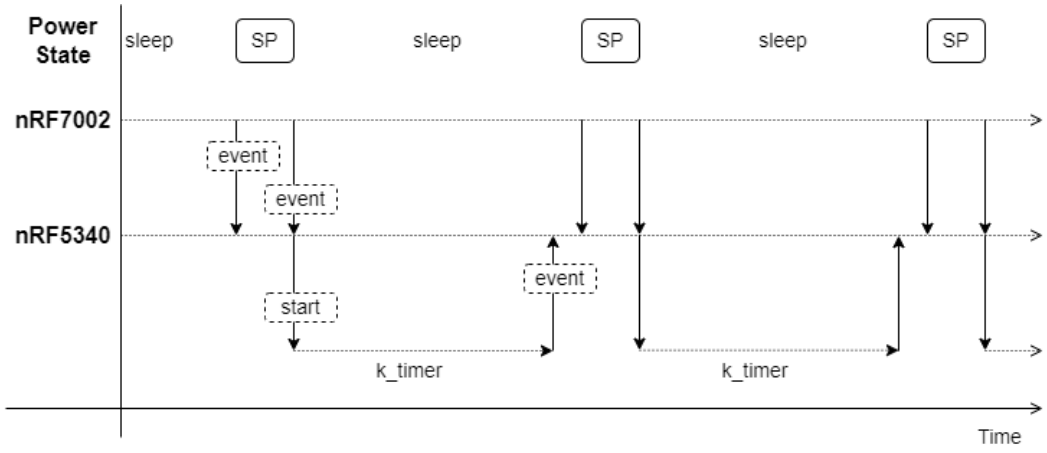


Figure 4.17: Wake-Ahead Timer

4.3.4 Wi-Fi Utils

This component implements various functions used by the other Wi-Fi components. The aim is to limit the lines of code in the other Wi-Fi components by relieving them of their straightforward functions.

The figure below shows the functions implemented in *wifi_utils*.

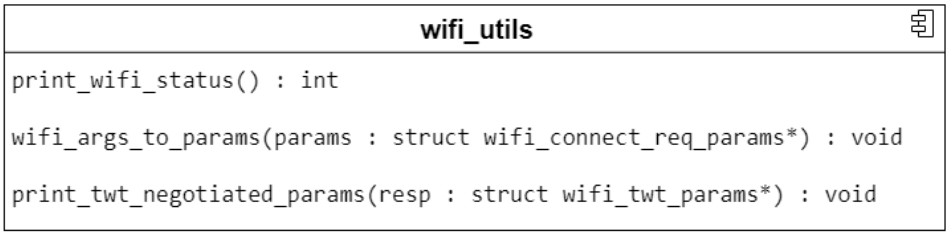


Figure 4.18: wifi_utils component

4.3.5 Profiler

The profiler component controls GPIOs and aims to link the tests with the power measurements. This module permits printing binary numbers on GPIOs, which can be connected to the logic analyzer of the Power Profiler. When a test is started, the profiler component will print the binary value of the test ID on the logic inputs of the Power Profiler.

The profiler component is optional and can be turned on and off in the testbed configuration. If enabled, the project must be built with the devicetree overlay file (*nrf7002dk_nrf5340_cpuapp.overlay*).

The figure below shows the function implemented by the *profiler* module.

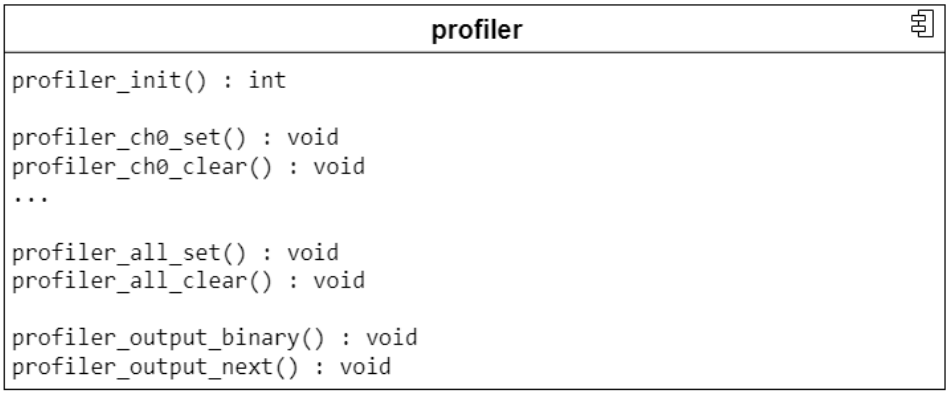


Figure 4.19: profiler component

The *profiler_init* function must be called first. Functions to set and clear each channel or every channel are available. The *profiler_output_binary* function controls all GPIOs to print a binary number. The *profiler_output_next* function outputs a binary number incremented each time the function is called, starting from 0.

4.3.6 CoAP

The CoAP component implements a CoAP client tailored explicitly for the testbed.

During the implementation phase, the *coap_client* library [45] provided by Zephyr was initially employed. However, its limitations in flexibility and suitability for the specific requirements of the testbed led to the development of a custom CoAP client.

Limitations of the coap_client library

The *coap_client* library [45] creates a dedicated thread to handle each pending request, which is used for the reliability mechanisms. However, in the context of the testbed, the primary objective is to evaluate packet loss, so only non-confirmable packets are used.

The reliance on a separate thread for each pending request is a significant drawback because it limits the number of concurrent pending requests that can be used, as each request consumes a lot of memory. The library only supports two concurrent pending requests.

Moreover, when using non-confirmable requests, the library does not proceed to retransmissions but still expects a response. If the packet is lost, the request will be pending forever. To address this problem, the library has a function that cancels pending requests. However, the function will cancel both requests, even if only one needs to be canceled. [1]

Furthermore, canceling an observation with the *coap_client* library is impossible. The observe mechanism allows the server to notify the client. The client sends a GET request with the observe option set to 0 to start the observation. To cancel the observation, the client must send a GET request with the observe option to 1. Both GET requests must have the same token. The problem is that the library manages the token, and every request has a different one. It does not let the user specify a token.

These limitations led to developing a client tailored to the testbed's needs. However, it is important to note that the *coap_client* library is still classified as experimental and remains under active development. Despite its current limitations, the library provides a solid foundation for reliable CoAP communication in many use cases.

[1] It is possible to cancel individual requests with Zephyr 4.0.0. The project was developed with NCS 2.8, which implements Zephyr 3.7.0.

CoAP Client Implementation

The CoAP client implementation of the testbed uses the *coap* library [46] instead of *coap_client*. Unlike *coap_client*, the *coap* library does not implement any advanced mechanisms, such as error detection and retransmission. The *coap* library only allows the creation and parsing of packets, which can be sent and received using a standard Zephyr UDP socket [47].

The CoAP components use two threads: one for sending and the other for receiving. The socket is shared between these two threads. It is protected against concurrent access by a mutex.

A buffer is used to store packets that are ready to be sent. When a packet is prepared, the *send_sem* semaphore is given, allowing the sender thread to transmit the buffered packet. Once the packet is sent, the *sent_sem* semaphore is given, enabling the sender process to retrieve the return code of the transmission. This mechanism allows the return codes to be sent back to the test component, facilitating the monitoring of potential errors.

The receiver thread handles incoming packets by receiving and parsing them. Upon receiving a packet, the thread takes actions depending on the content, such as invoking a callback, giving a semaphore, or sending an ACK.

The figure below shows the data flows between the threads.

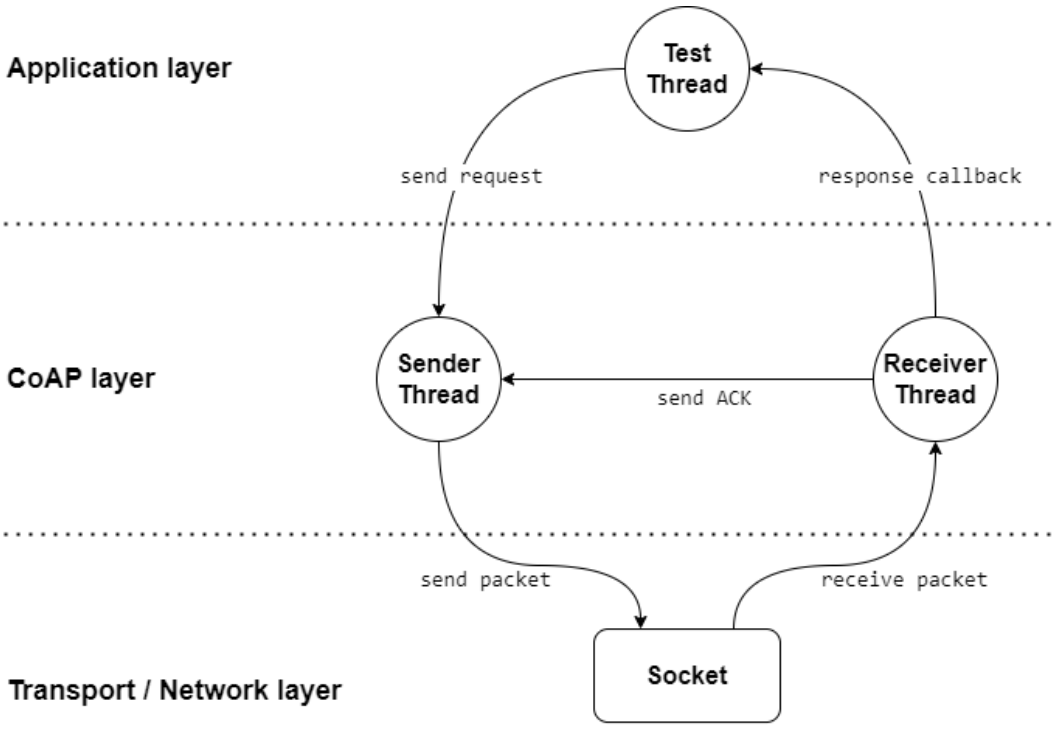


Figure 4.20: CoAP component data flow

The figure below shows the functions implemented by the *coap* component.

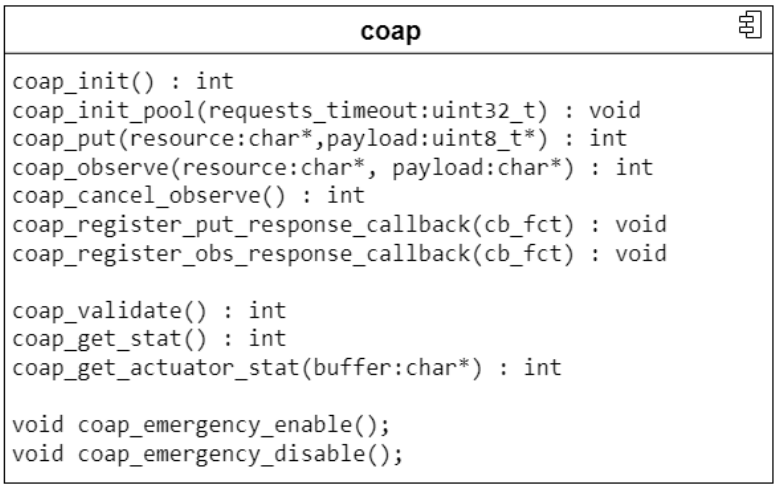


Figure 4.21: CoAP component

The *coap_init* function has to be called before the other functions are used. The network must be connected when calling the CoAP initialization function because it performs a DNS resolution. The function creates the socket according to the configuration (4.2). It can use IPv4 or IPv6 and UDP or DTLS. A DTLS socket is configured using a function of the *coap_security* component. After creating the socket, the function creates and starts both sender and receiver threads.

The *coap_put* function sends PUT requests. It is used by the sensor, large-packet, and multi-packet tests. The response time of the PUT requests is tracked. The *coap_pool_init* function allows for clearing and initializing a pool for pending requests identified by their token. Requests are removed from the pool after a timeout, which is configured as a parameter of the *coap_pool_init* function. If a response is received after the request has timed out, it is discarded. The pending requests pool is implemented in the *coap_utils* component. Once the response is received, the round trip time is calculated, and the callback function is invoked to notify the test component.

The *coap_observe* and *coap_cancel_observe* functions are used by the actuator test to start and stop observations. If the notifications are received as confirmable CoAP packets, as it is on the Californium public server, an ACK is automatically sent. When a notification is received, the callback function is invoked.

The *coap_validate*, *coap_get_stat*, and *coap_get_actuator_stat* functions are only available when using the private server. The validate function is called before each test. It sends a GET request to validate that everything is correctly configured. The response needs to be received before the test can proceed. This function also resets and initializes the statistics computed on the server. On the server, a counter counts the number of requests received for the sensor, large-packet, and multi-packet tests. Its value can be retrieved using the *coap_get_stat function*. For the actuator tests, the counter counts the number of notifications sent. When the echo feature is used, the *coap_get_actuator_stat function* retrieves the latency histogram.

The *coap_emergency_enable* and *coap_emergency_disable* functions activate and deactivate emergency transmissions. When emergency mode is enabled, uplink traffic is sent with an emergency priority, allowing it to be transmitted outside of a TWT session. This feature is used in the actuator tests.

Request Transmission Sequence

The figure below shows the sequence of the transmission of a PUT request.

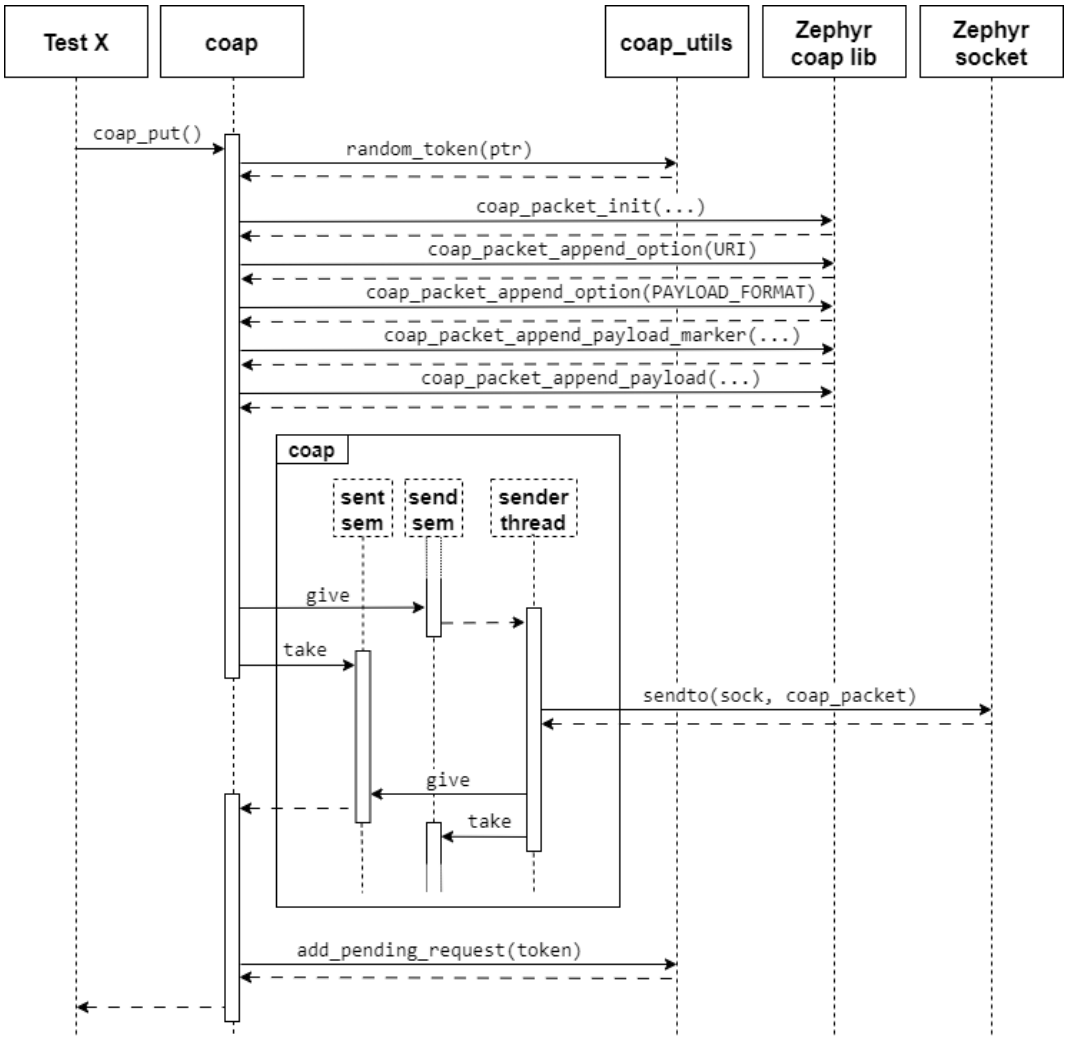


Figure 4.22: CoAP request transmission sequence

First, the CoAP packet is created. It is stored in a buffer. Then, the sender thread sends the packet. Once sent, the return code is stored in a buffer, and the sender thread waits for the next packet to be sent. Then the *coap_put* function reads the return code, and if it is not an error, the token is added to the pool for the pending requests, and the function returns.

Response Reception Sequence

The figure below shows the sequence of the reception of a PUT request's response.

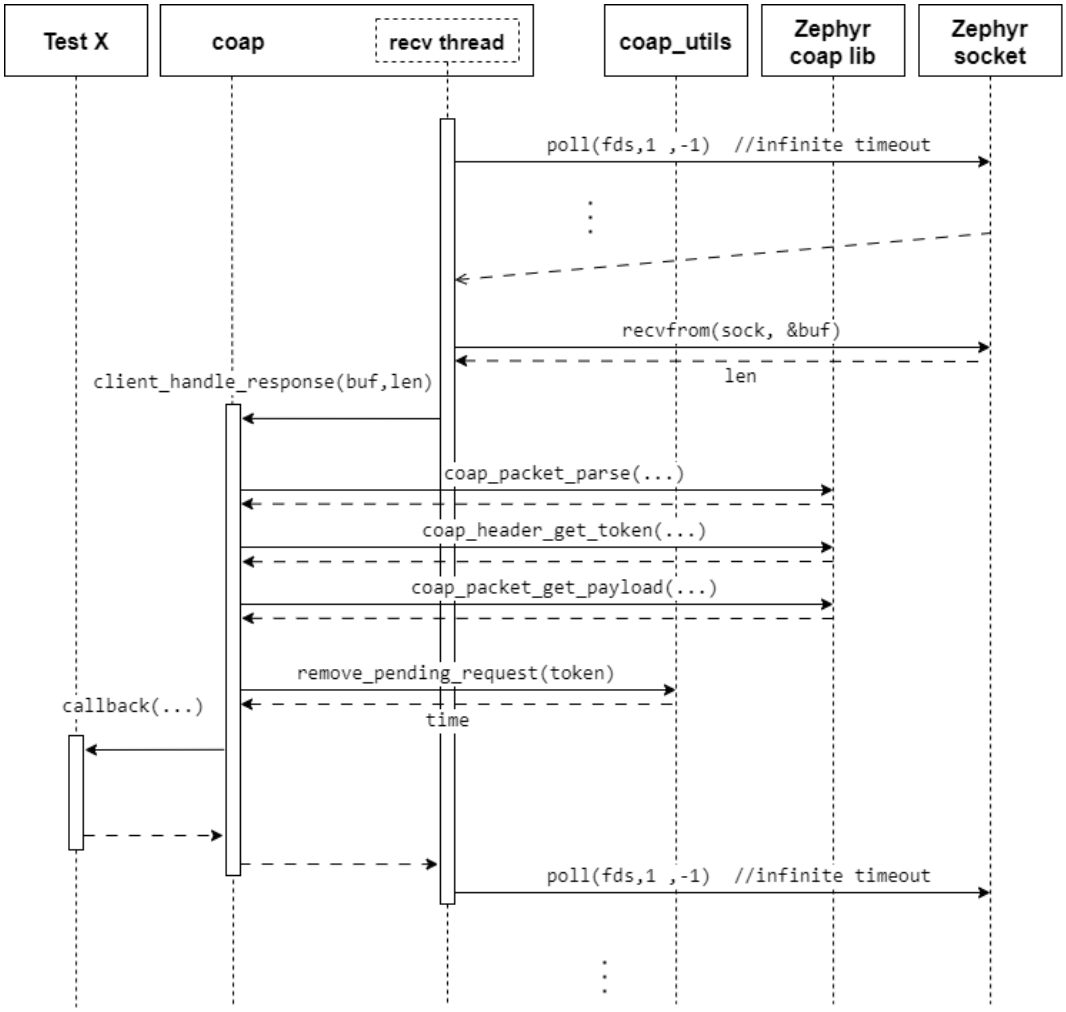


Figure 4.23: CoAP request reception sequence

The receiver thread uses the poll function of the socket library with an infinite timeout. The poll blocks the execution until a packet is received on the socket. Then, the packet is read using the *recvfrom* function, and removed from the pending request pool, and the round trip time is returned to the test using the callback. The reception of a notification of the actuator test is the same, with the only difference being that the pool for pending requests is not used, and another callback is invoked. the packet is parsed. In this case, it is a PUT response, so the token is removed from the pending request pool, and the round trip time is returned to the test using the callback. The reception of a notification of the actuator test is the same, with the only difference being that the pool for pending requests is not used, and another callback is invoked.

4.3.7 CoAP Utils

This component implements various functions used by the *coap* component. The aim is to limit the lines of code in the *coap* component, relieving it of its straightforward tasks.

The figure below shows the functions implemented in *coap_utils*.

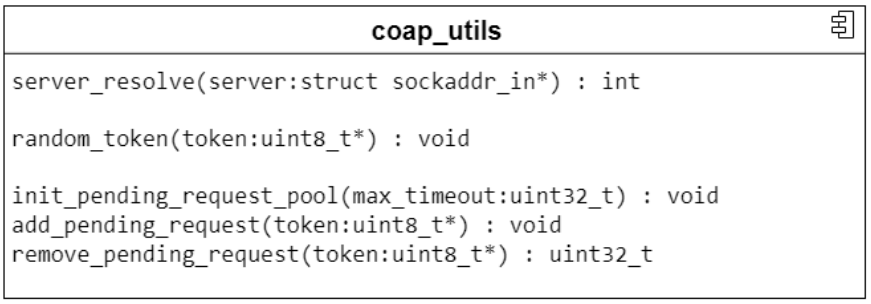


Figure 4.24: CoAP Utils component

The *server_resolve* function performs the DNS resolution. It finds the IPv4 or IPv6 address depending on which protocol is configured.

The *random_token* function generates a random token for a CoAP request. The token length is not configurable but is defined in the *coap_utils.h* file. Its value is 8.

As discussed in the previous section, the three last functions control the CoAP request pool. The pool can be initialized with a maximum timeout, after which the requests are discarded. There is a function to append a request token and a function to remove a request token. When a token is removed, the function returns the request round trip time. The pool size is not configurable but is defined in the *coap_utils.c* file. Its value is 100.

The pool stores the tokens and timestamps when added using the kernel timer [48]. When removed, the pending time is calculated using the current time and the timestamp recorded when added. The pool automatically removes the timed-out requests. It is updated every time the add or remove function is called.

4.3.8 CoAP Security

The *coap_security* component implements the functions needed to create a DTLS socket. The figure below shows the functions implemented by the *coap_security* component:

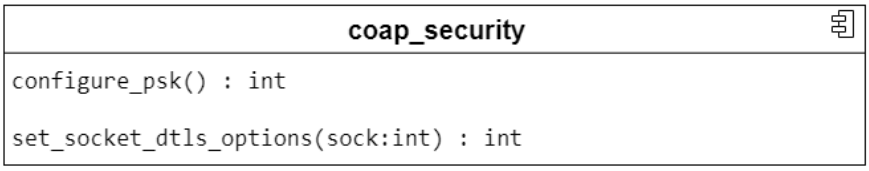


Figure 4.25: CoAP Security component

The *configure_psk* function stores the credentials. It is called during the initialization of the testbed.

The *set_socket_dtls_options* function is called after the socket creation in the *coap_init* function. This function sets the different socket options according to the configuration of the testbed. The peer verification, connection ID feature, cipher suite, and credentials are configured in this function.

4.3.9 Tests

The testbed implements four use cases with PS and TWT tests. Each test has a private server and a public server version. A test is implemented in one *.h* file and two *.c* files: one for the public server version and one for the private server version. Only one of the *.c* files is activated in a build, depending on whether the server is configured to use the private or public server. Every test follows the same structure and uses the same mechanism to be invoked. The figure below shows a sensor TWT test.

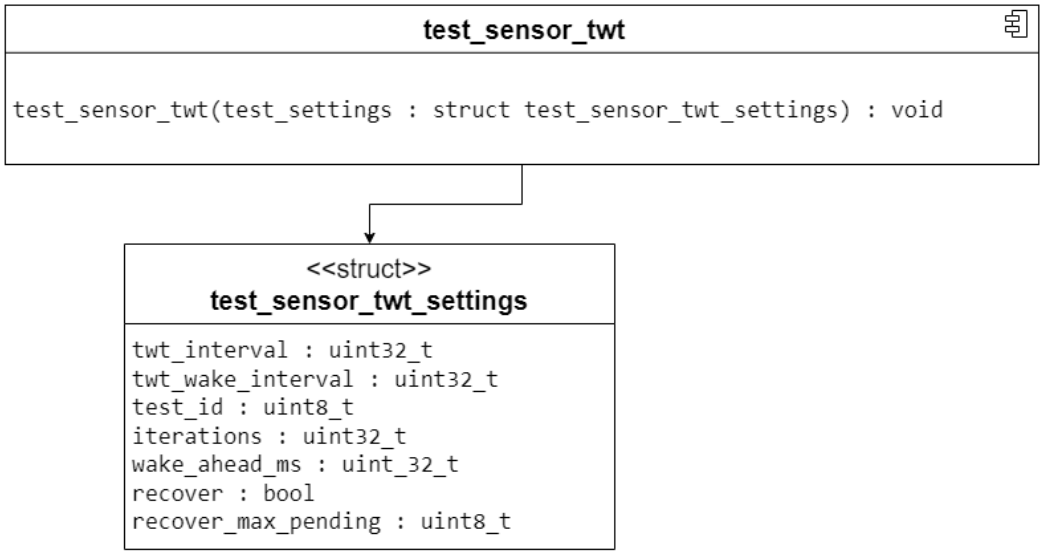


Figure 4.26: Sensor Test Component

To run a test, the test settings struct must be created and filled with the test parameters. Then, the test function can be called with the settings struct as a parameter. The function returns when the test is finished. The listing below shows how a sensor TWT test can be invoked.

```
struct test_sensor_twt_settings test_settings = {
    .iterations = 1000,
    .twt_interval = interval,
    .twt_wake_interval = duration,
    .test_id = 1,
    .recover = false,
    .wake_ahead_ms = 100,
};

test_sensor_twt(&test_settings);
```

Listing 4.1: Sensor TWT test invocation

Every test uses the same mechanism to be run; only the settings and names differ. The testbed automatically selects the private or public server version according to the configuration.

A thread is created for every test. The test is executed on the test thread. Then, when it is finished, the thread is aborted, and a new thread is created for the next test. A semaphore blocks the main thread while the test thread is running. As the test threads are executed sequentially, they all reuse the same stack to avoid consuming too much memory.

The diagram below shows the test execution.

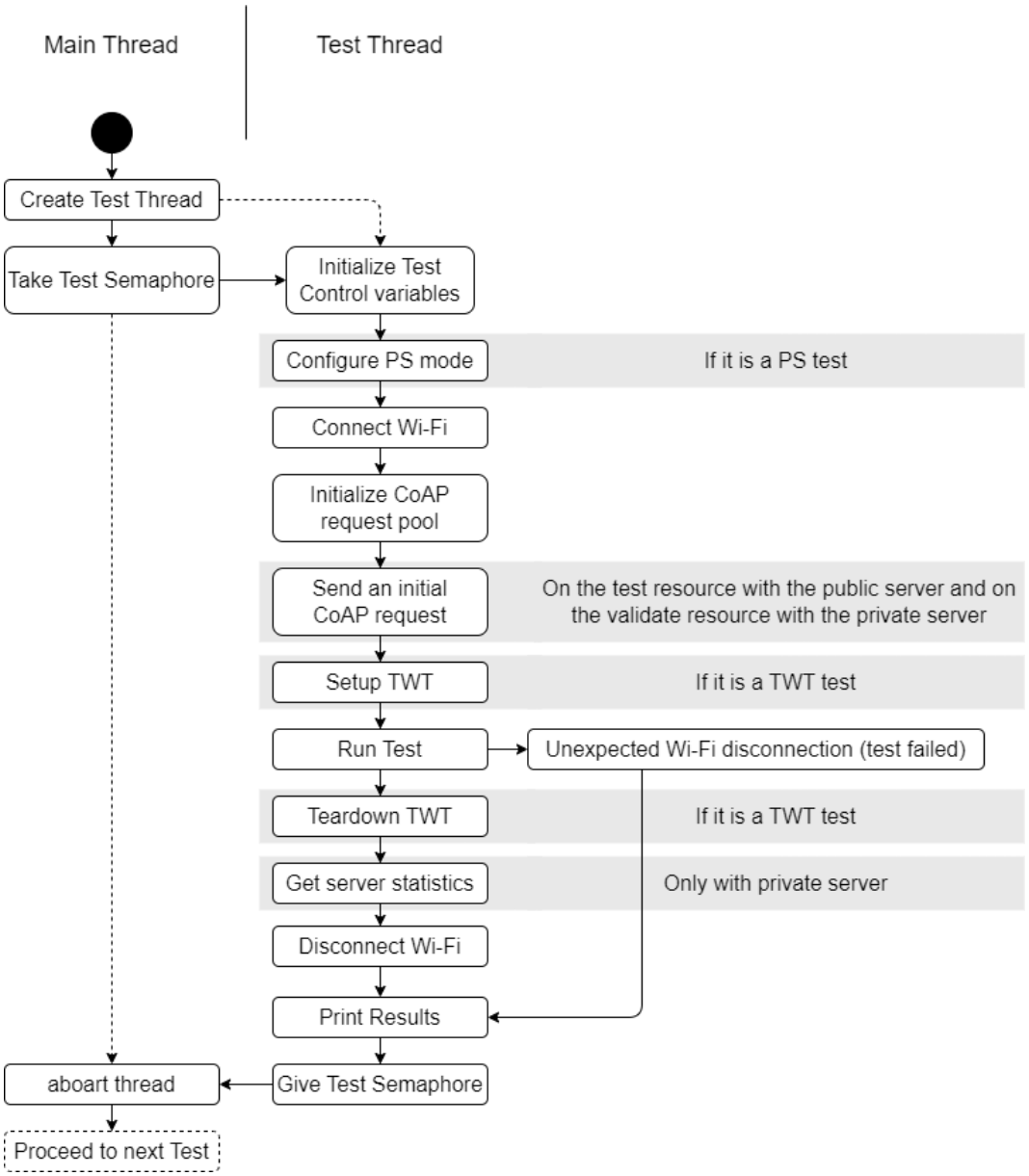


Figure 4.27: Test Execution Sequence

Each test contains two structs: *test_monitor* and *test_control*. These structs contain the variables necessary to control and monitor the tests, such as counters for the requests and responses. The tests also have a callback function that is invoked if the Wi-Fi connection is disrupted unexpectedly. If this happens, the test is stopped, and the testbed proceeds to the next one if it can reconnect.

The results are printed at the end of the test in a JSON format. The *test_report* component is used to print the results. It includes the testbed configuration, the test settings, and the test results.

Sensor Use Case

The sensor use case is described in section 3.3.1. The figure below shows one iteration of the TWT test.

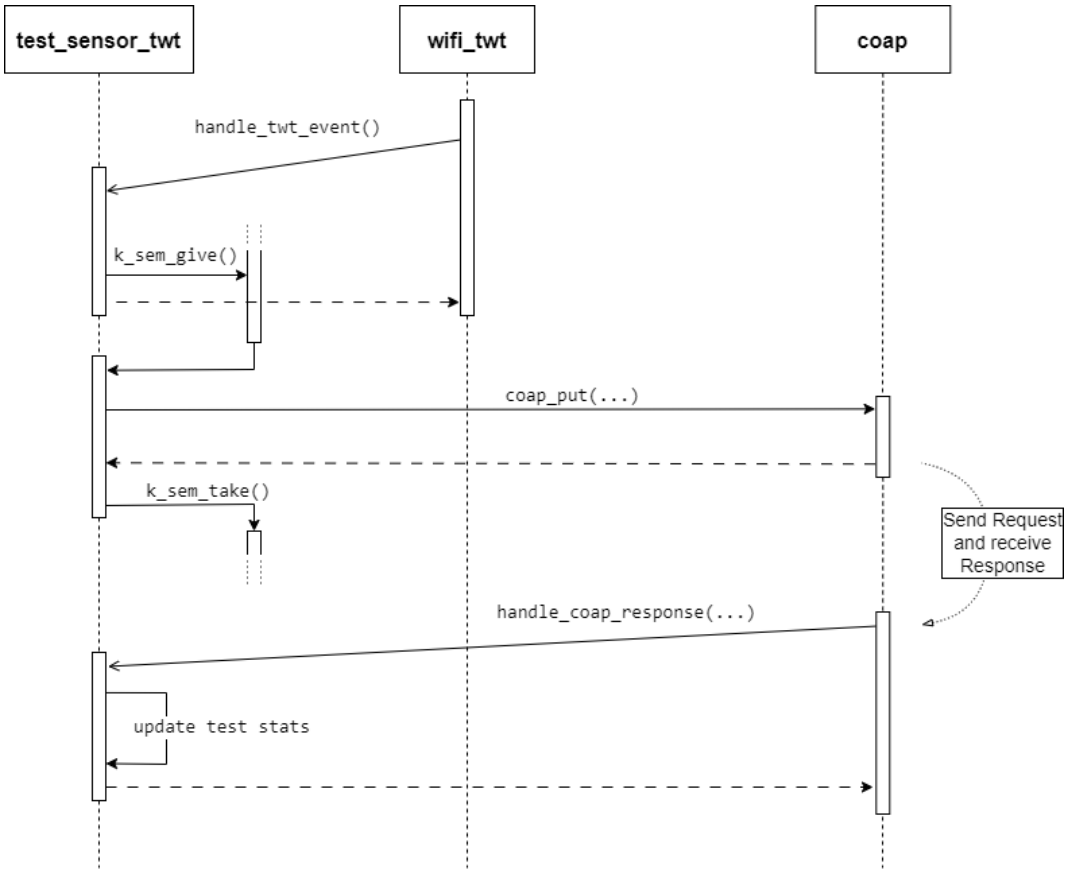


Figure 4.28: Sensor Test Sequence Diagram

The sensor TWT use case uses the TWT callback, invoked a short time ahead of the TWT session (see section 4.3.3). The callback gives a semaphore, which unblocks the test thread. In the PS tests, the semaphore is given using a timer instead of the TWT callback. The test thread sends a CoAP request using the CoAP component (see section 4.3.6). The statistics are updated in the response callback.

The sensor tests provide the following results.

PS - Public Server	**TWT - Public Server**
- Number of requests sent - Number of responses received - Average latency (ms)	- Number of requests sent - Number of responses received - Average latency (s) - Recovery count (if recovery is enabled) - Latency histogram
PS - Private Server	**TWT - Private Server**
- Number of requests sent - Number of requests received on the server - Number of responses received - Number of requests lost - Number of responses lost - Average latency (ms)	- Number of requests sent - Number of requests received on the server - Number of responses received - Number of requests lost - Number of responses lost - Average latency (s) - Recovery count (if recovery is enabled) - Latency histogram

Table 4.1: Sensor Use Case Test Outputs

Large Packet Test Case

The large packet test case is described in section 3.3.2. The test follows the same sequence as in the sensor tests (figure 4.28). The only difference is the size of the requests. The outputs are the same as those of the sensor use case (table 4.1), except for the recovery count, as the recovery mechanism is only implemented for the sensor use case.

Multi Packet Use Case

The multi-packet use case is described in section 3.3.3. The test follows the same sequence as in the sensor tests (figure 4.28). The only difference is that at every iteration, many requests are sent. The outputs are the same as those of the sensor use case (table 4.1), except for the recovery count, as the recovery mechanism is only implemented for the sensor use case.

Actuator Use Case

The actuator use case is described in section 3.3.4. The figure below shows one iteration of the TWT test.

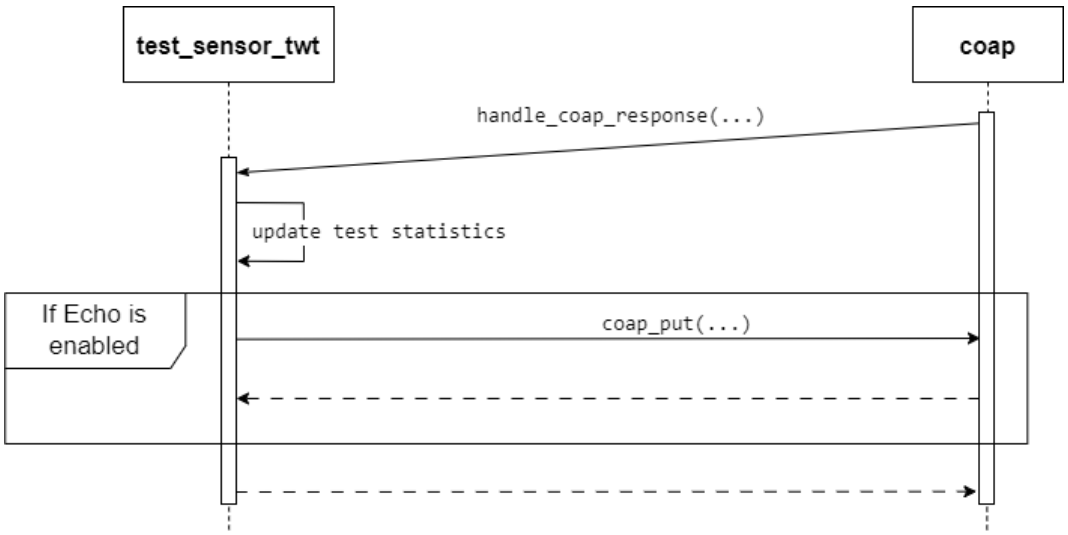


Figure 4.29: Actuator Test Sequence Diagram

When a notification is received, the response callback is invoked, and the test updates its statistics. If the echo feature is activated, the test copies the payload and sends it back to the server using the echo dedicated resource.

The actuator tests provide the following outputs.

PS - Public Server	**TWT - Public Server**
- Number of notifications received	- Number of notifications received
PS - Private Server	**TWT - Private Server**

- Number of notifications sent
- Number of notifications received
- If echo is enabled:
  - Echoes received on the server
  - Average latency (ms)
	- Number of notifications sent
	- Number of notifications received
	- If echo is enabled:
	  - Echoes received on the server
	  - Average latency (s)
	  - Latency histogram

Table 4.2: Actuator Use Case Test Outputs

4.3.10 Test Report

The *test_report* component is used by the tests to print the results at the end of the test. The listing below shows an example of a test result.

```
{
"test_title":"Sensor Use Case - TWT",
"testbed_setup": {
    "CoAP_Server":"californium.eclipseprojects.io",
    "DTLS":"Enabled",
    "DTLS_Peer_Verification":"Enabled",
    "DTLS_Connection_ID":"Enabled",
    "DTLS_Ciphersuite":"TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256",
    "IP_Protocol":"IPv4",
    "Wi-Fi_TWT_Implicit": true,
    "Wi-Fi_TWT_Announced": true,
    "Wi-Fi_TWT_Trigger": false,
    "Wi-Fi_PS_Listen_Interval": 10
},
"test_setup": {
    "Iterations": 1000,
    "Negotiated_TWT_Interval":"10 s",
    "Negotiated_TWT_Wake_Interval":"32 ms",
    "Recovery":"Disabled"
},
"results": {
    "Requests_Sent": 1000,
    "Responses_Received": 998,
    "Average_Latency": "23 s"
},
"latency_histogram": [
    { "latency": 0, "count": 0 },
    { "latency": 10, "count": 188 },
    { "latency": 20, "count": 368 },
    { "latency": 30, "count": 307 },
    { "latency": 40, "count": 135 },
    ......
    { "latency": 190, "count": 0 },
    { "latency": "lost", "count": 15 }
]
}
```

Listing 4.2: Test Result Example

Python scripts are used to parse the logs and extract the results of all the tests. The scripts are in the *scripts* folder of the testbed repository (see Appendix A). The *extractresults.py* script extracts the results from all the logs and generates a JSON file that contains the results of all the tests present in the logs. The *testreport.py* script generates a *.pdf* report that includes results and graphs from the JSON that contains all the results. The *histogram.py* file generates a histogram in a *.png* file from one test.

Many test reports generated by these scripts are available in the appendix (E, F, ...).

4.3.11 Test Global

The *test_global* component contains global definitions used by all the tests. It also includes the test thread stack and the test semaphore, which ensure that tests are executed sequentially, one after the other. The thread stack and test semaphore are declared as *extern* in the header file, allowing them to be accessed by all the tests.

4.3.12 Test Runner

The *test_runner* component runs the tests set in the configuration (see section 4.2). It allows different tests to be done without having to modify the code. The *test_runner* component only contains the *run_test* function. The function sets and runs the tests according to the configuration.

With this method, tests are statically configured, and only a limited number of tests are available (max 4 Sensor PS tests, max 4 Sensor TWT tests, ...). To run more tests in one build, it is also possible not to use the *test_runner* and instead directly code the tests in the *main.c*. The listing below shows the test invocation in the main with both options.

```
//this code run the tests defined in the configuration
//comment the following line to run custom tests instead
#define TESTRUNNER

#ifdef TESTRUNNER
run_tests();
#else
//example - run 12 sensor tests with different durations
//          of 8 to 64 ms and intervals of 5 to 20 seconds
for(int duration = 8; duration <= 64; duration *= 2)
{
    for(int interval = 5000; interval <= 20000; interval *= 2)
    {
        static int i = 0;
        struct test_sensor_twt_settings test_settings = {
                        .iterations = 1000,
                        .twt_interval = interval,
                        .twt_wake_interval = duration,
                        .test_id = i++,
                        .recover = false,
                        .wake_ahead_ms = 100,

        };
        test_sensor_twt(&test_settings);
    }
}
#endif //TESTRUNNER
```

Listing 4.3: Test Invocation

4.3.13 Main

The main function (in *main.c*) initializes the testbed.

First, it checks if there are inconsistencies in the configuration, such as a wrong port being used or the testbed configured to use the private server with the URL of the public server. Warnings are printed if inconsistencies are detected.

Then, it initializes and connects the Wi-Fi components. After that, the CoAP component is initialized. If the testbed is configured to use the private server, it sends a request to the *validate* resource to check if everything is correctly configured. After that, the Wi-Fi is disconnected, and the tests are started.

5 Testbed Server Implementation

This chapter describes the implementation of the testbed server. It explains the server behavior and implementation, including a general class diagram and descriptions of the different resources implemented. The chapter ends with a summary of the benefits and drawbacks of using the private server compared to the public server.

Contents

5.1 Testbed Server **92**
5.2 Class Diagram **92**
5.3 Resources **93**
5.3.1 ValidateResource 93
5.3.2 StatResource 93
5.3.3 ActuatorStatResource 93
5.3.4 SensorResource 93
5.3.5 ActuatorResource 93
5.3.6 ActuatorEchoResource 94
5.3.7 Large Packet Resources 94
5.4 Benefits of the Private Server **95**
5.5 Drawbacks of the Private Server **95**

5.1 Testbed Server

The testbed private server was developed with *Java* and the *Californium* library [41]. The goal of the private server is to give more information and to implement more advanced tests than what is possible with the public server. The tests are described in section 3.3. The server computes statistics that are retrieved by the client using dedicated resources. Moreover, the server provides logs, which are valuable for analyses.

A *GitHub* repository that contains the source code of the testbed server is available in Appendix A.

5.2 Class Diagram

The figure below shows the class diagram of the server *Java* application.

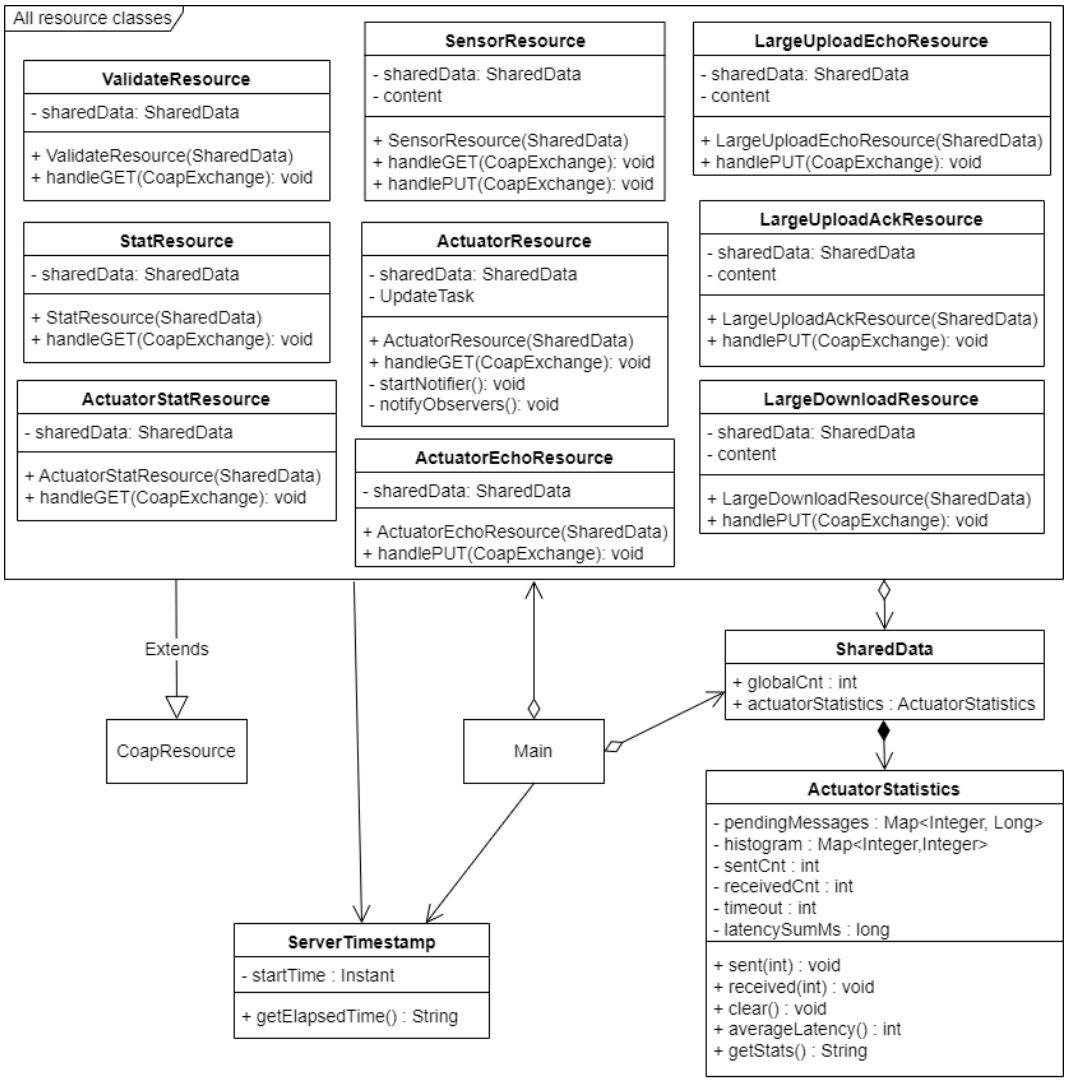


Figure 5.1: Testbed Server Class Diagram

All resource classes have the same relations. They all extend the *CoapResource* class from the *Californium* library. An instance of the *SharedData* class is shared between all the resources. It contains a global counter and a data structure used to store the actuator statistics. The *ServerTimestamp* class provides a function that returns the time since the server was started. It is used in the logs.

5.3 Resources

5.3.1 ValidateResource

The validate resource is used at the beginning of each test. It is used to check that the testbed is correctly configured. It handles GET requests and responds with a "valid" payload to confirm the communication works. The statistics stored on the server are reset when the validate resource is called.

5.3.2 StatResource

The stat resource is used at the end of the tests. It handles GET requests and responds with the value of the *globalCnt* variable of the *SharedData* class.

5.3.3 ActuatorStatResource

The stat resource is used at the end of the actuator tests. It handles GET requests and responds with the *actuatorStatistics* data of the *SharedData* class. It consists of a latency histogram and a latency average value.

5.3.4 SensorResource

The sensor and multi-packet tests use the sensor resource. When a PUT request is received, its payload is echoed in the response payload. The *globalCnt* variable of the *SharedData* class is incremented every time a PUT request is received on the sensor resource. The response is sent after a short delay to simulate the latency of a server on the Internet. The delay is a random value between 20 ms and 50 ms.

5.3.5 ActuatorResource

The actuator test uses the actuator resource. It handles GET requests using the observe option. The GET request's payload contains the minimum and maximum interval between the notifications. The *updateTask* calls the *notifyObservers* function sporadically with a random interval within the minimum and maximum specified in the initial GET request payload. The *globalCnt* variable of the *SharedData* class is incremented every time a notification is sent and the *ActuatorStatisics*'s *sent* function is called with the ID of the notification. [1]

[1] The ID of a notification is simply a number incremented at each iteration

5.3.6 ActuatorEchoResource

The actuator test also uses the actuator echo resource. The client echoes the notifications received in a GET request to the actuator echo resource. These requests are not responded to. When received, the *ActuatorStatistics*'s *received* function is called with the ID of the echo. Then, the *ActuatorStatistic* class measures the latency of the echo (total round trip time) and adds the value to the histogram.

5.3.7 Large Packet Resources

The large packet tests can use three configurations.

With a large request and a large response, it uses the *LargeUploadEchoResource*. The resource echoes the payload in the response.

For a large request and a small response, the *LargeUploadAckResource* is used. The resource only echoes the first chars of the payload that contains the ID of the request (The ID is a number incremented at each request).

For a small request and a large response, the *LargeDownloadResource* is used. In this configuration, the response payload size is indicated in the request.

These three resources use PUT requests and increment the *globalCnt* variable in the *SharedData* class. The response is sent after a short delay to simulate the latency of a server on the Internet. The delay is a random value between 20 ms and 50 ms.

5.4 Benefits of the Private Server

For the sensor, large-packet, and multi-packet tests, the private server provides the number of requests received on the server. Jointly with the number of requests sent and responses received, it indicates the number of lost requests and the number of lost responses, isolating the statistics of lost packets for the uplink and downlink traffic. On the public server, if a response is not received, it is impossible to know if the request or the response was lost.

For the large-packet tests, it adds the possibility to have three different configurations (large requests and large responses / large requests and small responses / small requests and large responses). On the public server, the responses do not contain payloads, so the only possible configuration is large requests and small responses.

The public server version is very limited for the actuator tests because the number of notifications sent is unknown. Moreover, the private server tests can include responses (echo) to the notifications and compute the total round trip time.

5.5 Drawbacks of the Private Server

The main drawback of the private server setup is that it is not located on the Internet but is very close to the client, with only two hops between them. This makes the setup less representative of a commercial application. To address this, a specific network topology is used (see Section 3.5), and latency is artificially introduced to simulate the response time of a server on the Internet. Nevertheless, this approach does not fully replicate the complexities of an actual Internet connection.

6 | Analysis

TODO - chapter intro

Contents

6.1 Limitations of employing TWT in practical applications 98
6.1.1 Limitations imposed by TCP 98
6.1.2 Limitations imposed by ARP 100
6.1.3 Limitations imposed by WPA Group Key Rotation 100
6.2 ARP Analysis 101
6.2.1 First Test 101
6.2.2 Second Test 102
6.2.3 Test Conclusion 103
6.2.4 Gratuitous ARP 103
6.3 NDP Analysis 104
6.3.1 Startup 104
6.3.2 Regular Traffic 105
6.3.3 Test Conclusion 105
6.4 Test Results Analysis 106
6.4.1 Sensor Use Case 106
6.4.2 Large Packet Test Case 118
6.4.3 Multi Packet Use Case 122
6.4.4 Actuator Use Case 125

6.1 Limitations of employing TWT in practical applications

TWT allows for high energy savings, permitting the activation of the Wi-Fi RF circuit only for short periods. However, it introduces many limitations as Wi-Fi and the protocols running on top of it were designed for responsive and always-connected systems. When TWT is activated, broadcast and multicast downlink traffic reception is not guaranteed, and frames are presumably lost because these frames are not buffered to be transmitted during the service period. Moreover, TWT is meant to operate with long intervals of many seconds, which results in significant link layer latency and, thus, imposes major limitations when using typical communication stack implementations. it were designed for responsive and always-connected systems. When TWT is activated, broadcast and multicast downlink traffic reception is not guaranteed, and frames are presumably lost because these frames are not buffered to be transmitted during the service period. Moreover, TWT is meant to operate with long intervals of many seconds, which results in significant link layer latency and, thus, imposes major limitations when using typical communication stack implementations.

6.1.1 Limitations imposed by TCP

The figure below shows a typical protocol stack of an IoT device using Wi-Fi.

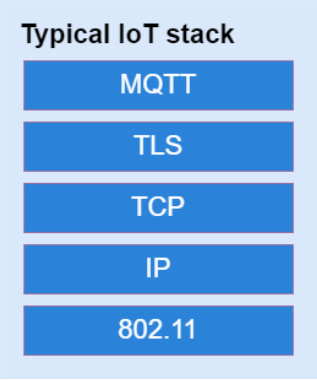


Figure 6.1: Typical protocol stack of an IoT device

MQTT is one of the most widely used protocols for IoT devices. Its simplicity, efficiency, and low bandwidth requirements make it particularly suited for constrained environments. However, TWT imposes significant constraints that make MQTT conflicting with TWT, precisely due to its reliance on TCP.

TCP is connection-oriented and establishes a reliable connection. This approach works well, but it requires a continuous flow. TWT interruptions may result in connection timeouts, retransmissions, or session termination.

Additionally, the repeated interruptions and the latency caused by TWT may cause TCP to misinterpret the situation as network congestion, further degrading performance.

In summary, TCP creates a fundamental mismatch with TWT. Addressing these limitations may require protocol-level modifications or alternative transport-layer solutions better aligned with TWT's low-power operation. Therefore, TCP-based communication stacks are considered incompatible with TWT.

6.1.2 Limitations imposed by ARP

In IPv4, routers use the ARP protocol to resolve IP addresses. The problem is that the standard implementation of the ARP protocol uses broadcast frames, which are typically not received under TWT. However, downlink broadcast frames can theoretically be avoided since the STA IP address typically does not change. In the worst case, the IP address could change when the DHCP lease time expires, usually once a day. In this case, we could imagine teardown TWT for a short time. Thus, the ARP problem could be resolved in theory. However, the AP would need to implement a "smart" ARP method for this solution to work. ARP is analyzed in the ARP Analysis section.

Another solution to the ARP problem is to use IPv6 instead of IPv4, which does not use ARP but NDP. NDP protocol is analyzed in the NDP Analysis section.

6.1.3 Limitations imposed by WPA Group Key Rotation

6.2 ARP Analysis

The testbench was used to test the ARP behavior. Two tests were made. These tests aim to understand how the AP implements ARP and see if it suits TWT usage. Note that the AP was rebooted before each test.

6.2.1 First Test

The first test used a sensor use case with no TWT and no power save. The STA sends a packet every five seconds and receives a serverresponse for each packet. The setup uses CoAP/UDP to send the packets, and the IP protocol used is IPv4. No DTLS security is used for this test, as it does not impact ARP.

The table below shows the first frames exchanged after the connection. For readability, duplicate frames were removed, and addresses were replaced with names. The raw logs are available in Appendix C.

Time	Source	Destination	Proto.	Info
25.008	AP_MAC	Broadcast	ARP	Who has STA_IP? Tell AP_IP
29.647	STA_MAC	Broadcast	ARP	Who has AP_IP? Tell STA_IP
29.650	AP_MAC	STA_MAC	ARP	AP_IP is at AP_MAC
29.654	STA_IP	SERV_IP	CoAP	CoAP request
29.658	SERV_IP	STA_IP	CoAP	CoAP response
33.940	AP_MAC	STA_MAC	ARP	Who has STA_IP? Tell AP_IP
33.947	STA_MAC	AP_MAC	ARP	STA_IP is at STA_MAC
39.665	STA_IP	SERV_IP	CoAP	CoAP request
39.668	SERV_IP	STA_IP	CoAP	CoAP response
44.665	STA_IP	SERV_IP	CoAP	CoAP request
44.668	SERV_IP	STA_IP	CoAP	CoAP response

Table 6.1: ARP and CoAP frames exchanged after the connection

These logs show the first ARP exchanges between the STA and AP. The first ARP frame is the AP asking for the MAC address associated with the STA IP address. This request is a broadcast frame, and the AP asks that the response be sent to its IP address. The STA needs to reply to this request, but it needs to know the MAC associated with the AP IP address.

So, the STA sends a request asking for the MAC address associated with the AP's IP address. This request is a broadcast frame, and the STA asks to send the response to its IP address. As the source of this request is the STA's MAC address, the AP can associate the STA's IP and MAC addresses without receiving the reply for its first request. At this point, it replies to the STA's request with a unicast frame, and both devices know which IP address is associated with which MAC address.

Notice that after the first CoAP exchange, the AP redoes its first request, which has never received an explicit response, to confirm it deduced the correct MAC address for the STA IP address. However, this time, the request is a unicast frame.

The interesting element about this test is that after the first exchange, the system has been working for hours without any other ARP message. This indicates the AP may employ the CoAP traffic to update its ARP cache. The second test aims to prove this hypothesis.

6.2.2 Second Test

The second test uses the same setup as the first one with only one modification. After sending the first CoAP packet, no more packets are sent for half an hour, and then, the traffic resumes. This test forces the ARP cache timeout to expire and the AP to reask for the STA's IP address.

The table below shows the first frames exchanged after the traffic resumed. For readability, address were replaced with names. The raw logs are available in Appendix C.

Time	Source	Destination	Proto.	Info
1846.112	STA_IP	SERV_IP	CoAP	Request
1846.118	SERV_IP	STA_IP	CoAP	Response
1851.112	STA_IP	SERV_IP	CoAP	Request
1851.115	SERV_IP	STA_IP	CoAP	Response
1851.273	AP_MAC	STA_MAC	ARP	Who has STA_IP? Tell AP_IP
1851.277	STA_MAC	AP_MAC	ARP	STA_IP is at STA_MAC
1856.112	STA_IP	SERV_IP	CoAP	Request
1856.115	SERV_IP	STA_IP	CoAP	Response

Table 6.2: ARP and CoAP frames exchanged after the traffic resumed

The test shows that after half an hour, the AP kept the STA's IP address in its ARP cache since two CoAP requests and responses could be exchanged. After the second CoAP exchange, the AP sent an ARP request to the STA in a unicast frame to confirm it had the correct addresses.

6.2.3 Test Conclusion

The two tests prove that the AP uses a "smart" implementation of ARP that avoids sending broadcast ARP frames once the AP knows the STA's MAC address. The only broadcast ARP frames are the first ones. The test also shows that unicast ARP is not due to TWT. In theory, it can also be helpful in other PS modes, such as Listen Interval.

In a TWT scenario, TWT can be set once the first ARP exchange is done, and then, the ARP cache will automatically refresh with the regular traffic. If there is no traffic for a long time, the AP will send an ARP request in a unicast frame, which will be buffered and received by the STA during the next TWT session.

The ARP analysis proves that the ARP limitation is not a real blocker when using TWT. Moreover, if an AP does not implement a "smart" ARP method, which is unlikely because APs implementing TWT are Wi-Fi 6 and thus modern, a recovery mechanism can be implemented to detect when the STA is not receiving data and teardown TWT a short time for an ARP exchange. Another solution would be to use gratuitous ARP.

6.2.4 Gratuitous ARP

If an AP does not implement a "smart" ARP method, gratuitous ARP is another solution to avoid the AP losing the STA's MAC address. With gratuitous ARP, the STA periodically sends an ARP response with its IP and MAC addresses. This response permits the AP never to lose the STA's MAC address. However, the AP must be able to process these frames. [1]

[1] This could not been tested because the AP used for the tests employs the IP traffic to update its ARP cache. Nevertheless, when gratuitous ARP is enabled on the STA, the gratuitous ARP frames are visible when monitoring the network.

6.3 NDP Analysis

The testbench was used to test the NDP behavior. Like the ARP tests, NDP was tested with a sensor use case, no TWT, and no power save. The STA sends a packet every five seconds and receives a response for each packet. The setup uses CoAP to send the packets, and the IP protocol used is IPv6. No DTLS security is used for this test, as it does not impact ARP.

6.3.1 Startup

The table below shows the first frames exchanged after the connection. For readability, addresses were replaced with names. The raw logs are available in Appendix D.

Time	Source	Destination	Proto.	Info
15.260	STA_IP_loc	multicast	ICMPv6	RS from STA_MAC
15.265	AP_IP_loc	multicast	ICMPv6	RA from AP_MAC
15.272	STA_IP_loc	multicast	ICMPv6	NS for AP_IP_loc from STA_MAC
15.276	AP_IP_loc	STA_IP_loc	ICMPv6	NA AP_IP_loc is at AP_MAC
15.278	::	multicast	ICMPv6	NS for STA_IP_glo
16.842	STA_IP_glo	multicast	ICMPv6	NS for SERV_IP_glo from STA_IP_glo
16.856	AP_IP_glo	STA_IP_glo	ICMPv6	NA SERV_IP_glo is at AP_MAC
16.860	STA_IP_glo	SERV_IP_glo	CoAP	CoAP request
16.863	SERV_IP_glo	STA_IP_glo	CoAP	CoAP response

Table 6.3: NDP and CoAP frames exchanged after the connection

The first NDP exchange is the router discovery. Once associated, the STA sends a Router Solicitation (RS), and the AP responds with a Router Advertisement (RA). These frames are both multicast.

Then, the STA sends a Neighbor Solicitation (NS) for the AP's local IP address in a multicast frame. The AP replies with the Neighbor Advertisement (NA) in a unicast frame directed to the STA's local IP address. At this point, the AP and the STA have associated the local IP addresses with the MAC addresses.

The following frame is an NS for the STA's global IP address. The STA sends this frame to check if its global address is unique, as with IPv6, the devices automatically generate addresses using SLAAC. This frame is multicast because it is directed to all devices. There was no answer, so the STA confirmed that its global IP address was valid.

The next NDP exchange is not part of the initialization but is sent just before the first CoAP packets. It is an NS for the server IP address in a multicast frame. The AP responds to it because it routes packets outside the subnet, so they must be destined to the AP at the link level. The response, which is the critical aspect because it is downlink, is sent in a unicast frame.

After these NDP exchanges, the CoAP packets can be sent using the global addresses.

6.3.2 Regular Traffic

Once the device is connected, four different NDP exchanges are regularly made. The tables below show these four exchanges.

Source	Destination	Proto.	Info
AP_IP_local	STA_IP_local	ICMPv6	NS for STA_IP_local from AP_MAC
STA_IP_local	AP_IP_local	ICMPv6	NA STA_IP_local is at STA_MAC

Table 6.4: AP asks for STA's local IP address

Source	Destination	Proto.	Info
AP_IP_local	STA_IP_global	ICMPv6	NS for STA_IP_global from AP_MAC
STA_IP_global	AP_IP_local	ICMPv6	NA STA_IP_global is at STA_MAC

Table 6.5: AP asks for STA's global IP address

Source	Destination	Proto.	Info
STA_IP_global	multicast	ICMPv6	NS for SERV_IP_global from STA_MAC
AP_IP_global	STA_IP_global	ICMPv6	NA SERV_IP_global is at AP_MAC

Table 6.6: STA asks for servers's global IP address

Source	Destination	Proto.	Info
STA_IP_local	multicast	ICMPv6	NS for AP_IP_local from STA_MAC
AP_IP_local	STA_IP_local	ICMPv6	NA AP_IP_global is at AP_MAC

Table 6.7: STA asks for AP's local IP address

These exchanges permit refreshing the cache of STA and AP for the local and global IP addresses. Unlike IPv4 and ARP, the CoAP traffic does not update the cache. All the frames are unicast except the NS sent by the STA.

6.3.3 Test Conclusion

This test shows that IPv6 is suitable for TWT, and NDP will not be a problem. The only multicast downlink frame is the Router Advertisement, which is done at startup when TWT is not set up.

6.4 Test Results Analysis

The thesis aims to evaluate the behavior of the STA and communication when the TWT feature is used in commercial application scenarios. Therefore, the tests were performed in an ordinary environment, with many other Wi-Fi networks nearby. No tests were conducted in an RF-isolated chamber.

6.4.1 Sensor Use Case

This section presents the results of the sensor use case tests. The use case is described in section 3.3.1. The tests were performed with various configurations. The raw test results are available in Appendix E.

Packet Loss and Response Time

The first element analyzed is the impact of TWT on communication. The performance metrics are the packet losses and the response time. Packet loss is theoretically not expected to increase. However, the constraints imposed by TWT create tighter conditions for communication, which may lead to occasional failures due to the system's limited flexibility. On the other hand, response time is expected to increase significantly due to TWT's inherent design, which requires devices to wait for scheduled transmission slots, introducing significant additional latency compared to conventional Wi-Fi communication.

The test was performed using the following testbed configuration:

- Public server (Californium)
- DTLS Enabled with Peer Verification, Connection ID, and TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256 cipher suite
- IPv4
- TWT implicit, announced, and trigger disabled

The test was performed multiple times with the following test settings:

- 1000 iterations
- TWT session durations from 8 to 64 ms
- TWT intervals from 5 to 20 seconds
- Recovery mode disabled

The test was also performed in PS mode (DTIM/Legacy) to serve as a reference for packet loss. However, response times are not comparable due to the significantly higher values in TWT, making the comparison with PS mode meaningless.

The first metric analyzed is the packet loss. The graph below shows the proportion of lost packets for each test. These results show combined uplink and downlink packet losses.

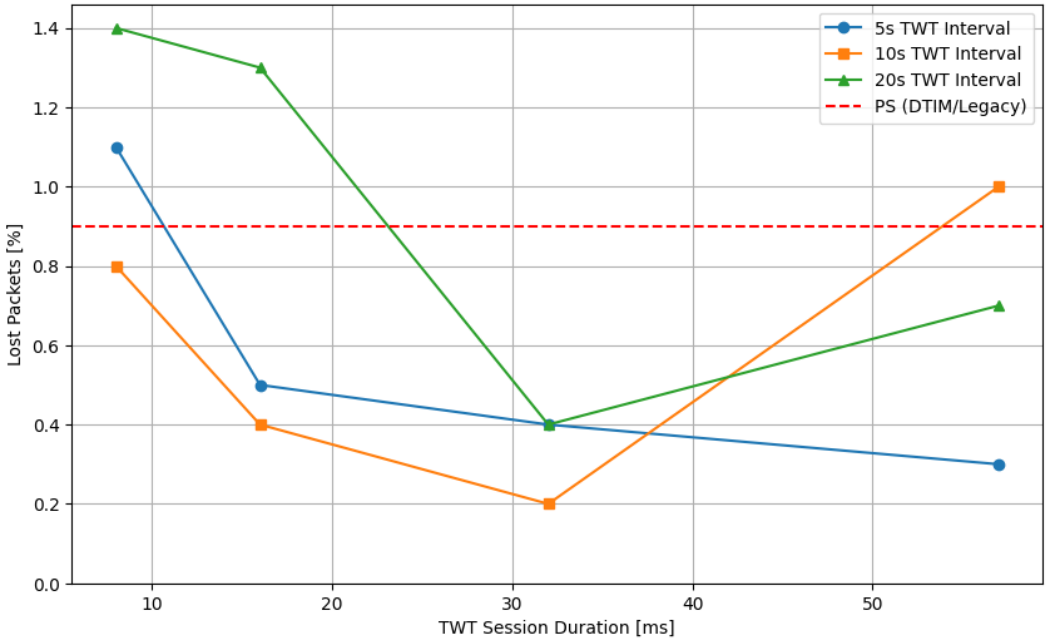


Figure 6.2: Packet loss for the sensor use case

The graph shows that the packet loss tends to decrease with longer sessions. However, it is essential to note that very few packets were lost during all tests. All tests have more than 98 % of their requests that are responded to, which is excellent performance.

Moreover, the PS test issued 0.9 % of unresponded requests. The results of the TWT tests show similar proportions of lost packets, indicating that TWT does not significantly impact performance in terms of lost packets in this test configuration.

It is important to note that all responded requests are counted as successfully responded, regardless of the response time. The testbed uses very long request timeouts, which can be considered effectively infinite. This highlights the advantage of using CoAP, which offers configurable timeout, retransmission, and backoff parameters.

The next metric analyzed is the latency. As the end-to-end latency cannot be measured because this would require a precise common time reference between the client and the server, the testbed measures the response time. The response time corresponds to the elapsed time between the sending of the request and the receiving of the response, as detailed in the Testbed Design chapter (3.3.1). In PS Mode (DTIM/Legacy), the average response time measured on the public server (Californium) is 139 ms. In the TWT test, a request is sent at every TWT session, and the response is expected to be received in the next TWT session. The testbed monitors the response time of each request and computes the average response time and a histogram.

The bin width of the histogram is set to the TWT interval, so each bar represents a TWT session. The figure below shows an example of a histogram. The testbed used a 16 ms TWT session duration and a 5 seconds TWT interval for this test. The results of all tests are available in Appendix E.

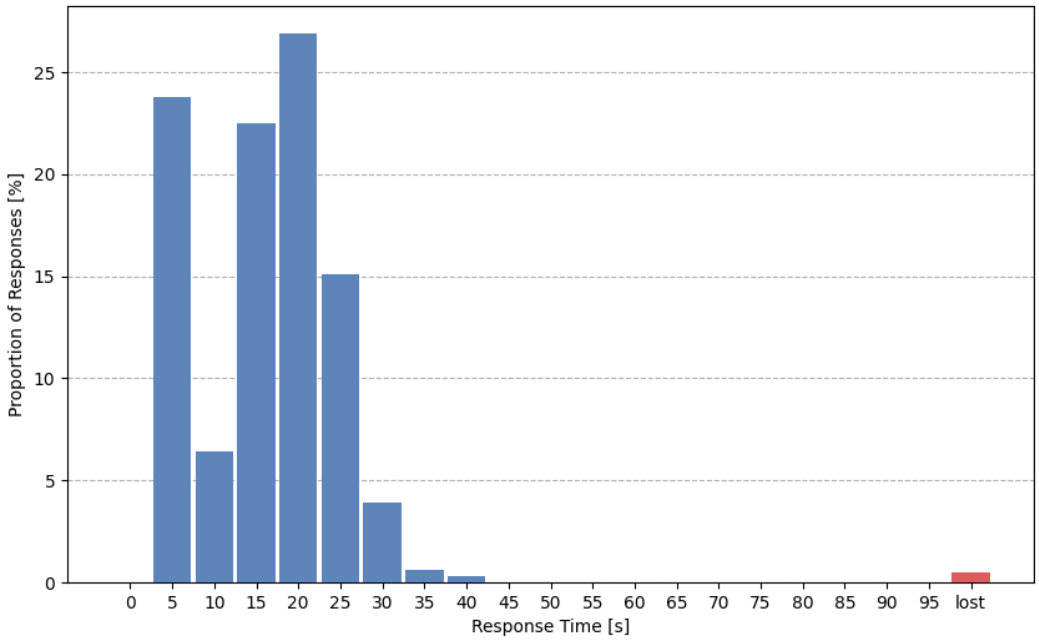


Figure 6.3: Response time histogram for a sensor use case test performed on the public server, with a 5 second TWT interval and 16 ms TWT session duration

In this test, no request was responded to within the same TWT session. 24 % of the requests were responded to after one TWT session, corresponding to the expected behavior. 6 % were responded to after 2 TWT sessions, 22 % after 3 sessions, etc.

These results show that the responses take too much time to arrive. Tests on the private server revealed similar results. Monitoring the wireless frames indicated that the uplink traffic is systematically transmitted on time, and downlink traffic is regularly transmitted after many TWT sessions. Tracking the traffic on the server side showed that the requests arrive on time on the server, and the server immediately sends the responses. This led to the conclusion that the AP regularly buffers the downlink traffic for many TWT sessions and transmits it at a later TWT session.

The graph below shows the average response time for all the tests. The response time is expressed in multiples of the TWT interval instead of seconds to allow for a meaningful comparison between tests with different TWT intervals.

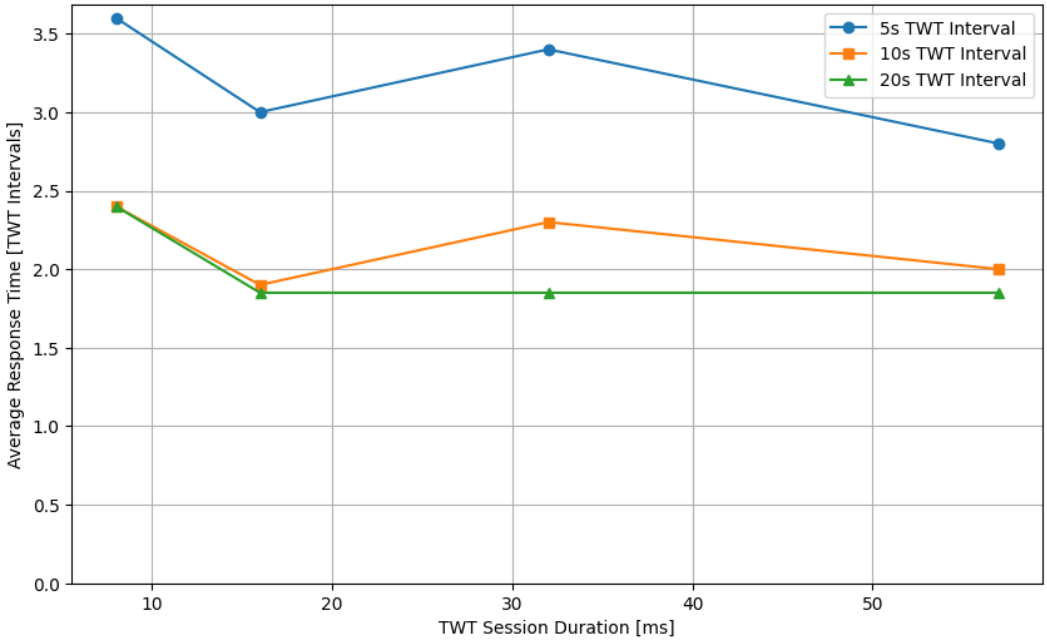


Figure 6.4: Average response time for the sensor use case

The tests with 10 and 20 seconds TWT intervals gave an average response time of around two TWT intervals. The test with a 5-second TWT interval gave a response time of around three TWT intervals.

Concerning the TWT session duration, the response time does not significantly decrease when using longer TWT sessions. A minimal decrease is visible but can be considered negligible when put in perspective with the standard deviation (see fig. 6.5).

This indicates that the problem of the AP buffering the packets is likely not directly due to contention, as the issue is still present with longer sessions. The hypothesis that the AP cannot transmit its packet because the delays imposed by the contention are longer than the TWT session duration is not valid because, in the best case, the response time falls by only 22 % when the session duration, which corresponds to the time when the AP has a chance to transmit its packet, is increased by 612 %. Furthermore, in practice, contention would never cause more than 10 ms delays.

Several approaches have been explored to address the issue of excessive response time, including changing the QoS priority in the MAC header to the maximum priority and changing the AP's aggregation and RTS threshold settings. However, none of these approaches gave better response times.

The figure below shows the response time and standard deviation for the 5-second TWT interval tests.

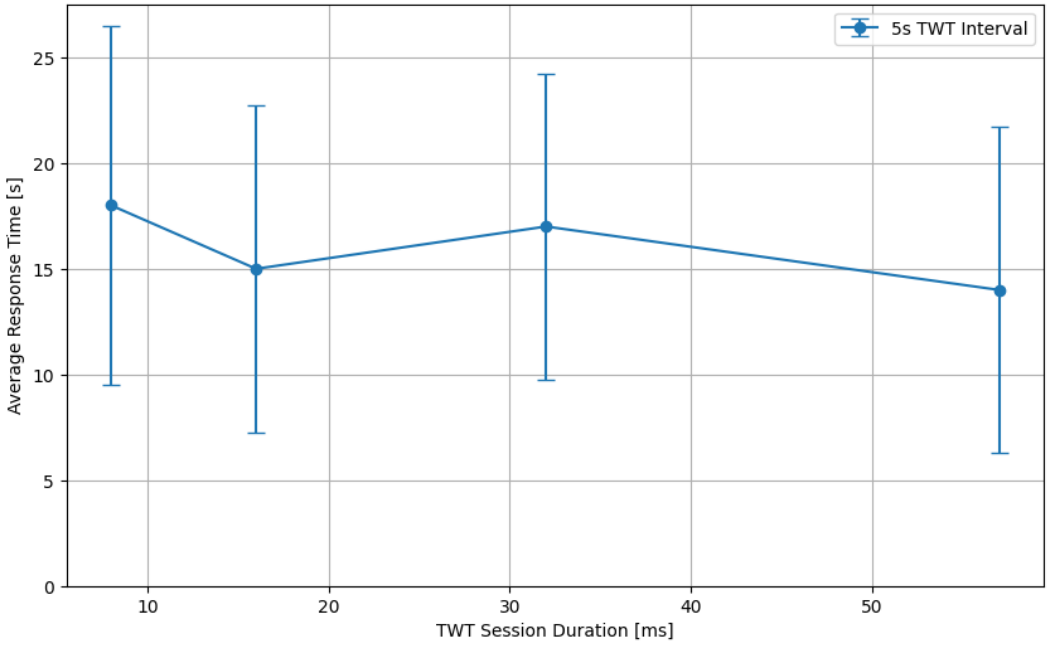


Figure 6.5: Standard deviation of the average response time for the sensor use case

The graph shows that, on average, longer TWT sessions have a slightly better response time, but individually, a response can have a short latency as well as a long latency for all session durations. The response time histogram (Figure 6.3 - corresponding to the 16 ms point in Figure 6.5) corroborates this, showing response times ranging from 1 to 8 TWT intervals.

These first tests reveal two significant findings. The first finding is positive: when TWT is employed, data transmission occurs reliably, with packet loss rates comparable to those observed in traditional power-saving modes. However, the second finding highlights a key limitation: transmission latency is higher than anticipated. The results indicate that packets may require multiple sessions to be transmitted, and given that TWT intervals are inherently extended periods, the resulting latency can become excessive. Additionally, the standard deviation of latency is considerable, with numerous outliers, leading to unpredictable transmission times. These findings suggest that while TWT is suitable for non-time-sensitive applications—where ensuring eventual data transmission is more critical than strict timing requirements—it poses challenges for real-time applications, where stable and controlled transmission times are essential.

Power Consumption

The power consumption of the nRF7002 module is measured to analyze the benefits of TWT. The figure below shows the current consumption during a TWT session.

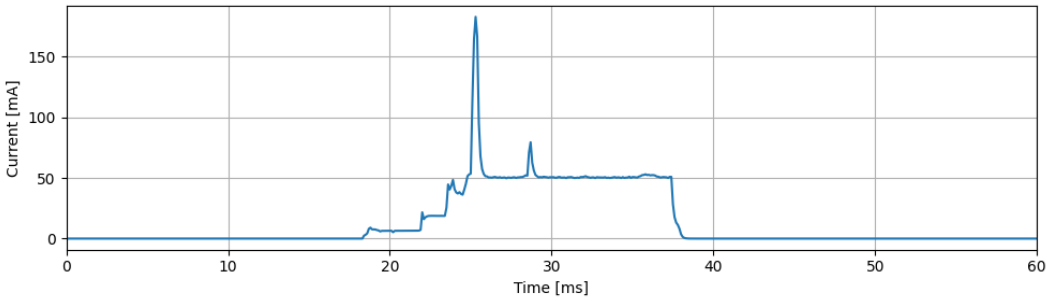


Figure 6.6: Current consumption during a TWT session

This measurement shows that during a TWT session, the current is at 51 mA with some variation and peaks at the beginning of the session.

The graph below shows the average current for each test. These measurements only include the consumption of the nRF7002 chip (Wi-Fi module). It is powered by 3.6 V. For the detailed numerical results corresponding to these measurements, please refer to the appendix (see Table E.1).

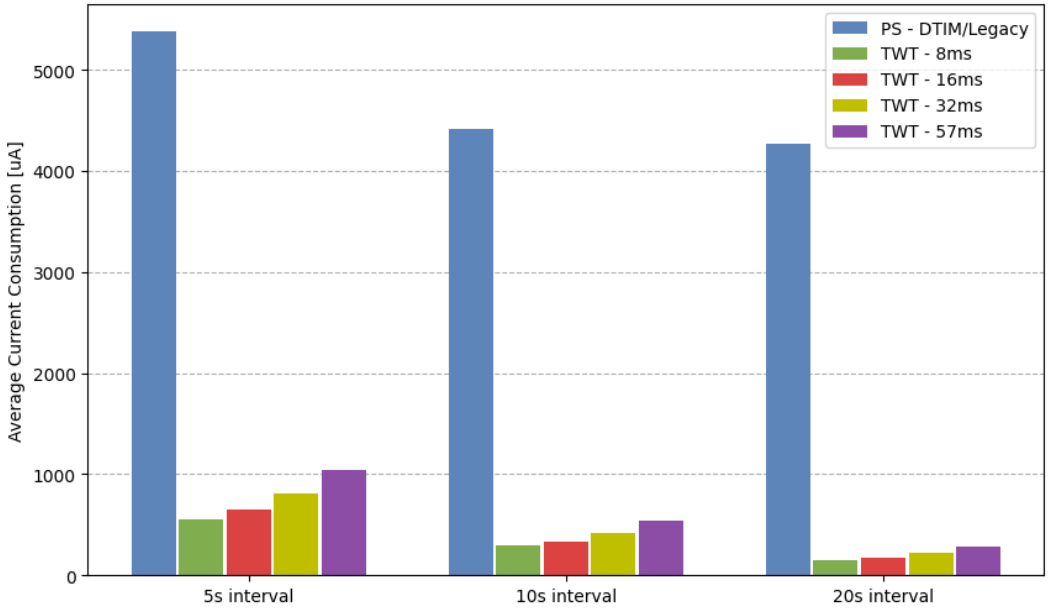


Figure 6.7: Average current for sensor use case

These results demonstrate a significant improvement compared to PS mode (DTIM/Legacy). They also show that, in PS-DTIM mode, the decrease in power consumption is non-negligible when the transmission frequency is reduced. Nevertheless, there is still a big room for improvement, which TWT exploits.

The graph below shows the same results but without the PS-DTIM for a better analysis of the TWT measurements.

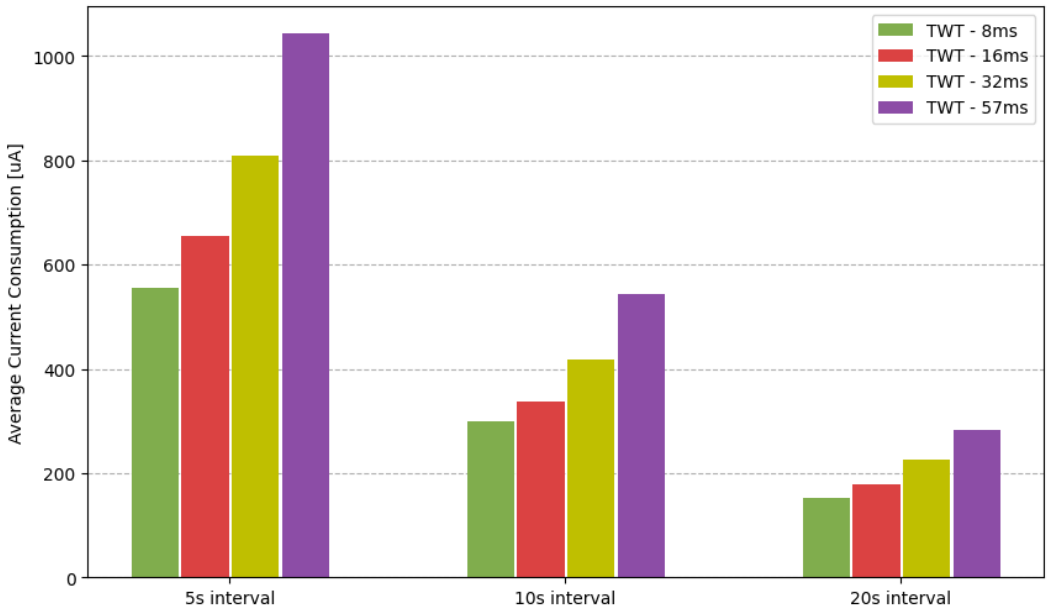


Figure 6.8: Average current for sensor use case

The results show that longer intervals and shorter sessions lead to lower power consumption. The results also show that the power consumption is not proportional to the session duration. For example, the 8 ms and 16 ms tests have very similar average currents. This is due to overheads.

The idealistic average current consumption can be calculated based on the active and sleep durations, as well as their corresponding current levels. The measured current when the nRF7002 module is active (excluding peaks) is 51 mA, while the current during the sleep state is 15 µA. The following calculation estimates the ideal average current for a TWT interval of 5 seconds and a TWT session duration of 8 ms.

$$
\begin{aligned}
I_{\text{ideal}} &= I_{\text{active}} \cdot \frac{T_{\text{session}}}{T_{\text{interval}}} + I_{\text{sleep}} \cdot \frac{T_{\text{interval}} - T_{\text{session}}}{T_{\text{interval}}} \\
&= 51\,000\,\mu\text{A} \cdot \frac{8\,\text{ms}}{5000\,\text{ms}} + 15\,\mu\text{A} \cdot \frac{5000\,\text{ms} - 8\,\text{ms}}{5000\,\text{ms}} \\
&= 96.58\,\mu\text{A}
\end{aligned}
$$

The ideal average current is 97 µA, which is significantly lower than the measured current of 556 µA. The most significant overhead is caused by clock drift error [3]. AP and STA timers cannot be exactly synchronized, so the timer of the STA is deliberately set a little shorter to ensure the STA does not miss the session. The consequence is that the device has to wake up slightly earlier, resulting in intervals gradually becoming longer. The timer is resynchronized every 5 minutes.

The average current consumption, including the clock drift error, can be estimated by measuring the actual wake time. In the 5 seconds interval and 8 ms test, the shortest active time is 12 ms, while the longest is 93 ms. This means the additional active time caused by the clock error ranges from 4 ms to 85 ms. Since the increase is linear, the average additional active time is 44.5 ms. The following calculation estimates the average current, including the impact of the clock drift error. Note that this model, although accounting for clock drift error, is still simplified, as it assumes a constant active current and does not consider radio ramp-up times, short peaks, and other minor overheads.

$$
\begin{aligned}
I_{\text{avg}} &= I_{\text{active}} \cdot \frac{T_{\text{session}} + T_{\text{error}}}{T_{\text{interval}}} + I_{\text{sleep}} \cdot \frac{T_{\text{interval}} - T_{\text{session}} - T_{\text{error}}}{T_{\text{interval}}} \\
&= 51\,000\,\mu\text{A} \cdot \frac{8\,\text{ms} + 44.5\,\text{ms}}{5000\,\text{ms}} + 15\,\mu\text{A} \cdot \frac{5000\,\text{ms} - 8\,\text{ms} - 44.5\,\text{ms}}{5000\,\text{ms}} \\
&= 550.34\,\mu\text{A}
\end{aligned}
$$

The estimation of 550 µA is very close to the measured value of 556 µA, confirming that the clock drift error is the most significant overhead. This also highlights why it is not recommended to use shorter TWT sessions, as they would not significantly improve power consumption while placing additional constraints on the network. Additionally, the AP used for these tests does not support session durations shorter than 8 ms.

The power measurements revealed no critical findings, as the clock drift error is an expected issue. The power consumption is significantly improved when using TWT, with shorter sessions and longer intervals leading to lower power consumption, as anticipated. Since the primary focus of this thesis is to analyze the impact of TWT on the upper layers of the communication stack, the power consumption analysis has not been studied in greater depth.

It is important to consider these results as indicative, as another AP with different clock drift characteristics could lead to varying outcomes. Furthermore, TWT is still under development, and synchronization parameters may be optimized in the future, potentially improving the results further.

Recovery Mode

The recovery mode was introduced to address the problem of the AP buffering the downlink packets. When enabled, the STA teardowns and re-setup TWT when it detects that the AP buffers the responses. The recovery mode is detailed in the Testbed Design chapter (3.3.1).

A test with the recovery mode was done using the setup that delivered the worst results in terms of latency, which is the 8 ms session duration and 5 s interval. The test was executed with the same parameters, except for the recovery mode enabled with a threshold value of 2. This is the most sensitive threshold. Recovery is accomplished every time the number of pending requests exceeds 2.

The figure below shows the test's latency histogram.

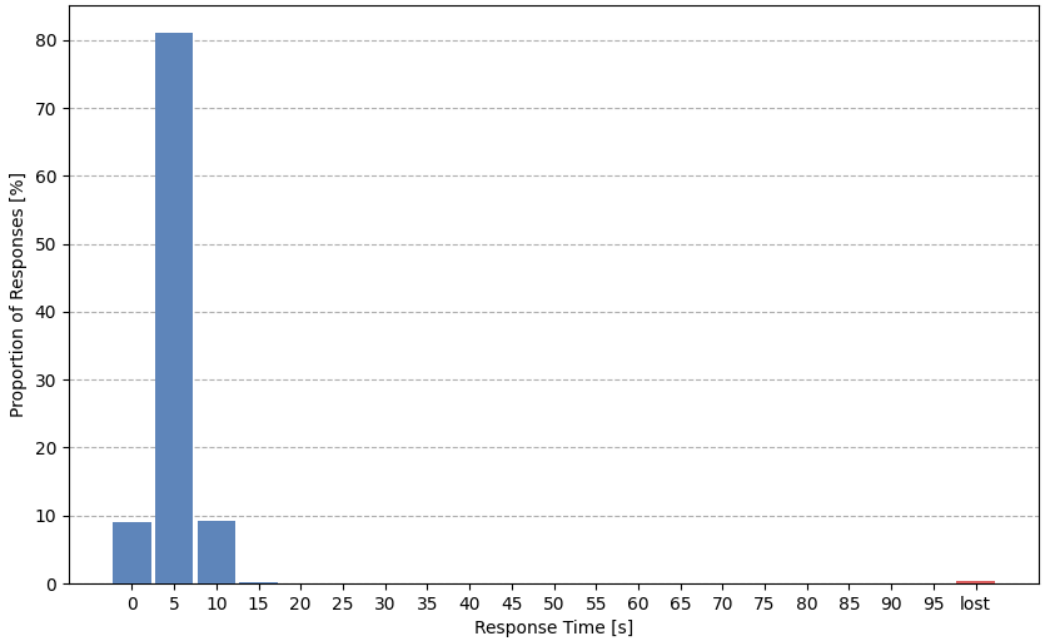


Figure 6.9: Response time histogram for a sensor use case test using recovery mode (threshold = 2), with a 5 second TWT interval and 8 ms TWT session duration

The test shows much better results than without recovery in terms of latency. More than 90 % of the responses are received in the next TWT session or within the same. The responses received after correspond to those that had triggered a recovery, and a few were lost.

Note that some responses are received within the same TWT session in this test, which is not the case in the previous tests (see fig. 6.3). The reason is that the test was performed on another network (and this is not due to the recovery mode). Nevertheless, the same AP was used with the same settings. On this network, the end-to-end latency is a little better, which explains that some responses can be received within the same

session. This has no impact on TWT. If the test were performed on the slightly slower network, the responses received within the same session would have been received after one TWT session.

The testbed had to recover 91 times in a test of 1000 iterations to accomplish such results. This corresponds to around a recovery every 10 requests, which is a lot because the TWT feature is intended to work during hours with no teardown while staying connected. The current consumption is expected to increase when the recovery mode is activated; however, the measurements produced counterintuitive results.

The test recorded an average current of 479 µA, corresponding to a decrease of 13 % compared to the non-recovery test, which recorded an average current consumption of 556 µA. The figure below compares the consumption of the PS mode and the TWT tests with the recovery mode enabled and disabled.

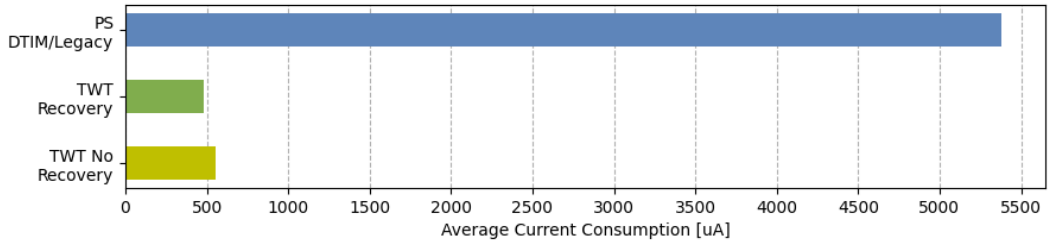


Figure 6.10: Recovery mode current consumption comparison

The power consumption in the test with recovery mode (5 s interval and 8 ms duration) is lower than in the test without it. This is explained by the frequent recoveries. The test recorded 91 recoveries out of 1000 sessions, corresponding to a recovery every 11 TWT sessions on average. Since a recovery involves the teardown and re-setup of TWT, it forces synchronization between the AP and STA counters, resulting in more frequent synchronization. In normal operation, synchronization occurs every 5 minutes. The frequent synchronization helps limit the clock drift error [3].

In the previous test, the measured active times gradually increased from 12 ms to 93 ms, resulting in an average active time of 52.5 ms (8 ms for the service period and 44.5 ms on average due to the clock drift error). In the eleventh TWT session, the active time was 28 ms. Since recovery is performed every 11 sessions on average, the average active time of a session is reduced to 20 ms (8 ms for the service period and 12 ms due to the clock drift error). The following calculation estimates the average current, excluding recoveries and including the clock drift error, with an additional 12 ms.

$$
\begin{aligned}
I_{\text{avg}} &= I_{\text{active}} \cdot \frac{T_{\text{session}} + T_{\text{error}}}{T_{\text{interval}}} + I_{\text{sleep}} \cdot \frac{T_{\text{interval}} - T_{\text{session}} - T_{\text{error}}}{T_{\text{interval}}} \\
&= 51\,000\,\mu\text{A} \cdot \frac{8\,\text{ms} + 12\,\text{ms}}{5000\,\text{ms}} + 15\,\mu\text{A} \cdot \frac{5000\,\text{ms} - 8\,\text{ms} - 12\,\text{ms}}{5000\,\text{ms}} \\
&= 218.94\,\mu\text{A}
\end{aligned}
$$

The increase in the average current consumption due to the recoveries can be calculated using the measurement of the actual current consumption and the estimation of the power consumption that excludes the recoveries.

$$\Delta I_{\text{recovery}} = I_{\text{measured}} - I_{\text{estimated}} = 479\,\mu\text{A} - 219\,\mu\text{A} = 260\,\mu\text{A}$$

The decrease in the average current consumption due to the reduced clock drift error can be estimated using the result of the test without recovery mode and the estimation of the power consumption that excludes the recoveries.

$$\Delta I_{\text{error}} = I_{\text{estimated}} - I_{\text{no recovery}} = 219\,\mu\text{A} - 556\,\mu\text{A} = -337\,\mu\text{A}$$

The results indicate that the recovery mode has two impacts on power consumption. First, it decreases the clock drift error, reducing the average current by 337 µA in this test. However, the recoveries involve some active periods, which increased the average current consumption by 260 µA in this test. As a result, the total average current consumption has decreased slightly. Nonetheless, the response time is significantly improved compared to the tests without recovery mode. Therefore, the recovery mode offers an excellent tradeoff between latency and power consumption.

These results are specific to this configuration. However, they demonstrate that the impact of the recoveries on power consumption is minimal. While the results may vary slightly in other configurations, the overall impact on power consumption will remain small, making the recovery mode a worthwhile compromise between latency and power efficiency.

These results also suggest that the current synchronization time of 5 minutes is not ideal and could be improved, as the recoveries, forcing the synchronization, effectively reduce the clock drift error. A better synchronization would improve the current consumption when the recovery mode is disabled. However, when enabled, the clock error's effect is smaller, leaving less room for improvement. Since the primary focus of this thesis is to analyze the impact of TWT on the upper layers of the communication stack, the power consumption analysis has not been studied in greater depth.

The relatively low impact of the recovery mode is explained by the fact that a recovery takes a very short time to be executed. Once the pending responses are received, TWT is immediately re-setup. The recovery takes between 100 and 200 ms if the responses are directly received. If the responses are lost, the recovery times out, and TWT is re-setup after 2 seconds. During the 2 seconds, the Wi-Fi module is in PS (DTIM/Legacy) mode. However, this situation is not frequent. The test recorded three lost packets out of 1000.

The figure below shows the current consumption during recovery for lost and buffered packets.

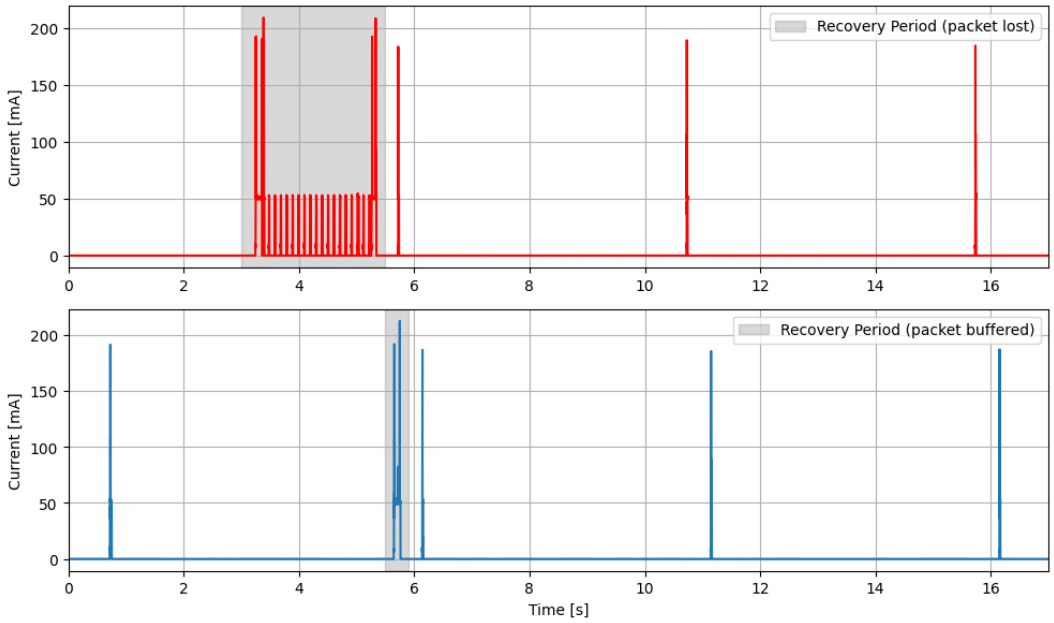


Figure 6.11: Recovery mode current trace

The recovery mode tests revealed that the approach of tearing down and re-setting up TWT when the AP buffers the packets is functional, even though it is not ideal. The tests showed that this mode effectively reduces the latency and brings it close to the expected value of one TWT interval. A negative aspect is that the recovery mode makes power consumption unpredictable, as the buffering phenomenon is unpredictable and directly impacts recovery, which affects power consumption. Nevertheless, the test shows that power consumption is not increased in practice because the recovery mode reduces the clock drift error, compensating for the increase due to the recoveries.

The recovery mode is not ideal because TWT is not designed to be used with frequent teardowns and setups. TWT is intended to operate for extended periods without interruptions or disconnections. While the recovery mode mitigates the effects of the issue, it does not address the root cause. The problem lies with the AP buffering the packets, which is an AP-side issue, and therefore cannot be resolved from the STA side.

6.4.2 Large Packet Test Case

The large packet test case is a variant of the sensor use case that uses large payloads. The test case is described in section 3.3.2. The tests were performed using the private server because the public server does not implement resources that respond with a large payload.

The tests were performed with 1 kB payloads. This corresponds to the maximum payload size authorized in CoAP. Sending more data in CoAP requires blockwise transfers. However, blockwise transfer is incompatible with TWT (see section 3.3.2).

Packet Loss and Response Time

Packet loss and response time are used to assess the impact of TWT on communication. This test aims to determine whether larger packet sizes have no effect or a negative impact on the disturbances caused by TWT, as observed in previous sensor tests.

The test was performed using the following testbed configuration:

- Private Server
- DTLS Enabled with Peer Verification, Connection ID, and TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256 cipher suite
- IPv4
- TWT implicit, announced, and trigger disabled

The test was performed multiple times with the following test settings:

- 1000 iterations
- TWT session durations from 8 to 64 ms
- TWT intervals from 5 to 10 seconds
- Large requests and large responses
- 1 kB payload size

The test was also performed in PS mode (DTIM/Legacy) to serve as a reference for packet loss. However, response times are not comparable due to the significantly higher values in TWT, making the comparison with PS mode meaningless.

The results of the large packet tests cannot be directly compared with the sensor use-case results because the sensor tests are performed on the public server, and the large packet tests are performed on the private server. The private server is two hops away from the client, and the Californium public server is 17 hops away, which increases the chances of losing packets between the AP and the server. This is the reason the test is also performed in PS mode. The increase (or absence of increase) in losses relative to the PS test can be compared between tests on public and private servers.

The graph below shows the number of lost packets for each test. For comparison, the PS test recorded 0.2 % of the lost packets. These results show combined uplink and downlink packet losses.

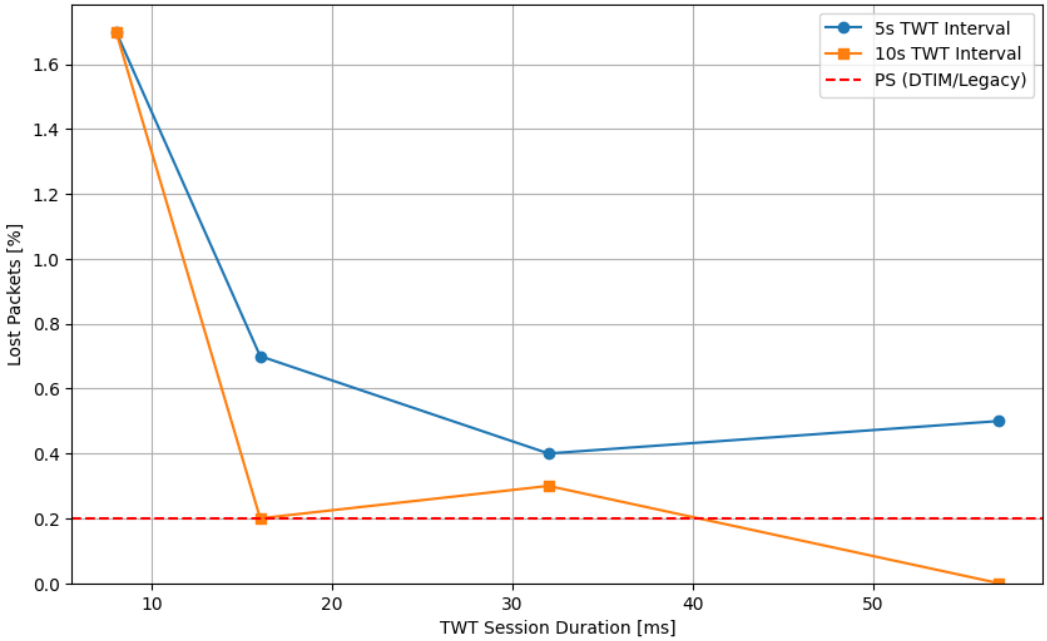


Figure 6.12: Packet loss for large packet test case

The graph indicates that packet loss decreases with longer sessions. It also shows that the TWT tests have significantly more lost packets than the PS test, with an average of 0.7 % of packets lost in the TWT tests compared to 0.2 % of packets lost in the PS test. For comparison, the sensor tests recorded a similar proportion of lost packets in both TWT and PS tests. This demonstrates that larger packets result in more losses, mainly when using small TWT session durations. Despite increased packet loss, the transmission remains reliable, with more than 98 % of requests successfully responded to.

The chart below gives more details about the lost packets by showing the proportion of requests and responses in the lost packets.

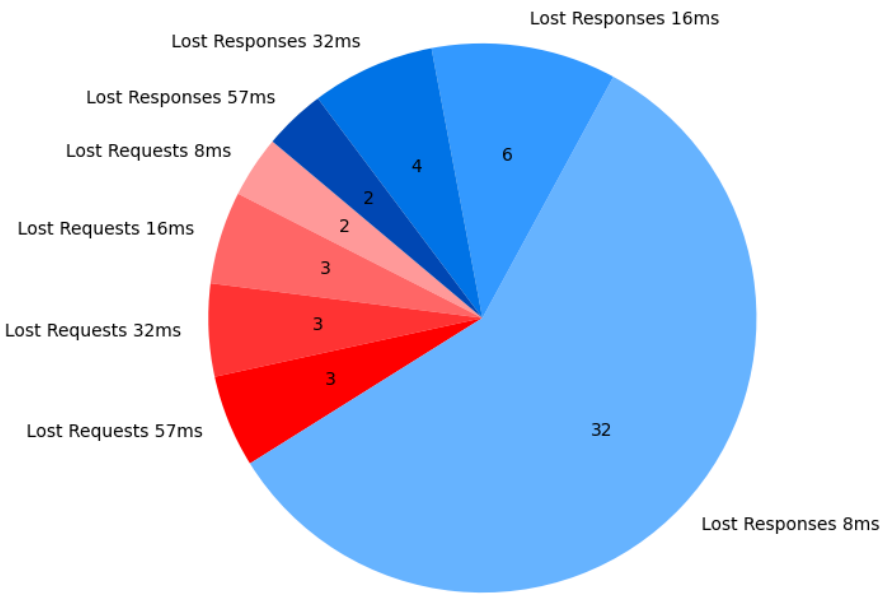


Figure 6.13: Proportion of lost requests and responses

The chart reveals that the majority of lost packets are responses. This indicates that TWT has a more significant impact on downlink traffic regarding packet loss.

Despite the tests showing increased packet loss, TWT's impact on the packet loss remains minimal. However, this test indicates that the downlink traffic is more impacted. A worse performance in downlink traffic has already been observed in the sensor response time tests.

The aspect that was more concerning in the previous tests was the response time. The graph below shows the average response time for the large packet tests.

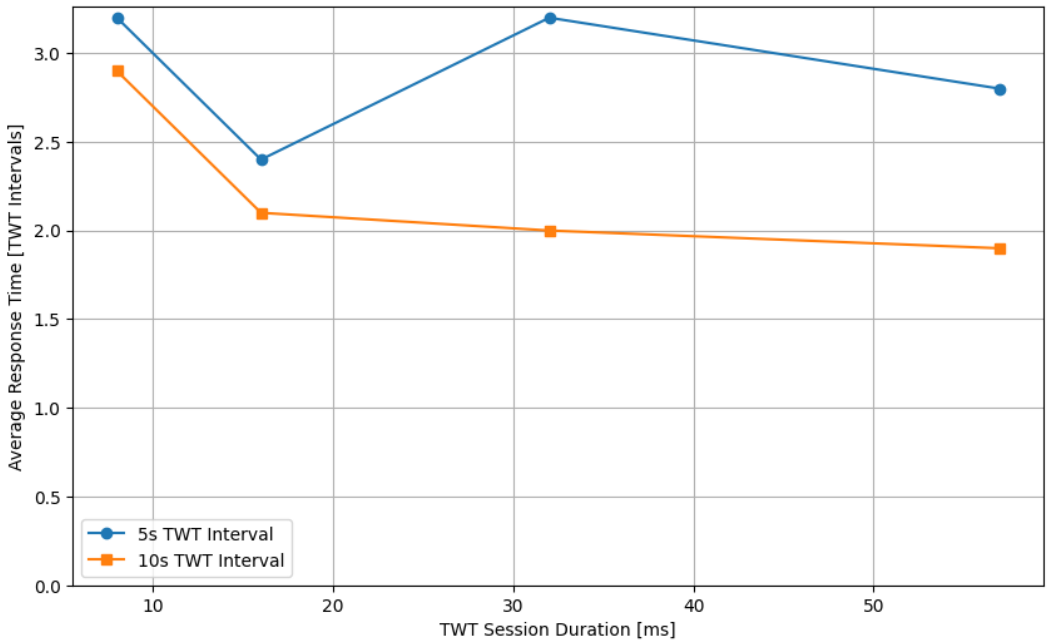


Figure 6.14: Average response time for large packet test case

The graph shows similar results in the sensor and the large packet tests, with responses arriving after 2 to 3 TWT intervals. Despite the response time being too high, as responses are expected to arrive after one TWT interval, the tests show that it is not increased when using large packets.

Power Consumption

The power measurements delivered results similar to those of the sensor tests (see Fig. 6.8). The large packet tests have recorded no significant variation in the power consumption. These results were expected because the large packet tests use the same TWT settings, which means the device stays active during the same period. For the detailed numerical results corresponding to these measurements, please refer to the appendix (see Table F.1).

The large packet tests showed that for all the elements analyzed, which are the packet loss, the response time, and the power consumption, the results are very similar to the sensor (small packet) tests. Therefore, the conclusion of the large packet tests is that the size of the payload has very little impact.

6.4.3 Multi Packet Use Case

The multi packet use case analyzes how the system reacts when multiple packets must be transmitted simultaneously. This use case is described in Section 3.3.3. The tests were performed using the public server.

The test was performed using the following testbed configuration:

- Public server (Californium)
- DTLS Enabled with Peer Verification, Connection ID, and TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256 cipher suite
- IPv4
- TWT implicit, announced, and trigger disabled

The test was performed multiple times with the following test settings:

- 1000 iterations
- TWT session durations from 8 to 64 ms
- TWT intervals from 5 to 10 seconds
- Recovery mode disabled

In this test, multiple requests are sent during an iteration. Then, the system waits for all the responses before starting the next iteration. A timeout forces the next iteration if a response is lost. Unlike the previous tests, one iteration can span multiple TWT sessions.

The first observation concerns the number of sessions needed to transmit the requests. This test showed that the system can transmit more data than can be buffered in one TWT session, even with the shortest duration (8 ms). This indicates that the buffer can be emptied entirely in a single TWT session.

Next, the number of messages was adjusted to avoid overwhelming the server. The server can respond to up to 10 messages simultaneously. With this configuration, the AP's buffer does not overflow, as all the responses can be received.

The logs below show that the system can transmit 10 requests in one TWT session and receive 10 responses in the next session.

```
[00:03:48.239] CoAP request sent, Token: 58 2c 62 21 b5 8b 87 b1
[00:03:48.244] CoAP request sent, Token: bf 15 0a 27 fb e6 80 d3
[00:03:48.249] CoAP request sent, Token: fc a8 73 7c 4a e6 07 35
[00:03:48.254] CoAP request sent, Token: 3b 83 1d 3d 20 24 97 d5
[00:03:48.258] CoAP request sent, Token: f3 81 3b 0b 3e 97 01 64
[00:03:48.263] CoAP request sent, Token: e1 5a c2 9c 97 9c dc df
```

```
[00:03:48.268] CoAP request sent, Token: 98 06 3f 75 f4 27 6f 5f
[00:03:48.273] CoAP request sent, Token: 73 2e 66 17 14 d0 6f bb
[00:03:48.277] CoAP request sent, Token: 6b 5f 42 cd 02 46 d0 de
[00:03:48.282] CoAP request sent, Token: e8 a4 5e 58 82 2c 50 f9
[00:03:53.733] CoAP response, Token: 58 2c 62 21 b5 8b 87 b1
[00:03:53.744] CoAP response, Token: bf 15 0a 27 fb e6 80 d3
[00:03:53.748] CoAP response, Token: fc a8 73 7c 4a e6 07 35
[00:03:53.752] CoAP response, Token: 3b 83 1d 3d 20 24 97 d5
[00:03:53.755] CoAP response, Token: f3 81 3b 0b 3e 97 01 64
[00:03:53.759] CoAP response, Token: e1 5a c2 9c 97 9c dc df
[00:03:53.763] CoAP response, Token: 98 06 3f 75 f4 27 6f 5f
[00:03:53.766] CoAP response, Token: 73 2e 66 17 14 d0 6f bb
[00:03:53.770] CoAP response, Token: 6b 5f 42 cd 02 46 d0 de
[00:03:53.774] CoAP response, Token: e8 a4 5e 58 82 2c 50 f9
```

Listing 6.1: Multi Packet Test Logs

The logs show that the system can transmit all the requests in one TWT session and then receive all the responses in the next session, even with an 8 ms session duration. However, the system does not consistently perform as expected when the test is run over a long period. The graph below shows the packet loss statistics.

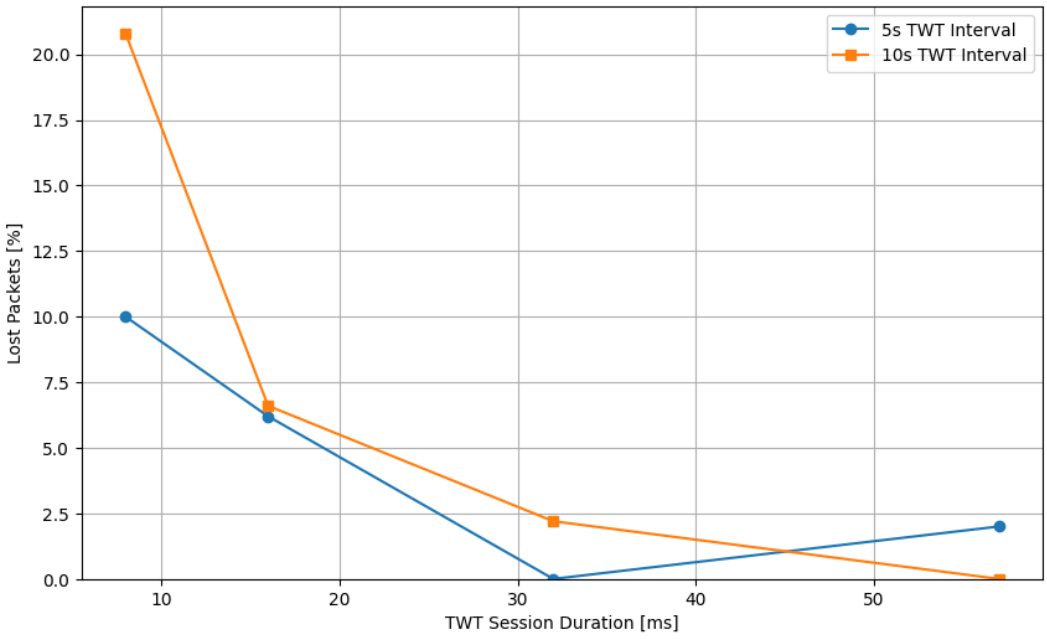


Figure 6.15: Packet loss for multi packet use case

A significant number of packets are lost when using short durations (8 ms and 16 ms). This is expected because the system is required to handle many packets in a very short time. Therefore, it is expected to experience a higher packet loss rate. Nevertheless, the losses are much lower for longer session durations (32 ms and 57 ms). This demonstrates that when timing is less constrained, the system can handle many requests without increasing the packet loss rate. The results align with the previous tests, with around 98 % of the requests successfully responded to.

The graph below shows the response time of the requests.

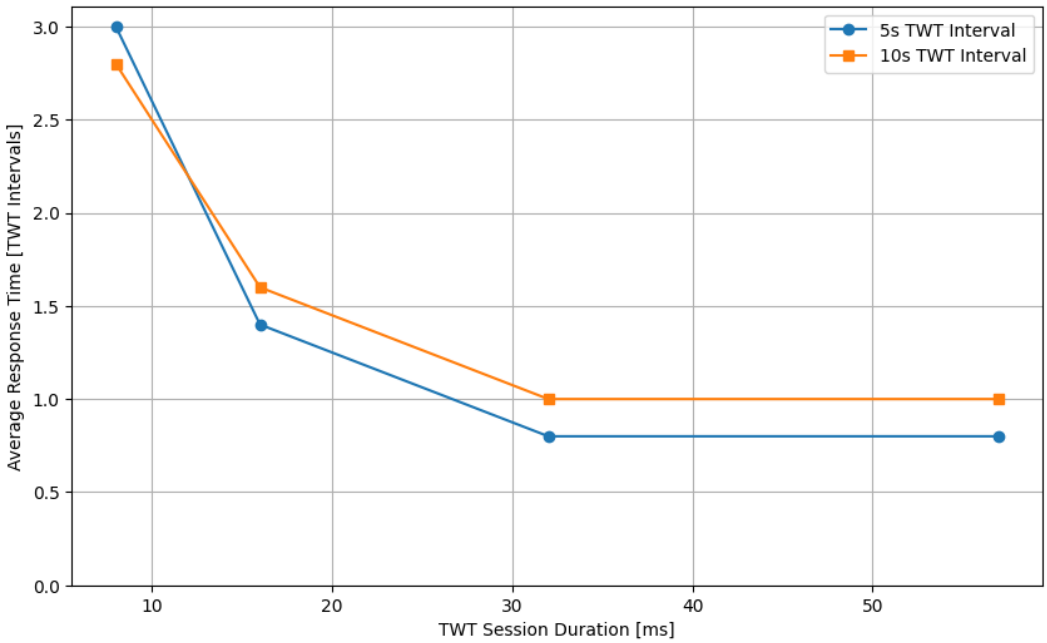


Figure 6.16: Average response time for multi packet use case

The graph indicates similar results for short sessions to those of the previous tests, with packets being buffered for 2 to 3 sessions. However, the results for longer sessions are particularly interesting, as latency drops to 1 TWT interval and even lower for the 5-second interval curve. This is because some requests are responded to within the same TWT session. This finding is noteworthy because it shows that the system performs better than in the previous tests. It may explain the issue of packets being buffered: the AP seems to prioritize transmission when many packets are buffered. The fact that packets are still buffered with shorter durations can be attributed to the tight timing constraints.

Power Consumption

The power measurements delivered results similar to those of the sensor tests (see Fig. 6.8). The multi packet tests have recorded no significant variation in the power consumption. These results were expected because the multi packet tests use the same TWT settings, which means the device stays active during the same period. For the detailed numerical results corresponding to these measurements, please refer to the appendix (see Table G.1).

6.4.4 Actuator Use Case

The actuator use case analyzes how the TWT application behaves when the traffic is downlink. This use case is described in Section 3.3.4. The tests were performed using the private server because the public server only allows for limited test scenarios. The public server only has observable resources that send notifications every 5 seconds, which is too frequent and predictable for a realistic actuator scenario. Furthermore, the notifications are confirmable packets, meaning they must be acknowledged. If an ACK arrives too late, the server retransmits the packet, which skews the results.

The test was performed using the following testbed configuration:

- Private Server
- DTLS Enabled with Peer Verification, Connection ID, and TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256 cipher suite
- IPv4
- TWT implicit, announced, and trigger disabled

The test was performed multiple times with the following test settings:

- 4 hours test
- Notifications sent by the server every 20 to 40 seconds (random interval)
- TWT session durations from 8 to 64 ms
- TWT interval of 5 seconds
- Notification Echoes enabled
- With and without emergency uplink (for the echoes)

The test was also performed in PS mode (DTIM/Legacy) to serve as a reference for packet loss. However, response times are not comparable due to the significantly higher values in TWT, making the comparison with PS mode meaningless.

Packet Loss and Response Time

Packet loss and response time are used to assess the impact of TWT on communication. In this use case, fewer packets are transmitted. Unlike the previous use cases, where packets were systematically transmitted at every TWT session, the behavior in this scenario differs significantly. Here, the server initiates the packet transmission, which is not synchronized with the TWT session schedule.

The server sends packets using random intervals that are longer than the TWT interval. This configuration introduces a level of unpredictability and the average frequency of transmissions is lower than in the sensor use case tests because the packets are not sent at every session. By analyzing the packet loss and response time under these conditions, the efficiency and reliability of TWT in scenarios with sporadic communication can be evaluated.

The graph below shows the proportion of lost packets for each test.

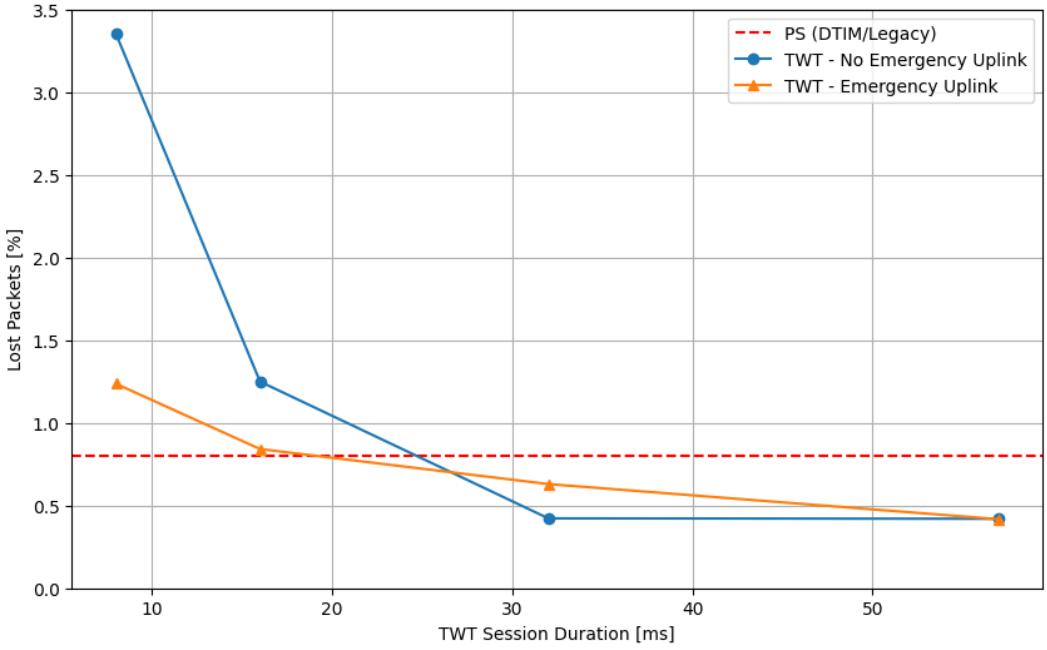


Figure 6.17: Packet loss for the actuator use case

The graph shows that shorter durations result in more lost packets, while longer durations produce results similar to the PS test. However, the results remain very good, with notifications being successfully echoed in more than 98 % of cases on average.

The next metric analyzed is the latency. As in the sensor use case, the end-to-end latency cannot be measured because this would require a precise common time reference between the client and the server. Instead, the testbed measures the response time. When a notification is sent by the server, the client responds to it (echo), and the elapsed time is measured on the server. The test also includes an option to use emergency uplink transmissions for the echoes. When enabled, the echoes can be immediately transmitted, even outside of a TWT session. The response time results of the emergency tests can be considered as downlink latency because the uplink

transmission only takes a few milliseconds, which is negligible compared to the many seconds of latency imposed by TWT. When the emergency uplink is disabled, the echo is transmitted in the next TWT session.

The bin width of the response time histogram is set to one second and not to the TWT interval as in the previous use cases, because the transmissions are not synchronized with the TWT sessions. The figure below shows an example of a histogram. This example is the result of the test using no emergency transmission and a 16 ms session duration. The results of all tests are available in Appendix H.

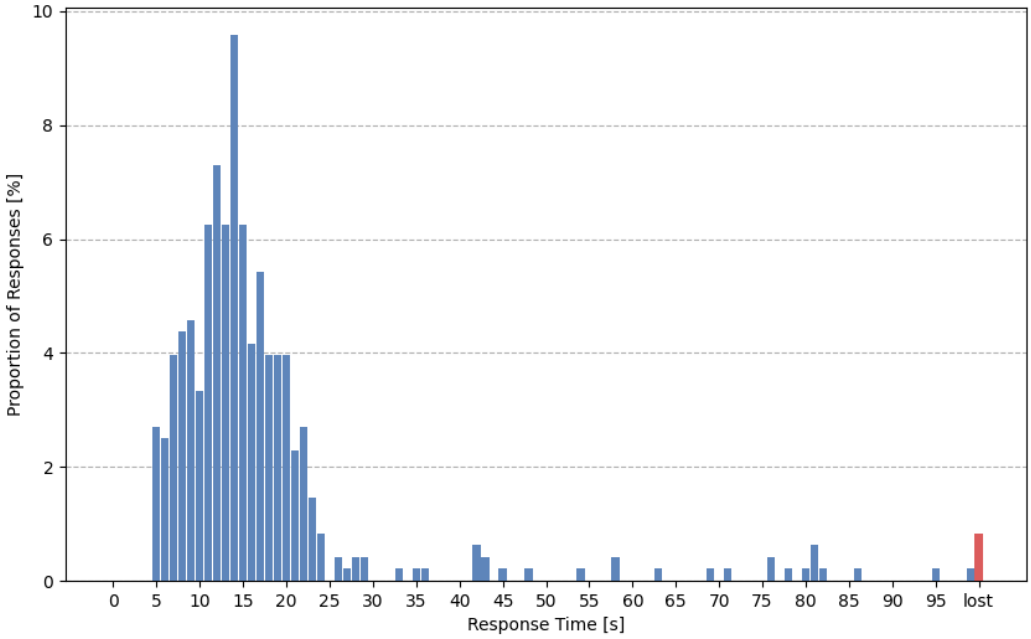


Figure 6.18: Echo time histogram for an actuator use case test using a 5 second TWT interval and 16 ms TWT session duration

The histogram shows that, as in the previous use case, the response time is much longer than expected. In this test, we would expect all the responses to arrive with a response time of 5 to 10 seconds. Most of the responses take between 5 and 25 seconds to arrive. However, there is a significant number of outliers, and some have very long response times. Monitoring the traffic at the Wi-Fi link and comparing the client and server logs revealed that the downlink traffic is buffered at the AP. The uplink traffic is systematically transmitted in the next TWT session when the emergency uplink is not used and immediately when the emergency uplink is used. This indicates that the problem of the AP buffering the downlink traffic, already observed in the previous use cases, is also present in actuator use cases.

The spread of the response time indicates that, as in the previous use cases, the standard deviation is significant. Therefore, averages must be interpreted with caution. This significant standard deviation highlights the unpredictability of the buffering phenomenon.

The graph below shows the average and median response times of all the tests.

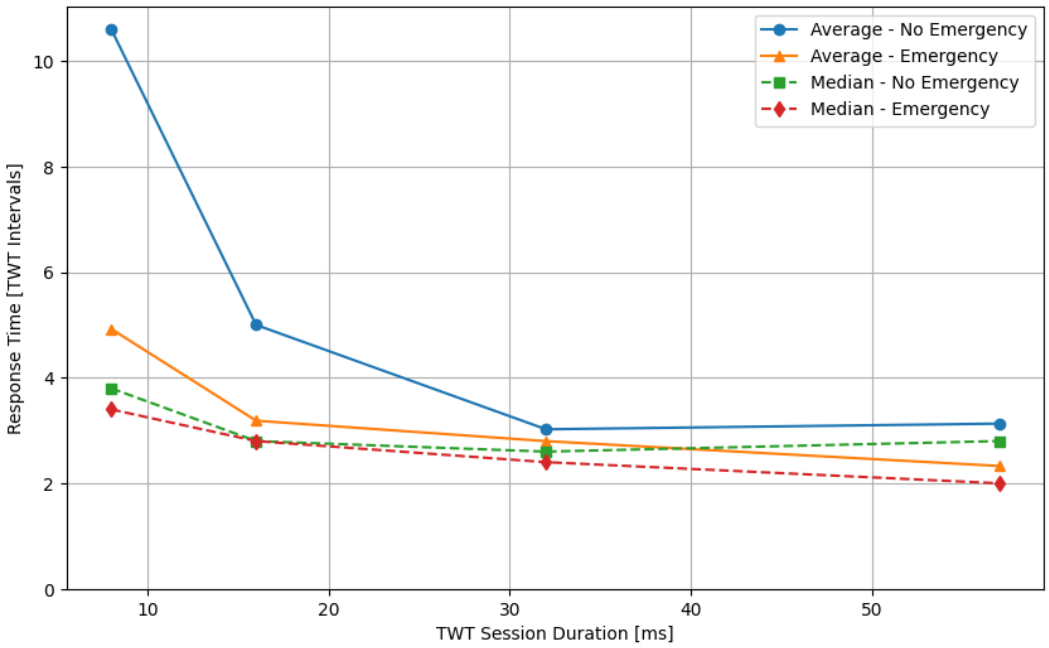


Figure 6.19: Response time for the actuator use case

This graph shows that the average latency decreases with longer durations. However, it is important to examine all the response time histograms (see Appendix H). The histograms show similar profiles, with notifications being echoed between 5 and 25 seconds. However, the histograms reveal that tests with short TWT sessions have many significant outliers. This indicates that the outliers may skew the average, which is why the graph also includes the medians. The medians confirm that the high response time averages of the short-duration tests are due to the frequent outliers.

These tests confirm the buffering problem already observed in the sensor use case, with downlink traffic being buffered for 1 to 5–6 TWT sessions. However, they show that some packets can also be buffered for very long periods, especially when using small TWT sessions. This was not observed in the sensor use case and can be explained by the traffic density being lower in the actuator use case, with a packet being sent every 6 TWT sessions on average. The actuator tests also further confirm the unpredictability of the buffering phenomenon, with high standard deviations and significant outliers.

Another aspect that can be analyzed is the effectiveness of the emergency feature. Observing the general shape of the curves suggests that the emergency feature is effective, as the total round-trip time is reduced. Drawing stronger conclusions about the uplink performance from this test is not valid because it shows the total round-trip time, which includes both downlink and uplink, and the tests demonstrated that the downlink traffic latency has a significant standard deviation. Since the emergency and no-emergency averages result from two different tests with substantial standard deviations, drawing conclusions from a direct comparison of these results is not meaningful.

Emergency Transmission Analysis

To further analyze the uplink traffic and the emergency feature, the test case was slightly modified to include two echoes for every notification: one sent with emergency priority and another that follows the TWT schedule. The diagram below shows the modified test case.

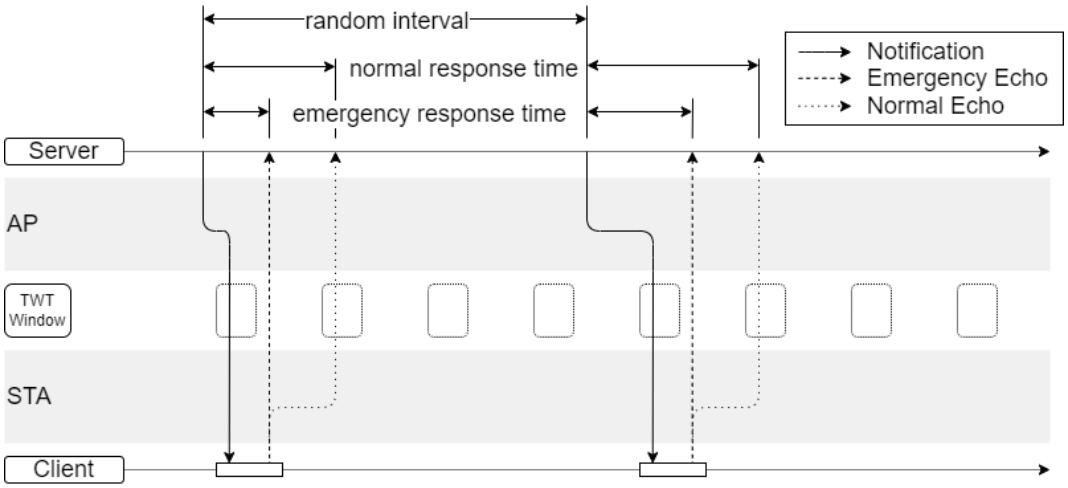


Figure 6.20: Modified test case with double echo

With this configuration, the emergency and normal response times can be compared, even if the downlink latency is longer than expected and has a high standard deviation.

The test was performed using the following testbed configuration:

- Private Server
- DTLS Enabled with Peer Verification, Connection ID, and TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256 cipher suite
- IPv4
- TWT implicit, announced, and trigger disabled

The test was performed four times with the following test settings:

- 1 hour test
- Notifications sent by the server every 20 to 40 seconds (random interval)
- TWT session durations from 8 to 64 ms
- TWT interval of 5 seconds
- Notification Echoes enabled
- Double echo, one with emergency priority and one without

The graph below shows the results of the tests.

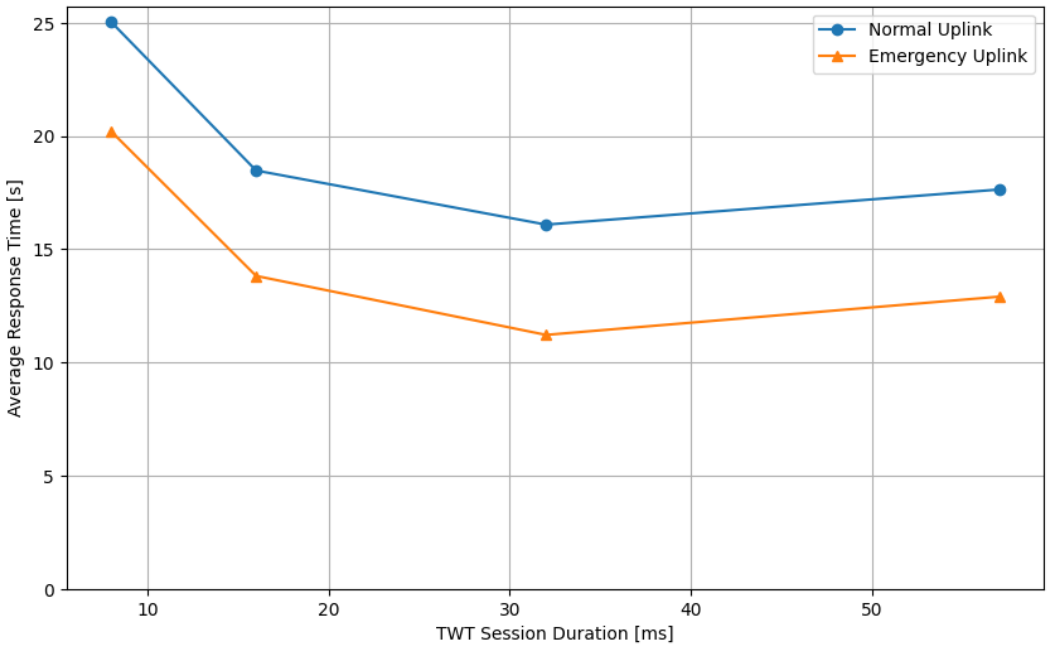


Figure 6.21: Response time for actuator use case with double echo

The graph shows that the average response time for every test is 5 seconds (one TWT interval) lower when the emergency mode is enabled. This indicates that the emergency feature works as expected.

The graph below shows the histograms of the response time for the normal echo and the emergency echo for the test with a 57ms TWT session duration [2].

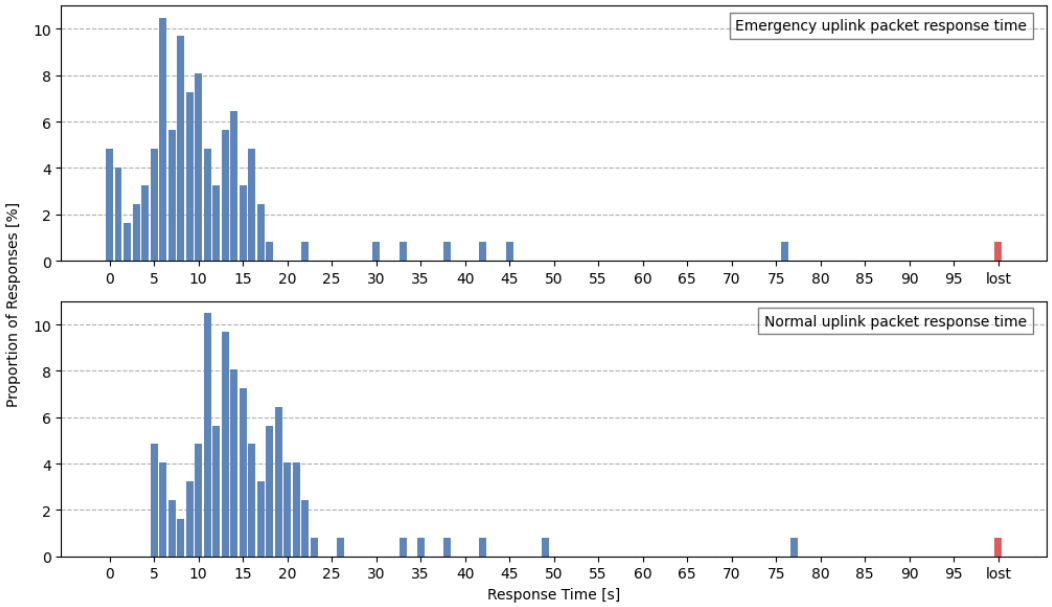


Figure 6.22: Response time histogram for an actuator use case test with double echo, using a 5 second TWT interval and 57 ms TWT session duration

The test results show nearly identical histograms, with a 5 second shift between the emergency and normal echoes. This furhter confirms that the emergency feature is functioning as expected, as nearly all emergency priority packets arrive 5 seconds earlier than the normal priority packets sent simultaneously. Both packets are sent immediately after the TWT session. The emergency packet is transmitted directly, while the normal packet is buffered and transmitted during the next TWT session, explaining the 5 second delay (one TWT interval). These results also confirm that the uplink traffic operates as expected when using TWT, as observed in the sensor and actuator tests while monitoring the Wi-Fi link.

The actuator use case showed that, as in the previous use cases, TWT is not causing significant packet losses. However, the actuator tests show longer response times than expected and corroborate the results of the earlier tests, confirming that the AP is buffering the packets for many TWT sessions, causing excessive downlink latency. As in the sensor tests, the buffering behavior is unpredictable, and response time averages show a significant standard deviation. Additionally, the actuator tests revealed a new finding: when using small TWT session durations in the actuator use case, the response time histograms show many strong outliers, indicating that some downlink packets are buffered for excessive periods. Finally, the actuator tests demonstrated that the emergency feature works as desired and that the uplink traffic is transmitted as expected without being buffered for more than a TWT interval.

[2]The test is configured for 64ms, but the negotiated TWT session duration is 57ms

Power Consumption

The power measurements of the tests without emergency uplink delivered results similar to those of the sensor tests (see Fig. 6.8). The actuator tests (without emergency) have recorded no significant variation in the power consumption. These results were expected because the actuator tests use the same TWT settings, which means the device stays active during the same period.

The power consumption is expected to increase when using the emergency mode, as the Wi-Fi module needs to remain active outside of TWT sessions, thereby increasing the total active time. To avoid inconsistencies, the random notification interval was replaced with a fixed notification interval of 10 seconds. The TWT interval is set to 5 seconds, and session durations range from 8 to 64 ms. Since the buffering of downlink packets at AP may cause inconsistencies in the uplink echoes, the echoes were replaced with an uplink emergency packet sent every 10 seconds outside of the TWT sessions, regardless of whether the downlink notification was received.

The figure below compares the power consumption with the emergency uplink disabled and enabled. For the detailed numerical results corresponding to these measurements, please refer to the appendix (see Table H.1).

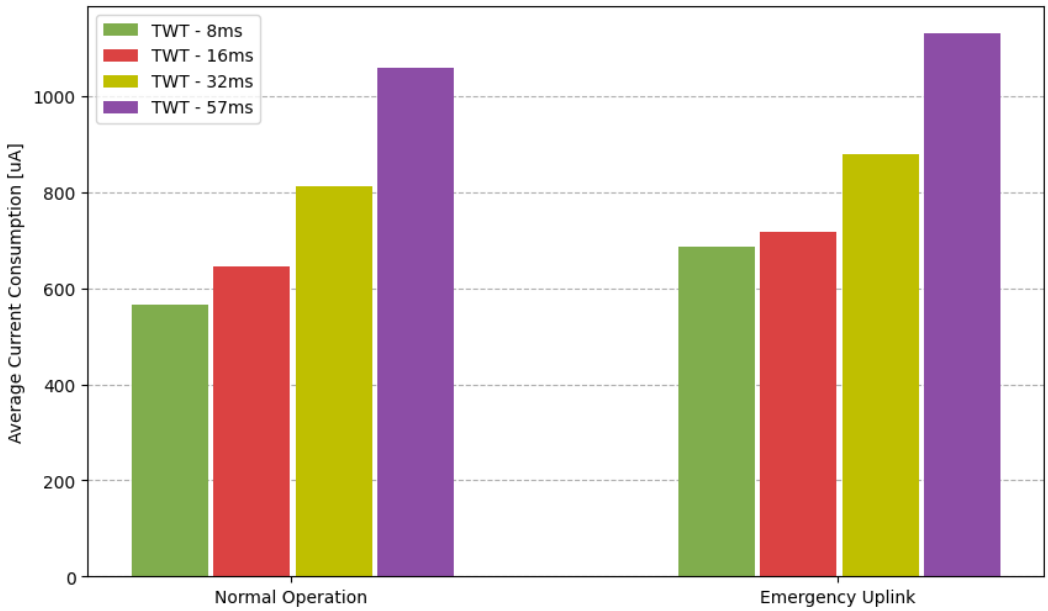


Figure 6.23: Average current consumption with and without emergency uplink

The results show an increase of 83 µA on average. This indicates that the impact of the emergency uplink transmissions is limited, although non-negligible. Increasing the frequency and/or the number of emergency uplink transmissions would directly impact power consumption. To conclude, the emergency uplink tests showed that the feature is operational but affects the power consumption, as expected.

7 | Conclusions

7.1 Project summary

7.2 Comparison with the initial objectives

7.3 Encountered difficulties

7.4 Future perspectives

A | Source Code Repositories

The source code developed for this project is available on GitHub. The repositories are listed below:

- **TWT Testbed Repository**

  https://github.com/svankappel/twt-testbed

  This repository contains the testbed application developed for the nRF7002 DK. It implements a CoAP client for testing the Target Wake Time (TWT) functionality. The application is designed to evaluate power-saving features in Wi-Fi 6 environments.

- **TWT Testbed Server Repository**

  https://github.com/svankappel/twt-testbed-server

  This repository contains the implementation of a CoAP server, programmed in Java using the Californium library. It allows for more advanced testing scenarios than those achievable with the Californium public server available on the internet.

Both repositories include comprehensive README files with detailed instructions for setup, usage, and additional information to facilitate replication and further development.

B | Wi-Fi Monitor Mode on Linux

The script below can be used to set the Wi-Fi interface of a Linux computer in monitor mode.

```
#!/bin/bash

# check channel
if [ -z "$1" ]; then
    echo "Usage: $0 <channel>"
    exit 1
fi

CHANNEL=$1

INTERFACE="wlp0s20f3"

sudo rfkill unblock wifi

sudo ip link set $INTERFACE down
sudo iw $INTERFACE set monitor control
sudo ip link set $INTERFACE up

sudo iw $INTERFACE set channel $CHANNEL

sudo iwconfig $INTERFACE
```

Listing B.1: Wi-Fi Monitor Mode on Linux

Usage example for setting monitor mode on ch10:

```
sudo ./set_monitor_mode 10
```

The name of the Wi-Fi interface can be found with the following command:

```
iw dev
```

The following command displays the modes supported by the Wi-Fi interface:

```
iw list | grep -A 10 "Supported interface modes"
```

C | ARP Tests

The following pages present the raw results of the ARP tests.

- The first test shows that after the initial ARP requests, no further ARP requests are made. This suggests that the AP updates its ARP table using uplink traffic.
- The second test supports this finding, as an ARP request is triggered after 30 minutes of inactivity. Additionally, the ARP request is unicast, indicating that the AP retains the STA's MAC address and sends the request to verify that the MAC address is still associated with the same IP address.

No.	Time	Source	Destination	Protoco	Length	Info
354	24.958	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	EAPOL	193	Key (Message 1 of 4)
356	24.971	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	EAPOL	215	Key (Message 2 of 4)
358	24.974	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	EAPOL	249	Key (Message 3 of 4)
360	24.983	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	EAPOL	193	Key (Message 4 of 4)
370	25.008	ASUSTekCOMPU_65:45:58	Broadcast	ARP	136	Who has 192.168.50.229? Tell 192.168.50.1
418	26.066	ASUSTekCOMPU_65:45:58	Broadcast	ARP	136	Who has 192.168.50.229? Tell 192.168.50.1
432	27.091	ASUSTekCOMPU_65:45:58	Broadcast	ARP	136	Who has 192.168.50.229? Tell 192.168.50.1
492	29.647	NordicSemico_00:6d:37	Broadcast	ARP	154	Who has 192.168.50.1? Tell 192.168.50.229
494	29.648	NordicSemico_00:6d:37	Broadcast	ARP	136	Who has 192.168.50.1? Tell 192.168.50.229
498	29.65	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	ARP	154	192.168.50.1 is at cc:28:aa:65:45:58
501	29.654	192.168.50.229	192.168.1.228	CoAP	175	CON, MID:45703, GET, TKN:ed 91 83 7e 9b ce a8 83, /validate
505	29.658	192.168.1.228	192.168.50.229	CoAP	172	ACK, MID:45703, 2.05 Content, TKN:ed 91 83 7e 9b ce a8 83, /validate
572	33.94	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	ARP	154	Who has 192.168.50.229? Tell 192.168.50.1
575	33.947	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	ARP	154	192.168.50.229 is at f4:ce:36:00:6d:37
670	39.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45704, PUT (text/plain), TKN:55 94 38 f1 43 b1 fb f1, /test
674	39.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45704, 2.04 Changed, TKN:55 94 38 f1 43 b1 fb f1, /test
767	44.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45705, PUT (text/plain), TKN:1c 26 c9 3f 13 3d 53 2c, /test
774	44.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45705, 2.04 Changed, TKN:1c 26 c9 3f 13 3d 53 2c, /test
778	44.669	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45705, 2.04 Changed, TKN:1c 26 c9 3f 13 3d 53 2c, /test
858	49.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45706, PUT (text/plain), TKN:ab 35 b2 14 d4 4f d3 a1, /test
865	49.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45706, 2.04 Changed, TKN:ab 35 b2 14 d4 4f d3 a1, /test
869	49.669	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45706, 2.04 Changed, TKN:ab 35 b2 14 d4 4f d3 a1, /test
937	54.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45707, PUT (text/plain), TKN:95 90 f2 dd c1 b5 89 90, /test
946	54.669	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45707, 2.04 Changed, TKN:95 90 f2 dd c1 b5 89 90, /test
1084	59.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45708, PUT (text/plain), TKN:c1 59 0e 77 34 83 8b e5, /test
1091	59.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45708, 2.04 Changed, TKN:c1 59 0e 77 34 83 8b e5, /test
1201	64.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45709, PUT (text/plain), TKN:bd 46 8a 6f 03 74 24 aa, /test
1208	64.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45709, 2.04 Changed, TKN:bd 46 8a 6f 03 74 24 aa, /test
1288	69.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45710, PUT (text/plain), TKN:b5 b2 cd 0e 83 1a 29 1c, /test
1297	69.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45710, 2.04 Changed, TKN:b5 b2 cd 0e 83 1a 29 1c, /test
1374	74.668	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45711, PUT (text/plain), TKN:56 f3 da b3 2f f1 97 b0, /test
1382	74.672	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45711, 2.04 Changed, TKN:56 f3 da b3 2f f1 97 b0, /test
1458	79.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45712, PUT (text/plain), TKN:eb 2a 26 89 71 0a 54 54, /test
1465	79.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45712, 2.04 Changed, TKN:eb 2a 26 89 71 0a 54 54, /test
1597	84.665	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:45713, PUT (text/plain), TKN:46 03 3f ed 0b 33 23 c4, /test
1604	84.668	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:45713, 2.04 Changed, TKN:46 03 3f ed 0b 33 23 c4, /test
1677	89.665	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45714, PUT (text/plain), TKN:ab d6 36 08 17 30 46 37, /test
1684	89.668	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45714, 2.04 Changed, TKN:ab d6 36 08 17 30 46 37, /test
1759	94.667	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45715, PUT (text/plain), TKN:18 fc e5 ad 4b 6b e1 0d, /test
1771	94.692	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45715, 2.04 Changed, TKN:18 fc e5 ad 4b 6b e1 0d, /test
1880	99.665	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45716, PUT (text/plain), TKN:a0 06 b2 d9 cb f1 49 87, /test
1881	99.665	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45716, PUT (text/plain), TKN:a0 06 b2 d9 cb f1 49 87, /test [Retransmission]
1888	99.668	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45716, 2.04 Changed, TKN:a0 06 b2 d9 cb f1 49 87, /test
1992	104.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45717, PUT (text/plain), TKN:ae 8b 30 07 7e 75 58 78, /test
1999	104.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45717, 2.04 Changed, TKN:ae 8b 30 07 7e 75 58 78, /test
2074	109.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45718, PUT (text/plain), TKN:15 c3 19 5b 32 e2 67 67, /test
2081	109.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45718, 2.04 Changed, TKN:15 c3 19 5b 32 e2 67 67, /test
2152	114.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45719, PUT (text/plain), TKN:73 58 0a 7c 07 ea c0 74, /test
2159	114.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45719, 2.04 Changed, TKN:73 58 0a 7c 07 ea c0 74, /test
2253	119.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45720, PUT (text/plain), TKN:26 78 eb 69 3c 19 59 3e, /test
2262	119.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45720, 2.04 Changed, TKN:26 78 eb 69 3c 19 59 3e, /test
2266	119.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45720, 2.04 Changed, TKN:26 78 eb 69 3c 19 59 3e, /test
2346	124.67	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45721, PUT (text/plain), TKN:b5 d4 47 fb 22 76 bf 89, /test
2359	124.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45721, 2.04 Changed, TKN:b5 d4 47 fb 22 76 bf 89, /test
2441	129.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45722, PUT (text/plain), TKN:5f f1 cb 35 4c 1b 6b b5, /test
2448	129.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45722, 2.04 Changed, TKN:5f f1 cb 35 4c 1b 6b b5, /test
2523	134.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45723, PUT (text/plain), TKN:bb e4 81 e1 cf 4e 68 d1, /test
2532	134.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45723, 2.04 Changed, TKN:bb e4 81 e1 cf 4e 68 d1, /test
2624	139.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45724, PUT (text/plain), TKN:3a 35 0c 36 f4 5b 82 27, /test
2631	139.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45724, 2.04 Changed, TKN:3a 35 0c 36 f4 5b 82 27, /test
2730	144.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45725, PUT (text/plain), TKN:af fc 81 74 03 f5 81 a3, /test
2739	144.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45725, 2.04 Changed, TKN:af fc 81 74 03 f5 81 a3, /test
2806	149.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45726, PUT (text/plain), TKN:1b c0 15 24 10 a4 26 92, /test
2817	149.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45726, 2.04 Changed, TKN:1b c0 15 24 10 a4 26 92, /test
2883	154.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45727, PUT (text/plain), TKN:86 fb 67 bb 75 c1 ab 59, /test
2891	154.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45727, 2.04 Changed, TKN:86 fb 67 bb 75 c1 ab 59, /test
2950	159.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45728, PUT (text/plain), TKN:93 b1 dd 25 21 b0 bd 2f, /test
2957	159.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45728, 2.04 Changed, TKN:93 b1 dd 25 21 b0 bd 2f, /test
3050	164.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45729, PUT (text/plain), TKN:8a 0e e7 b7 21 be f3 d2, /test
3057	164.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45729, 2.04 Changed, TKN:8a 0e e7 b7 21 be f3 d2, /test
3124	169.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45730, PUT (text/plain), TKN:52 69 8b ec 29 0e 3c 0a, /test
3131	169.67	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:45730, 2.04 Changed, TKN:52 69 8b ec 29 0e 3c 0a, /test
3192	174.66	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:45731, PUT (text/plain), TKN:0c 83 a5 52 40 98 1e 5e, /test

No.	Time	Source	Destination	Protocol	Length	Info
620	31.699	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	EAPOL	193	Key (Message 1 of 4)
622	31.713	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	EAPOL	215	Key (Message 2 of 4)
624	31.716	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	EAPOL	249	Key (Message 3 of 4)
626	31.725	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	EAPOL	193	Key (Message 4 of 4)
633	31.748	ASUSTekCOMPU_65:45:58	Broadcast	ARP	136	Who has 192.168.50.229? Tell 192.168.50.1
681	32.793	ASUSTekCOMPU_65:45:58	Broadcast	ARP	136	Who has 192.168.50.229? Tell 192.168.50.1
693	33.818	ASUSTekCOMPU_65:45:58	Broadcast	ARP	136	Who has 192.168.50.229? Tell 192.168.50.1
752	36.123	NordicSemico_00:6d:37	Broadcast	ARP	142	Who has 192.168.50.1? Tell 192.168.50.229
753	36.123	NordicSemico_00:6d:37	Broadcast	ARP	154	Who has 192.168.50.1? Tell 192.168.50.229
755	36.124	NordicSemico_00:6d:37	Broadcast	ARP	136	Who has 192.168.50.1? Tell 192.168.50.229
766	36.125	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	ARP	154	192.168.50.1 is at cc:28:aa:65:45:58
769	36.128	192.168.50.229	192.168.1.228	CoAP	175	CON, MID:34422, GET, TKN:ef f2 75 a2 e8 e4 ab bf, /validate
770	36.129	192.168.50.229	192.168.1.228	CoAP	175	CON, MID:34422, GET, TKN:ef f2 75 a2 e8 e4 ab bf, /validate [Retransmission]
774	36.132	192.168.1.228	192.168.50.229	CoAP	172	ACK, MID:34422, 2.05 Content, TKN:ef f2 75 a2 e8 e4 ab bf, /validate
906	40.347	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	ARP	154	Who has 192.168.50.229? Tell 192.168.50.1
909	40.351	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	ARP	154	192.168.50.229 is at f4:ce:36:00:6d:37
32827	1846.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34423, PUT (text/plain), TKN:93 98 af dd c3 68 8d 55, /test
32833	1846.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34423, 2.04 Changed, TKN:93 98 af dd c3 68 8d 55, /test
32904	1851.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34424, PUT (text/plain), TKN:81 2d f7 85 f0 24 fc a1, /test
32913	1851.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34424, 2.04 Changed, TKN:81 2d f7 85 f0 24 fc a1, /test
32922	1851.3	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	ARP	154	Who has 192.168.50.229? Tell 192.168.50.1
32925	1851.3	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	ARP	154	192.168.50.229 is at f4:ce:36:00:6d:37
33000	1856.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34425, PUT (text/plain), TKN:bb 0a 6b 97 4b d7 c7 c2, /test
33007	1856.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34425, 2.04 Changed, TKN:bb 0a 6b 97 4b d7 c7 c2, /test
33075	1861.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34426, PUT (text/plain), TKN:c9 4a 12 01 cc 3f dd 1e, /test
33082	1861.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34426, 2.04 Changed, TKN:c9 4a 12 01 cc 3f dd 1e, /test
33171	1866.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34427, PUT (text/plain), TKN:31 f7 fc 0d 76 f9 94 80, /test
33178	1866.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34427, 2.04 Changed, TKN:31 f7 fc 0d 76 f9 94 80, /test
33237	1871.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34428, PUT (text/plain), TKN:56 7e ad 56 c5 80 4f 0b, /test
33247	1871.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34428, 2.04 Changed, TKN:56 7e ad 56 c5 80 4f 0b, /test
33320	1876.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34429, PUT (text/plain), TKN:d0 bd bf 85 2a f3 63 b9, /test
33327	1876.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34429, 2.04 Changed, TKN:d0 bd bf 85 2a f3 63 b9, /test
33423	1881.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34430, PUT (text/plain), TKN:a9 83 f1 50 56 ff b3 13, /test
33430	1881.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34430, 2.04 Changed, TKN:a9 83 f1 50 56 ff b3 13, /test
33494	1886.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34431, PUT (text/plain), TKN:51 a0 48 af 45 3e b8 8a, /test
33501	1886.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34431, 2.04 Changed, TKN:51 a0 48 af 45 3e b8 8a, /test
33596	1891.1	192.168.50.229	192.168.1.228	CoAP	191	CON, MID:34432, PUT (text/plain), TKN:da d8 6f 43 65 86 99 21, /test
33603	1891.1	192.168.1.228	192.168.50.229	CoAP	195	ACK, MID:34432, 2.04 Changed, TKN:da d8 6f 43 65 86 99 21, /test
33680	1896.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34433, PUT (text/plain), TKN:46 e3 23 07 97 3d 37 81, /test
33687	1896.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34433, 2.04 Changed, TKN:46 e3 23 07 97 3d 37 81, /test
33774	1901.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34434, PUT (text/plain), TKN:65 33 ac 40 c9 74 43 68, /test
33775	1901.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34434, PUT (text/plain), TKN:65 33 ac 40 c9 74 43 68, /test [Retransmission]
33776	1901.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34434, PUT (text/plain), TKN:65 33 ac 40 c9 74 43 68, /test [Retransmission]
33777	1901.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34434, PUT (text/plain), TKN:65 33 ac 40 c9 74 43 68, /test [Retransmission]
33778	1901.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34434, PUT (text/plain), TKN:65 33 ac 40 c9 74 43 68, /test [Retransmission]
33787	1901.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34434, 2.04 Changed, TKN:65 33 ac 40 c9 74 43 68, /test
33858	1906.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34435, PUT (text/plain), TKN:67 34 4d a5 d7 5c 37 38, /test
33865	1906.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34435, 2.04 Changed, TKN:67 34 4d a5 d7 5c 37 38, /test
33948	1911.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34436, PUT (text/plain), TKN:b3 20 29 b3 44 13 6c b9, /test
33955	1911.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34436, 2.04 Changed, TKN:b3 20 29 b3 44 13 6c b9, /test
34071	1916.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34437, PUT (text/plain), TKN:84 90 e7 5a 65 55 e9 97, /test
34079	1916.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34437, 2.04 Changed, TKN:84 90 e7 5a 65 55 e9 97, /test
34173	1921.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34438, PUT (text/plain), TKN:5a 1b d2 b7 9e 8c 74 ab, /test
34180	1921.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34438, 2.04 Changed, TKN:5a 1b d2 b7 9e 8c 74 ab, /test
34275	1926.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34439, PUT (text/plain), TKN:55 a1 99 ab 5f 83 5e 30, /test
34288	1926.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34439, 2.04 Changed, TKN:55 a1 99 ab 5f 83 5e 30, /test
34370	1931.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34440, PUT (text/plain), TKN:a7 64 2e 67 05 13 c1 cb, /test
34377	1931.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34440, 2.04 Changed, TKN:a7 64 2e 67 05 13 c1 cb, /test
34476	1936.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34441, PUT (text/plain), TKN:23 38 97 19 b0 ab 4c db, /test
34485	1936.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34441, 2.04 Changed, TKN:23 38 97 19 b0 ab 4c db, /test
34491	1936.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34441, 2.04 Changed, TKN:23 38 97 19 b0 ab 4c db, /test
34572	1941.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34442, PUT (text/plain), TKN:57 cc 79 53 22 10 c7 56, /test
34579	1941.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34442, 2.04 Changed, TKN:57 cc 79 53 22 10 c7 56, /test
34654	1946.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34443, PUT (text/plain), TKN:65 50 66 c2 66 9e 52 80, /test
34662	1946.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34443, 2.04 Changed, TKN:65 50 66 c2 66 9e 52 80, /test
34743	1951.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34444, PUT (text/plain), TKN:57 d5 eb 9d f7 c3 2f 94, /test
34750	1951.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34444, 2.04 Changed, TKN:57 d5 eb 9d f7 c3 2f 94, /test
34843	1956.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34445, PUT (text/plain), TKN:f1 9c 45 55 f7 62 3b 66, /test
34852	1956.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34445, 2.04 Changed, TKN:f1 9c 45 55 f7 62 3b 66, /test
34960	1961.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34446, PUT (text/plain), TKN:04 cd 19 35 fd c9 cf 9a, /test
34967	1961.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34446, 2.04 Changed, TKN:04 cd 19 35 fd c9 cf 9a, /test
35105	1966.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34447, PUT (text/plain), TKN:85 da 52 0c 85 61 c0 e7, /test
35204	1971.1	192.168.50.229	192.168.1.228	CoAP	192	CON, MID:34448, PUT (text/plain), TKN:9f ab b7 02 aa ea 31 7e, /test
35212	1971.1	192.168.1.228	192.168.50.229	CoAP	196	ACK, MID:34448, 2.04 Changed, TKN:9f ab b7 02 aa ea 31 7e, /test

D | NDP Test

The following pages present the raw results of the NDP test.

The results display the various NDP exchanges that occurred during the operation in IPv6.

No.	Time	Source	Destination	Protocol	Leng	Info
213	14.3	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	EAPOL	193	Key (Message 1 of 4)
216	14.3	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	EAPOL	215	Key (Message 2 of 4)
218	14.3	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	EAPOL	249	Key (Message 3 of 4)
219	14.3	ASUSTekCOMPU_65:45:58	NordicSemico_00:6d:37	EAPOL	249	Key (Message 3 of 4)
221	14.3	NordicSemico_00:6d:37	ASUSTekCOMPU_65:45:58	EAPOL	193	Key (Message 4 of 4)
261	15.3	fe80::f6ce:36ff:fe00:6d37	ff02::2	ICMPv6	182	Router Solicitation from f4:ce:36:00:6d:37
263	15.3	fe80::f6ce:36ff:fe00:6d37	ff02::2	ICMPv6	164	Router Solicitation from f4:ce:36:00:6d:37
264	15.3	fe80::ce28:aaff:fe65:4558	ff02::1	ICMPv6	244	Router Advertisement from cc:28:aa:65:45:58
265	15.3	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
267	15.3	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
268	15.3	::	ff02::1:ff00:6d37	ICMPv6	190	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37
272	15.3	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
281	15.3	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
283	15.3	::	ff02::1:ff00:6d37	ICMPv6	172	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37
284	15.3	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
289	15.3	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
314	16.8	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	186	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
318	16.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
321	16.9	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
324	16.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	195	CON, MID:55768, GET, TKN:95 25 f3 bb 21 0a 01 d5, /validate
328	16.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	192	ACK, MID:55768, 2.05 Content, TKN:95 25 f3 bb 21 0a 01 d5, /validate
406	20.6	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
409	20.6	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
436	22.1	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
439	22.1	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
510	26.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55769, PUT (text/plain), TKN:4b 66 0e 6a 23 e3 16 01, /test
514	26.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55769, 2.04 Changed, TKN:4b 66 0e 6a 23 e3 16 01, /test
638	31.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55770, PUT (text/plain), TKN:c4 36 32 1b 15 1a 37 2f, /test
650	31.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55770, 2.04 Changed, TKN:c4 36 32 1b 15 1a 37 2f, /test
725	36.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55771, PUT (text/plain), TKN:71 27 38 3f 3a 31 d5 92, /test
732	36.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55771, 2.04 Changed, TKN:71 27 38 3f 3a 31 d5 92, /test
816	41.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55772, PUT (text/plain), TKN:79 49 5f e9 39 75 c3 f7, /test
823	41.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55772, 2.04 Changed, TKN:79 49 5f e9 39 75 c3 f7, /test
902	46.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55773, PUT (text/plain), TKN:e7 f8 16 22 2a 2b a0 bd, /test
914	46.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55773, 2.04 Changed, TKN:e7 f8 16 22 2a 2b a0 bd, /test
1014	51.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55774, PUT (text/plain), TKN:2f 04 40 47 c5 ca bd a5, /test
1016	51.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
1018	51.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
1024	51.9	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
1029	51.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55774, 2.04 Changed, TKN:2f 04 40 47 c5 ca bd a5, /test
1103	56.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55775, PUT (text/plain), TKN:3f 00 f1 94 1e 4e e6 4b, /test
1110	56.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55775, 2.04 Changed, TKN:3f 00 f1 94 1e 4e e6 4b, /test
1193	61.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55776, PUT (text/plain), TKN:02 b1 00 f5 41 64 7e 26, /test
1200	61.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55776, 2.04 Changed, TKN:02 b1 00 f5 41 64 7e 26, /test
1275	66.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55777, PUT (text/plain), TKN:b9 a6 12 af dd 88 cd 52, /test
1282	66.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55777, 2.04 Changed, TKN:b9 a6 12 af dd 88 cd 52, /test
1376	71.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	211	CON, MID:55778, PUT (text/plain), TKN:dd 31 5e 8f cc 21 b3 b5, /test
1383	71.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	215	ACK, MID:55778, 2.04 Changed, TKN:dd 31 5e 8f cc 21 b3 b5, /test
1447	76.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55779, PUT (text/plain), TKN:1c 24 aa 8f 1b 47 40 b1, /test
1454	76.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55779, 2.04 Changed, TKN:1c 24 aa 8f 1b 47 40 b1, /test
1529	81.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55780, PUT (text/plain), TKN:4f 72 26 15 18 94 44 da, /test
1536	81.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55780, 2.04 Changed, TKN:4f 72 26 15 18 94 44 da, /test
1597	86.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55781, PUT (text/plain), TKN:45 8d ef 35 65 6a 97 f4, /test
1599	86.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
1601	86.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
1607	86.9	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
1612	86.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55781, 2.04 Changed, TKN:45 8d ef 35 65 6a 97 f4, /test
1698	91.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55782, PUT (text/plain), TKN:0a 41 bc e8 1e fb 50 bc, /test
1705	91.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55782, 2.04 Changed, TKN:0a 41 bc e8 1e fb 50 bc, /test
1717	92.3	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
1720	92.3	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
1774	96.9	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55783, PUT (text/plain), TKN:91 78 02 76 8b 53 be e7, /test
1781	96.9	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55783, 2.04 Changed, TKN:91 78 02 76 8b 53 be e7, /test
1789	97.3	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
1791	97.3	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
1797	97.3	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
1851	102	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55784, PUT (text/plain), TKN:21 62 cc 06 1a 9f c0 a2, /test
1858	102	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55784, 2.04 Changed, TKN:21 62 cc 06 1a 9f c0 a2, /test
1880	103	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
1883	103	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
1953	107	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55785, PUT (text/plain), TKN:b9 69 b2 4b de 29 54 4d, /test
1960	107	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55785, 2.04 Changed, TKN:b9 69 b2 4b de 29 54 4d, /test
2070	112	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55786, PUT (text/plain), TKN:d3 24 85 e2 b9 0b 37 0b, /test
2077	112	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55786, 2.04 Changed, TKN:d3 24 85 e2 b9 0b 37 0b, /test
2163	117	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55787, PUT (text/plain), TKN:57 1f 01 b2 28 20 71 fd, /test
2170	117	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55787, 2.04 Changed, TKN:57 1f 01 b2 28 20 71 fd, /test
2250	122	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55788, PUT (text/plain), TKN:c0 05 ae 9e e8 8f 5c 08, /test
2252	122	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
2254	122	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
2260	122	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
2265	122	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55788, 2.04 Changed, TKN:c0 05 ae 9e e8 8f 5c 08, /test

2347	127	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55789, PUT (text/plain), TKN:8a c7 ca c8 69 a1 b5 af, /test
2354	127	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55789, 2.04 Changed, TKN:8a c7 ca c8 69 a1 b5 af, /test
2451	132	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55790, PUT (text/plain), TKN:0c 7b a7 fb fe 9c 95 e2, /test
2458	132	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55790, 2.04 Changed, TKN:0c 7b a7 fb fe 9c 95 e2, /test
2520	137	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55791, PUT (text/plain), TKN:41 7b b8 df 0b 06 a0 e3, /test
2527	137	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55791, 2.04 Changed, TKN:41 7b b8 df 0b 06 a0 e3, /test
2601	142	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55792, PUT (text/plain), TKN:50 3d 4c cc 05 61 2f 0a, /test
2608	142	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55792, 2.04 Changed, TKN:50 3d 4c cc 05 61 2f 0a, /test
2704	147	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55793, PUT (text/plain), TKN:4a df 3a 62 73 c7 56 81, /test
2712	147	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55793, 2.04 Changed, TKN:4a df 3a 62 73 c7 56 81, /test
2816	152	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55794, PUT (text/plain), TKN:ef dc 80 a0 5c 3f ec b7, /test
2823	152	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55794, 2.04 Changed, TKN:ef dc 80 a0 5c 3f ec b7, /test
2895	157	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55795, PUT (text/plain), TKN:1e 2c 9b 4c 3b 15 da ac, /test
2899	157	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
2906	157	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55795, 2.04 Changed, TKN:1e 2c 9b 4c 3b 15 da ac, /test
2908	157	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
2911	157	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
2987	162	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55796, PUT (text/plain), TKN:fc 2c 22 e8 bd 0f 73 e9, /test
2994	162	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55796, 2.04 Changed, TKN:fc 2c 22 e8 bd 0f 73 e9, /test
3001	162	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
3004	162	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
3081	167	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55797, PUT (text/plain), TKN:38 45 76 62 6d 1b 11 d2, /test
3088	167	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55797, 2.04 Changed, TKN:38 45 76 62 6d 1b 11 d2, /test
3091	167	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
3093	167	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
3096	167	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
3194	172	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55798, PUT (text/plain), TKN:e0 a8 67 af f6 8b 86 7b, /test
3201	172	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55798, 2.04 Changed, TKN:e0 a8 67 af f6 8b 86 7b, /test
3211	172	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
3214	172	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
3345	177	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55799, PUT (text/plain), TKN:1c 93 03 b6 01 22 1d 92, /test
3355	177	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55799, 2.04 Changed, TKN:1c 93 03 b6 01 22 1d 92, /test
3359	177	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55799, 2.04 Changed, TKN:1c 93 03 b6 01 22 1d 92, /test
3363	177	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55799, 2.04 Changed, TKN:1c 93 03 b6 01 22 1d 92, /test
3436	182	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55800, PUT (text/plain), TKN:5c 14 00 0a 21 b6 72 48, /test
3443	182	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55800, 2.04 Changed, TKN:5c 14 00 0a 21 b6 72 48, /test
3515	187	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55801, PUT (text/plain), TKN:b6 2f 3d 45 50 91 03 24, /test
3522	187	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55801, 2.04 Changed, TKN:b6 2f 3d 45 50 91 03 24, /test
3623	192	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55802, PUT (text/plain), TKN:0a 62 2d 16 a6 0d 1a 69, /test
3625	192	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
3627	192	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
3633	192	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
3638	192	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55802, 2.04 Changed, TKN:0a 62 2d 16 a6 0d 1a 69, /test
3723	197	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55803, PUT (text/plain), TKN:4b 14 9e 0b 7c f3 26 5e, /test
3730	197	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55803, 2.04 Changed, TKN:4b 14 9e 0b 7c f3 26 5e, /test
3737	197	fe80::ce28:aaff:fe65:4558	ff02::1	ICMPv6	244	Router Advertisement from cc:28:aa:65:45:58
3800	202	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55804, PUT (text/plain), TKN:5c e3 86 df 75 bc a7 97, /test
3807	202	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55804, 2.04 Changed, TKN:5c e3 86 df 75 bc a7 97, /test
3883	207	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55805, PUT (text/plain), TKN:bc cc fb 18 ba f7 c9 72, /test
3890	207	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55805, 2.04 Changed, TKN:bc cc fb 18 ba f7 c9 72, /test
4032	212	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55806, PUT (text/plain), TKN:ef 23 2d 50 e2 77 c6 e4, /test
4039	212	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55806, 2.04 Changed, TKN:ef 23 2d 50 e2 77 c6 e4, /test
4138	217	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55807, PUT (text/plain), TKN:ab e3 50 b7 bb ca e0 fc, /test
4145	217	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55807, 2.04 Changed, TKN:ab e3 50 b7 bb ca e0 fc, /test
4229	222	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55808, PUT (text/plain), TKN:47 c2 a6 8b 3e 8c db f9, /test
4236	222	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55808, 2.04 Changed, TKN:47 c2 a6 8b 3e 8c db f9, /test
4339	227	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55809, PUT (text/plain), TKN:46 2d 12 2f 04 86 01 b3, /test
4341	227	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
4343	227	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
4349	227	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
4354	227	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55809, 2.04 Changed, TKN:46 2d 12 2f 04 86 01 b3, /test
4478	232	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55810, PUT (text/plain), TKN:3f 47 15 5b e6 3d c8 30, /test
4485	232	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55810, 2.04 Changed, TKN:3f 47 15 5b e6 3d c8 30, /test
4496	232	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
4499	232	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
4578	237	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55811, PUT (text/plain), TKN:ca 48 68 99 6d c3 36 26, /test
4585	237	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55811, 2.04 Changed, TKN:ca 48 68 99 6d c3 36 26, /test
4591	237	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
4593	237	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
4599	237	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
4698	242	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55812, PUT (text/plain), TKN:ab 48 22 c8 93 2d 3c 56, /test
4707	242	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55812, 2.04 Changed, TKN:ab 48 22 c8 93 2d 3c 56, /test
4726	242	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
4731	242	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
4790	247	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55813, PUT (text/plain), TKN:1d f0 1c 68 6d c4 c7 b9, /test
4797	247	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55813, 2.04 Changed, TKN:1d f0 1c 68 6d c4 c7 b9, /test
4924	252	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55814, PUT (text/plain), TKN:f8 bb 9c 7f 63 01 c2 f5, /test
4926	252	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55814, PUT (text/plain), TKN:f8 bb 9c 7f 63 01 c2 f5, /test [Retransmission]
4933	252	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55814, 2.04 Changed, TKN:f8 bb 9c 7f 63 01 c2 f5, /test
5012	257	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55815, PUT (text/plain), TKN:46 08 c6 62 f8 1d 62 7f, /test
5019	257	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55815, 2.04 Changed, TKN:46 08 c6 62 f8 1d 62 7f, /test
5122	262	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55816, PUT (text/plain), TKN:a9 9b 59 63 71 1f 83 62, /test

5124	262	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
5126	262	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
5134	262	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
5139	262	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55816, 2.04 Changed, TKN:a9 9b 59 63 71 1f 83 62, /test
5217	267	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	200	CON, MID:55817, PUT (text/plain), TKN:91 5a e6 96 79 71 c8 5a, /test
5224	267	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55817, 2.04 Changed, TKN:91 5a e6 96 79 71 c8 5a, /test
5331	272	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55818, PUT (text/plain), TKN:62 bf 42 42 30 dc 72 ad, /test
5338	272	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55818, 2.04 Changed, TKN:62 bf 42 42 30 dc 72 ad, /test
5411	277	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55819, PUT (text/plain), TKN:68 79 4d 66 e4 87 87 6b, /test
5418	277	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55819, 2.04 Changed, TKN:68 79 4d 66 e4 87 87 6b, /test
5485	282	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55820, PUT (text/plain), TKN:3c 50 b8 cf 24 85 e8 03, /test
5496	282	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55820, 2.04 Changed, TKN:3c 50 b8 cf 24 85 e8 03, /test
5578	287	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55821, PUT (text/plain), TKN:cf 13 9c 83 d2 e9 06 1f, /test
5585	287	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55821, 2.04 Changed, TKN:cf 13 9c 83 d2 e9 06 1f, /test
5712	292	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55822, PUT (text/plain), TKN:00 4d 8d 22 a5 95 72 e2, /test
5720	292	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55822, 2.04 Changed, TKN:00 4d 8d 22 a5 95 72 e2, /test
5815	297	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55823, PUT (text/plain), TKN:2b b2 24 cc 01 f2 ff 59, /test
5817	297	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
5819	297	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
5825	297	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
5830	297	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55823, 2.04 Changed, TKN:2b b2 24 cc 01 f2 ff 59, /test
5917	302	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55824, PUT (text/plain), TKN:06 9a c2 6b 74 fa e9 17, /test
5924	302	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55824, 2.04 Changed, TKN:06 9a c2 6b 74 fa e9 17, /test
5936	302	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
5939	302	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
6004	307	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55825, PUT (text/plain), TKN:a6 8d 24 06 74 82 46 cc, /test
6011	307	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55825, 2.04 Changed, TKN:a6 8d 24 06 74 82 46 cc, /test
6029	307	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
6031	307	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
6037	307	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
6153	312	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55826, PUT (text/plain), TKN:a4 fb 79 78 bf bd 50 1e, /test
6160	312	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55826, 2.04 Changed, TKN:a4 fb 79 78 bf bd 50 1e, /test
6174	312	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
6177	312	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
6234	317	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55827, PUT (text/plain), TKN:6d 10 f5 95 8c e4 e8 90, /test
6242	317	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55827, 2.04 Changed, TKN:6d 10 f5 95 8c e4 e8 90, /test
6330	322	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55828, PUT (text/plain), TKN:f6 c7 e0 94 0b 1b c2 a8, /test
6337	322	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55828, 2.04 Changed, TKN:f6 c7 e0 94 0b 1b c2 a8, /test
6421	327	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55829, PUT (text/plain), TKN:32 4a c1 76 2a 71 bb 36, /test
6428	327	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55829, 2.04 Changed, TKN:32 4a c1 76 2a 71 bb 36, /test
6560	332	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55830, PUT (text/plain), TKN:a0 39 f9 0f ca 9c 37 95, /test
6562	332	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
6564	332	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
6570	332	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
6575	332	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55830, 2.04 Changed, TKN:a0 39 f9 0f ca 9c 37 95, /test
6646	337	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55831, PUT (text/plain), TKN:04 d7 81 79 76 c8 b8 4c, /test
6653	337	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55831, 2.04 Changed, TKN:04 d7 81 79 76 c8 b8 4c, /test
6750	342	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55832, PUT (text/plain), TKN:12 7e b0 6d 87 39 5f 85, /test
6757	342	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55832, 2.04 Changed, TKN:12 7e b0 6d 87 39 5f 85, /test
6844	347	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55833, PUT (text/plain), TKN:8c 77 38 34 bd 49 a9 13, /test
6852	347	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55833, 2.04 Changed, TKN:8c 77 38 34 bd 49 a9 13, /test
6989	352	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55834, PUT (text/plain), TKN:b0 12 4c be f6 67 42 3e, /test
6996	352	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55834, 2.04 Changed, TKN:b0 12 4c be f6 67 42 3e, /test
7111	357	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55835, PUT (text/plain), TKN:5a 7f 3e 3d 52 5e df ab, /test
7118	357	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55835, 2.04 Changed, TKN:5a 7f 3e 3d 52 5e df ab, /test
7203	362	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55836, PUT (text/plain), TKN:00 a1 9c 8a 4b c3 d0 54, /test
7210	362	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55836, 2.04 Changed, TKN:00 a1 9c 8a 4b c3 d0 54, /test
7286	367	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55837, PUT (text/plain), TKN:17 fb e6 cf 7d 13 a4 94, /test
7288	367	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
7290	367	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
7296	367	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
7301	367	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55837, 2.04 Changed, TKN:17 fb e6 cf 7d 13 a4 94, /test
7420	372	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55838, PUT (text/plain), TKN:08 36 a6 b1 21 9f 6e 94, /test
7431	372	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55838, 2.04 Changed, TKN:08 36 a6 b1 21 9f 6e 94, /test
7444	372	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
7447	372	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
7551	377	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55839, PUT (text/plain), TKN:3c 5e 20 37 f1 68 4d db, /test
7558	377	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55839, 2.04 Changed, TKN:3c 5e 20 37 f1 68 4d db, /test
7566	377	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
7568	377	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
7574	377	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
7674	382	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55840, PUT (text/plain), TKN:04 78 58 6d 66 0b dc dc, /test
7681	382	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55840, 2.04 Changed, TKN:04 78 58 6d 66 0b dc dc, /test
7738	383	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
7741	383	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
7816	387	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55841, PUT (text/plain), TKN:4a 60 78 d9 8e fe a1 80, /test
7820	387	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55841, 2.04 Changed, TKN:4a 60 78 d9 8e fe a1 80, /test
7953	392	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55842, PUT (text/plain), TKN:b6 84 0c 69 db 8e 14 64, /test
7960	392	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55842, 2.04 Changed, TKN:b6 84 0c 69 db 8e 14 64, /test
8060	397	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55843, PUT (text/plain), TKN:7d 3d 4a e8 fd f7 e8 04, /test
8067	397	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55843, 2.04 Changed, TKN:7d 3d 4a e8 fd f7 e8 04, /test
8213	402	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55844, PUT (text/plain), TKN:8c 2c 65 ea e1 1f 8a be, /test

8215	402	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
8217	402	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
8223	402	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
8228	402	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55844, 2.04 Changed, TKN:8c 2c 65 ea e1 1f 8a be, /test
8321	407	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55845, PUT (text/plain), TKN:7f 36 f1 c2 c1 80 e0 fa, /test
8328	407	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55845, 2.04 Changed, TKN:7f 36 f1 c2 c1 80 e0 fa, /test
8442	412	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55846, PUT (text/plain), TKN:4d fb 17 5a 92 ed 99 ce, /test
8449	412	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55846, 2.04 Changed, TKN:4d fb 17 5a 92 ed 99 ce, /test
8569	417	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55847, PUT (text/plain), TKN:61 d7 8d 65 c2 05 e4 72, /test
8576	417	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55847, 2.04 Changed, TKN:61 d7 8d 65 c2 05 e4 72, /test
8668	422	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55848, PUT (text/plain), TKN:fb ba 91 2e ff b7 ad 31, /test
8675	422	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55848, 2.04 Changed, TKN:fb ba 91 2e ff b7 ad 31, /test
8783	427	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55849, PUT (text/plain), TKN:f0 b0 7c a3 d5 9e 65 96, /test
8790	427	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55849, 2.04 Changed, TKN:f0 b0 7c a3 d5 9e 65 96, /test
8923	432	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55850, PUT (text/plain), TKN:60 51 13 18 41 88 a8 46, /test
8930	432	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55850, 2.04 Changed, TKN:60 51 13 18 41 88 a8 46, /test
9007	437	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55851, PUT (text/plain), TKN:3a 09 51 9f 11 c4 57 cc, /test
9013	437	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
9016	437	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55851, 2.04 Changed, TKN:3a 09 51 9f 11 c4 57 cc, /test
9019	437	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
9021	437	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
9024	437	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
9094	442	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55852, PUT (text/plain), TKN:f1 85 d8 fe 0a 31 1a 71, /test
9101	442	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55852, 2.04 Changed, TKN:f1 85 d8 fe 0a 31 1a 71, /test
9113	442	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
9116	442	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
9211	447	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	200	CON, MID:55853, PUT (text/plain), TKN:b9 40 68 18 06 4f d0 3f, /test
9218	447	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55853, 2.04 Changed, TKN:b9 40 68 18 06 4f d0 3f, /test
9222	447	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
9224	447	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
9230	447	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
9365	452	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55854, PUT (text/plain), TKN:93 79 df fb 7e aa d1 fe, /test
9372	452	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55854, 2.04 Changed, TKN:93 79 df fb 7e aa d1 fe, /test
9389	452	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
9392	452	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
9481	457	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55855, PUT (text/plain), TKN:d7 72 44 de d3 fb f3 ee, /test
9488	457	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55855, 2.04 Changed, TKN:d7 72 44 de d3 fb f3 ee, /test
9587	462	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55856, PUT (text/plain), TKN:c1 02 90 91 ff ab 34 56, /test
9595	462	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55856, 2.04 Changed, TKN:c1 02 90 91 ff ab 34 56, /test
9657	467	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55857, PUT (text/plain), TKN:6f 94 7a d7 ec c9 c1 4f, /test
9664	467	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55857, 2.04 Changed, TKN:6f 94 7a d7 ec c9 c1 4f, /test
9668	467	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55857, 2.04 Changed, TKN:6f 94 7a d7 ec c9 c1 4f, /test
9782	472	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55858, PUT (text/plain), TKN:cd c3 0c 54 b9 4d bc e4, /test
9784	472	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
9786	472	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
9792	472	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
9797	472	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55858, 2.04 Changed, TKN:cd c3 0c 54 b9 4d bc e4, /test
9887	477	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55859, PUT (text/plain), TKN:59 94 1c 4d 7a 07 22 1c, /test
9894	477	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55859, 2.04 Changed, TKN:59 94 1c 4d 7a 07 22 1c, /test
9907	477	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
9910	477	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
9998	482	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55860, PUT (text/plain), TKN:84 0f 69 d3 ae 0f 3a af, /test
10005	482	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55860, 2.04 Changed, TKN:84 0f 69 d3 ae 0f 3a af, /test
10012	482	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
10014	482	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
10020	482	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
10119	487	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55861, PUT (text/plain), TKN:4c d2 c0 6f 6a dc c2 39, /test
10126	487	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55861, 2.04 Changed, TKN:4c d2 c0 6f 6a dc c2 39, /test
10157	488	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
10160	488	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
10269	492	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55862, PUT (text/plain), TKN:77 66 ed 10 2c 20 c5 f9, /test
10276	492	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55862, 2.04 Changed, TKN:77 66 ed 10 2c 20 c5 f9, /test
10339	497	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55863, PUT (text/plain), TKN:20 d0 ab 80 a5 57 a1 4f, /test
10346	497	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55863, 2.04 Changed, TKN:20 d0 ab 80 a5 57 a1 4f, /test
10449	502	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55864, PUT (text/plain), TKN:8f 94 44 31 96 75 81 3e, /test
10456	502	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55864, 2.04 Changed, TKN:8f 94 44 31 96 75 81 3e, /test
10542	507	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55865, PUT (text/plain), TKN:51 ac 14 43 cf 7d 6e 65, /test
10544	507	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
10546	507	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
10552	507	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
10557	507	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55865, 2.04 Changed, TKN:51 ac 14 43 cf 7d 6e 65, /test
10683	512	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55866, PUT (text/plain), TKN:36 4d ef bf 25 b1 bd 47, /test
10690	512	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55866, 2.04 Changed, TKN:36 4d ef bf 25 b1 bd 47, /test
10704	512	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
10707	512	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
10812	517	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55867, PUT (text/plain), TKN:5c 16 f1 01 14 bb 0a 75, /test
10816	517	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55867, 2.04 Changed, TKN:5c 16 f1 01 14 bb 0a 75, /test
10822	517	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
10824	517	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
10830	517	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
10940	522	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	212	CON, MID:55868, PUT (text/plain), TKN:fe 48 d7 10 cc 96 24 e1, /test
10947	522	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	216	ACK, MID:55868, 2.04 Changed, TKN:fe 48 d7 10 cc 96 24 e1, /test

10961	522	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
10964	522	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
11043	527	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55869, PUT (text/plain), TKN:f5 6c e3 29 87 df 6b a2, /test
11050	527	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55869, 2.04 Changed, TKN:f5 6c e3 29 87 df 6b a2, /test
11190	532	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55870, PUT (text/plain), TKN:e5 1b c9 dc 19 fa 9f b7, /test
11197	532	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55870, 2.04 Changed, TKN:e5 1b c9 dc 19 fa 9f b7, /test
11281	537	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55871, PUT (text/plain), TKN:10 fe 0d 58 7e 35 9c 1d, /test
11288	537	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55871, 2.04 Changed, TKN:10 fe 0d 58 7e 35 9c 1d, /test
11376	542	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55872, PUT (text/plain), TKN:3f 0b fa ec e2 eb 67 68, /test
11378	542	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
11380	542	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ff02::1:ff95:dc2f	ICMPv6	180	Neighbor Solicitation for 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f from f4:ce:36:00:6d:37
11386	542	2a00:801:7b6:be7c:ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:39ed:6c76:3695:dc2f (rtr, sol) is at cc:28:aa:65:45:58
11391	542	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55872, 2.04 Changed, TKN:3f 0b fa ec e2 eb 67 68, /test
11487	547	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55873, PUT (text/plain), TKN:e8 eb bb e0 aa b2 5f af, /test
11494	547	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55873, 2.04 Changed, TKN:e8 eb bb e0 aa b2 5f af, /test
11499	547	fe80::ce28:aaff:fe65:4558	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
11502	547	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement 2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
11623	552	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55874, PUT (text/plain), TKN:06 f7 fd 1a 41 47 04 6c, /test
11630	552	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55874, 2.04 Changed, TKN:06 f7 fd 1a 41 47 04 6c, /test
11633	552	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	198	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
11635	552	fe80::f6ce:36ff:fe00:6d37	ff02::1:ff65:4558	ICMPv6	180	Neighbor Solicitation for fe80::ce28:aaff:fe65:4558 from f4:ce:36:00:6d:37
11638	552	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Advertisement fe80::ce28:aaff:fe65:4558 (rtr, sol, ovr) is at cc:28:aa:65:45:58
11714	557	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55875, PUT (text/plain), TKN:a5 c1 94 11 85 a9 bd 34, /test
11721	557	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55875, 2.04 Changed, TKN:a5 c1 94 11 85 a9 bd 34, /test
11733	557	fe80::ce28:aaff:fe65:4558	fe80::f6ce:36ff:fe00:6d37	ICMPv6	198	Neighbor Solicitation for fe80::f6ce:36ff:fe00:6d37 from cc:28:aa:65:45:58
11736	557	fe80::f6ce:36ff:fe00:6d37	fe80::ce28:aaff:fe65:4558	ICMPv6	198	Neighbor Advertisement fe80::f6ce:36ff:fe00:6d37 (sol, ovr) is at f4:ce:36:00:6d:37
11809	562	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	CoAP	213	CON, MID:55876, PUT (text/plain), TKN:26 98 73 d4 7d b1 80 8a, /test
11816	562	2a00:801:7b6:be7c:39ed:6c76:3695:dc2f	2a00:801:7b6:be7c:f6ce:36ff:fe00:6d37	CoAP	217	ACK, MID:55876, 2.04 Changed, TKN:26 98 73 d4 7d b1 80 8a, /test

E | Sensor Tests Results

The following pages present the results of the sensor tests, including:

- Test report for the PS mode test
- Test reports for the TWT tests with TWT intervals ranging from 5 to 20 seconds and TWT session durations between 8 and 64 ms.
- Test report for the recovery mode test.
- Power measurements.

Test Report - Sensor Use Case - PS

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

Test Setup

Iterations: 1000
PS Mode: Legacy
PS Wake Up Mode: DTIM

Test Results

Requests Sent: 1000
Responses Received: 991
Average Latency: 139 ms

Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

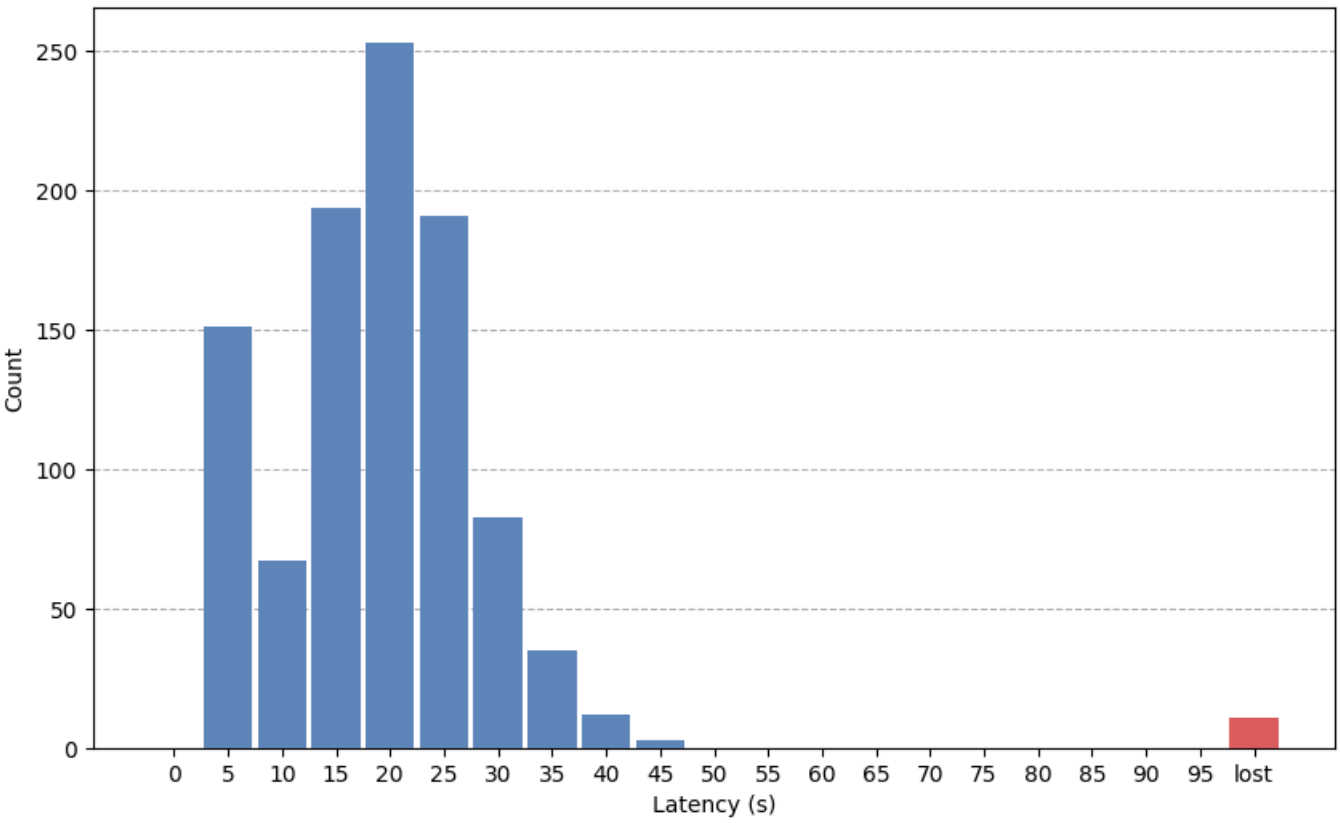
Test Setup

Iterations: 1000
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 8 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 989
Average Latency: 18 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

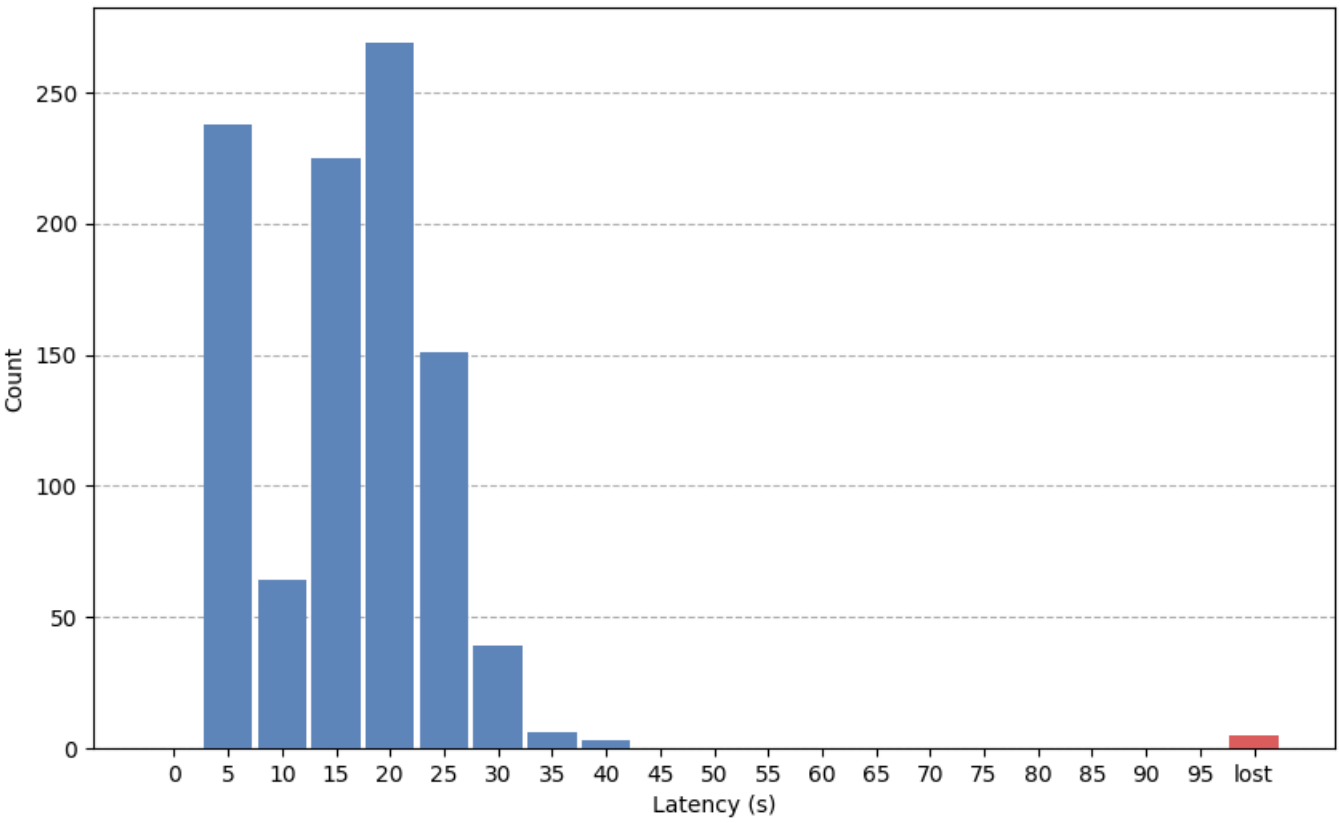
Test Setup

Iterations: 1000
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 16 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 995
Average Latency: 15 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

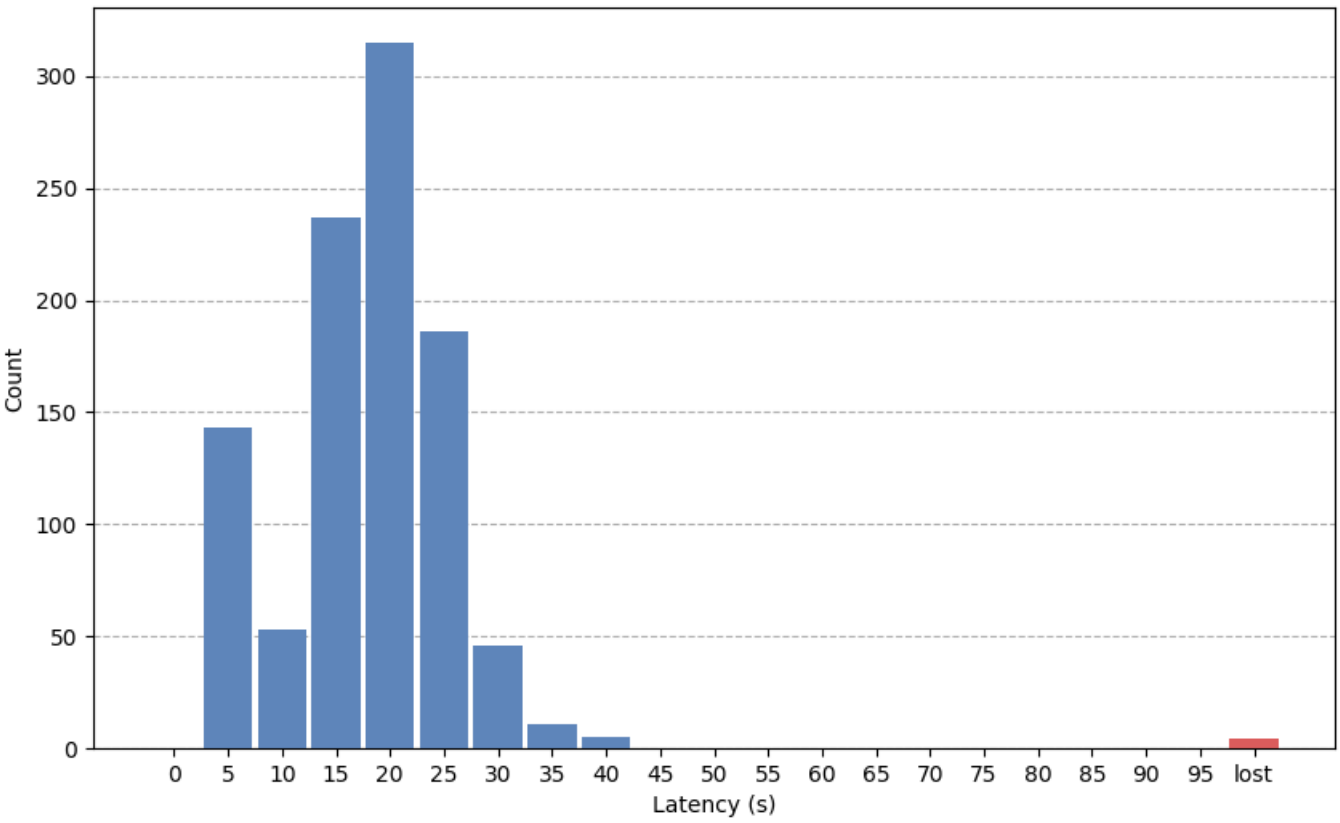
Test Setup

Iterations: 1000
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 32 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 996
Average Latency: 17 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

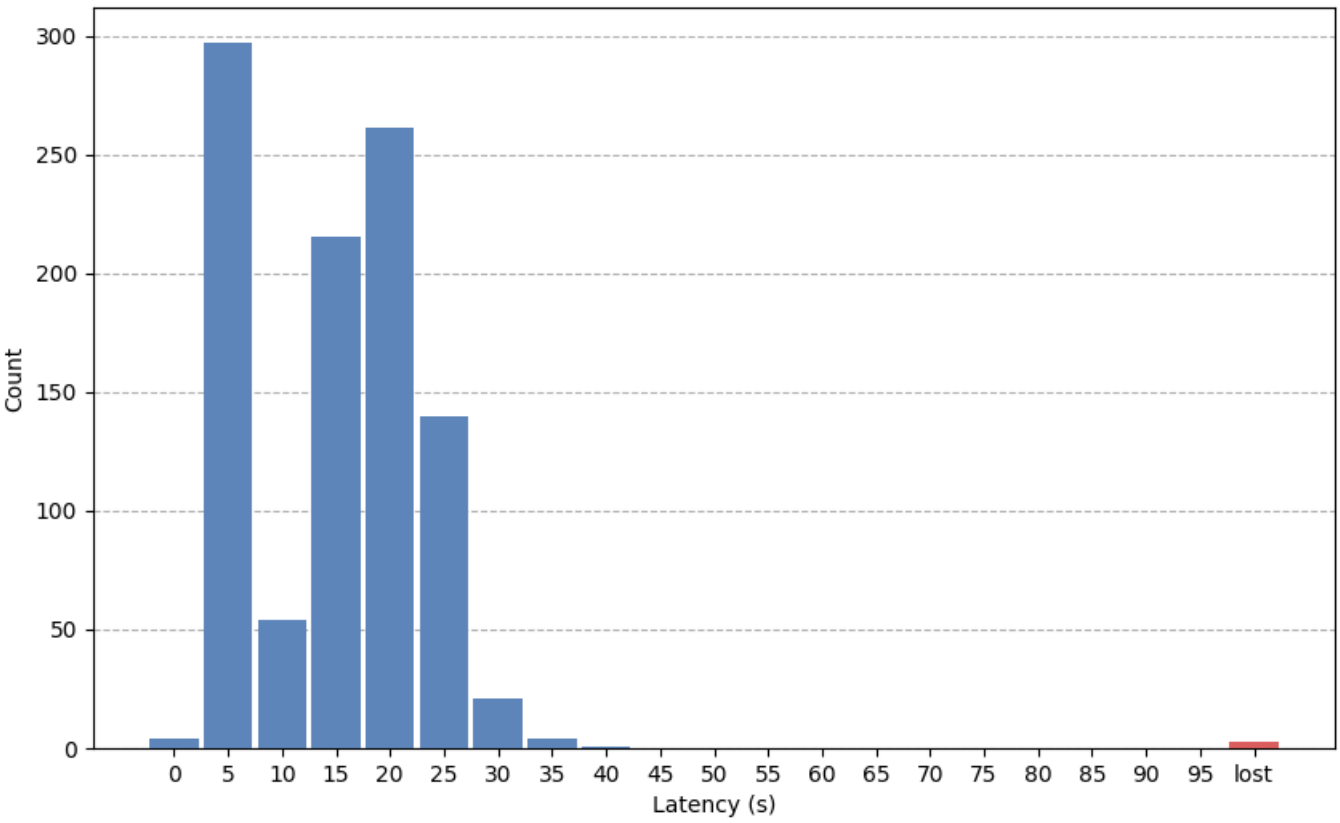
Test Setup

Iterations: 1000
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 57 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 997
Average Latency: 14 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

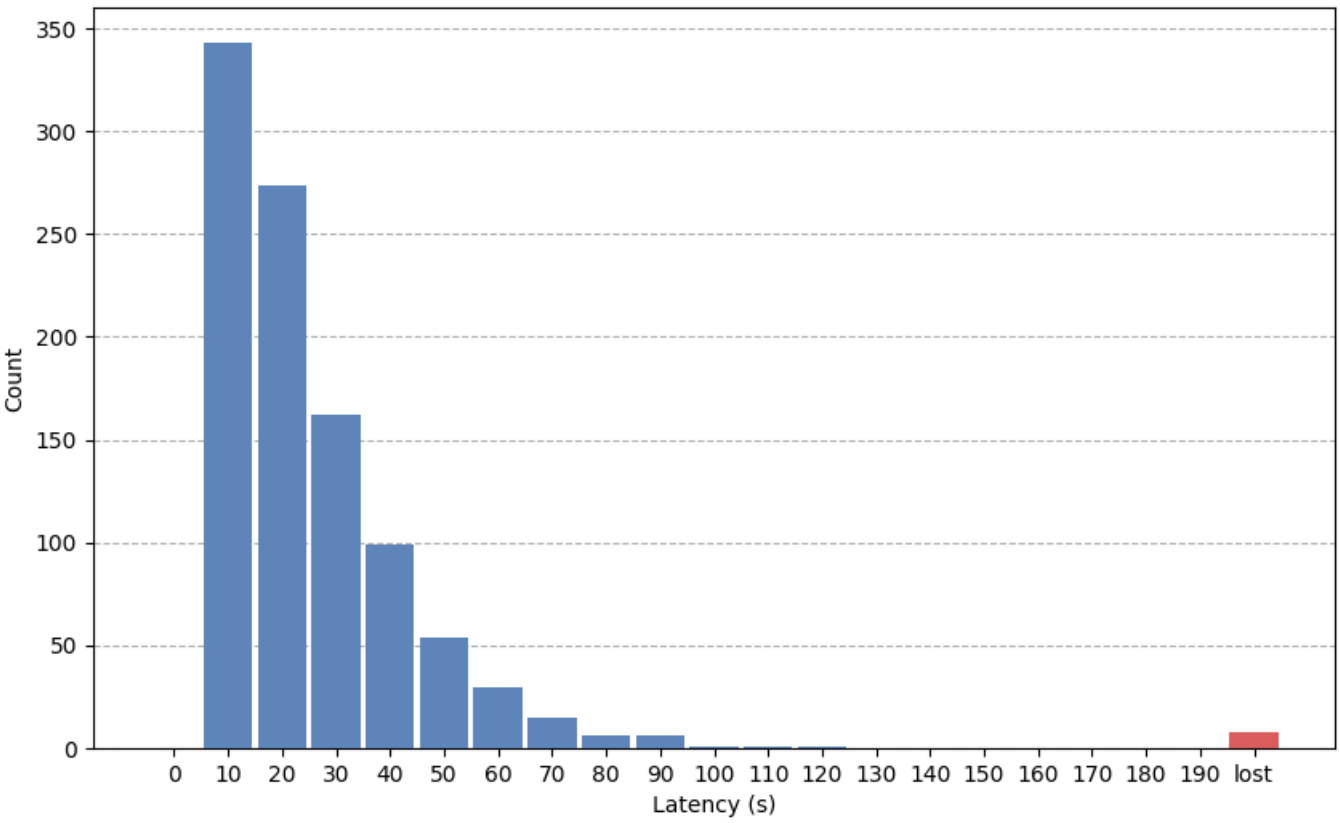
Test Setup

Iterations: 1000
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 8 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 992
Average Latency: 24 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

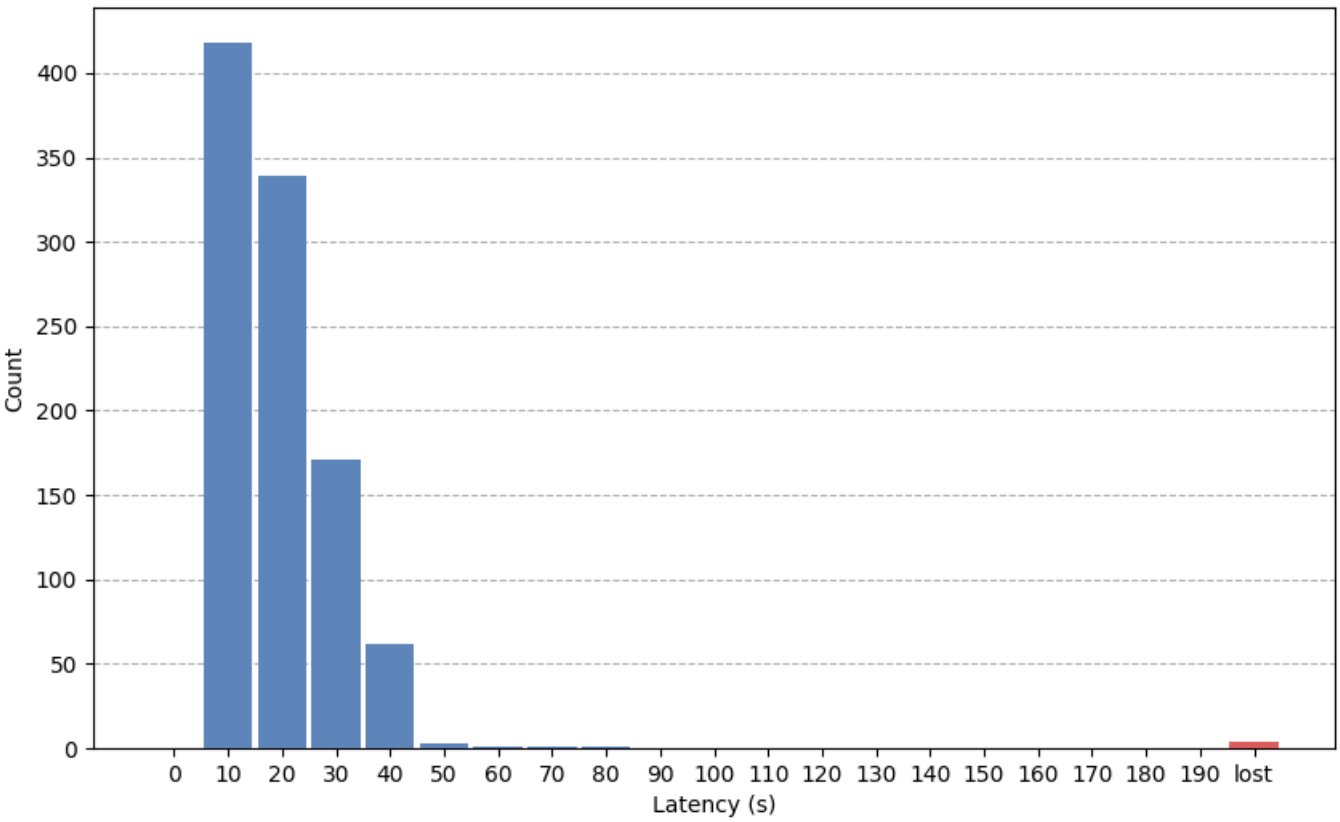
Test Setup

Iterations: 1000
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 16 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 996
Average Latency: 19 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

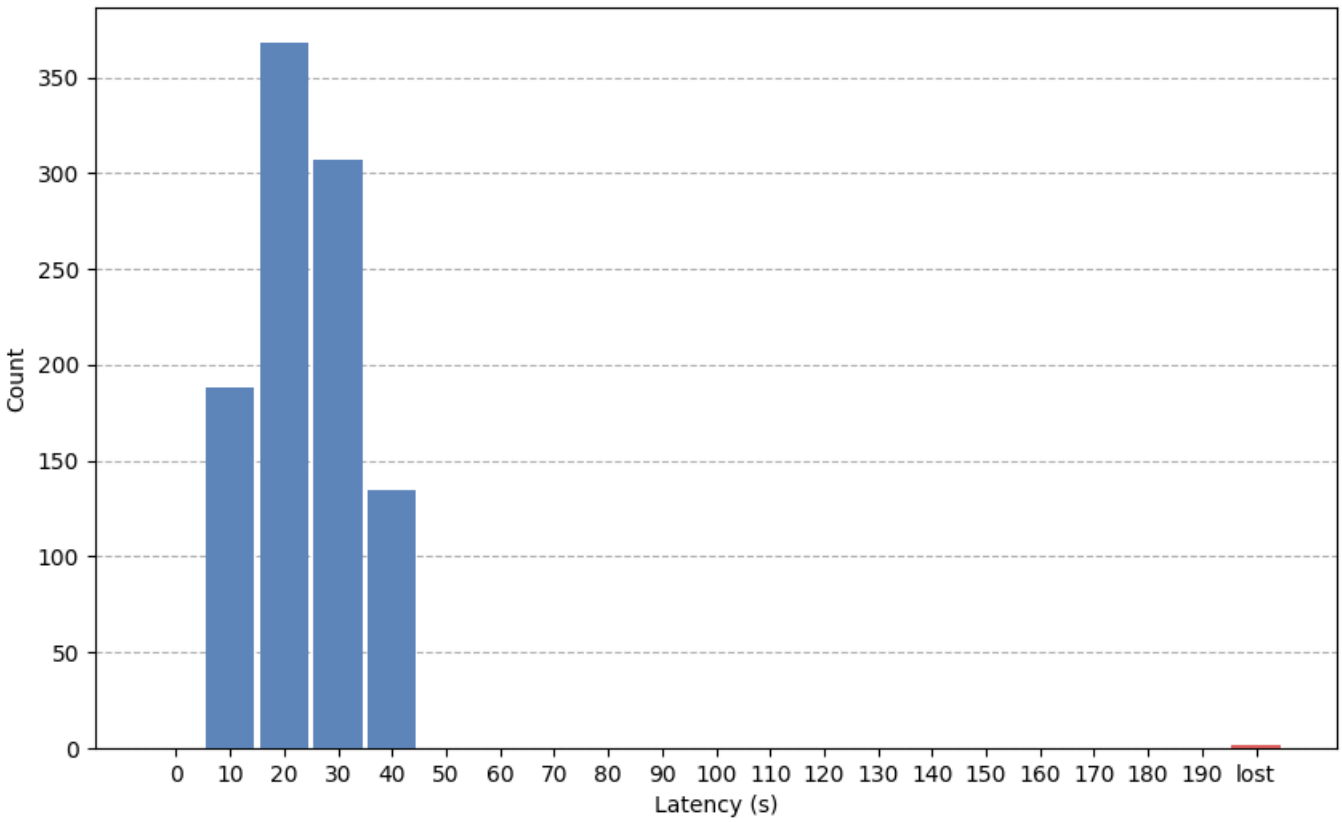
Test Setup

Iterations: 1000
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 32 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 998
Average Latency: 23 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

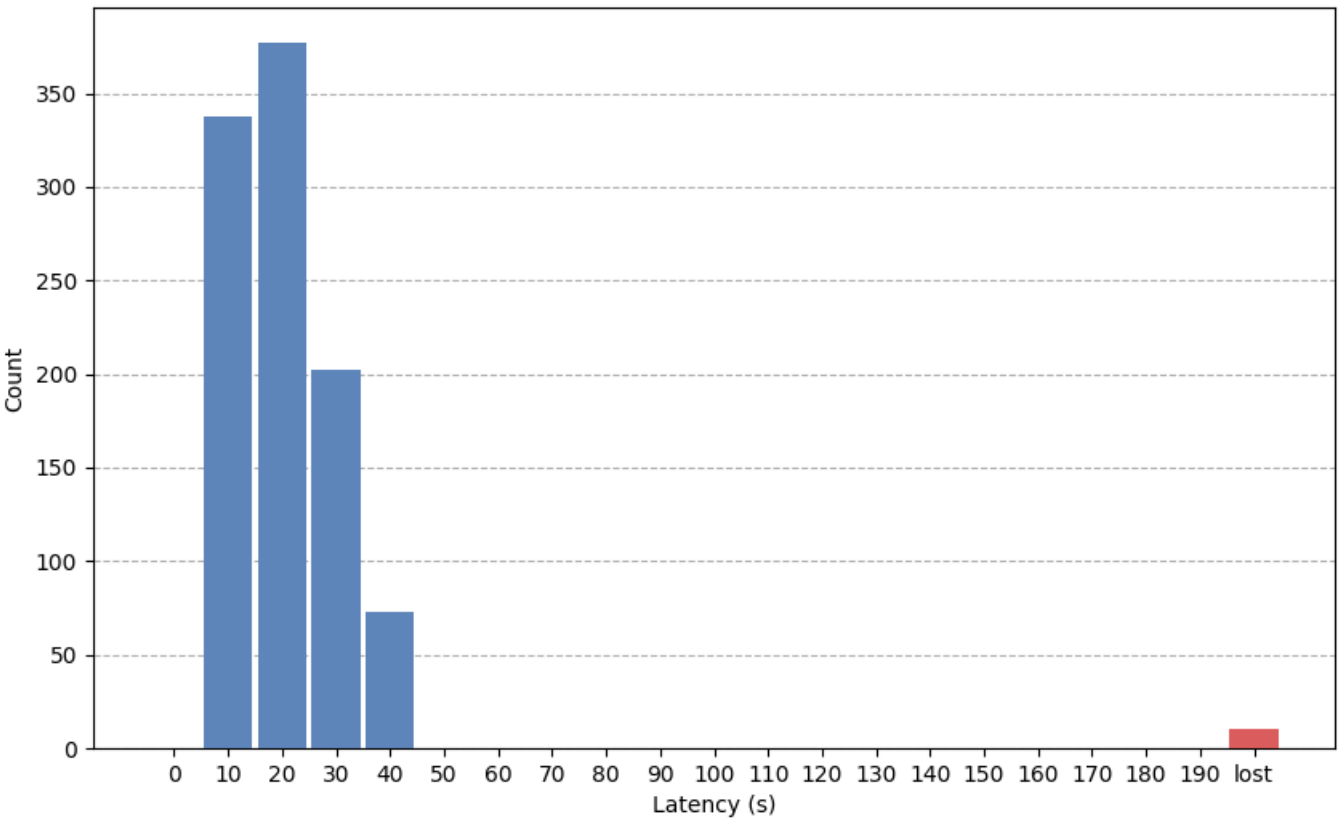
Test Setup

Iterations: 1000
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 57 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 990
Average Latency: 20 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

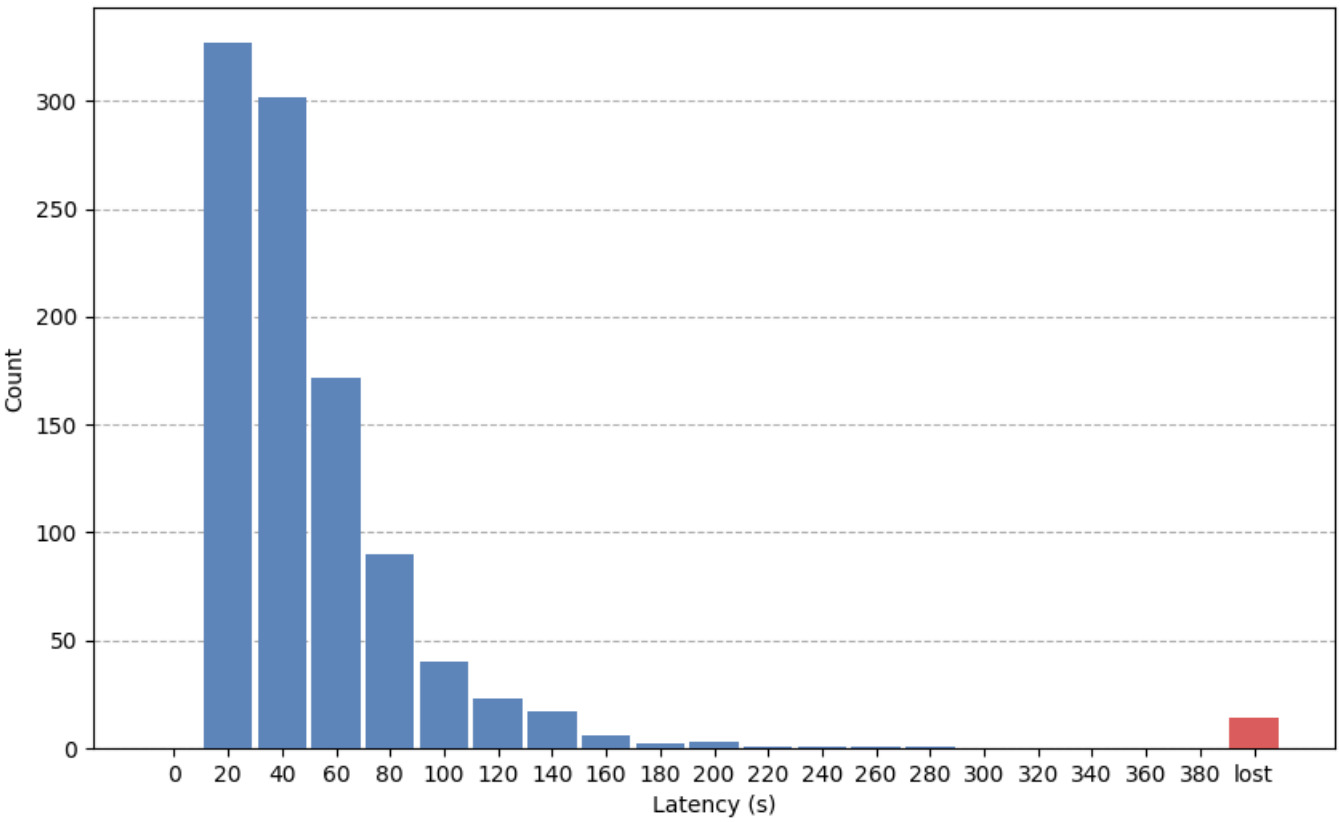
Test Setup

Iterations: 1000
Negotiated TWT Interval: 20 s
Negotiated TWT Wake Interval: 8 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 986
Average Latency: 48 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

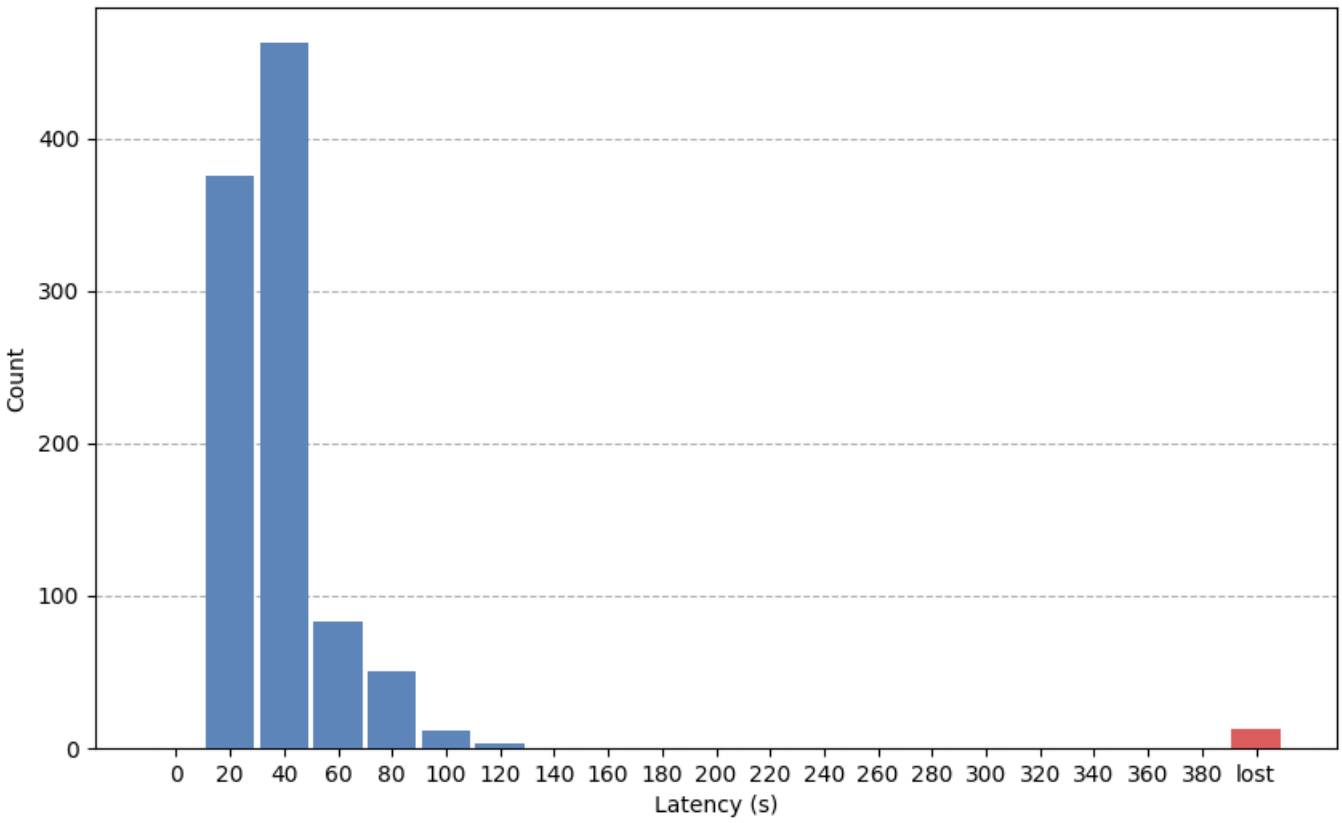
Test Setup

Iterations: 1000
Negotiated TWT Interval: 20 s
Negotiated TWT Wake Interval: 16 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 987
Average Latency: 37 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

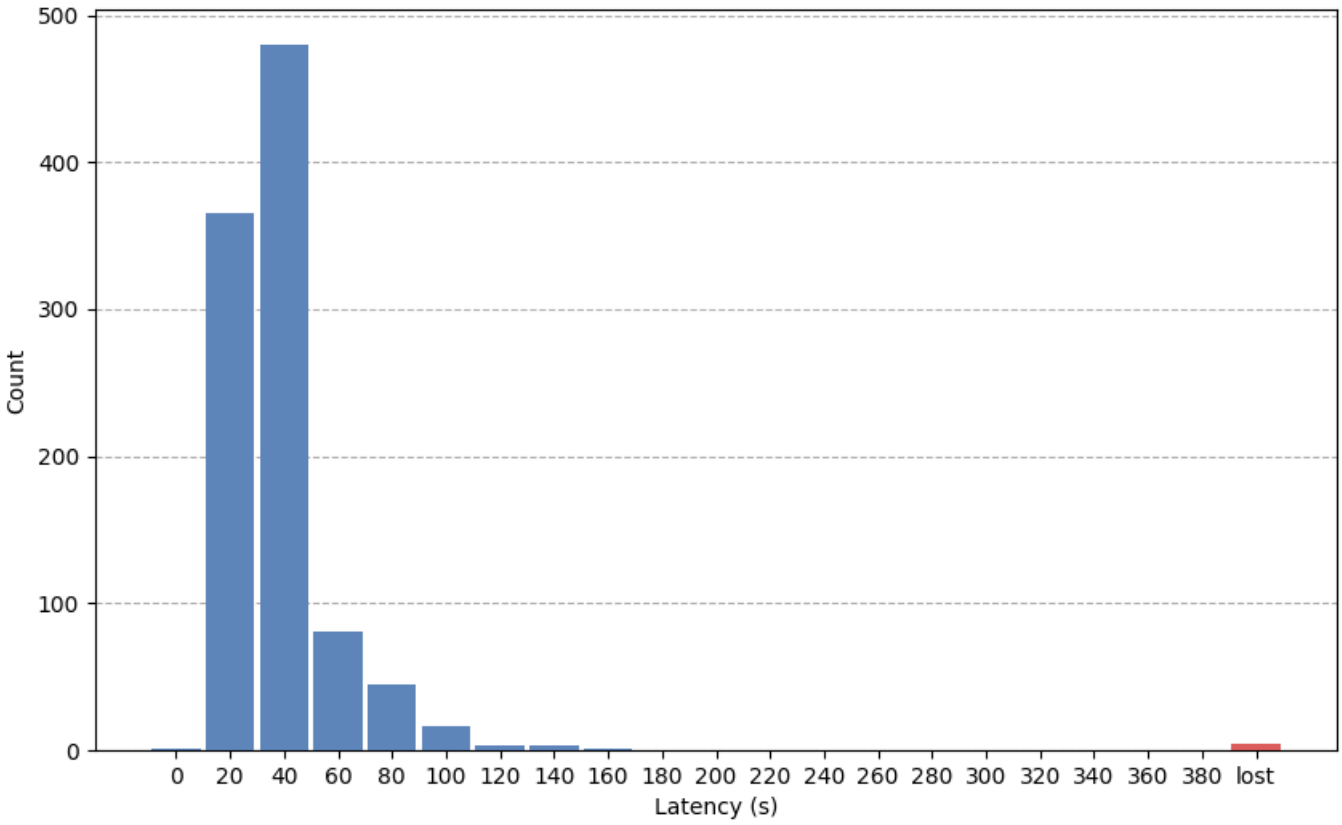
Test Setup

Iterations: 1000
Negotiated TWT Interval: 20 s
Negotiated TWT Wake Interval: 32 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 996
Average Latency: 37 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

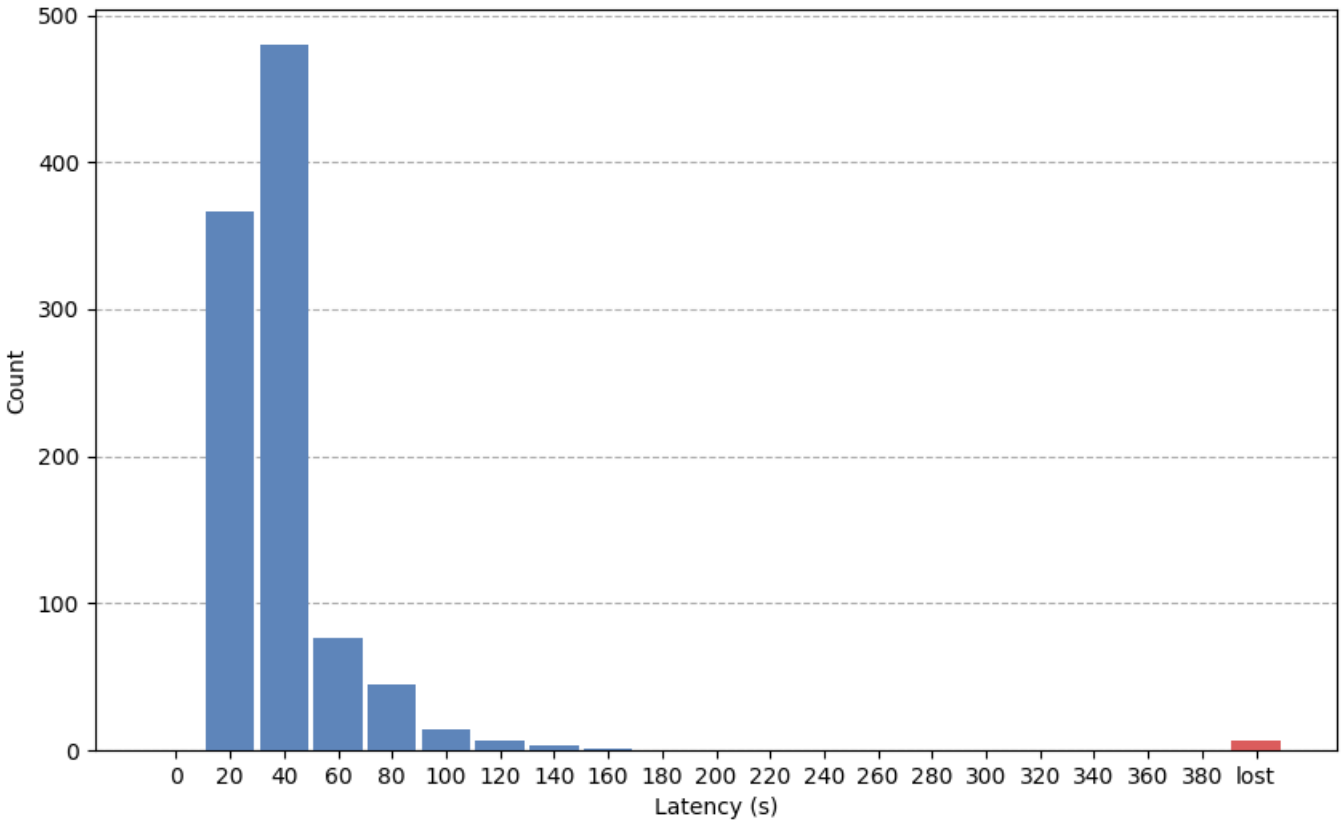
Test Setup

Iterations: 1000
Negotiated TWT Interval: 20 s
Negotiated TWT Wake Interval: 57 ms
Recovery: Disabled

Test Results

Requests Sent: 1000
Responses Received: 993
Average Latency: 37 s

Response Time Histogram



Test Report - Sensor Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

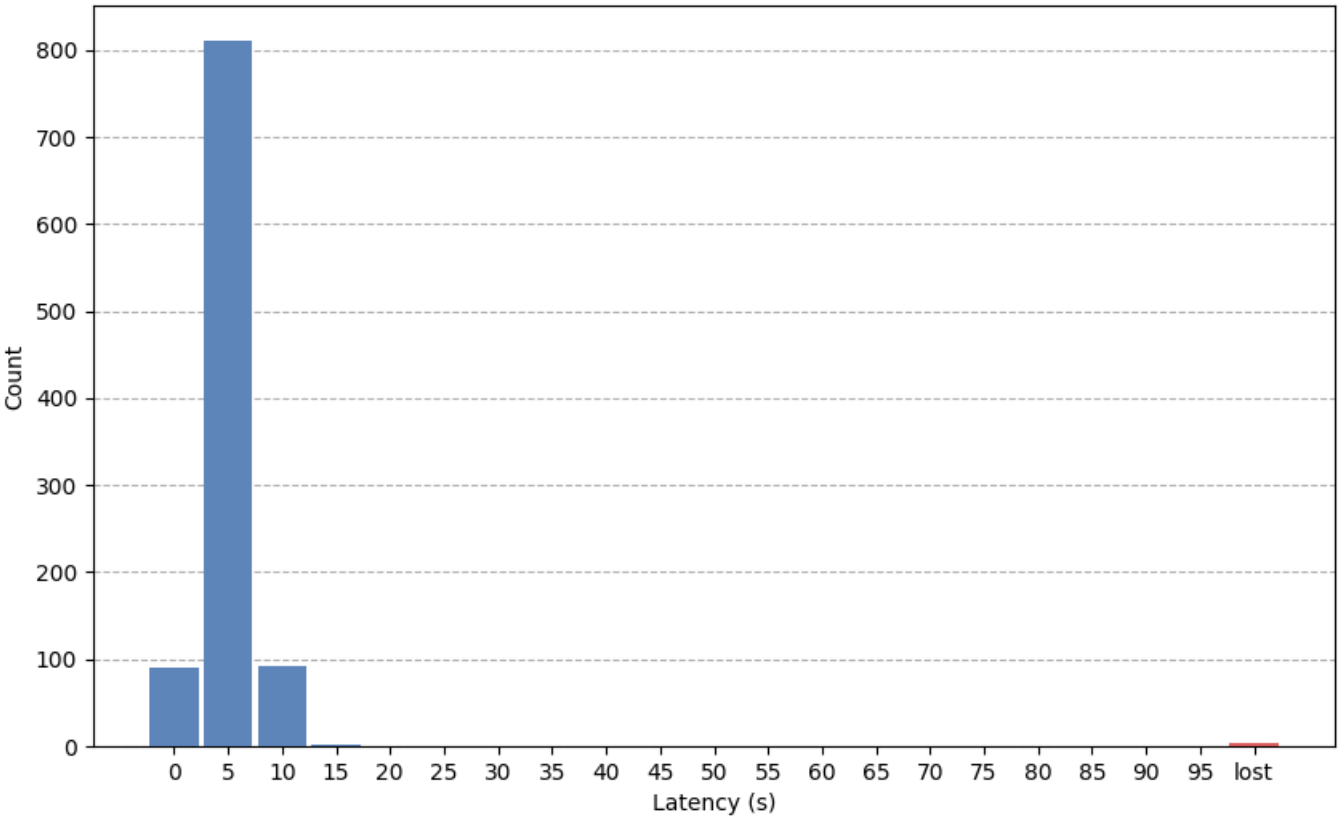
Test Setup

Iterations: 1000
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 8 ms
Recovery: Enabled
Max Pending Requests Before Recover: 2

Test Results

Requests Sent: 1000
Responses Received: 997
Average Latency: 5 s
Recovery Count: 91

Response Time Histogram



Power measurements

The table below shows the average current consumption of the nRF7002 Wi-Fi module for each test.

	5 Seconds Interval	10 Seconds Interval	20 Seconds Interval	Unit
PS Mode				
PS Mode (DTIM/Legacy)	5380	4420	4270	µA
TWT - Recovery mode disabled				
TWT - 8 ms SP	556	300	153	µA
TWT - 16 ms SP	656	337	178	µA
TWT - 32 ms SP	808	417	227	µA
TWT - 57 ms SP	1043	544	282	µA
TWT - Recovery mode enabled				
TWT - 8 ms SP	479	-	-	µA

Table E.1: Power measurements for the sensor use case

The power measurements were performed using the Power Profiler Kit II, following the procedure described in [49].

Note that the displayed SP durations correspond to the negotiated durations, not the requested duration, which explains the 57 ms instead of 64 ms.

F | Large Packet Tests Results

The following pages present the results of the large packet tests, including:

- Test report for the PS mode test
- Test reports for the TWT tests with TWT intervals ranging from 5 to 10 seconds and TWT session durations between 8 and 64 ms.
- Power measurements

Test Report - Large Packet Use Case - PS

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
PS Mode: Legacy
PS Wake Up Mode: DTIM

Test Results

Requests Sent: 1000
Requests Received on Server: 998
Responses Received: 998
Requests Lost: 2
Responses Lost: 0
Average Latency: 76 ms

Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

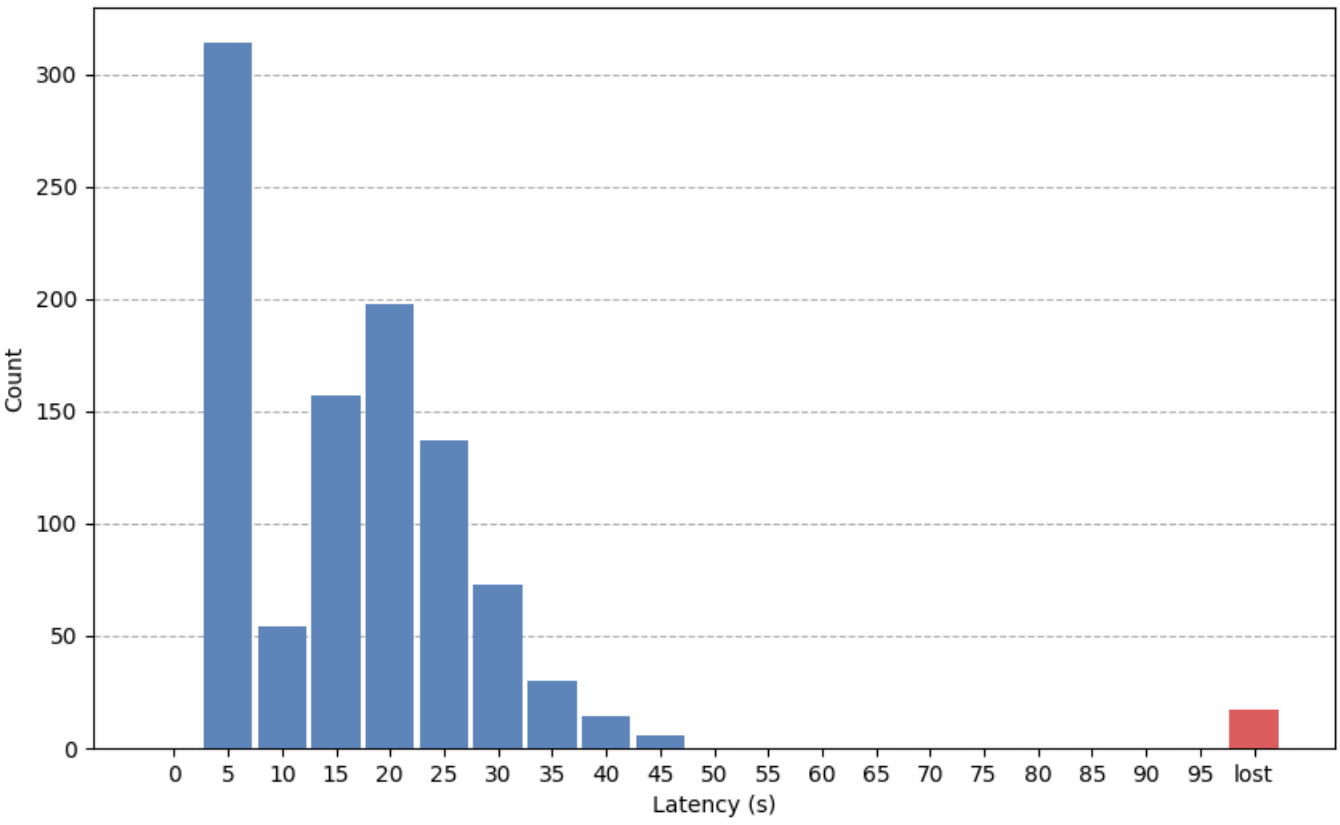
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 8 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 1000
Responses Received: 983
Requests Lost: 0
Responses Lost: 17
Average Latency: 16 s

Response Time Histogram



Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

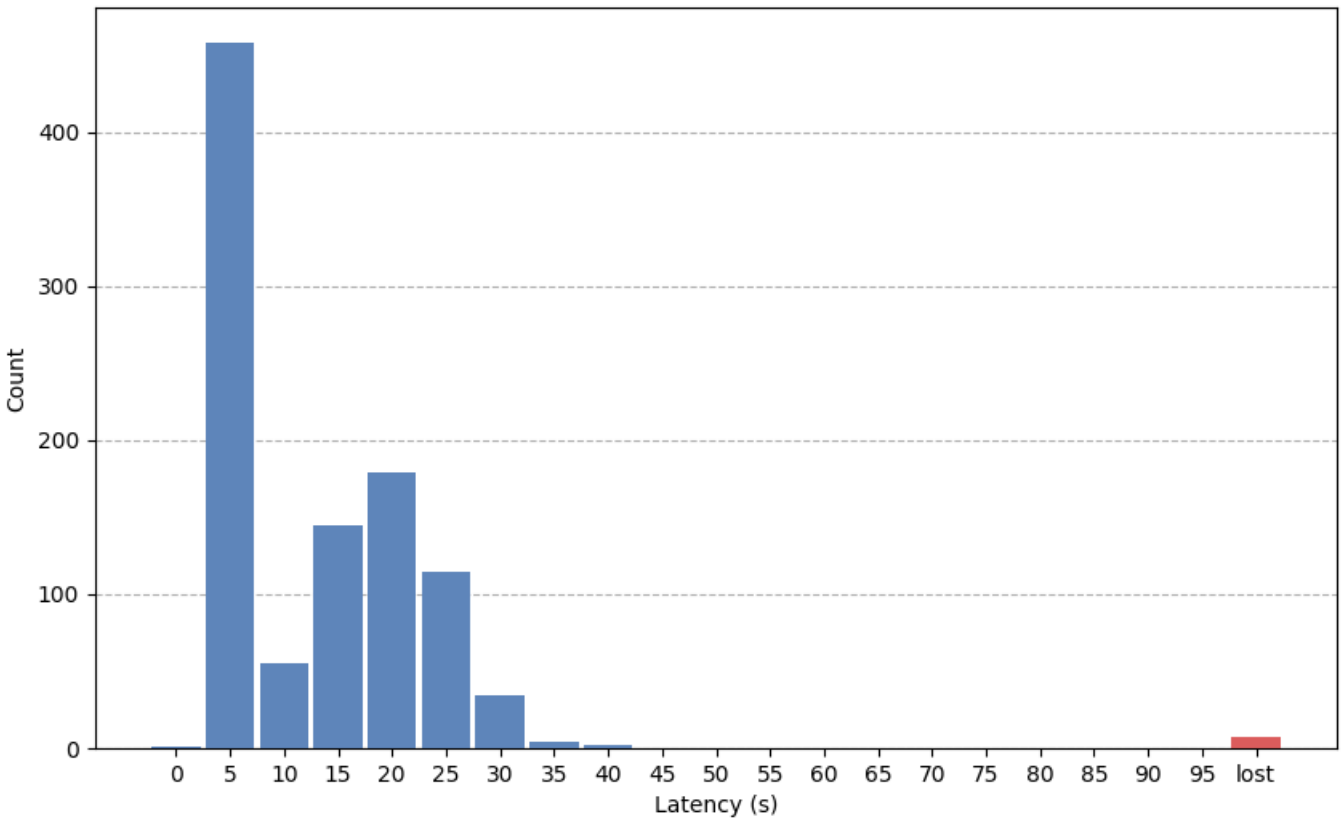
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 16 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 997
Responses Received: 993
Requests Lost: 3
Responses Lost: 4
Average Latency: 12 s

Response Time Histogram



Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

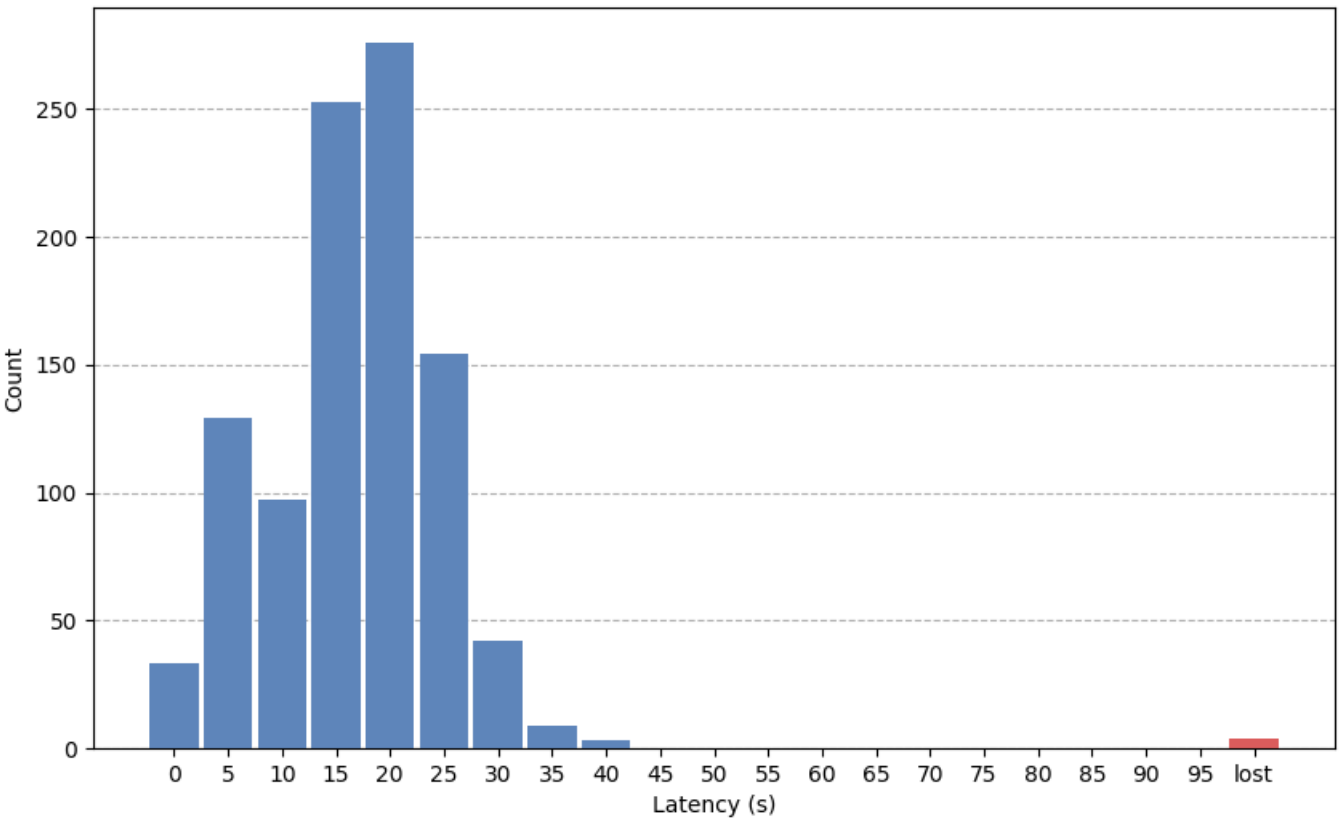
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 32 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 999
Responses Received: 996
Requests Lost: 1
Responses Lost: 3
Average Latency: 16 s

Response Time Histogram



Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

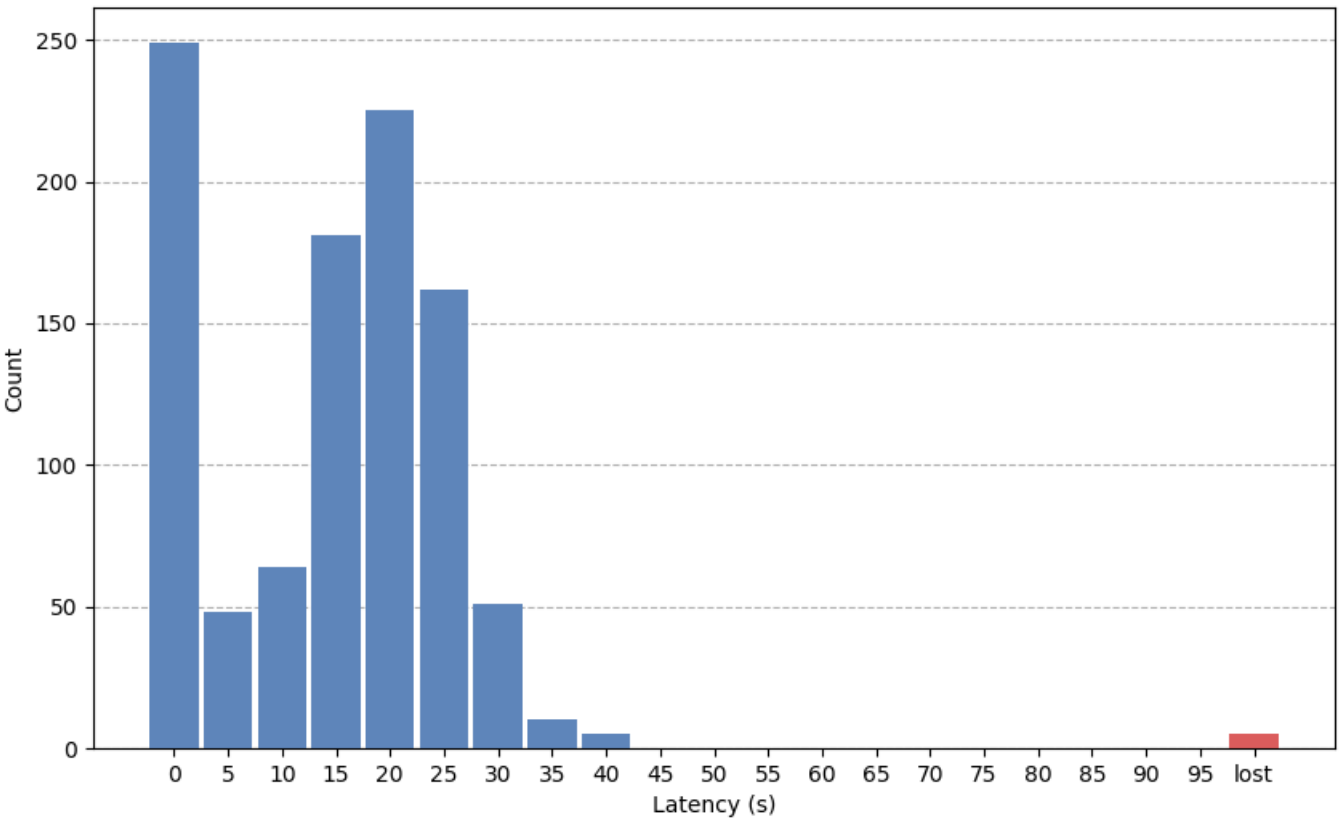
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 57 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 997
Responses Received: 995
Requests Lost: 3
Responses Lost: 2
Average Latency: 14 s

Response Time Histogram



Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

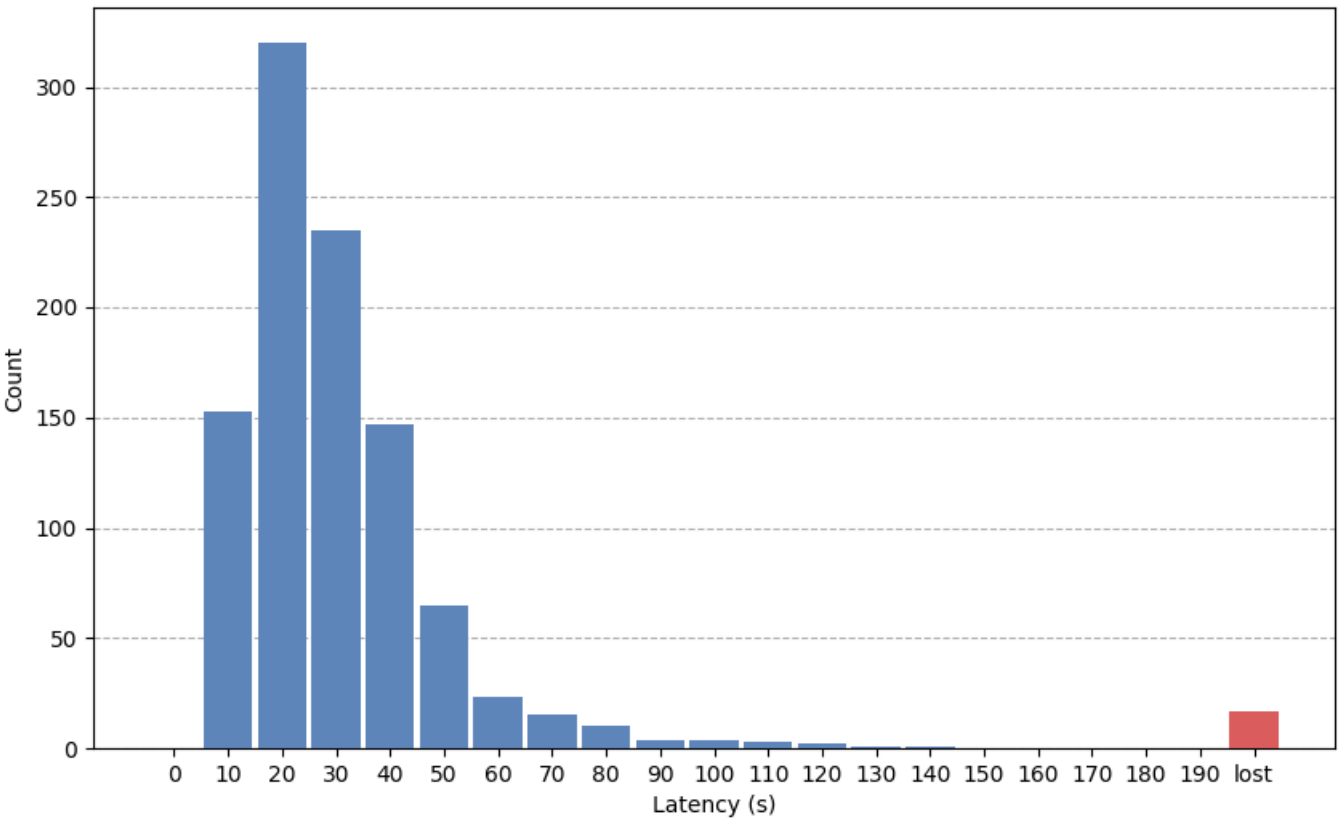
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 8 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 998
Responses Received: 983
Requests Lost: 2
Responses Lost: 15
Average Latency: 29 s

Response Time Histogram



Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

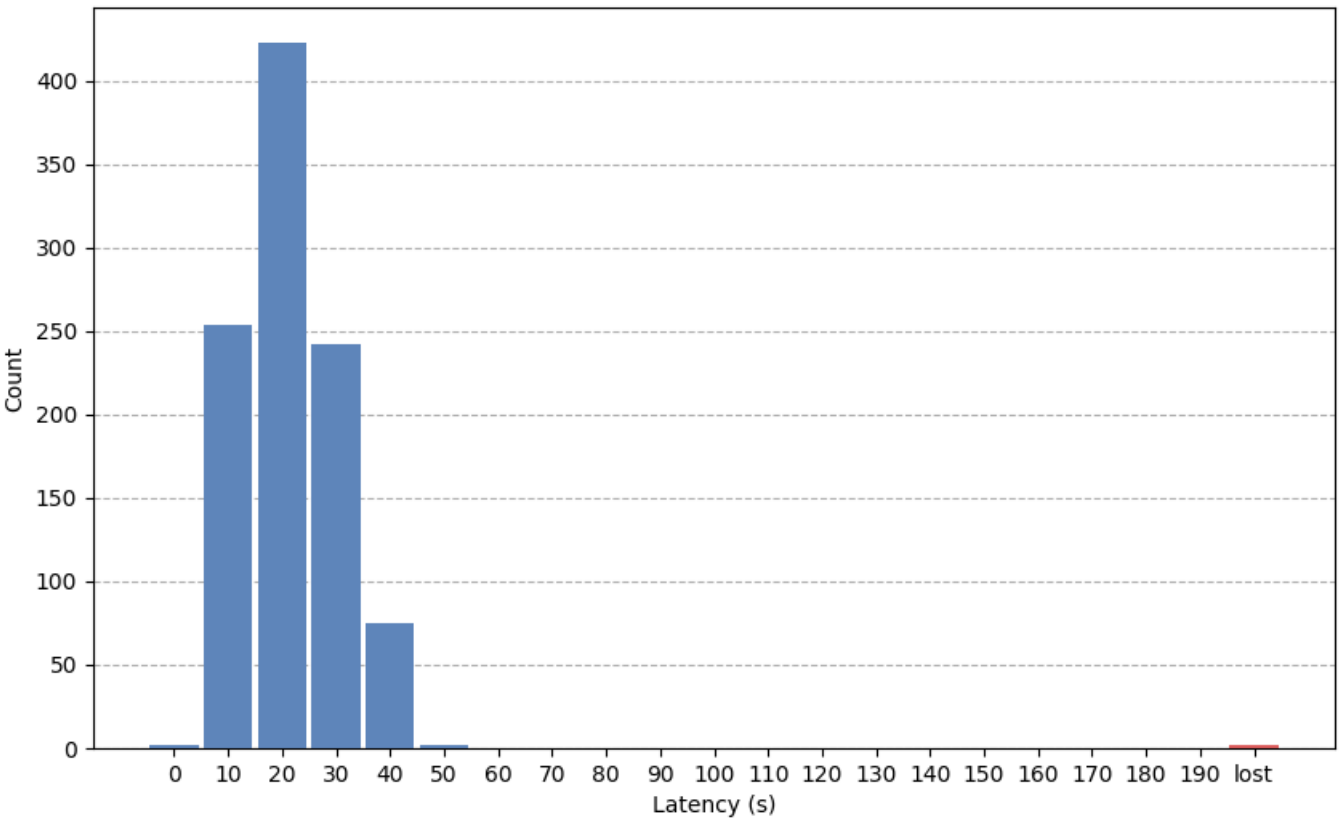
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 16 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 1000
Responses Received: 998
Requests Lost: 0
Responses Lost: 2
Average Latency: 21 s

Response Time Histogram



Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

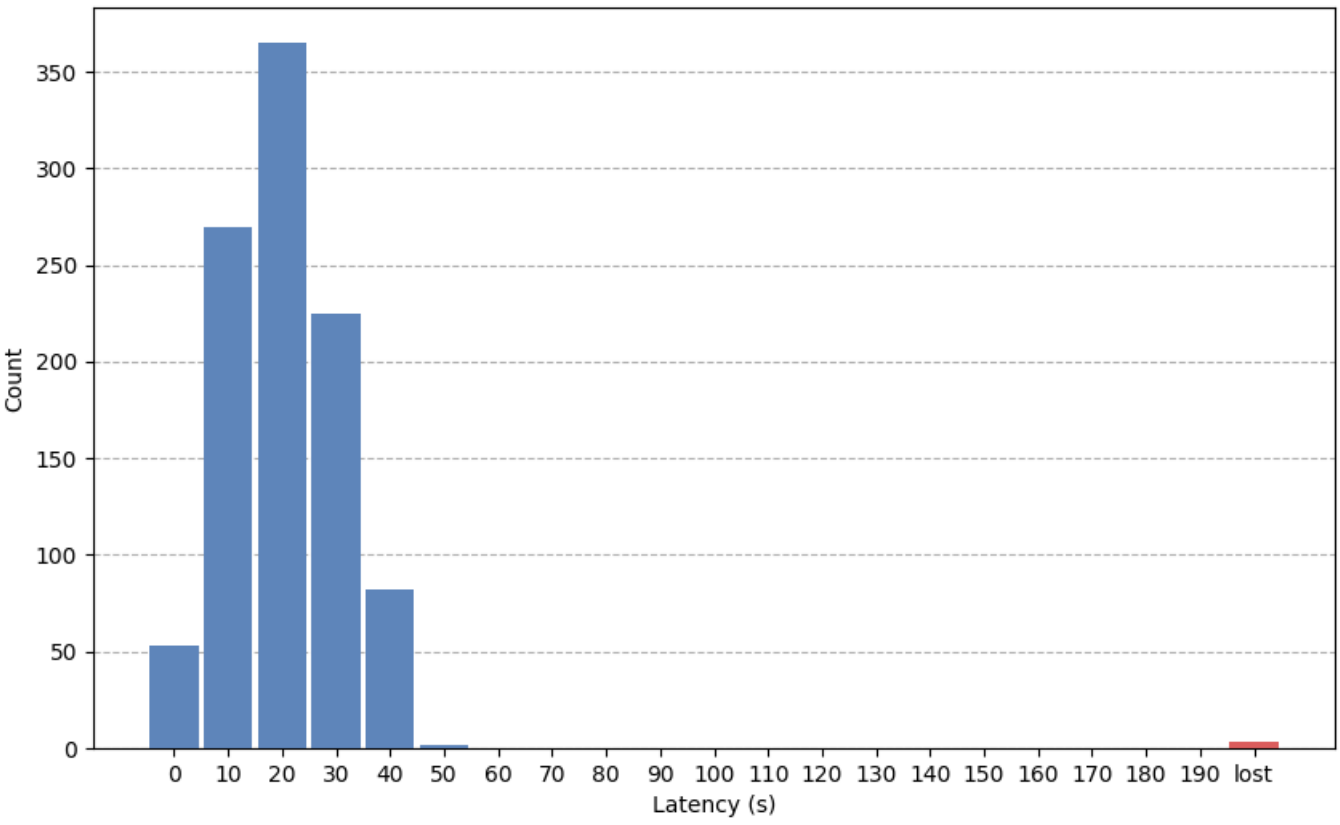
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 32 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 998
Responses Received: 997
Requests Lost: 2
Responses Lost: 1
Average Latency: 20 s

Response Time Histogram



Test Report - Large Packet Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.103
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

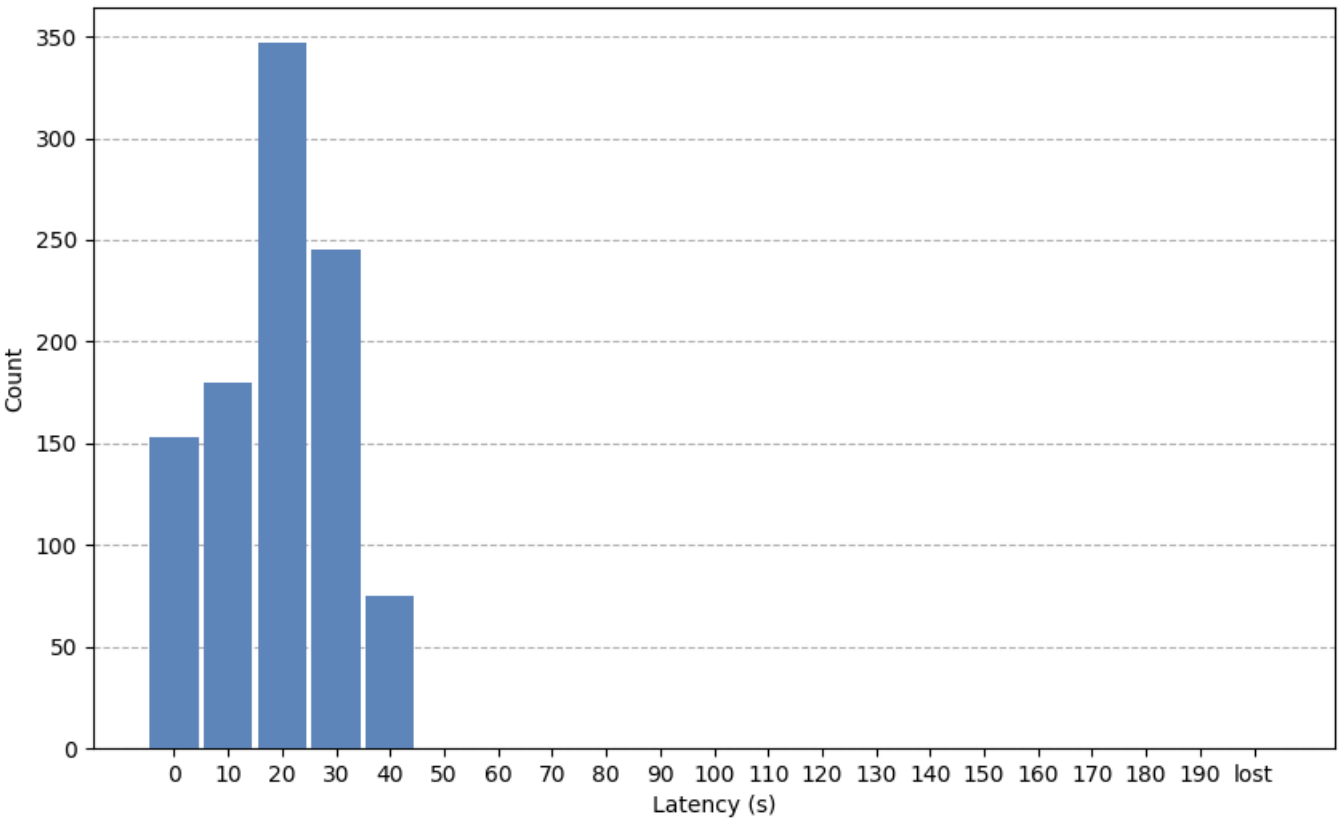
Test Setup

Iterations: 1000
Request Payload Size: 1000 bytes
Response Payload Size: 1000 bytes
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 57 ms

Test Results

Requests Sent: 1000
Requests Received on Server: 1000
Responses Received: 1000
Requests Lost: 0
Responses Lost: 0
Average Latency: 19 s

Response Time Histogram



Power measurements

The table below shows the average current consumption of the nRF7002 Wi-Fi module for each test.

	5 Seconds Interval	**10 Seconds Interval**	**Unit**
PS Mode			
PS Mode (DTIM/Legacy)	5380	4420	µA
TWT			
TWT - 8 ms SP	556	300	µA
TWT - 16 ms SP	656	348	µA
TWT - 32 ms SP	808	422	µA
TWT - 57 ms SP	1043	539	µA

Table F.1: Power measurements for the large packet test case

The power measurements were performed using the Power Profiler Kit II, following the procedure described in [49].

Note that the displayed SP durations correspond to the negotiated durations, not the requested duration, which explains the 57 ms instead of 64 ms.

G | Multi Packet Tests Results

The following pages present the results of the multi packet tests, including:

- Test report for the PS mode test
- Test reports for the TWT tests with TWT intervals ranging from 5 to 10 seconds and TWT session durations between 8 and 64 ms.
- Power measurements

Test Report - Multi Packet Use Case - PS

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

Test Setup

Iterations: 50
Number of packet per iteration: 10
PS Mode: Legacy
PS Wake Up Mode: DTIM

Test Results

Requests Sent: 500
Responses Received: 500
Average Latency: 65 ms

Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

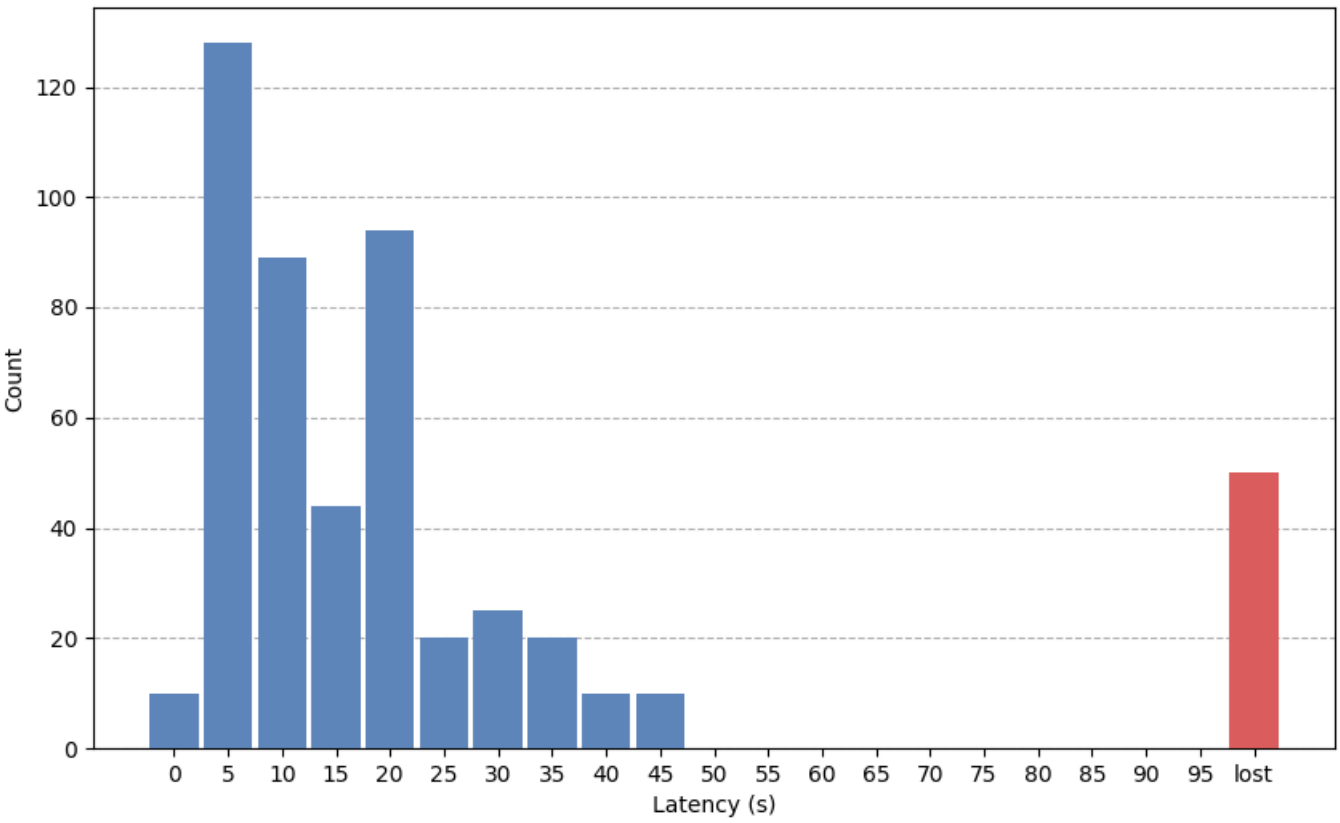
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 8 ms

Test Results

Requests Sent: 500
Responses Received: 450
Average Latency: 15 s

Response Time Histogram



Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

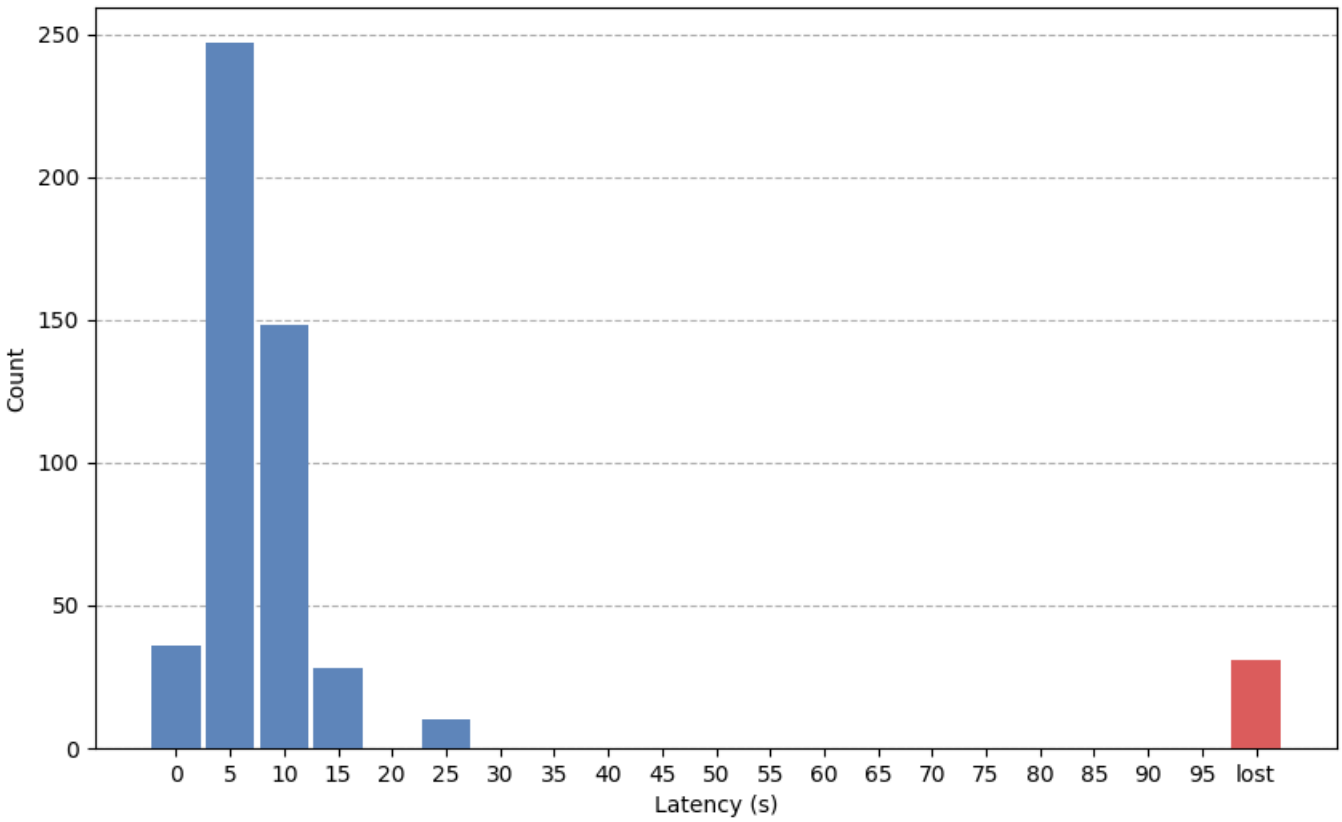
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 16 ms

Test Results

Requests Sent: 500
Responses Received: 469
Average Latency: 7 s

Response Time Histogram



Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

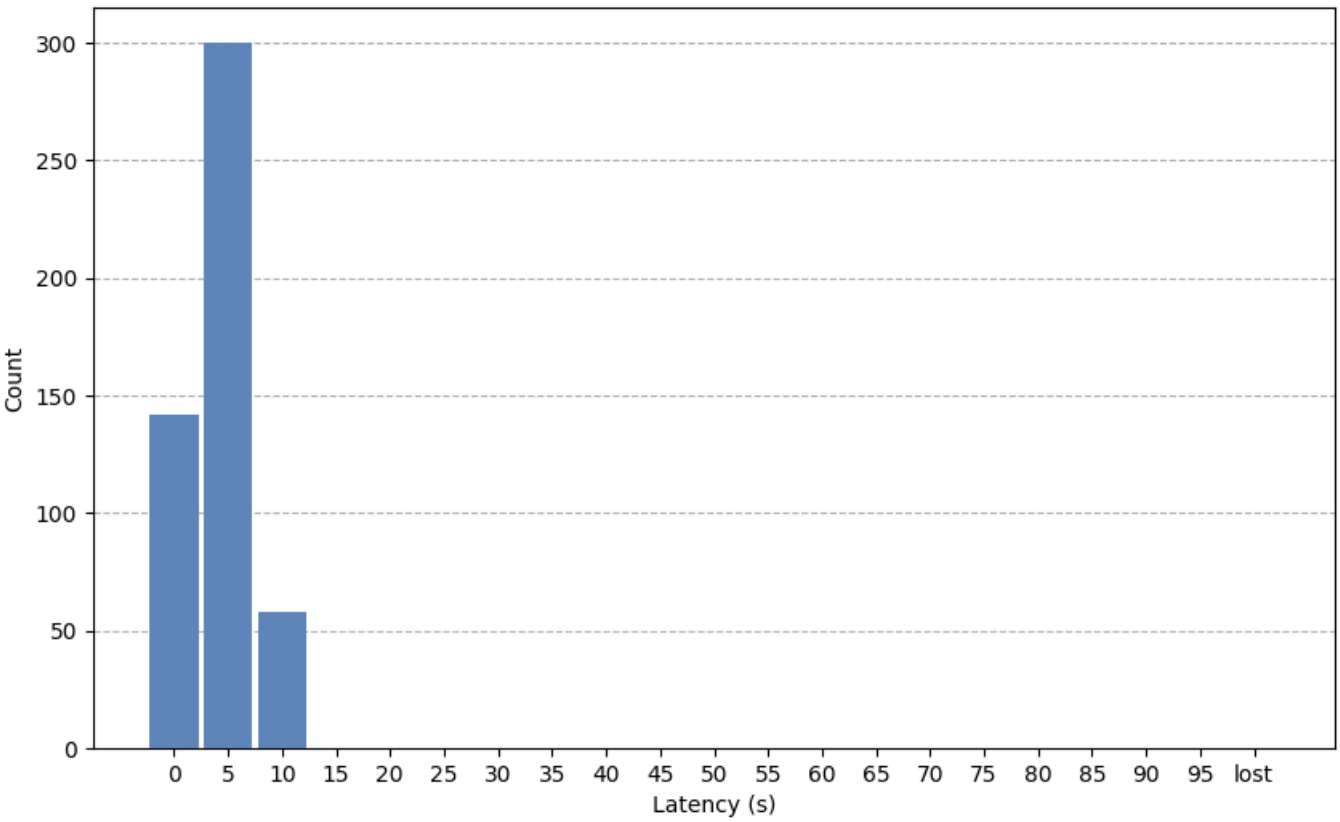
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 32 ms

Test Results

Requests Sent: 500
Responses Received: 500
Average Latency: 4 s

Response Time Histogram



Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

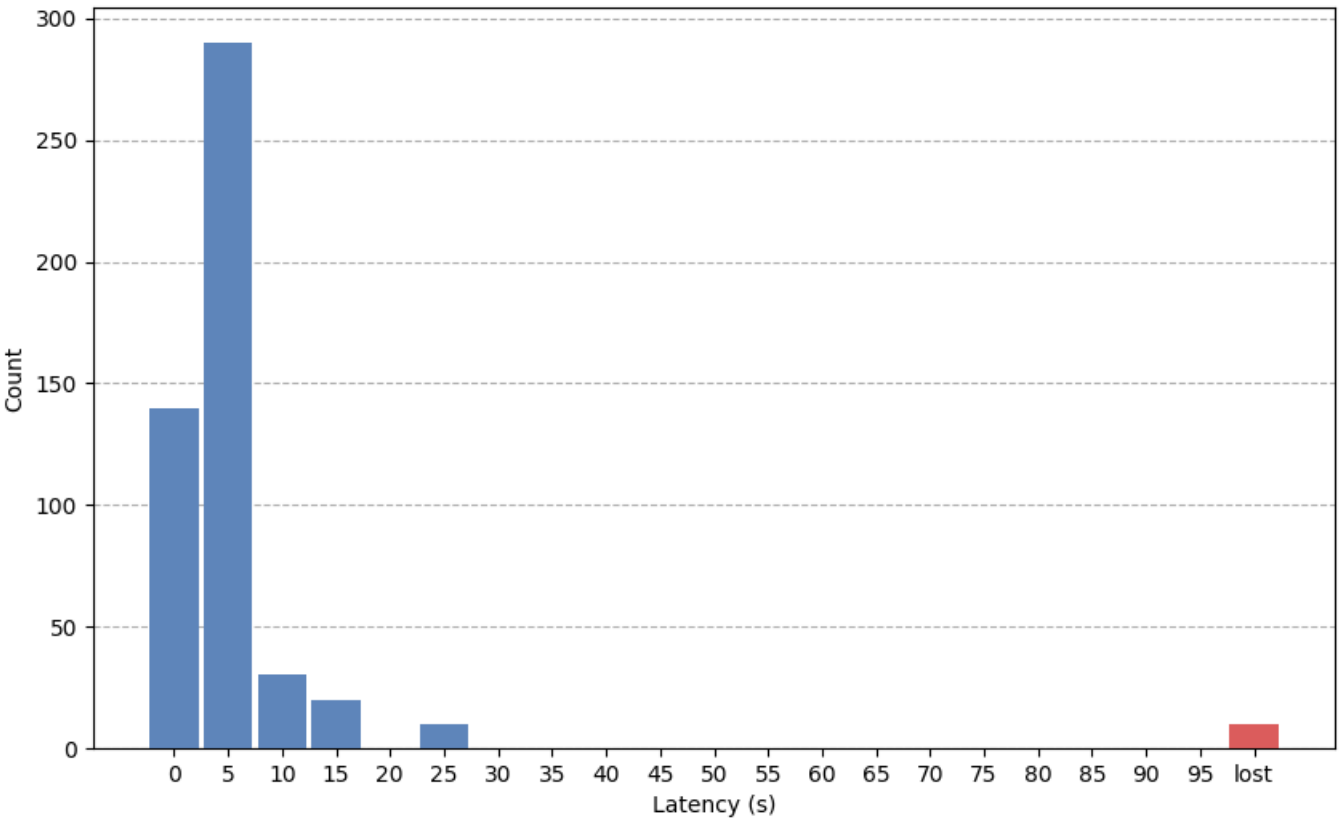
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 5 s
Negotiated TWT Wake Interval: 57 ms

Test Results

Requests Sent: 500
Responses Received: 490
Average Latency: 4 s

Response Time Histogram



Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

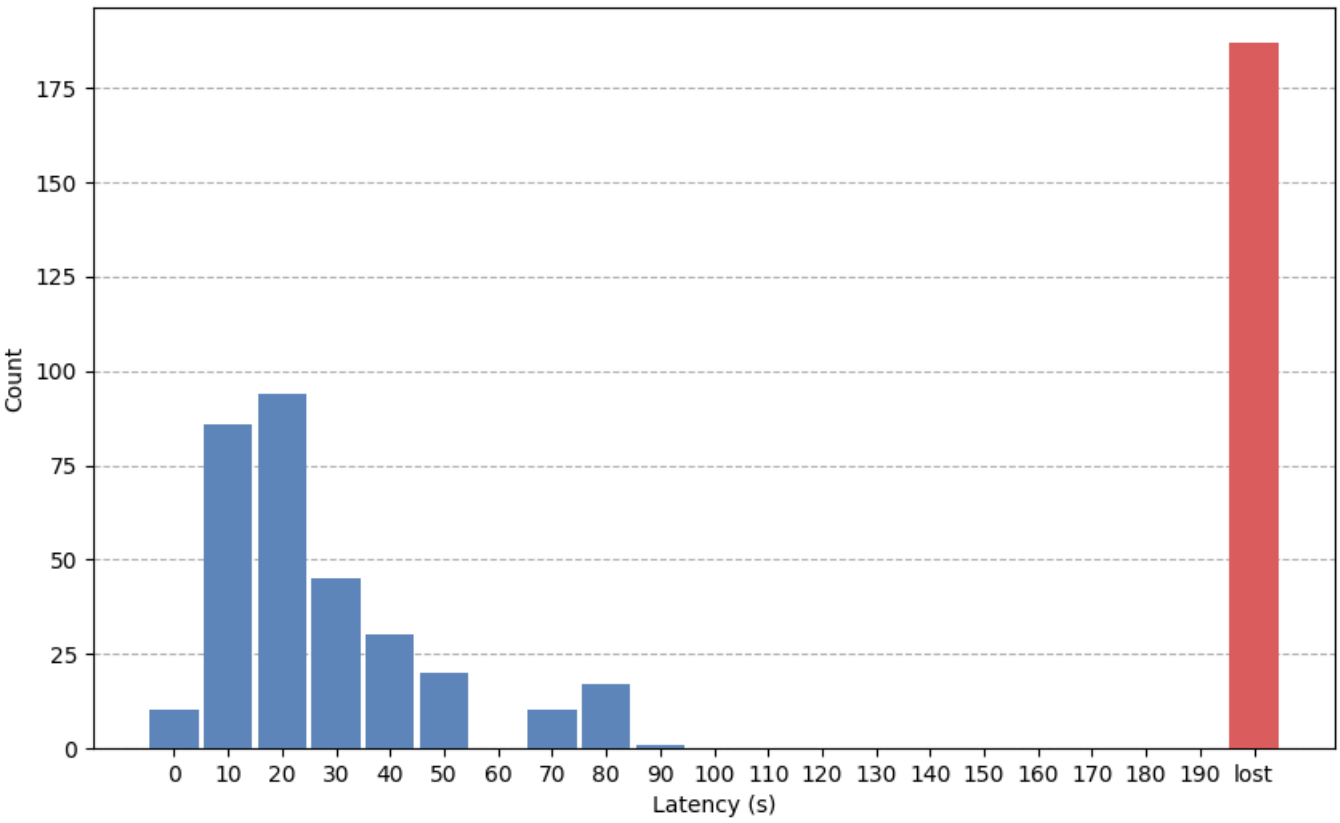
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 8 ms

Test Results

Requests Sent: 500
Responses Received: 313
Average Latency: 26 s

Response Time Histogram



Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

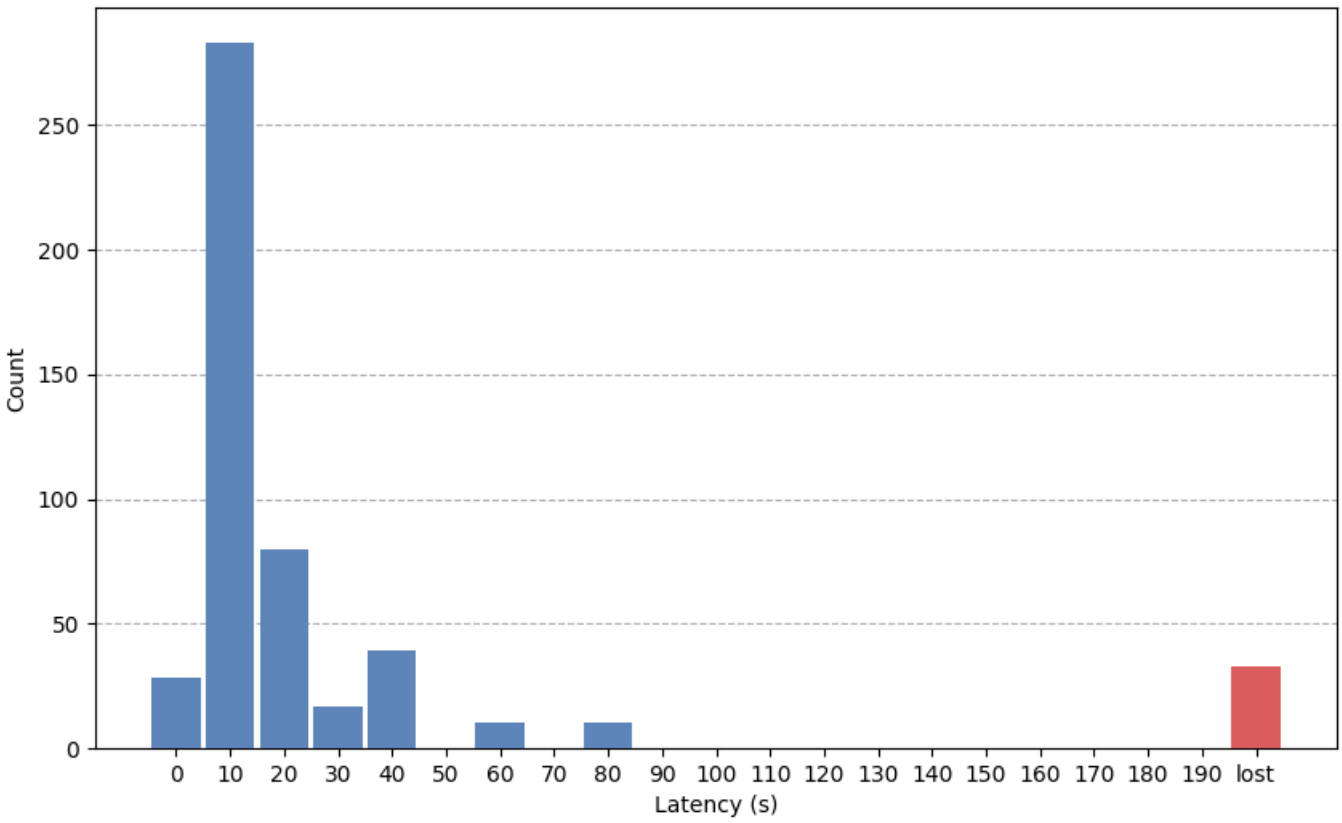
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 16 ms

Test Results

Requests Sent: 500
Responses Received: 467
Average Latency: 16 s

Response Time Histogram



Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

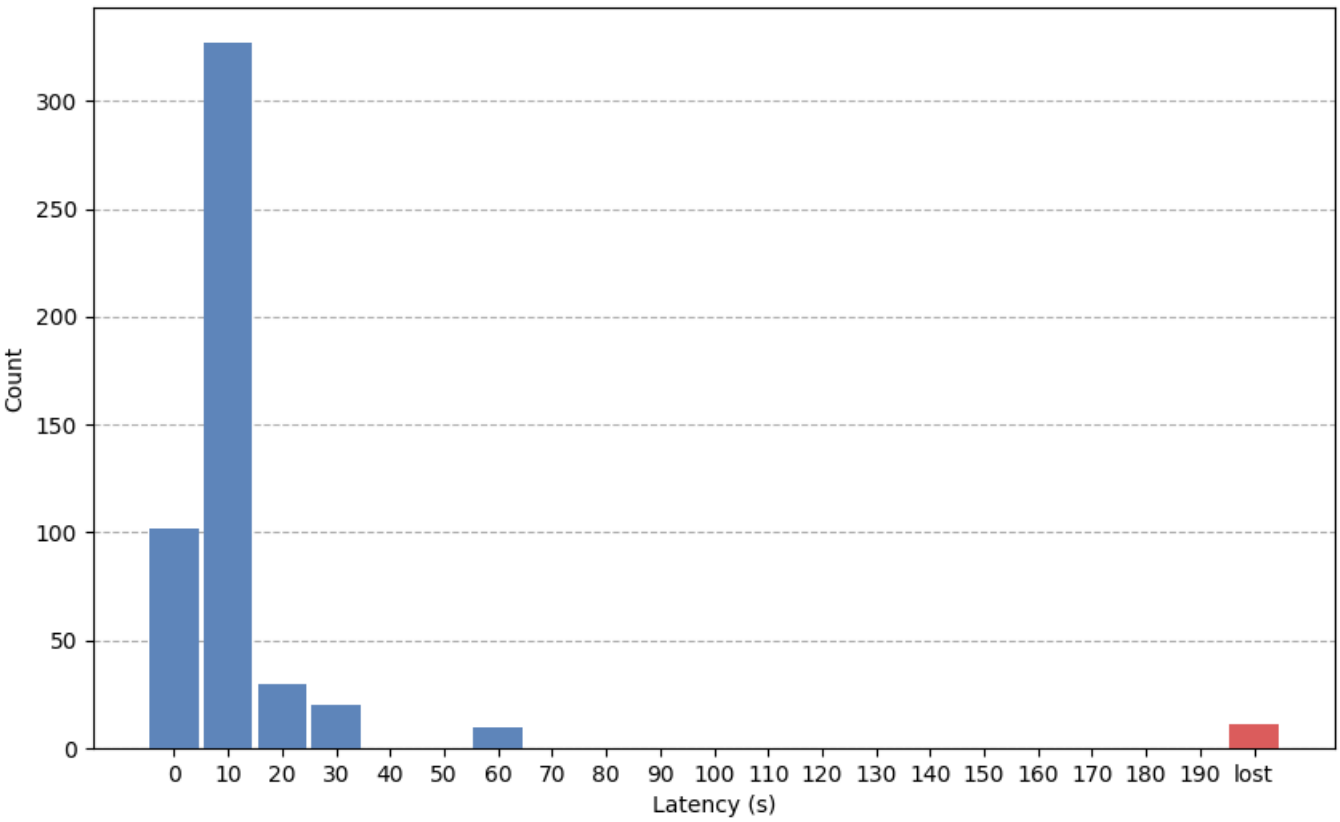
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 32 ms

Test Results

Requests Sent: 500
Responses Received: 489
Average Latency: 10 s

Response Time Histogram



Test Report - Multi Packet Use Case - TWT

Testbed Setup

CoAP Server: californium.eclipseprojects.io
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

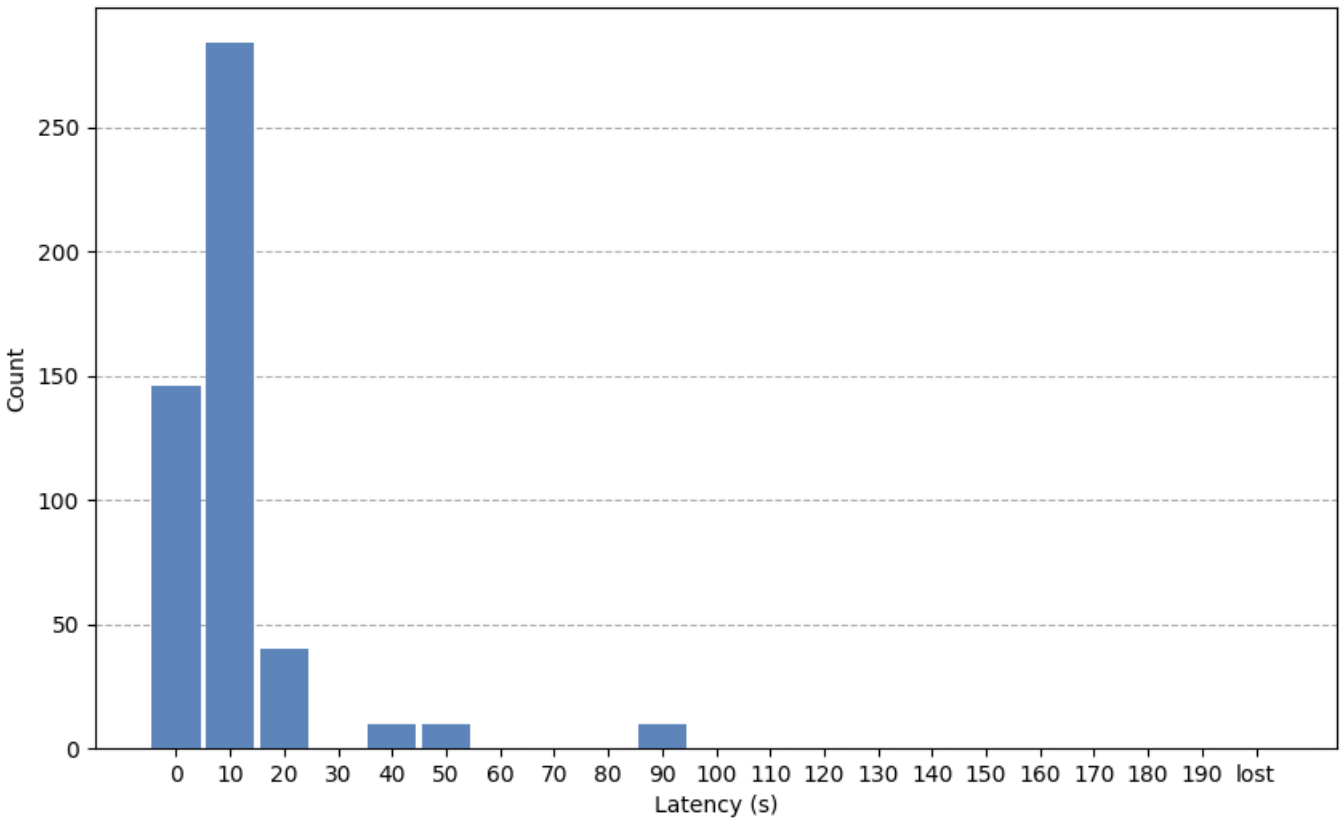
Test Setup

Iterations: 50
Number of packet per Iteration: 10
Negotiated TWT Interval: 10 s
Negotiated TWT Wake Interval: 57 ms

Test Results

Requests Sent: 500
Responses Received: 500
Average Latency: 10 s

Response Time Histogram



Power measurements

The table below shows the average current consumption of the nRF7002 Wi-Fi module for each test.

	5 Seconds Interval	**10 Seconds Interval**	**Unit**
TWT - 8 ms SP	578	312	µA
TWT - 16 ms SP	659	354	µA
TWT - 32 ms SP	799	420	µA
TWT - 57 ms SP	1050	555	µA

Table G.1: Power measurements for the multi packet use case

The power measurements were performed using the Power Profiler Kit II, following the procedure described in [49].

Note that the displayed SP durations correspond to the negotiated durations, not the requested duration, which explains the 57 ms instead of 64 ms.

H | Actuator Tests Results

The following pages present the results of the actuator tests, including:

- Test report for the PS mode test
- Test reports for the TWT tests with TWT intervals of 5 s and TWT session durations between 8 and 64 ms.
- Test reports for the emergency uplink TWT tests with TWT intervals of 5 s and TWT session durations between 8 and 64 ms.
- Power measurements

Test Report - Actuator Use Case - PS

Testbed Setup

CoAP Server: 192.168.50.229
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

Test Setup

Test Time: 3600 s
PS Mode: Legacy
PS Wake Up Mode: DTIM
Notifications Echo: Enabled

Test Results

Notifications sent by Server: 119
Notifications received on Client: 119
Echo received on Server: 118
Average Latency: 92 ms

Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

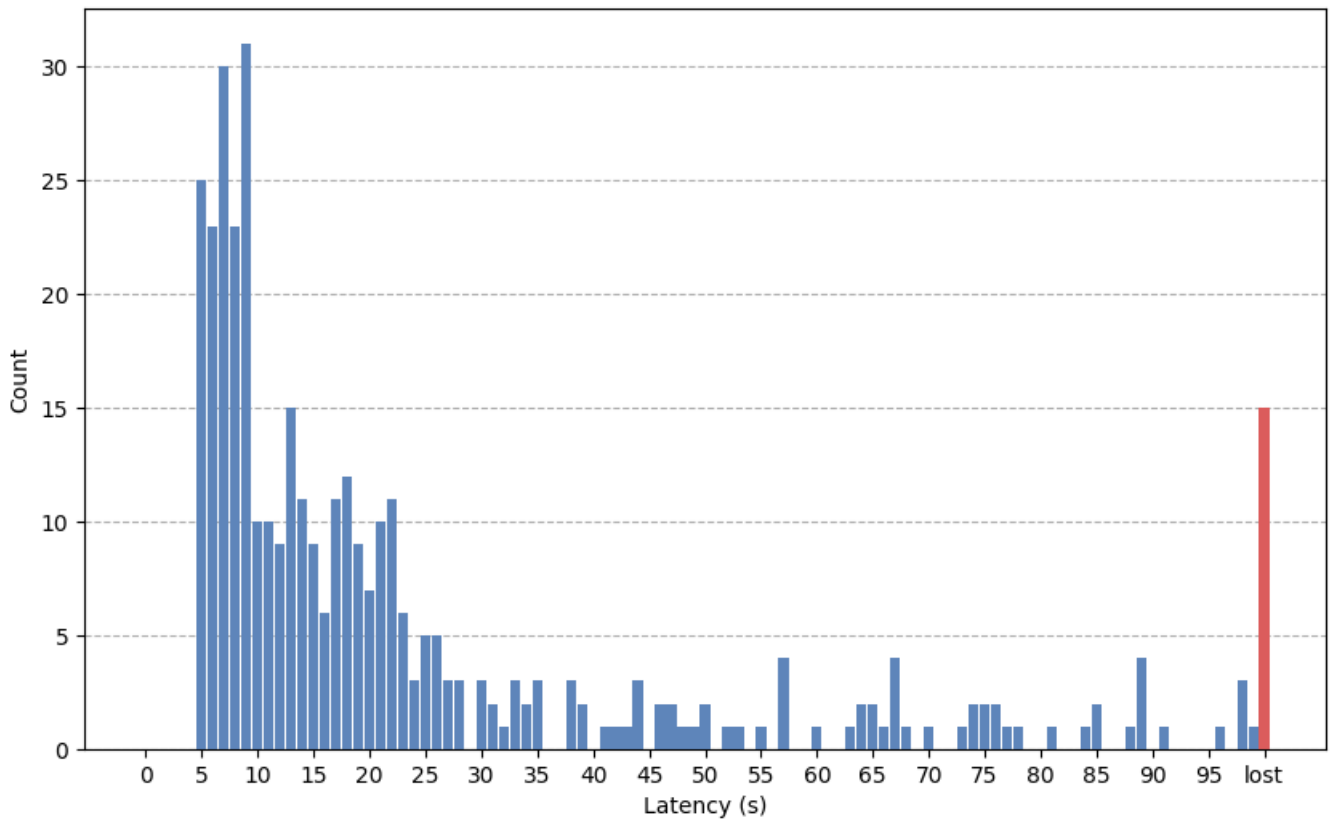
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 8 ms
Notifications Echo: Enabled
Emergency Uplink: Disabled

Test Results

Notifications sent by Server: 477
Notifications received on Client: 462
Echo received on Server: 461
Average Latency: 53047 ms

Response Time Histogram



Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

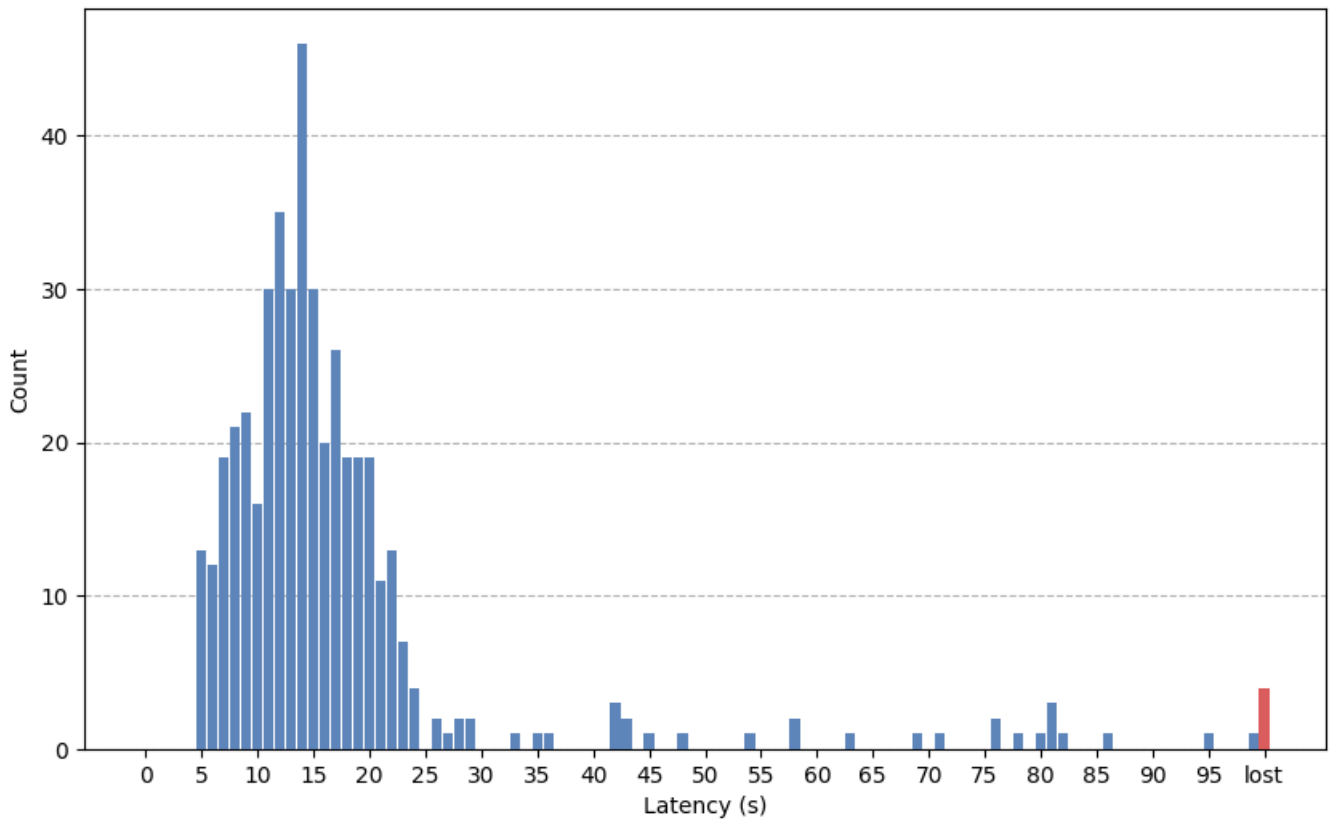
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 16 ms
Notifications Echo: Enabled
Emergency Uplink: Disabled

Test Results

Notifications sent by Server: 480
Notifications received on Client: 476
Echo received on Server: 474
Average Latency: 25006 ms

Response Time Histogram



Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

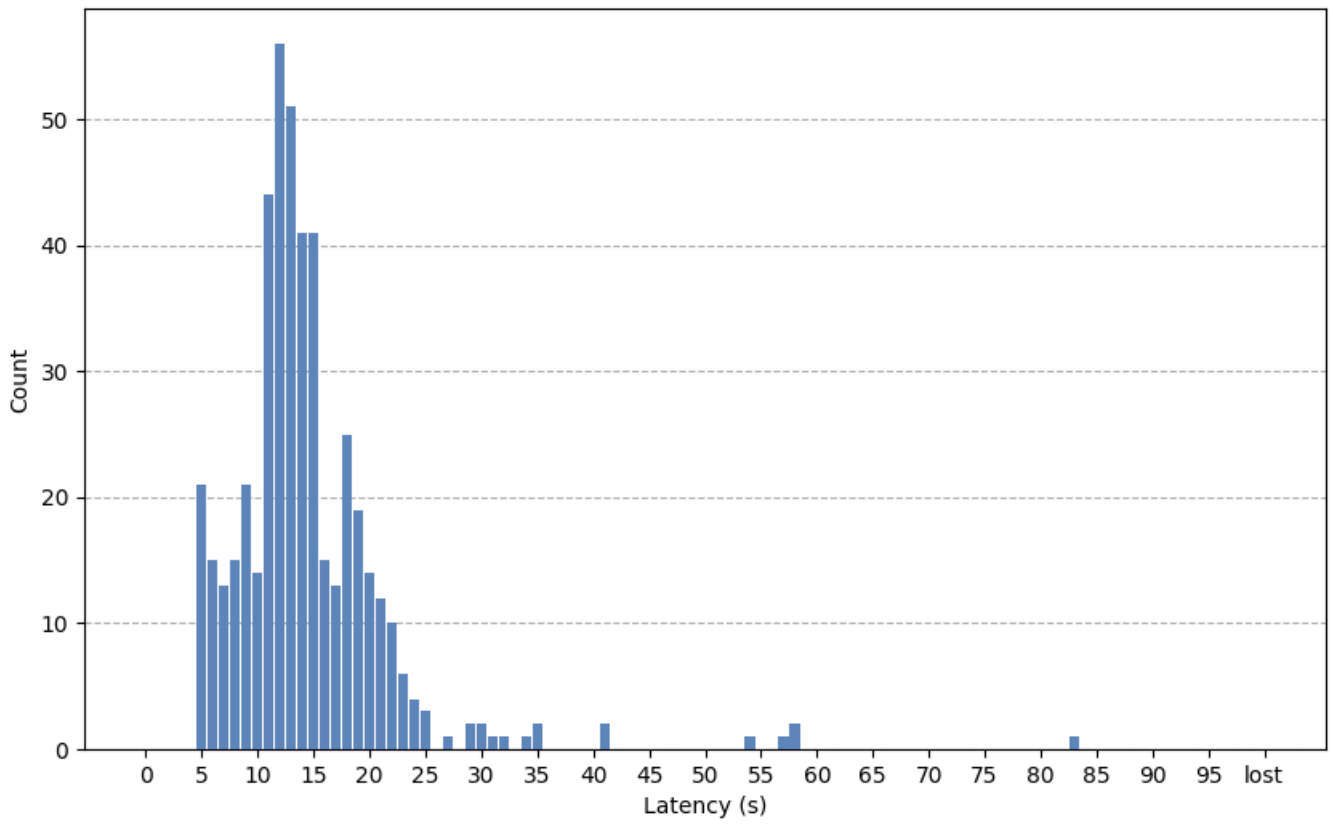
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 32 ms
Notifications Echo: Enabled
Emergency Uplink: Disabled

Test Results

Notifications sent by Server: 473
Notifications received on Client: 473
Echo received on Server: 471
Average Latency: 15115 ms

Response Time Histogram



Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

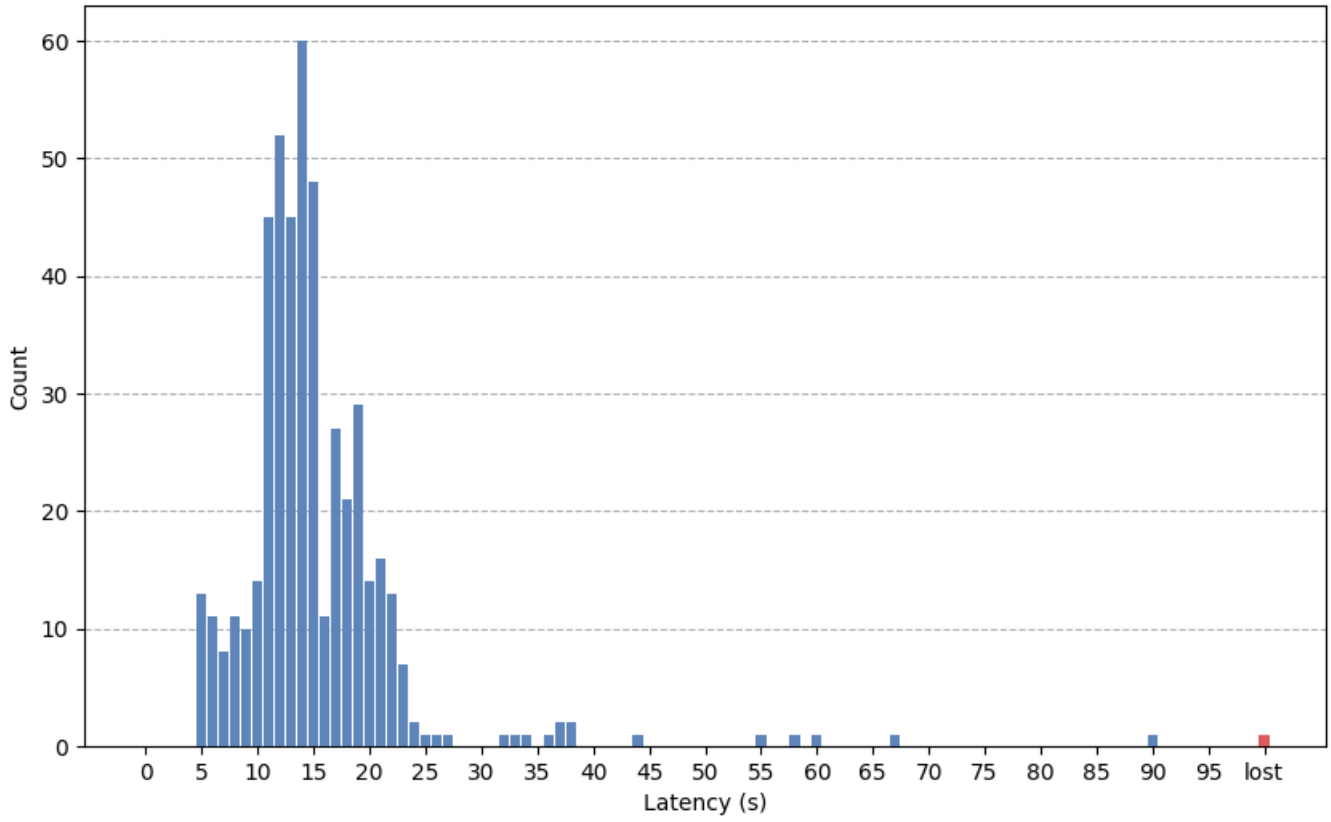
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 57 ms
Notifications Echo: Enabled
Emergency Uplink: Disabled

Test Results

Notifications sent by Server: 476
Notifications received on Client: 475
Echo received on Server: 474
Average Latency: 15653 ms

Response Time Histogram



Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

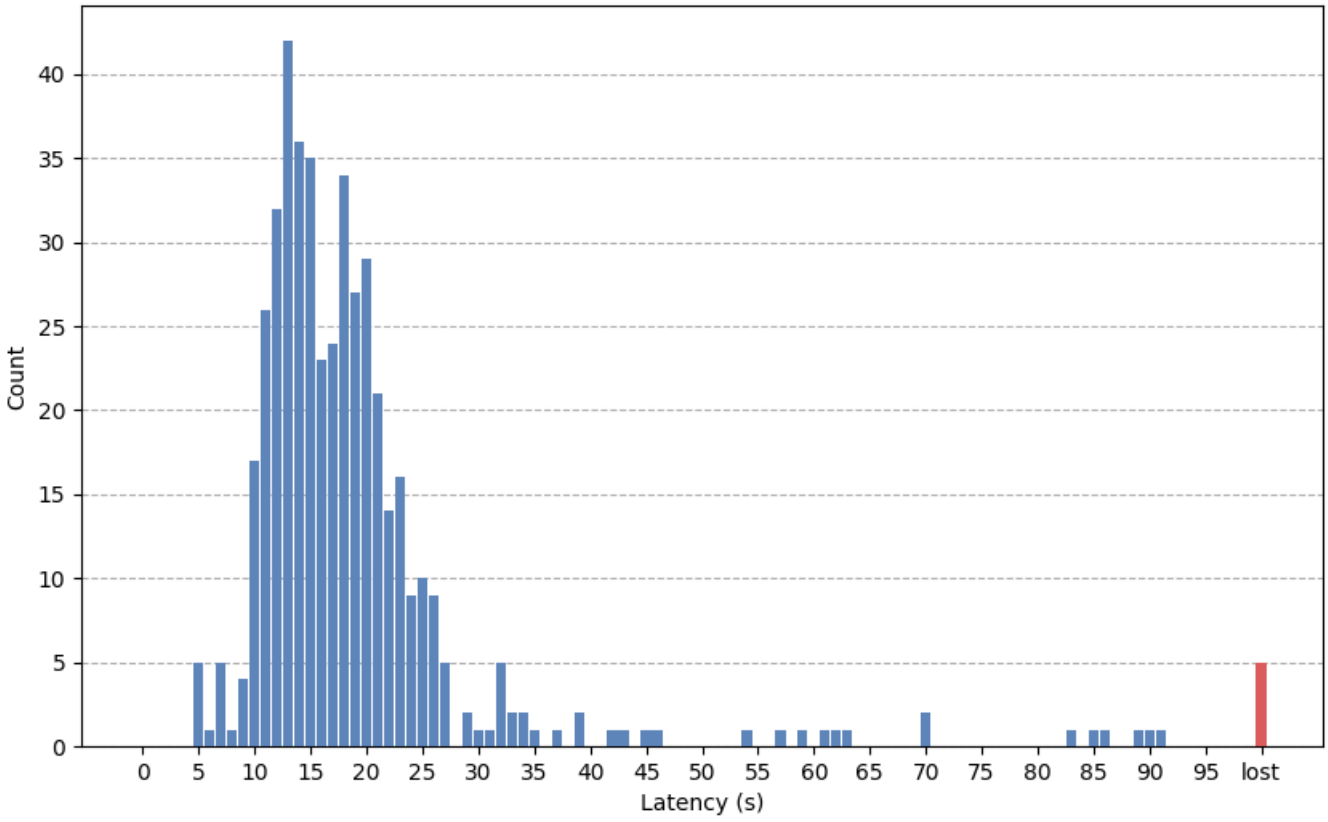
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 8 ms
Notifications Echo: Enabled
Emergency Uplink: Enabled

Test Results

Notifications sent by Server: 485
Notifications received on Client: 480
Echo received on Server: 479
Average Latency: 24633 ms

Response Time Histogram



Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

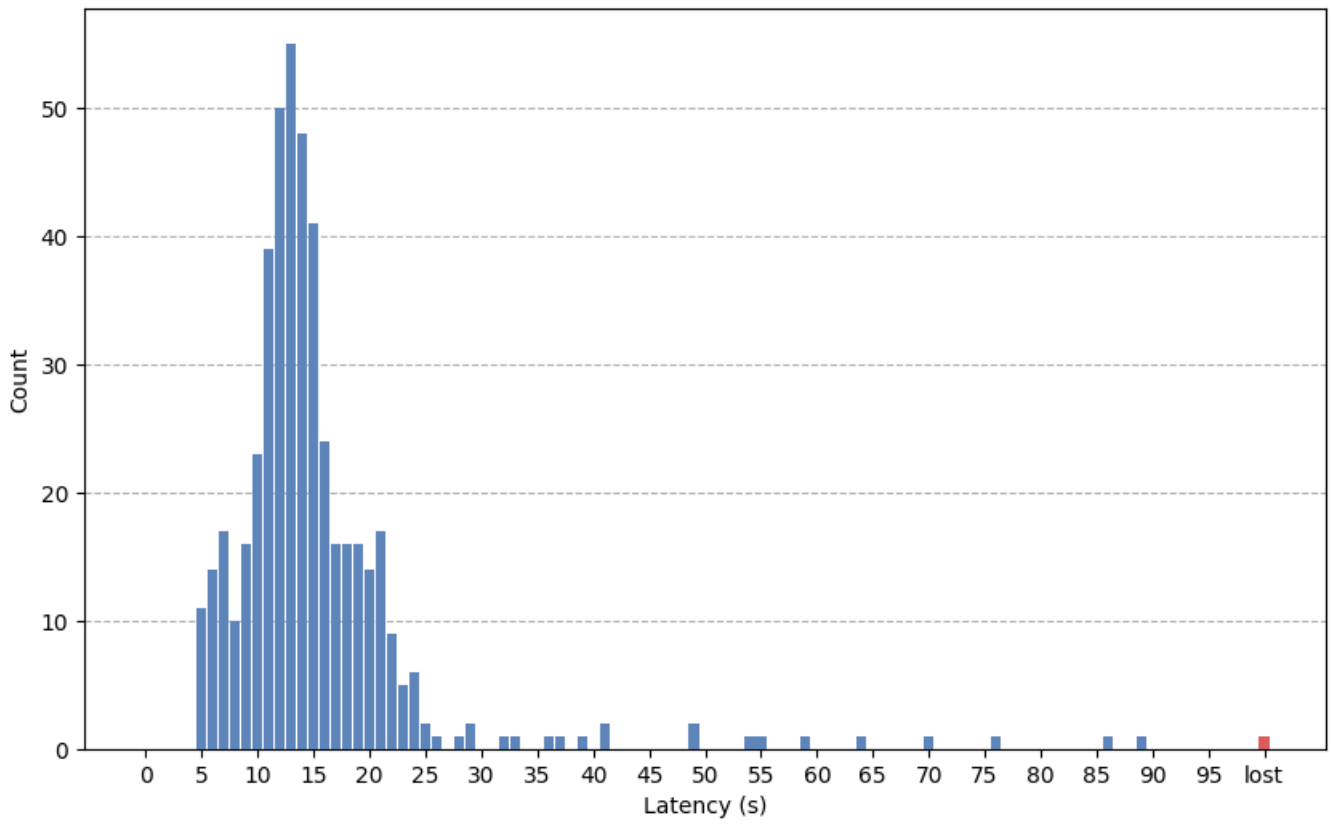
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 16 ms
Notifications Echo: Enabled
Emergency Uplink: Enabled

Test Results

Notifications sent by Server: 475
Notifications received on Client: 474
Echo received on Server: 471
Average Latency: 15930 ms

Response Time Histogram



Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

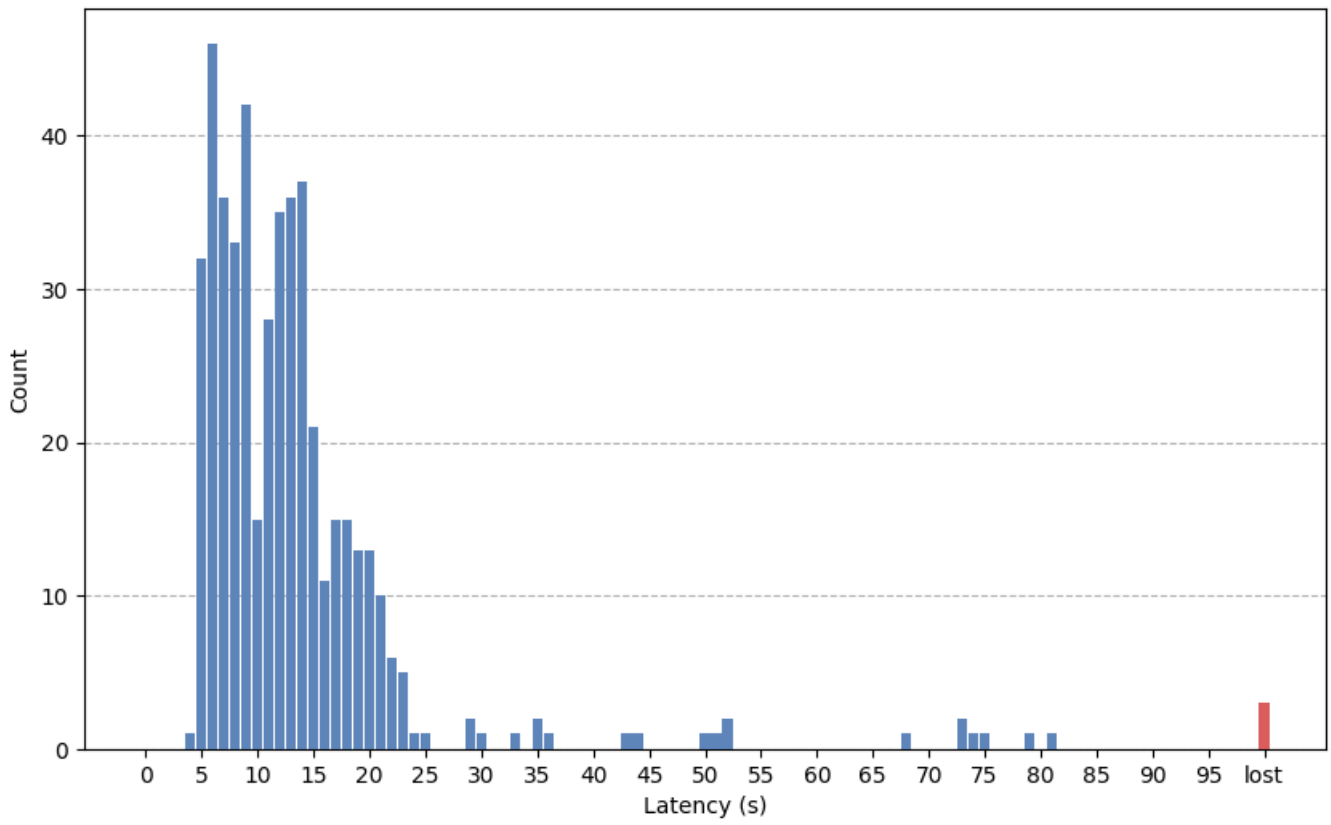
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 32 ms
Notifications Echo: Enabled
Emergency Uplink: Enabled

Test Results

Notifications sent by Server: 476
Notifications received on Client: 473
Echo received on Server: 473
Average Latency: 13993 ms

Response Time Histogram



Test Report - Actuator Use Case - TWT

Testbed Setup

CoAP Server: 192.168.1.228
DTLS: Enabled
DTLS Peer Verification: Enabled
DTLS Connection ID: Enabled
DTLS Ciphersuite: TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
IP Protocol: IPv4
Wi-Fi TWT Implicit: True
Wi-Fi TWT Announced: True
Wi-Fi TWT Trigger: False
Wi-Fi PS Listen Interval: 10

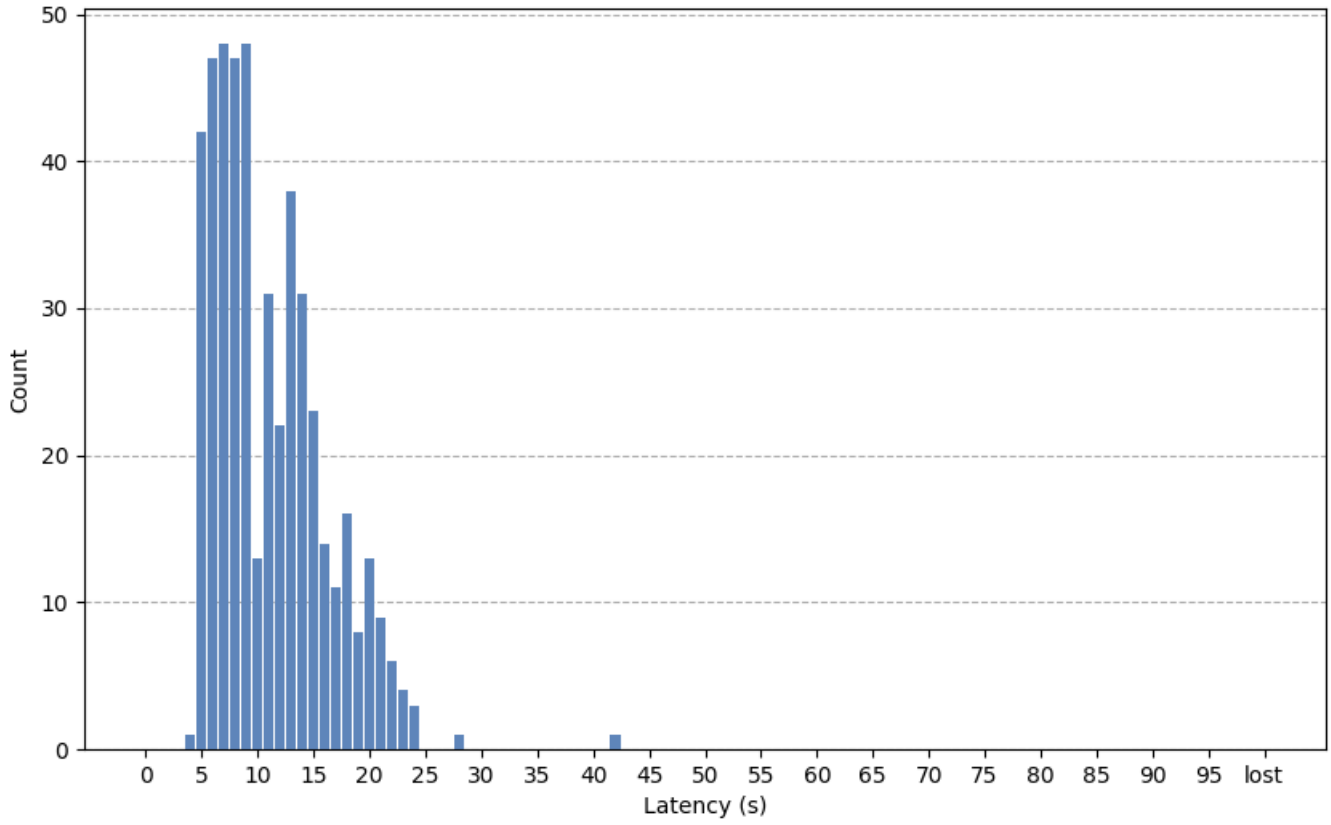
Test Setup

Test Time: 14400 s
TWT Interval: 5 s
TWT Wake Interval: 57 ms
Notifications Echo: Enabled
Emergency Uplink: Enabled

Test Results

Notifications sent by Server: 479
Notifications received on Client: 479
Echo received on Server: 477
Average Latency: 11638 ms

Response Time Histogram



Power measurements

The table below shows the average current consumption of the nRF7002 Wi-Fi module for each test.

	5 Seconds Interval Normal Mode	**5 Seconds Interval Emergency Uplink**	**Unit**
PS Mode			
PS Mode (DTIM/Legacy)	5160	-	µA
TWT			
TWT - 8 ms SP	565	686	µA
TWT - 16 ms SP	645	718	µA
TWT - 32 ms SP	814	879	µA
TWT - 57 ms SP	1060	1131	µA

Table H.1: Power measurements for the actuator use case

The power measurements were performed using the Power Profiler Kit II, following the procedure described in [49].

Note that the displayed SP durations correspond to the negotiated durations, not the requested duration, which explains the 57 ms instead of 64 ms.

References

[1] Ekaterina Stepanova et al. "On the Joint Usage of Target Wake Time and 802.11ba Wake-Up Radio". In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 221061–221076. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3043535. URL: https://ieeexplore.ieee.org/document/9288659 (visited on 01/20/2025).

[2] Shyam Krishnan Venkateswaran et al. "IEEE 802.11ax Target Wake Time: Design and Performance Analysis in ns-3". In: *2024 Workshop on ns-3*. WNS3 2024: 2024 Workshop on ns-3. Barcelona Spain: ACM, June 5, 2024, pp. 10–18. ISBN: 9798400717635. DOI: 10.1145/3659111.3659115. URL: https://dl.acm.org/doi/10.1145/3659111.3659115 (visited on 01/17/2025).

[3] Dmitry Bankov et al. "Clock Drift Impact on Target Wake Time in IEEE 802.11ax/ah Networks". In: *2018 Engineering and Telecommunication (EnT-MIPT)*. 2018 Engineering and Telecommunication (EnT-MIPT). Nov. 2018, pp. 30–34. DOI: 10.1109/EnT-MIPT.2018.00014. URL: https://ieeexplore.ieee.org/document/8757473 (visited on 01/20/2025).

[4] Govind Rajendran et al. "Performance Evaluation of Video Streaming Applications with Target Wake Time in Wi-Fi 6". In: *2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. Jan. 3, 2023, pp. 802–807. DOI: 10.1109/COMSNETS56262.2023.10041325. arXiv: 2310.02590[cs]. URL: http://arxiv.org/abs/2310.02590 (visited on 01/20/2025).

[5] Paul Simoneau. "The OSI Model: Understanding the Seven Layers of Computer Networks". In: ().

[6] *OSI model | Mevspace Docs*. URL: https://docs.mevspace.com/en/articles/articles-content/osi-model (visited on 11/05/2024).

[7] Steven Prugar. *The OSI Model: A Beginner's Guide for Techies*. NetBeez. May 5, 2021. URL: https://netbeez.net/blog/osi-model-beginners-guide-for-techies/ (visited on 11/05/2024).

[8] *Encapsulation (networking)*. In: *Wikipedia*. Page Version ID: 1237876021. July 31, 2024. URL: https://en.wikipedia.org/w/index.php?title=Encapsulation_(networking)&oldid=1237876021 (visited on 11/05/2024).

[9] *Wi-Fi Fundamentals*. Nordic Developer Academy. URL: https://academy.nordicsemi.com/courses/wi-fi-fundamentals/ (visited on 09/25/2024).

[10] *Understanding Wi-Fi 4/5/6/6E/7 (802.11 n/ac/ax/be)*. URL: https://www.wiisfi.com/ (visited on 09/25/2024).

[11] Matthew S. Gast. *802.11 Wireless Networks: The Definitive Guide: The Definitive Guide*. Google-Books-ID: IX3WatnVUe4C. "O'Reilly Media, Inc.", Apr. 25, 2005. 654 pp. ISBN: 978-1-4493-1952-6.

[12] *802.11 Frame Types and Formats*. How I WI-FI. July 13, 2020. URL: https://howiwifi.com/2020/07/13/802-11-frame-types-and-formats/ (visited on 10/25/2024).

[13] *[802.11] Wi-Fi Connection/Disconnection process*. NXP Community. Section: Wireless Connectivity Knowledge Base. Aug. 25, 2020. URL: https://community.nxp.com/t5/Wireless-Connectivity-Knowledge/802-11-Wi-Fi-Connection-Disconnection-process/ta-p/1121148 (visited on 10/24/2024).

[14] Matthew Li. *What You Should Know About Beacon Frame Format*. MOKOBlue: Original Bluetooth/BLE IoT & Smart Devices Manufacturer. Mar. 12, 2021. URL: https://www.mokoblue.com/what-you-should-know-about-beacon-frame-format/ (visited on 10/24/2024).

[15] *802.11 Frame Exchanges*. How I WI-FI. July 16, 2020. URL: https://howiwifi.com/2020/07/16/802-11-frame-exchanges/ (visited on 10/25/2024).

[16] *KRACK Attacks: Breaking WPA2*. URL: https://www.krackattacks.com/ (visited on 10/25/2024).

[17] *Security | Wi-Fi Alliance*. URL: https://www.wi-fi.org/discover-wi-fi/security (visited on 10/24/2024).

[18] *WPA3 Dragonfly Handshake -*. URL: https://sarwiki.informatik.hu-berlin.de/WPA3_Dragonfly_Handshake#What_is_new_in_WPA3 (visited on 10/25/2024).

[19] *Wi-Fi 6*. In: *Wikipedia*. Page Version ID: 1247702908. Sept. 25, 2024. URL: https://en.wikipedia.org/w/index.php?title=Wi-Fi_6&oldid=1247702908 (visited on 10/14/2024).

[20] *Multi-user MIMO*. In: *Wikipedia*. Page Version ID: 1237716950. July 31, 2024. URL: https://en.wikipedia.org/w/index.php?title=Multi-user_MIMO&oldid=1237716950 (visited on 10/14/2024).

[21] *Beamforming*. In: *Wikipedia*. Page Version ID: 1235788941. July 21, 2024. URL: https://en.wikipedia.org/w/index.php?title=Beamforming&oldid=1235788941 (visited on 10/14/2024).

[22] *Learn About BSS Color in 802.11ax: Background, Definition, Set-up*. Extreme Networks - United Kingdom. July 17, 2020. URL: https://uk.extremenetworks.com/extreme-networks-blog/what-is-bss-color-in-802-11ax/ (visited on 10/22/2024).

[23] *Operating in power save modes*. URL: https://docs.nordicsemi.com/bundle/ncs-latest/page/nrf/protocols/wifi/station_mode/powersave.html (visited on 09/24/2024).

[24] *Power Save Methods*. How I WI-FI. June 25, 2020. URL: https://howiwifi.com/2020/06/25/power-save-methods/ (visited on 11/07/2024).

[25] Kevin R. Fall and W. Richard Stevens. *TCP/IP Illustrated: The Protocols, Volume 1*. Google-Books-ID: a23OAn5i8R0C. Addison-Wesley, Nov. 8, 2011. 1060 pp. ISBN: 978-0-13-280818-7.

[26] Peter G. Johansson. *IPv4 over IEEE 1394*. RFC 2734. Dec. 1999. DOI: 10.17487/RFC2734. URL: https://www.rfc-editor.org/info/rfc2734.

[27] Ralph Droms. *Dynamic Host Configuration Protocol*. Request for Comments RFC 2131. Num Pages: 45. Internet Engineering Task Force, Mar. 1997. DOI: 10.17487/RFC2131. URL: https://datatracker.ietf.org/doc/rfc2131 (visited on 11/10/2024).

[28] Bob Hinden and Dr. Steve E. Deering. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Dec. 1998. DOI: 10.17487/RFC2460. URL: https://www.rfc-editor.org/info/rfc2460.

[29] *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826. Nov. 1982. DOI: 10.17487/RFC0826. URL: https://www.rfc-editor.org/info/rfc826.

[30] William A. Simpson et al. *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. Sept. 2007. DOI: 10.17487/RFC4861. URL: https://www.rfc-editor.org/info/rfc4861.

[31] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[32] *Extend your battery life with nRF Cloud's CoAP interface - Blogs - Nordic Blog - Nordic DevZone*. Sept. 28, 2023. URL: https://devzone.nordicsemi.com/nordic/nordic-blog/b/blog/posts/extend-your-battery-life-with-nrf-cloud-coap-interface (visited on 12/12/2024).

[33] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347. URL: https://www.rfc-editor.org/info/rfc6347.

[34] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[35] Henrik Nielsen, Roy T. Fielding, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. May 1996. DOI: 10.17487/RFC1945. URL: https://www.rfc-editor.org/info/rfc1945.

[36] *Cellular IoT Fundamentals*. Nordic Developer Academy. URL: https://academy.nordicsemi.com/courses/cellular-iot-fundamentals/ (visited on 12/12/2024).

[37] Cigdem Sengul and Anthony Kirby. *Message Queuing Telemetry Transport (MQTT) and Transport Layer Security (TLS) Profile of Authentication and Authorization for Constrained Environments (ACE) Framework*. RFC 9431. July 2023. DOI: 10.17487/RFC9431. URL: https://www.rfc-editor.org/info/rfc9431.

[38] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[39] *nRF7002 DK Hardware*. URL: https://docs.nordicsemi.com/bundle/ug_nrf7002_dk/page/UG/nrf7002_DK/intro.html (visited on 12/11/2024).

[40] *zephyrproject-rtos/zephyr*. original-date: 2016-05-26T17:54:19Z. Dec. 11, 2024. URL: https://github.com/zephyrproject-rtos/zephyr (visited on 12/11/2024).

[41] Eclipse Californium Eclipse project. *Eclipse Californium*. URL: https://eclipse.dev/californium/ (visited on 12/11/2024).

[42] *eclipse-californium/californium*. original-date: 2015-09-24T15:27:15Z. Jan. 6, 2025. URL: https://github.com/eclipse-californium/californium (visited on 01/07/2025).

[43] *Power Profiler Kit II*. URL: https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2 (visited on 01/17/2025).

[44] *Network Management — Zephyr Project Documentation*. URL: https://docs.zephyrproject.org/latest/connectivity/networking/api/net_mgmt.html (visited on 11/14/2024).

[45] *CoAP client — Zephyr Project Documentation*. URL: https://docs.zephyrproject.org/latest/connectivity/networking/api/coap_client.html (visited on 01/03/2025).

[46] *CoAP — Zephyr Project Documentation*. URL: https://docs.zephyrproject.org/latest/connectivity/networking/api/coap.html (visited on 01/03/2025).

[47] *BSD Sockets — Zephyr Project Documentation*. URL: https://docs.zephyrproject.org/latest/connectivity/networking/api/sockets.html (visited on 01/03/2025).

[48] *Kernel Timing — Zephyr Project Documentation*. URL: https://docs.zephyrproject.org/latest/kernel/services/timing/clocks.html (visited on 01/03/2025).

[49] *Power profiling of nRF7002 DK*. URL: https://docs.nordicsemi.com/bundle/ncs-2.6.0/page/nrf/device_guides/working_with_nrf/nrf70/developing/power_profiling.html (visited on 01/27/2025).

Glossary

ACK Acknowledgment. 13, 27, 41, 49, 51, 53, 76, 77, 125

AID Association ID. 20, 26, 27

AP Access Point. 2, 3, 12, 13, 15, 18–20, 23, 24, 26–28, 30–32, 37, 49, 50, 55, 61, 68, 71, 100–105, 108, 109, 112–115, 117, 119, 122, 124, 127, 131, 132, 139

API Application Programming Interface. 68, 71–73

ARP Address Resolution Protocol. 37, 100–105, 139

ASCII American Standard Code for Information Interchange. 10

BSS Basic Service Set. ix, 12, 22, 24

BSSID Basic Service Set Identifier. 12, 15, 18

CoAP Constrained Application Protocol. ix, 3, 7, 10, 43, 46–49, 51, 52, 58, 60, 63, 65, 75–78, 80, 84, 89, 101, 102, 104, 105, 107, 118, 135

CSMA/CA Carrier-Sense Multiple Access with Collision Avoidance. 12, 16, 24

CTS Clear to Send. 13, 16

DAD Duplicate Address Detection. 38

DHCP Dynamic Host Configuration Protocol. 35, 68, 100

DHCPv4 Dynamic Host Configuration Protocol version 4. 33

DHCPv6 Dynamic Host Configuration Protocol version 6. 35

DNS Domain Name System. 33, 60, 80

DS Differentiated Services. 34, 36

DTIM Delivery Traffic Indication Message. 2, 3, 5, 25–27, 29, 30, 63, 106, 108, 111, 112, 116, 118, 125

DTLS Datagram Transport Layer Security. ix, 7, 40, 46, 47, 60, 77, 81, 101, 104, 106, 118, 122, 125

ECDHE Elliptic Curve Diffie-Hellman Ephemeral. 40

EOSP End of Service Period. 28

GPIO General-Purpose Input/Output. 54, 74

HT High Throughput. 15

HTML HyperText Markup Language. 10, 42

HTTP Hypertext Transfer Protocol. 3, 10, 42, 43

IC Integrated Circuit. 46, 72

ICMPv6 Internet Control Message Protocol version 6. 37

IHL Internet Header Length. 34

IoT Internet of Things. x, 1–5, 7, 12, 21, 23, 29, 40, 42, 98

IP Internet Protocol. 7, 9, 10, 12, 13, 15, 32–35, 37–40, 42, 49, 60, 61, 68, 100–105, 139

IPv4 Internet Protocol version 4. 32–34, 36, 46, 47, 61, 77, 80, 100, 101, 105, 106, 118, 122, 125

IPv6 Internet Protocol version 6. 32, 36, 46, 47, 61, 77, 80, 100, 104, 105, 143

JSON JavaScript Object Notation. 10, 84, 87

LAN Local Area Network. 55

LLC Logical Link Control. 9

MAC Media Access Layer. 9, 12, 15, 27, 28, 32, 37, 49, 55, 101–104, 109, 139

MCU Microcontroller Unit. 72, 73

MQTT Message Queuing Telemetry Transport. ix, 3, 10, 42, 98

MU-MIMO Multi-User Multiple Input Multiple Output. 22, 23

NA Neighbor Advertisement. 37, 38

NAT Network Address Translation. 32, 33, 35

NDP Neighbor Discovery Protocol. 37, 38, 100, 104, 105, 143

NS Neighbor Solicitation. 37, 38, 105

OFDM Orthogonal Frequency-Division Multiplexing. 22

OFDMA Orthogonal Frequency Division Multiple Access. 22, 23

OSI Open Systems Interconnection. 1, 7, 9, 10, 32, 42

PHY Physical. 12, 24

PPK Power Profiler Kit. 54

PS Power Save. 2, 3, 5, 13, 27, 31, 45, 48, 50, 61–66, 82, 84, 88, 103, 106–108, 111, 112, 115, 116, 118, 119, 125, 126, 189

PSK Pre-Shared Key. 20, 40, 60

QAM Quadrature Amplitude Modulation. 22, 23

QoS Quality of Service. 15, 28, 34, 36, 109

QSPI Quad Serial Peripheral Interface. 46

RA Router Advertisement. 35, 37

RAM Random Access Memory. 46

RF Radio Frequency. 25, 98, 106

RPC Remote Procedure Call. 9

RS Router Solicitation. 37

RTOS Real-Time Operating System. 46

RTS Request to Send. 13, 16, 109

RU Resource Unit. 22, 23

SAE Simultaneous Authentication of Equals. 20, 23

SDK Software Development Kit. 46

SEND Secure Neighbor Discovery. 38

SLAAC Stateless Address Autoconfiguration. 35, 37, 38, 104

SoC System on Chip. 46

SP Service Period. 164, 175, 187, 199

SPI Serial Peripheral Interface. 46

SSID Service Set Identifier. 12, 18, 26, 54

SSL Secure Sockets Layer. 10

STA Station. 2, 4, 12, 13, 18–20, 23, 25–32, 37, 45, 49, 50, 55, 68, 71, 100–105, 112, 114, 115, 117, 139

TCP Transmission Control Protocol. 1, 3, 5, 9, 10, 32, 34, 36, 39, 41–43, 98, 99

TIM Traffic Indication Map. ix, 26, 27

TLS Transport Layer Security. 10, 40

TTL Time To Live. 34, 36

TWT Target Wake Time. ix, 1–5, 22, 23, 25, 30, 31, 45, 46, 48–51, 53, 61–66, 71–73, 82, 84, 86, 88, 98–101, 103–132, 149, 165, 177, 189

UDP User Datagram Protocol. ix, 1, 3, 7, 9, 32, 34, 36, 39–41, 43, 46, 47, 76, 77, 101

ULA Unique Local Address. 35

URI Universal Resource Identifier. 42, 43

WAN Wide Area Network. 55

WEP Wired Equivalent Privacy. 19

WLAN Wireless Local Area Network. 12, 55

WMM Wi-Fi Multimedia. ix, 26–28, 63

WPA Wi-Fi Protected Access. 19, 20, 54

WPA2 Wi-Fi Protected Access 2. 19, 20, 23

WPA3 Wi-Fi Protected Access 3. 19, 20, 22, 23

XML eXtensible Markup Language. 10