CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Optimizing Message Distribution for Bluetooth Mesh Gateways

Harnessing Dynamic Scenes and Caching: A Strategy for Network Optimization

Master's thesis in Computer science and engineering

Victor Lanner

Titus Blosse

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

MASTER'S THESIS 2024

Optimizing Message Distribution for Bluetooth Mesh Gateways

Harnessing Dynamic Scenes and Caching: A Strategy for Network Optimization

Victor Lanner

Titus Blosse

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Optimizing Message Distribution for Bluetooth Mesh Gateways
Harnessing Dynamic Scenes and Caching: A Strategy for Network Optimization
Victor Lanner, Titus Blosse

Supervisor: Ahmed Ali-Eldin, Computer Science and Engineering
Examiner: Ahmed Ali-Eldin, Computer Science and Engineering

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

Typeset in LaTeX
Gothenburg, Sweden 2024

Optimizing Message Distribution for Bluetooth Mesh Gateways
Harnessing Dynamic Scenes and Caching: A Strategy for Network Optimization
Victor Lanner, Titus Blosse
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis project revolves around optimizing and evaluating a message distribution service for Bluetooth mesh gateways. The gateway in this context is a specific node that features an Internet connection alongside its Bluetooth connectivity, which allows the Bluetooth mesh to communicate with the outside world over the Internet. The specific problem this project addresses concerns broadcast storms occurring on the Bluetooth mesh caused by an increased amount of messages sent to the gateway that are then forwarded to the mesh and how they can cause inefficiency in the gateway and the Bluetooth mesh. Using a combination of dynamic scenes and caching, it was shown that response time can be decreased to 1/6th of a naive, non-caching solution and halved in demanding scenarios.

Keywords: Computer, science, IoT, mesh, network, gateway, smart home, caching, Bluetooth

Acknowledgements

We want to thank our examiner and supervisor Ahmed Ali-Eldin Hassan, for continually supporting us throughout this project. We are also incredibly grateful to the company for giving us the opportunity to write our thesis at their company; this endeavor would not have been possible without them; we especially would like to thank the entire team we worked with for their invaluable support and hospitality during our stay.

Lastly, we want to thank our respective families for their invaluable support and encouragement during the work on this thesis.

Victor Lanner & Titus Blosse, Gothenburg, 2024-08-08

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
- 1.1 Smart Home Networking . . . 1
- 1.2 Gap . . . 2
- 1.3 Problem statement . . . 3
- 1.4 Delimitations . . . 4

2 Background **5**
- 2.1 Mesh Networking . . . 5
- 2.2 Bluetooth Mesh . . . 7
- 2.3 Gateway System Architecture . . . 8
  - 2.3.1 Microservice Architectures . . . 8
  - 2.3.2 Message Distribution Service . . . 9
- 2.4 Caching and Cache Replacement Algorithms . . . 10

3 Method **13**
- 3.1 Service Implementation . . . 13
- 3.2 Dynamic scene algorithm . . . 13
  - 3.2.1 Identifying a scene . . . 14
  - 3.2.2 Parsing setpoints . . . 14
  - 3.2.3 Comparing scene identifiers . . . 16
  - 3.2.4 Creating scenes . . . 16
  - 3.2.5 Caching of scenes . . . 17
- 3.3 Development Setup . . . 18

4 Evaluation **19**
- 4.1 Test Setup . . . 19
  - 4.1.1 Test Software . . . 19
  - 4.1.2 Test Hardware . . . 20
  - 4.1.3 Test Data . . . 21
- 4.2 Cache Hit and Cache Miss Evaluation . . . 22
- 4.3 Scene Cache Performance Evaluation . . . 23

5 Discussion **27**
5.1 Algorithm Performance 27
5.2 Limitations and Drawbacks 28
5.3 Future Work 30

6 Conclusion **31**

Bibliography **33**

List of Figures

2.1 An illustration of how a message may propagate from a sender node to a receiver in a mesh topology. The sender node broadcasts its message to its two adjacent nodes, the adjacent nodes in turn forward the message onward. The process is repeated until the message reaches the receiver node. 6
2.2 Illustration of how a setpoint is triggered by a user in proprietary implementation, the setpoints are sent to the company's cloud service which then forwards it to the gateway or in the case of HomeKit sent directly to the gateway. Worth noting is that HomeKit can sends commands via the cloud during specific circumstances, otherwise they are sent directly to the gateway. 8
2.3 Illustration of a monolithic application and a microservice application. A monolithic application encapsulates all functionality within a single process while a microservice application consists of many separate processes which together make up the application. Each colored shape represents an element of functionality. 9

3.1 Overview illustrating the logic used for the dynamic scene algorithm. A notable discrepancy in the diagram is the fact that the timer in reality works as an interrupt and not a state machine. Messages are abbreviated to msg. 14

4.1 Cumulative hit rate for each algorithm in every test case. 25
4.2 Total amount of messages sent for each test case and algorithm. . . . 26

List of Tables

3.1 Four examples of encoded setpoints. 15

4.1 Metrics recorded in the scene cache performance evaluation. 20
4.2 The different test scenarios. The random column indicates the interval for which the scene index can be called. 21
4.3 The execution time for a cache hit and miss 22
4.4 Results for the Easy scenario. 24
4.5 Results for the Medium scenario. 24
4.6 Results for the Random scenario. 24

1

Introduction

Home automation has been a topic of research for more than a century, starting with the invention of vacuum cleaners and dish washers. Over the last few decades, these devices have become prevalent and exist in most households. A recent development in the field of home automation is the 'smart home'. What differentiates smart home devices from traditional home automation devices is that they feature a network connection and are capable of operating autonomously to some extent. Some of the most common smart home devices today are, for example, smart lights, thermostats, and security cameras.

Given that these devices are connected to the Internet, they become a part of the Internet of Things (IoT). This allows users to control and monitor their home from anywhere, as long as they are connected to the Internet. Since the purpose of home automation is to make user life easier, it is crucial that the experience is as seamless as possible. Performance aspects such as low response time and high reliability are therefore a high priority.

This thesis was written at a company that offers smart home products with a focus on smart lighting products. The thesis investigates how caching can be implemented in their mesh gateway to potentially improve the performance of their products.

1.1 Smart Home Networking

With the emergence of more and more IoT devices in the home, making sure that all these devices connect efficiently and seamlessly to each other and the Internet has become a non-trivial issue. The traditional approach to connecting a device to the Internet is simply to connect it by cable. However, the cost and inconvenience of wiring a whole house is a significant hurdle. To avoid the need for wiring up every single device, most smart home devices today use one of the many wireless solutions available, such as Bluetooth or Wi-Fi, for example. However, a drawback of using wireless technology is the limited range and flexibility. That is why most smart homes utilize wireless mesh networking.

What differentiates a wireless mesh network from a traditional wireless network is that the individual devices in the network, called nodes, dynamically connect and communicate with each other and cooperate to transmit information over the network. The nodes communicate over the network by relaying messages from node to node until the messages reach their destination node [1].

A drawback of this approach is that it generates a lot of unnecessary traffic on the network. The unnecessary traffic generated by the nodes can in certain scenarios result in so-called broadcast storms. A broadcast storm is a phenomenon in which the network is overwhelmed by traffic, leaving it unable to function as intended. Multiple elements influence the probability of a broadcast storm occurring, with the volume of traffic and the layout of the network being notable contributors. It is generally desirable to avoid broadcast storms as increased traffic can lead to increased power consumption and poor network performance [2]. However, avoiding broadcast storms may be extra critical in certain scenarios. For example, with battery-powered mesh devices, where minimal power consumption is highly desired in order to maximize battery life. In the case of smart home devices, avoiding broadcast storms will result in a better user experience.

The size and topology of smart home meshes can vary greatly. A small home deployment may feature only a handful of devices, while a deployment found in a corporate office may utilize the maximum of 255 devices allowed by the proprietary mesh.

Users can control smart home meshes in multiple ways, via native apps or via so-called third-party integrations such as Apple HomeKit and Google Home. To allow users to control multiple devices simultaneously, wireless mesh implementations commonly support a feature called scenes. Scenes allow multiple nodes to be controlled with a single mesh command [3], a command being a control message sent to one or many mesh nodes. However, these third-party integrations do not support sending scene commands [4]. Rather, a message per device to be controlled will be sent into the mesh. This approach can be inefficient and may increase the chances that a broadcast storm will occur. In addition, all traffic generated by third-party integrations will be passed through a specific node, the so-called gateway node.

The gateway is a specific node type that acts as an interface between the mesh network and other networks, for example the Internet. This positioning makes the gateway by definition a bottleneck, since all messages sent from the Internet to the mesh network need to go through the gateway. Therefore, it is necessary for the gateway to handle passing traffic in a fast and reliable way to provide a good user experience. A frequently adopted strategy to reduce latency is the act of storing data which is likely to reoccur; in the context of a smart home such reoccurring data are, for example, scenes. The act of storing reoccurring data so that it can be recalled more quickly is called caching; it is a common concept in the field of computer science.

1.2 Gap

The topic of data caching in mesh networks has previously been explored. For example, the 2007 paper by Das [5] proposed cooperative caching between nodes to alleviate the gateway bottleneck. Other approaches to alleviating the gateway bottleneck have also been investigated, for example, the 2010 paper by Ancillotti [6], which proposes load balancing between multiple gateways in a mesh network.

The two examples above look at the general case of wireless mesh networks; however, smart home devices commonly implement mesh standards such as Bluetooth mesh. As Bluetooth mesh is a technical standard, implementations of it cannot be freely altered or improved, as the implementation must adhere to the Bluetooth standard specification. Therefore, solutions intended for the general case of wireless meshes may not be applicable to networks that implement Bluetooth mesh. However, there is research on improving the performance of Bluetooth mesh implementations in particular. For example, the 2022 paper by Perlioni [7] which found that the reliability of message delivery could be improved by randomizing a delay in which each node publishes its acknowledgment of a successfully received message.

Despite this, there appears to be a lack of research on the role of gateways in affecting the performance of Bluetooth mesh implementations. This thesis therefore investigates how caching in conjunction with the scene commands available in the Bluetooth mesh standard can be utilized to improve the performance of the gateway and the mesh. The concept of caching has been well researched, and using caching to improve the performance of a system is nothing new [8][9].

1.3 Problem statement

The research questions which this thesis project seeks to investigate are the following:

> *How can the amount of traffic in a Bluetooth mesh be reduced by optimizing the message distribution service on the mesh gateway using caching?*

This thesis aims to investigate how caching in conjunction with scene commands can be used to optimize message distribution for mesh network gateways. Using scene commands as a caching mechanism to control multiple nodes simultaneously, we will investigate if the maximum throughput of the gateway can be improved. These performance improvements may then translate into improved execution time and better user experience in the case of mesh networking in home automation.

> *How can broadcast storms in Bluetooth mesh networks be avoided by utilizing scene commands to a greater extent?*

The use of scene commands allows the gateway to control the devices on the mesh with less traffic, which should decrease the risk of a broadcast storm occurring on the mesh. This not only results in a more robust and reliable network, but also reduces overall network traffic.

> *How does the choice of cache replacement algorithm affect the caching performance in the context of caching scene commands?*

As the choice of cache replacement algorithm may have a significant effect on the overall caching performance, two different cache replacement algorithms will be investigated.

1.4 Delimitations

The domain of this thesis is closely related to computer networking; however, the scope of the thesis is limited to the processing performed on the mesh gateway. That is, modifying the actual network protocol used by devices to communicate on the mesh is beyond the scope of this thesis. Rather, the solutions investigated in this thesis are created within the limitations of the network protocol and the gateway hardware. This thesis only investigates mesh networks with a single gateway; networks with multiple gateways are outside the scope of this thesis.

2

Background

The following section presents the relevant theoretical background for the thesis. We will first give an overview of mesh networking before presenting the relevant parts of Bluetooth mesh. We then discuss microservice architectures before finally giving an overview of caching and the cache replacement policies explored in this thesis.

2.1 Mesh Networking

In a mesh network nodes are connected to each other in a non-hierarchical manner and each node will attempt to connect with as many other nodes as possible, thereby forming a mesh topology. The nodes will then cooperate to transmit information within the network; how this transmission is handled is defined by the routing algorithm used [10].

Data within a mesh network are commonly relayed between nodes using either flooding or routing algorithms [11]. For the purposes of this thesis, only flooding will be explored since that is what the company's and Bluetooth Low Energy (BLE) mesh use. More specifically, both utilize a form called managed flooding, which works in the following way. A sender node creates a message containing data, such as a payload, that it wants to transmit to a receiver node. The sender node will then transmit the message on the network. That is, it will send the message to all nodes to which it is connected. All nodes that receive the message will then retransmit it until the message reaches the receiver node to which it was originally addressed. To prevent nodes from retransmitting messages indefinitely, messages have a Time-to-Live, which decreases with each retransmission. In addition, the nodes store the messages they have sent and if they receive the same message again it is discarded [12]. Figure 2.1 illustrates how a message can propagate in a mesh topology.

An undesirable side effect of flooding methods is that they are increasingly susceptible to broadcast storms compared to routing approaches [13]. Broadcast storms are a phenomenon in which significant amounts of traffic accumulates on a network, which flooding networks are extra susceptible to due to the fact that all nodes on the mesh will forward all incoming messages. The consequence of this accumulation is that in severe cases it can result in the network not being able to transport normal traffic due to message collisions [2]. Broadcast storms can occur in many situations, for

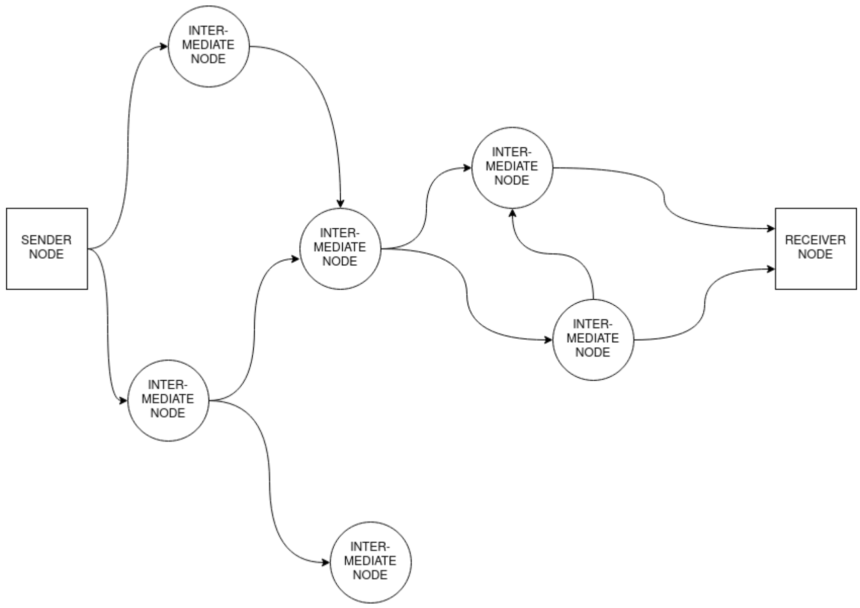


Figure 2.1: An illustration of how a message may propagate from a sender node to a receiver in a mesh topology. The sender node broadcasts its message to its two adjacent nodes, the adjacent nodes in turn forward the message onward. The process is repeated until the message reaches the receiver node.

example if many nodes transmit simultaneously or if a single node transmits a large amount of data continuously.

A solution to minimize the risk of a broadcast storm occurring is to stagger the transmission of messages out on the mesh, the drawback of this being the increased delay and inconsistent execution of commands on the mesh. This delay becomes especially obvious when controlling a group of lights on a mesh network, as the lights will turn on at different times, hence being dubbed the "popcorn effect" [14].

The gateway in the context of mesh networks is a type of node that allows communication between the mesh network and other networks, such as the Internet. The gateway functions as a regular node on the mesh, but will also feature one or more connections to other networks. The gateway therefore acts as the interface between the mesh network and the Internet outside. All information sent to or from the mesh must therefore pass through a gateway. A mesh network may feature multiple gateways. In such cases, load balancing between gateways as well as the positioning of the gateways on the mesh can have an effect on the performance of the mesh [6]. However, as discussed in section 1.4, for the purposes of this thesis, we are only investigating mesh networks that feature a single gateway.

2.2 Bluetooth Mesh

For implementation and testing this project uses a proprietary wireless mesh network that utilize Bluetooth Low Energy (BLE) for communication developed by the company. However, the implementation is similar to the official Bluetooth mesh standard and, therefore, we refer to the Bluetooth mesh standard when detailing the functionality of the Bluetooth mesh in the following sections [3].

Bluetooth mesh functions similarly to a publish/subscribe model, where publishers can publish to topics, and subscribers then subscribe to topics of interest. However, the Bluetooth mesh standard defines the node addresses to which the nodes can subscribe rather than the topics. The standard defines four types of node addresses; however, for the purpose of this thesis, only two are relevant: unicast addresses and group addresses [3]. The unicast address is unique to each node in the network and is assigned when a node joins the network. The group address represents a set of nodes. A node can join a group by adding the group address to its subscriber list. Every node on the network maintains a subscriber list where its unicast address is stored along with the group addresses to which the node is subscribed. In the company's implementation, all nodes are required to subscribe to group address 0. Group 0 can therefore be used to send messages to all nodes in the network.

Scenes are a way for nodes to store a specific state which can be recalled at a later time. A state is a configuration of the parameters available to the node. For example, if the node in question is a light in a smart home, the state could be a brightness level and a color temperature. Scenes are configured by sending scene configuration commands to each node involved in the scene. The command specifies the desired state and corresponding scene number which the node is to store. The scenes are then triggered using scene execute commands that are broadcast on the network. When a node receives this message, it will enter the state specified by the scene number in the execution command. Different nodes can have different states corresponding to a specific scene number, which means that one light can be set to turn on in response to a scene, while another light may be set to turn off in response to the same scene [15]. This is in contrast to group addresses, which would set all subscribers to the same state. In the proprietary Bluetooth mesh implementation, scene execute commands are broadcast to all nodes in the network. Using scene commands allows multiple nodes to be controlled with a single command, which is beneficial as it reduces the amount of traffic on the network [15].

Third-party integrations, in the case of smart home networks, are services provided by third parties which can control the mesh network. Examples of this are Google Home and Apple HomeKit. In the case of the proprietary implementation, these services communicate with the gateway, which in turn processes the requests and sends them out to the mesh. A request sent from an integration to the gateway is referred to as a setpoint. A setpoint specifies a state and a node that should enter the desired state. A simplified overview of how a user controls their devices via third-party integrations can be seen in Figure 2.2.

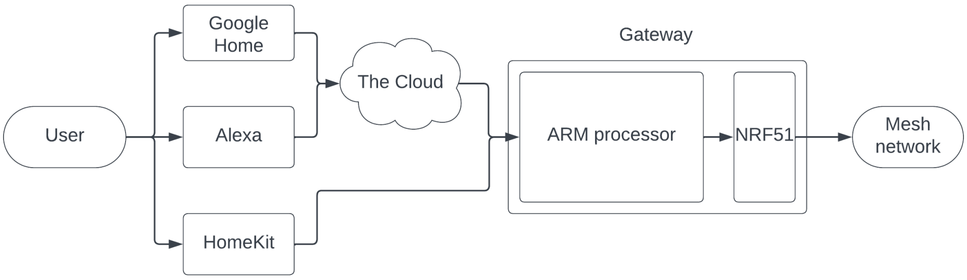


Figure 2.2: Illustration of how a setpoint is triggered by a user in proprietary implementation, the setpoints are sent to the company's cloud service which then forwards it to the gateway or in the case of HomeKit sent directly to the gateway. Worth noting is that HomeKit can sends commands via the cloud during specific circumstances, otherwise they are sent directly to the gateway.

Users of third-party integrations have the possibility to create scenes, referred to as action sets in the HomeKit documentation, and will be referred to as such from here on; the purpose is similar to that of scenes in a mesh. However, action sets do not automatically translate to a scene on the mesh. When an action set is sent, every action in that set is sent as an individual setpoint [4]. Furthermore, an action set created in such an integration is not visible to the gateway either. If every action is sent out individually on the mesh, it results in a lot of extra traffic on the mesh compared to if a single scene command was sent.

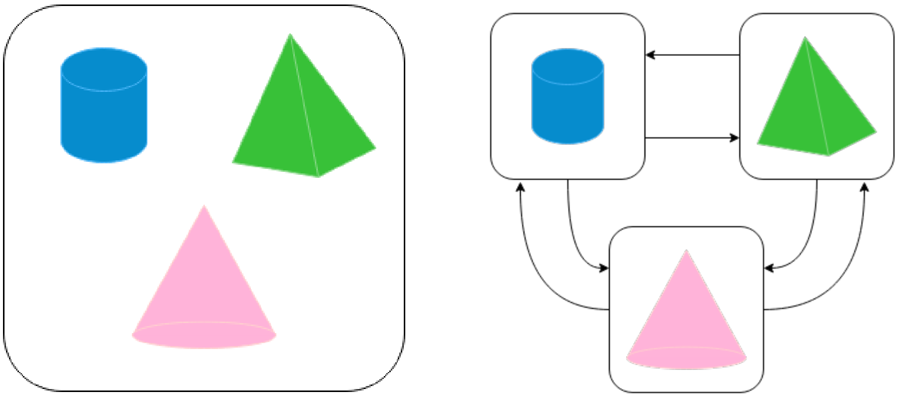
2.3 Gateway System Architecture

The system architecture of a gateway can of course be constructed in many ways. However, in the context of smart home meshes, the gateway is often a small IoT device with limited processing power, as is the case with the target device for this thesis. The target device, as mentioned in the Introduction, is the company's gateway. The hardware architecture of the target device consists of two main parts: an ARM processor running the gateway operating system and an NRF radio chip that handles the BLE communication between the mesh devices and the gateway processor. A more in-depth explanation of the target hardware for this thesis is provided in Section 3.3.

The ARM processor and the NRF chip communicate via an NATS messaging infrastructure. NATS is a messaging technology designed for edge devices and distributed systems. At its core, NATS functions according to a publisher/subscriber model, where "Data is encoded and framed as a message and sent by a publisher. The message is received, decoded, and processed by one or more subscribers."[16]

2.3.1 Microservice Architectures

The software which runs on the target gateway's ARM processor is constructed as a microservice architecture. Microservice architectures can, in short, be described as "an



(a) A monolithic application (b) A microservice application

Figure 2.3: Illustration of a monolithic application and a microservice application. A monolithic application encapsulates all functionality within a single process while a microservice application consists of many separate processes which together make up the application. Each colored shape represents an element of functionality.

approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms". Therefore, each element of functionality in the microservice application should be built as a separate service where only the service endpoints are exposed as an Application Programming Interface (API). This means that the internal structure of a service is hidden from every other service, and only the input and output parameters specified in the service API are visible. [17]

Microservice architectures have rapidly grown in popularity in the last two decades as an alternative to the more traditional approach of monolithic architectures. In monolithic architectures, an application is built as a single unit. That is, the entire system is built and compiled into a single logical executable. The two architectural styles are illustrated in Figure 2.3. Both architectural styles have benefits and drawbacks. One of the major benefits of microservice architectures and one of the reasons why the architectural style is becoming so widely adopted is its modularity. Since each element of functionality is developed as a separate, stand-alone service, each service can be easily modified and updated without affecting the other services in the system. This makes development of microservice applications very flexible as different services can be worked on separately by different developers. [18]

2.3.2 Message Distribution Service

As mentioned in the section above, the target gateway software functions as a microservice architecture where discrete services run separately and together make up the software. These services utilize NATS to communicate between each other. For the purpose of this thesis, the most relevant of these services is, which we will refer to as, the message distribution service. This service handles how the gateway communicates with the mesh, mainly by creating and configuring the messages that are to be transmitted on the mesh.

The message distribution service has multiple entry points corresponding to specific NATS topics. The service is triggered in response to NATS messages published on one of the topics to which the service is subscribed. Generally, a specific function is triggered as a callback in response to a specific topic, and the NATS message is passed as an argument to the function.

This service also handles how setpoints received from the third-party integration of Apple HomeKit are forwarded via the gateway to the mesh. It is this specific functionality which this thesis seeks to investigate. In the context of the company's gateway, only Apple HomeKit directly passes setpoints to the gateway, creating the gateway bottleneck issue discussed in Section 1.2. Other third-party integrations, such as Google Home or Amazon Alexa, communicate with the gateway via a cloud solution where the abundant processing power of cloud solutions are leveraged to alleviate the gateway bottleneck. Figure 2.2 illustrates how the three integrations communicate with the gateway. However, in the general case, the gateway bottleneck problem is not limited to Apple HomeKit; rather it can be generalized to any application or device which directly communicates mesh commands to the gateway and expects it to forward them onto the mesh.

2.4 Caching and Cache Replacement Algorithms

Caching is the act of storing data that is likely to reoccur such that it can be accessed more quickly upon next access. Caching is a common concept in computing and is utilized in many different domains. For example, in computer processors, where Central Processing Unit (CPU) caches exist on most modern processors. A CPU cache is a small fast memory located close to the CPU that is faster to read than the main memory [19]. When the CPU tries to read or write to a location in main memory, the CPU will first check if the requested data is stored in the cache. If so, it will read from or write to the cache rather than the location in main memory, which would be slower. When a piece of data is found in the cache, it is referred to as a cache hit. The cache hit time is the time it takes the CPU to find and read the data in the cache. When a piece of data is not found in the cache, it is referred to as a cache miss. A cache miss has an associated cache miss penalty, which is the time it takes the CPU to read the requested data from the main memory. Caching performance is often evaluated by the cache hit time, cache miss penalty, and hit rate. The hit rate being the percentage of all memory accesses where the requested data is found in the cache, thereby resulting in a cache hit.

Cache replacement policies are algorithms for managing the contents of caches. Due to caches generally being limited in size, managing the contents of the cache becomes important in order to optimize cache performance and, in turn, the performance of the system utilizing the cache. The optimal replacement policy would be one that replaces the content of the cache that will be accessed the farthest into the future. This policy is, of course, impossible to implement, as it requires knowledge of the future [20].

The First-in-First-out (FIFO) algorithm is one of the simplest cache replacement policies. The FIFO algorithm works as follows: the data item with the earliest creation time is replaced when a new item is to be stored. Therefore, the FIFO algorithm can be implemented using a list. New items are inserted at the head of the list, and the item at the tail is removed. The performance of the algorithm is considered poor because all data items will eventually be evicted when new items are referenced, regardless of the importance of the item [21].

The Least Recently Used (LRU) algorithm selects the contents that have not been referenced for the longest time for replacement. The policy is based on the principle of locality, which states that program and data references within a program tend to cluster. The performance of LRU can be near optimal in programs with high locality. However, it can impose significant overhead [20]. One possible implementation of the LRU algorithm is to store an associated age with each data item in the cache. The age of a data item is updated each time it is referenced, which means that a data item with a high age has been recently used. When a new data item needs to be written to the cache, the data item with the lowest age will be replaced.

3
Method

The core idea for this thesis is to evaluate the potential performance improvement by parsing action sets sent from third-party integrations and triggering them on the Bluetooth mesh with a single command instead of setting each setpoint individually. This section starts off by explaining the concept and how it is implemented, and then proceeds with presenting how it is evaluated.

3.1 Service Implementation

The algorithm presented in the following section that will be implemented as a service in the microservice architecture of the company gateway software. Specifically, the algorithm will be implemented into the preexisting message distribution service in the gateway software. Utilizing a pre-existing microservice enabled faster development, since a lot of pre-existing functions could be reused for the purposes of this thesis whilst also preserving functionality of the gateway, since it is quite a complex device.

In practice, this was done by hijacking a callback function triggered by a specific NATS message topic. The callback then acts as the entry point of the message distribution service. The input to the callback function is the payload of the NATS message, and the payload consists of a single setpoint that was initially created by some third party integration. The message distribution service then processes the setpoint following the algorithm presented in the following section.

3.2 Dynamic scene algorithm

For the purposes of this thesis, an algorithm was developed that dynamically parses, identifies, and creates scenes in the mesh and caches them on the gateway when an action set is sent from a third-party integration. This enables the gateway to trigger a scene with a single message compared to having to send an individual message for each setpoint in the scene. The algorithm will then be used to evaluate the performance effects of caching and different caching policies in the context of caching scene mesh commands. A huge advantage of this approach compared to the standard approach is that it completely eliminates the popcorn effect. An overview of the algorithm can be seen in Figure 3.1.

The idea behind this algorithm is to cache sets of setpoints that are likely to reoccur.

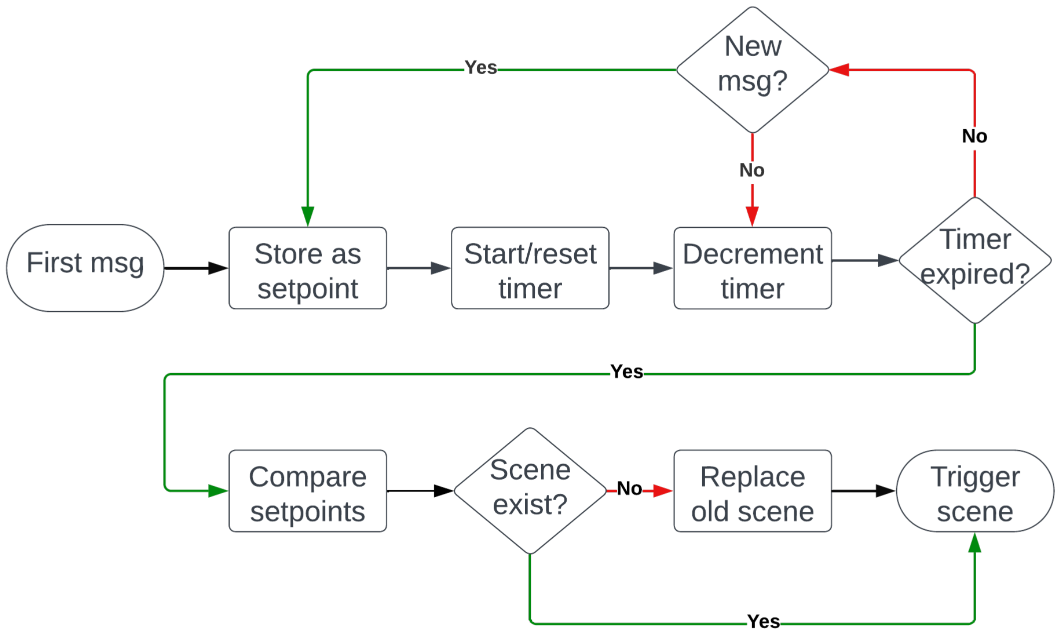


Figure 3.1: Overview illustrating the logic used for the dynamic scene algorithm. A notable discrepancy in the diagram is the fact that the timer in reality works as an interrupt and not a state machine. Messages are abbreviated to msg.

Due to the nature of smart homes, users are likely to group the nodes that they will frequently want to control simultaneously; this is commonly done with action sets. Action sets created by the user via third-party services are not advertised on the gateway or on the proprietary mesh. The algorithm therefore utilizes the scene feature available in the Bluetooth mesh standard that is described in Section 2.2 to effectively cache sets of setpoints on the gateway.

3.2.1 Identifying a scene

Setpoints are gathered by utilizing an inherent property of a scene, namely that multiple nodes are to be controlled simultaneously. The third-party integrations send the setpoints to the gateway in rapid succession and by utilizing this property, it is possible to detect that an action set has been triggered solely on the transmission window of the incoming setpoints. The algorithm does this by starting a short timer when a setpoint is received, and every time the callback function is triggered, the timer will be reset. In addition to resetting the timer, the payload of each setpoint is saved for parsing once all setpoints are received. If no setpoints reset the timer, it will eventually expire.

3.2.2 Parsing setpoints

When the timer expires, all setpoints are parsed individually and then combined to create a unique scene identifier. The purpose of creating a scene identifier is to compress the contents of the setpoint payload that are relevant to the algorithm and discard the rest. The only information needed for the algorithm to work properly is the address, state, and level of the node. A benefit of this approach is that it should make comparisons between scene identifiers significantly faster and requires less memory compared to storing and comparing payloads. The algorithm used to create unique scene identifiers can be seen in Algorithm 1.

Algorithm 1: How setpoints are encoded

Data: *Setpoints*
Result: Setpoints for every node are encoded at $S_{cache}[0]$ at index *node*

$S_{Cache}[0][node] \leftarrow []$;
for *each node in setpoints* **do**
  if *node.state == 1* **then**
    $S_{Cache}[0][node]$ = 0x80 | (node.level / 65535 * 100);
  else
    $S_{Cache}[0][node]$ = (node.level / 65535 * 100);

The state attribute indicates whether a device is on or off and is therefore stored as a Boolean value. The level attribute, utilized by dimmable devices, reflects brightness and is recorded as a 16-bit unsigned integer. It ranges from 0 to 65535 and therefore offers a high resolution.

The scene identifier is a vector consisting of 255 8-bit fields, which is stored at index 0 in S_{cache}. The vector has a size of 255 as that is the maximum number of nodes in the proprietary mesh implementation. Every bit field represents a single setpoint. The index of the bit field in the scene identifier vector corresponds to its unique node ID on the mesh. Each bit in the field indicates the properties of that setpoint, where the most significant bit indicates the state of the device, and the remaining seven bits encode a compressed level from 0-100 in binary as that was deemed a satisfactory resolution. To keep track of which nodes are in the scene, all node IDs that are not in the scene are encoded by setting all eight bits in the bit field corresponding to the specific node to 1. As a valid node configuration cannot result in the bit field '11111111' it was chosen to represent nodes that are not a part of the scene. Some examples of possible setpoints and their encoded counterparts can be seen in Table 3.1.

Table 3.1: Four examples of encoded setpoints.

	Bit							
Setpoint property	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Device On, 50 % brightness	1	0	1	1	0	0	1	0
Device Off, 100 % brightness	0	1	1	0	0	1	0	0
Device Off, 0 % brightness	0	0	0	0	0	0	0	0
Unused node	1	1	1	1	1	1	1	1

3.2.3 Comparing scene identifiers

When the unique scene identifier has been generated, it is used to check if it corresponds to a previously saved scene identifiers which are stored at different indexes in S_{cache} with each index being a separate scene. This is why bitfield vectors are used for scene encoding, since it enables rapid comparison of two vectors through a bitwise 'exclusive OR' operation, which helps in identifying if they are identical. If there is a match, the corresponding scene is simply triggered by broadcasting a set-scene command with the 'Scene ID' that corresponds to that scene identifier. The pseudocode for this implementation can be seen in Algorithm 2.

Algorithm 2: How matches are checked

Data: S_{cache}

Result: Returning true if a match was found or false if not.

```
scene ← 1;
for scene in S_cache do
    match ← True;
    for node in Scene do
        if S_Cache[0][node] ≠ S_Cache[scene][node] then
            scene++;
            match ← False;
            break;
    if match is True then
        return match;
return match;
```

3.2.4 Creating scenes

If no matching scene identifier is found, a new scene will always be created on the mesh. This is done by first deciding to what 'Scene ID' the scene identifier corresponds to. All previous references for that 'Scene ID' will first be erased from all devices on the mesh. After that, the gateway will configure the nodes that are part of the scene to associate this 'Scene ID' with the requested setpoints. This requires sending individual messages to each node with the required information, as mentioned in Section 2.2. After the scene is configured, it is triggered by broadcasting a set-scene command in a fashion similar to when a match is found.

For the algorithm to effectively use scene commands to dynamically control nodes, it must confirm that all nodes acknowledge that they are a part of the scene at the time of configuration. Therefore, the algorithm has logic that ensures that all nodes participating in the scene have sent an acknowledgment message before proceeding to send out the set-scene command mentioned above. The pseudocode for the algorithm can be viewed in Algorithm 3. The use of sleep between sending messages is to allow the message to propagate through the mesh and reach the intended recipient. The longer delay is intended to allow all nodes to respond with acknowledgments before transmitting more messages to the mesh. The precise delay values were determined through brief experimentation.

Algorithm 3: Pseudocode for ensuring all nodes are part of the scene

Data: S_{cache}

Result: Processes setpoints and handles acknowledgments

```
1  setpoints ← S_cache[0];
2  node ← 0;
3  while node < number of setpoints do
4      if node has not responded with ACK then
5          Send configure scene to node;
6          node++;
7          sleep for 75 ms;
8          go to Line3;
9      else
10         node++;
11         resolved setpoints++;
12 if resolved setpoints < number of setpoints then
13     resolvedSetpoints ← 0;
14     node ← 0;
15     sleep for 100 ms;
16     go to Line3;
17 Send set scene command to mesh;
```

3.2.5 Caching of scenes

Ideally, it would be possible to store every permutation of the mesh network as a scene beforehand and simply call it when necessary; however, for obvious reasons, this is not realistically feasible except for the smallest of networks. Second to that would be to store every scene that has ever been called, but this is quite a wasteful approach, for example, if a scene is only called once it would still be stored, or if its removed/modified at a later date the old version would still be stored. The solution to this problem is to employ caching as that results in the most important and qualified scenes actually being stored and can easily be replaced when they are no longer deemed a priority. Using caching, it would likely be possible to achieve almost the same level of performance as storing all scenes with fewer slots for dynamic scenes.

Although the gateway supports 63 user configurable scenes by default, this thesis seeks to evaluate how the choice and implementation of caching affects the overall performance, therefore, this implementation will be intentionally limited in the amount of storage slots available for dynamic scenes. Given that the matching algorithm has an O(N*M) time complexity where N is the number of scenes and M is the number of nodes, having to compare fewer scenes also makes the matching faster.

3.3 Development Setup

Development is conducted using a gateway development Board connected to a Raspberry Pi 4B. The Raspberry Pi enables development to be conducted in an isolated environment and simply adds flexibility to the setup since all the required software to monitor the gateway is kept on a separate machine.

The gateway development Board consists of two distinct parts on a single PCB: The first part is the ATSAMA5D27 CPU that runs the software and handles all the code and communication with the internet. The second part being an nRF51822 System-On-Chip (SoC) that communicates with the mesh that runs a separate firmware, that is similar to the one that the nodes on the mesh run. These two parts communicate through NATS messages. NATS messaging is used since the gateway's main CPU cannot address the antenna directly and thus broadcast messages to the mesh.

The Nordic Semiconductor nRF51822 is an Ultra Low Power SoC suitable for wireless applications, due to its built-in radio which supports both BLE and other 2.4 GHz wireless protocols such as the proprietary mesh [22]. The SoC is powered by a 16 MHz ARM Cortex M0 processor, which is the processor that communicates with the gateway's main processor. The nRF51 is used by all devices compatible with the proprietary mesh and simply connects the gateway to its own nRF51 module which is responsible for broadcasting messages to the mesh.

4

Evaluation

This chapter describes the test setup used to evaluate the algorithm and how the choice of caching algorithm affects the performance of the mesh. Furthermore, the evaluation of the algorithm and caching performance is divided into two parts. The first part establishes the cache hit time and the cache miss penalty. The cache hit time corresponds to the time it takes to match and execute a cached dynamic scene, and the cache miss penalty corresponds to the time it takes to generate a new dynamic scene if the incoming setpoints do not match any of the cached scenes. This data will be used to show how the caching algorithms perform compared to what can be seen as ideal performance.

The second part evaluates the performance of our caching implementation against a naïve cache-less implementation. This is essentially to find out how the caching algorithms would perform under differing conditions meant to simulate real-world usage. Since actual usage will depend on the specific user, multiple scenarios with a single variable change demonstrate inherent properties for the chosen cache algorithm for that scenario.

4.1 Test Setup

The following section is split into three parts presenting the test software, hardware and test data respectively. As implied by the name of the sections, they will focus on the two different aspects of the test setup and the considerations that had to be taken into account. The main consideration for the overall design of the tests was that they evaluated performance in a realistic setting, which is also why the tests measure the time for a scene to actually propagate on an actual mesh network, rather than simulating a network.

4.1.1 Test Software

Three different message distribution service implementations are used in the evaluation. All three test-services are implemented using the C programming language and are designed to work in the microservice architecture of the the company gateway. Two of the test services implement the algorithm presented in Section 3.2. What differentiates the two implementations is the choice of caching algorithm. The test services implement FIFO and LRU, respectively. The caching algorithms are implemented as suggested in Section 2.4. The third test-service implementation is a cacheless solution which is used as reference, and this implementation is referred to as the baseline. The baseline is also implemented as a service using a callback as the entry point as explained in 3.1. However, the baseline lacks the caching logic presented in Algorithm 1 and 2. Instead, Algorithm 3 is used to directly send mesh commands to execute the desired setpoints.

The evaluation of the three algorithms is carried out using a script written in Python. The core functionality of the script is that it communicates with the gateway via NATS messages and listens for specific responses from the gateway. Specifically, the script accepts a .csv-file containing test data in the form of a set of setpoints. The script then sends the sets to the gateway as NATS messages, and the script then expects a scene execute command to be communicated from the gateway as a NATS message, indicating that the gateway has finished processing the given set of setpoints. The script records timestamps of the setpoints and the scene execute command and calculates the time difference. The test metrics shown in Table 4.1 are then calculated from these timestamps and whether any scene configuration commands were sent.

Table 4.1: Metrics recorded in the scene cache performance evaluation.

Metric	Unit
Min Execution time	Seconds
Max Execution time	Seconds
Avg Execution time	Seconds
Std Deviation	Seconds
Cache Hit Rate	Percent of all cache accesses
Messages Sent	Number of messages

4.1.2 Test Hardware

The following hardware was used in the evaluation of the message distribution service:

- Gateway development board
- Raspberry Pi 4B
- Two test boards containing 10 physical devices each.

The three test-services were evaluated running on the gateway development board. The exact specification of the development board is detailed in Section 3.3. The test script was run on a Raspberry Pi 4B. The two boards communicated over an Ethernet connection.

In the evaluation a wireless mesh network consisting of 40 nodes was used. The network consisted of 20 unique devices. Specifically, 20 devices mounted on two separate boards. Each device has two outputs and can function as two separate nodes. Each device on the test boards is wired to two small LED lights, each light acting as a node.

Since the layout of the mesh network affects the performance of the network, using the test boards will affect the results of the tests. However, having a cluster of 40 nodes so close to each other should increase the load on the network because all nodes retransmit messages and thus demonstrate the performance of the algorithm under suboptimal mesh configurations.

4.1.3 Test Data

Three different test scenarios where used in the evaluation of the algorithm. A test scenario in this context is a series of action sets, exactly how the series looks depends on the specific scenario chosen, as seen in 4.2. The setpoints for the chosen action sets are then sent to the site.state.in topic in rapid succession, with them all being sent in approximately 1 ms. As mentioned previously, action sets is a collection of setpoints. These setpoints are generated with a Python script that creates a unique action set with random states and levels for each setpoint. Each set has 40 setpoints, meaning that all nodes in the test mesh are included in every set. The test scenarios are intended to emulate real use by having different levels of recurrence. This means that specific sets of setpoints reoccur in a predetermined pattern. To ensure that the tests were repeatable, the same random seed was used for the function responsible for choosing a random action set between all test scenarios. The three different cases represent different workloads, where case "Easy" and "Medium" feature recurring sets, and case "Random" is completely randomized.

Table 4.2: The different test scenarios. The random column indicates the interval for which the scene index can be called.

Scenario	Action set pattern	Nodes	Repetitions	Random
Easy	0, random, 1, random	40	50	2-3
Medium	0, random, 1, random	40	50	2-7
Random	random, random, random, random	40	50	0-7

4.2 Cache Hit and Cache Miss Evaluation

This test is designed to evaluate the cache hit time and the cache miss penalty of the algorithm running on the gateway. We define cache hit time and cache miss penalty as follows: The time between the gateway receiving the first setpoint and the gateway issuing a scene execution command onto the mesh.

The algorithm is evaluated by having sets of setpoints sent to the gateway in a predetermined pattern that emulates real use and ensures that both hits and misses occur. The case "Easy" in Table 4.2 shows the pattern used to generate the test data. The pattern consists of two reoccurring sets with a randomly selected set in between. The pattern features four different sets in total. The test pattern is run for 50 iterations, resulting in 200 action sets being sent in total.

For this test, LRU was chosen as the cache replacement policy with four available cache slots. The number of slots will affect the measured performance of the algorithm in this test, as fewer slots result in fewer scene identifiers having to be compared against the identifiers of the incoming setpoints. Fewer cache slots should therefore be beneficial for both cache hit time and cache miss time.

Another consideration is that the topology of the test mesh and the number of setpoints in each scene will affect the results. In the case of a miss, where a new scene must be created and configured, a scene configuration command must be sent to each node in the scene. The number of nodes in the scene will therefore affect the overall miss time as each message that has to be sent will incur a time penalty. The time penalty for sending a message is dictated to some extent by the mesh topology. If the mesh is large, a message will have to propagate further to reach its destination, resulting in higher time penalties per message.

The data for this test were generated by running the first pattern for a total of 50 repetitions. The results are presented in Table 4.3. The result is a mix of both hits and misses, to ensure reliable results. Worth noting is that the occurrence of calling scenes was higher, the sample size for both groups is still satisfactory.

Table 4.3: The execution time for a cache hit and miss

Type	Avg	Std dev	Min	Max	Amount
Hit (setting scene)	0.98 s	0.02 s	0.90 s	1.03 s	121
Miss (configure scene)	6.22 s	0.97 s	5.40 s	9.11 s	79

The data show a large time difference between a hit and a miss, with a hit being 6.48 times faster than a miss on average. The average cache miss penalty of 6.22 seconds is significant and may not be considered as 'responsive' from a usability perspective. The performance difference between a hit and a miss highlights the importance of a good caching algorithm that minimizes misses.

The min and max execution times for a miss are significantly different. The maximum execution time is 1.69 times slower than the fastest execution times. For a hit, the difference between the max and min was smaller. The slowest execution time recorded for a hit was 1.14 times slower than the fastest execution time.

The reason for the large differences in the min and max execution times for both hits and misses may be the fact that the testing was performed using a real mesh. Due to the flooding algorithm used in the mesh, messages may propagate differently between test iterations.

4.3 Scene Cache Performance Evaluation

This test is designed to evaluate the overall performance of the scene caching algorithm compared to a cache-less solution and to determine whether the cache replacement policy affects caching performance in the context of smart home scenes. Two implementations of the algorithm are tested, one implementation using FIFO as the cache replacement policy and one implementation using LRU. A cache-less implementation will also be tested as a baseline. Table 4.1 lists the metrics that are recorded in the evaluation. Execution times are measured as the time between the gateway receiving the first setpoint and the gateway issuing a scene execution command onto the mesh. Additionally, the total number of messages generated by the scene caching algorithm will be recorded in order to compare the total amount of traffic generated by the algorithm as compared to the baseline.

The algorithms are evaluated by having sets of setpoints sent to the gateway, in a similar way to how third-party integrations send out action sets, using predetermined patterns that is intended to emulate real use. Multiple sets of test data are used for evaluation with various degrees of randomization. The patterns used to generate the test data are listed in Table 4.2. The test cases "easy" and "medium" feature two reoccurring sets of setpoints. The reoccurring sets are designed to emulate real use in a smart home where the user is likely to have a scene configured that they repeat with some consistency. For example, a user may wish to turn off all lights in the home every day before going to bed; the reoccurring sets in the test data are there to emulate this behavior. The intention of test case "random" is to show how the algorithm performs in what could be deemed a worst-case scenario. Here, the cache is significantly smaller than the number of scenes and there is no pattern which could potentially 'help' the cache replacement algorithm.

The topology of the test mesh and the size of the scenes affect the outcome of this evaluation for the same reasons as discussed in the final paragraph of Section 4.2.

The baseline execution time is constant in terms of how many nodes there are in the scene; therefore, it is not necessary to test it for multiple cases since the number of nodes is constant in these tests. That is why the baseline numbers are identical for all scenarios.

Table 4.4: Results for the Easy scenario.

Algorithm	Case	Avg	Std dev	Min	Max	Hit rate
LRU	Easy	0.94 s	0.64 s	0.77 s	5.88 s	98 %
FIFO	Easy	0.92 s	0.61 s	0.76 s	5.53 s	98 %
Baseline	Easy	6.47 s	0.43 s	5.58 s	7.72 s	N/A

Table 4.5: Results for the Medium scenario.

Algorithm	Case	Avg	Std dev	Min	Max	Hit rate
LRU	Medium	2.26 s	2.06 s	0.78 s	6.42 s	70 %
FIFO	Medium	3.03 s	2.24 s	0.77 s	6.17 s	51.5 %
Baseline	Medium	6.47 s	0.43 s	5.58 s	7.72 s	N/A

Table 4.6: Results for the Random scenario.

Algorithm	Case	Avg	Std dev	Min	Max	Hit rate
LRU	Random	3.43 s	2.31 s	0.77 s	8.45 s	45 %
FIFO	Random	3.19 s	2.19 s	0.76 s	5.99 s	47 %
Baseline	Random	6.47 s	0.43 s	5.58 s	7.72 s	N/A

Tables 4.4, 4.5, and 4.6 show the results of the scenarios 'easy', 'medium' and 'random', respectively. For scenario 'easy', FIFO has the best average execution time with a 7.03 times speed-up compared to the baseline and a 98% hit rate. For scenario 'medium', LRU has the highest hit rate and the fastest average execution time with a 2.86 times speed-up over the baseline and a 1.34 times speed-up over FIFO. For scenario 'random', FIFO has the best average execution time with a 2.03 times speed-up compared to the baseline and a 47% hit rate. Note that FIFO has a slightly faster average execution time over LRU in Scenario 'easy' even though both implementations show a 98% hit rate. This may be due to the fact that the LRU implementation is more complicated than FIFO, causing a longer processing time.

The results include average, standard deviation, minimum, and maximum execution times to provide an estimate of scene execution duration, but these should not be the primary focus of the tables, as they can vary based on implementation. The column that should be focused on is the one that contains the hit rate.

The performance over time was deemed an interesting metric due to the algorithm having to configure scenes on the mesh, to evaluate this Figure 4.1 was created, which plots the current hit rate of each algorithm in the different scenarios. Regardless of the algorithm, all test cases demonstrate a sharp increase in hit rate in the beginning before slowly decreasing and appearing to settle at a somewhat stable level. The exception to this is the easy case, where once the four different action sets are configured, the hit rate approaches 100 %. As can be seen, 200 iterations seems to be a reasonable amount of iterations as the hit rate in all cases flattens out.

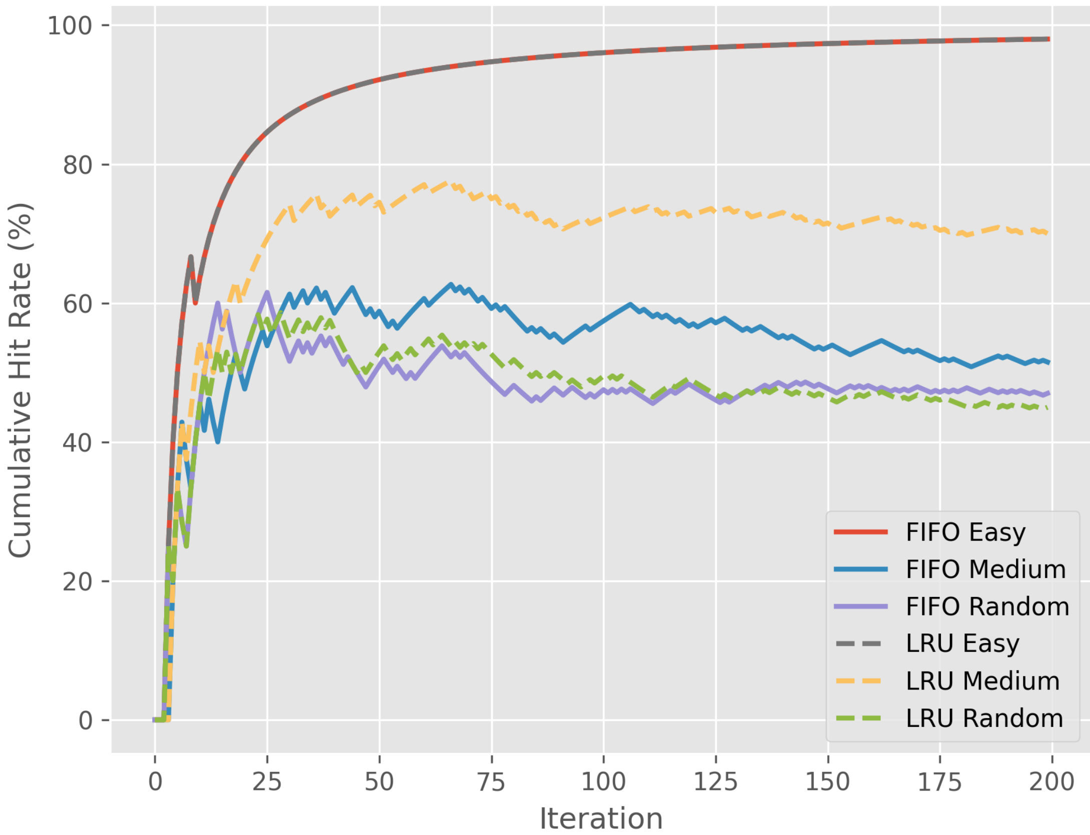


Figure 4.1: Cumulative hit rate for each algorithm in every test case.

To evaluate how much traffic is being sent by each algorithm, the total messages sent for all scenarios were totaled and the plotted in Figure 4.2. From the plot, it is clear that regardless of scenario and cache algorithm, the total amount of traffic generated by the gateway is reduced. When comparing the hits shown in Tables 4.4, 4.5 and 4.6 and the traffic generated, there appears to be a correlation between the hit rate and traffic, where a higher hit rate corresponds to less traffic. This was expected as a higher hit rate means that less scenes have to be configured on the mesh, leading to less traffic.

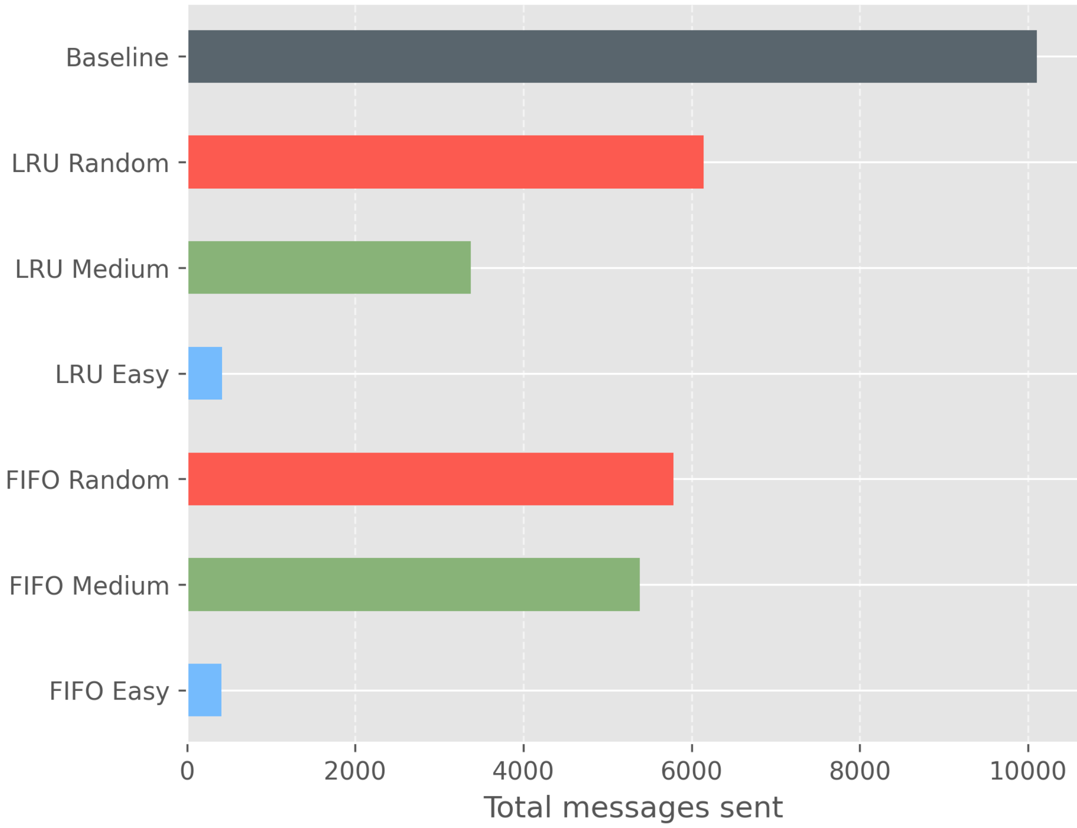


Figure 4.2: Total amount of messages sent for each test case and algorithm.

5 Discussion

The purpose of this thesis was to evaluate whether the use of caching on a BLE gateway could improve the efficacy of the message distribution service and how such an implementation affects the execution time. An algorithm was developed that dynamically creates and stores scenes, effectively using the scene functionality available in the Bluetooth mesh standard as a caching mechanism. Two implementations were developed, one that used FIFO as the cache replacement algorithm and one that utilized LRU. The two implementations were then evaluated and compared with a baseline implementation without cache. These implementations were evaluated by their raw cache hit-and-miss execution times and how well they performed in scenarios meant to simulate different workloads.

The following sections of this chapter will discuss the results gathered in the evaluation of the algorithm in more detail and highlight benefits and drawbacks. We will also suggest possible future work for the continuation of this research.

5.1 Algorithm Performance

From the results of Section 4.2, a significant difference in execution time between a hit and a miss can be observed. This is reasonable given that, for a hit, only a single message needs to be sent to the mesh compared to a single message per node to be controlled, which in this testing is 40. This shows that there are significant performance improvements to be had by using dynamic scenes. However, the choice of caching algorithm is also a design choice that affects performance, as reducing the number of misses is key to improving the average throughput.

From the results of 4.3, presented in Tables 4.4, 4.5, and 4.6, it is clear that performance varies drastically between the three scenarios. Performance differences between the two caching algorithms were also evident, especially in the 'medium' test scenario. It was expected that both algorithms would perform well when the number of scenes was kept within the size of the cache. In that case, the choice of caching algorithm becomes irrelevant as all scenes can be stored in the cache. The 'easy' test scenario confirmed this expectation, since both algorithms performed similarly, having an identical hit rate and similar execution times. Both algorithms achieved a 98% hit rate because the cache started empty, leading to initial misses. With four distinct action sets sent 200 times in total, four misses correspond to a hit rate of 98%.

The more demanding 'medium' scenario showed that there is indeed a difference between the caching algorithms, LRU being 1.34 times faster than FIFO on average. This outcome was expected, as LRU is regarded as a more sophisticated algorithm. For the same reason, our expectations were that LRU would also be faster in the 'random' scenario. However, this was not the case; FIFO showed a slightly higher hit rate and a faster average execution time in the 'random' scenario. A reason for this could be that since the test data used in this scenario were completely randomized, there is no locality in the data that the LRU algorithm could take advantage of. This may be why the two algorithms perform similarly in this scenario. Although the algorithms showed similar results in the random scenario, FIFO performed slightly better. This may be due to the simplicity of the FIFO implementation. As discussed in Section 2.4, FIFO is one of the simplest algorithms to implement, and LRU being one of the more difficult, this fact may be evident in the performance of the two implementations.

The results of Section 4.2 also show that there is no clear winner in the choice of cache replacement policy when it comes to hit rate performance, especially because the FIFO algorithm slightly beats the LRU algorithm in the 'random' scenario. However, that is overshadowed by the significant performance improvement of the LRU algorithm in the 'medium' scenario. As the 'medium' scenario was intended to reflect real-world usage most accurately, the results from that scenario therefore indicate that LRU would be the most appropriate choice in an implementation intended for the end consumer. The 'normal' results also demonstrate how the more advanced logic of the LRU algorithm yields better performance when the cache size is exceeded by the number of scenes, and there are some scenes repeated often.

From Figure 4.2, it is clear that using scenes as a caching mechanism to reduce the traffic generated by the gateway can be effective regardless of workload. Since even in the 'random' scenario where both LRU and FIFO performed worst, the traffic is almost halved. This indicates that for certain use cases where reducing traffic is a high priority, for example in the case of battery powered devices on the mesh, implementing even a small cache in the gateway may be beneficial.

5.2 Limitations and Drawbacks

For the purpose of this thesis a number of technical limitations have been imposed. The most significant being the limitation on cache slots; the number of dynamic scenes that are stored on the gateway. If the solution was to be adapted for a production environment, the number of cache slots could perhaps be increased to improve the overall hit rate. However, by increasing the number of cache slots the speed of the caching algorithm may decrease as more scene identifiers have to be compared in order to determine if the incoming scene exists in the cache.

Although a hit in the scene cache shows a significant gain compared to the baseline, it is worth recognizing that even when the algorithm gets a cache hit, the execution time still leaves a bit to be desired. This is likely due to how the setpoints are handled in the gateway operating system running on the test hardware. In order for the gateway to properly communicate with the mesh, incoming setpoints are stored in an SQLite database. SQLite is a C-library which implements a small SQL database engine, SQL being a language for creating and managing databases. In testing unrelated to the evaluation of the scene caching algorithm as described in chapter 4, it was found that writing setpoints to the database introduces an overhead of approximately 20 ms per setpoint on average. This results in a noticeable delay when processing 40 setpoints in quick succession. However, as a proof-of-concept the algorithm still shows promise, and with further optimization to setpoint handling further gains could most likely be realized. Since this delay is closely correlated to how many setpoints the scene has, the delay would be significantly less noticeable on a smaller mesh network, which would also be more realistic, since 40 devices is quite a large scene based on user data.

The most significant downside of the algorithm is that it only significantly improves performance when a scene has been configured on the mesh. If the scene has not been registered the improvement in response time will be non-existent compared to just sending the command directly to the devices without creating a scene. This drawback is of course not unique to this algorithm; rather, it is universal for all caching solutions. However, creating a scene yields the advantage of eliminating the popcorn effect as discussed in Section 3.2. This is an aspect which should not be neglected as it can be argued that it improves the user experience.

Another negative aspect of this algorithm is that it still sends out quite a lot of messages on the mesh when it configures a scene, which might lead to dropped messages on the network. This is quite counter-intuitive to the main idea behind the algorithm, which is to decrease the overall traffic on the mesh. To alleviate this, the retry logic was implemented as mentioned in Section 3.2. A drawback of this retry logic is that the reliance on acknowledgment from the nodes results in twice as many messages being sent on the network.

Another limitation in this study is with respect to the test data used in the algorithm evaluation. All test data was generated using a pattern designed by us. Generating data that accurately simulate demanding, yet realistic, real-world usage was challenging. It would have been beneficial to use real user data for testing; however, collecting and using such data raises serious privacy concerns, so instead multiple tests with varying properties were used. We are confident that the test results highlight both the strengths and weaknesses of this implementation and their impact on performance, despite the limitations of the available data.

The final drawback we wish to highlight is that the algorithm as we present it is not very general in its current state. The algorithm is designed for a specific node type that responds to a specific type of mesh command, namely a setpoint specifying a state as 'on' or 'off' and a brightness level that can be encoded as a number between 0 and 126. These specifics, such as the size of the scene identifier, can, of course, be altered if the algorithm were to be implemented in a different context. As the algorithm has only been implemented and evaluated for a specific node type, it is unclear to what different smart home devices it could be extended to.

5.3 Future Work

For future work, the most critical next step is possibly to evaluate the algorithm on real-user data. Since our results show that the performance of the algorithm varies greatly under different workloads, testing the algorithm under real workloads may allow for more accurate tuning of the algorithm. For example, by changing the number of scenes that are cached and choosing the most appropriate cache replacement algorithm. Also, evaluating more caching algorithms may be interesting as there are many available.

It should be noted that if an algorithm like this were to be implemented in a production environment, utilizing some sort of dynamic allocation for the amount of available dynamic scenes would make it possible to get a cache hit every time. Due to the fact that, in most cases, scenes are set up once and rarely changed or reconfigured. How the logic for such an algorithm would look for this specific use case would be a suitable extension of this work. Especially considering that the number of available scenes is shared between the user and the dynamic scene algorithm.

An aspect that this thesis does not evaluate is a cache replacement algorithm specifically designed for caching of scenes. This could be achieved by, for example, having scene size taken into account when choosing which cache slot to replace. This would make for some interesting future research since scenes often vary in size in real-world scenarios and since larger scenes would see more significant improvements in performance. Therefore, developing a cache replacement algorithm specifically designed for the caching of scenes and integrating it into our scene caching algorithm could result in additional performance benefits.

Another possible improvement to the caching algorithm would be for it to be able to identify a partial match between an action set and a scene. Currently, if the user triggers an action set that consists of a cached scene and a device marked as unused in the cached scene, we would get a cache miss and set up a new scene that includes that device. A better approach would probably be to trigger the scene and then send a separate message for that device, since that would most likely still be noticeable faster than configuring a scene.

Another way for the algorithm to be improved in the future could be to improve how the algorithm configures scenes on the mesh. Currently, it requires a lot of messages to be broadcast on the mesh, which takes a significant amount of time. Finding a solution to this that requires fewer messages and does not compromise performance or reliability could be a challenge.

6
Conclusion

The approach taken to answer the stated research questions was to develop an algorithm that parses action sets generated by third-party integrations and dynamically creates scenes on a Bluetooth mesh corresponding to that action set, effectively using Bluetooth mesh scene commands as a caching mechanism, as scene commands can be used to control multiple nodes with a single mesh command. The algorithm was implemented as a service on a gateway development board, which was in turn used to evaluate the algorithm.

In evaluation, it was shown that using scene commands in this manner decreases network traffic generated by the gateway, reducing the chance of broadcast storms occurring. The utilization of scene commands also came with the advantage of completely eliminating the popcorn effect.

In addition to the above, the proposed algorithm was shown to reduce the time required to control a given set of nodes. In the best case where all incoming action sets already are cached on the gateway, a speed up of 6.35 times over a non-scene caching implementation was shown. In a higher workload scenario where the number of different incoming action sets was higher than the number of available cache slots, an average speed up of 2.86 times was shown. Thus, showing that a significant performance gain can be achieved by using scene caching in the gateway message distribution service.

The main drawback of this approach is that there is no improvement in execution time when there is a miss in the cache and a new scene has to be configured on the mesh. The performance gains provided by the algorithm are therefore greatly dependent on how many different action sets the user utilizes and the number of scenes we are able to store on the gateway, i.e. the number of cache slots. Therefore, a main focus for future research should be improving the amount of cache hits, which may be possible by, for example, developing a more advanced cache replacement algorithm tailored to this specific use case.

Bibliography

[1] M. Baert, J. Rossey, A. Shahid, and J. Hoebeke, "The Bluetooth Mesh standard: An overview and experimental evaluation," *Sensors (Switzerland)*, vol. 18, no. 8, Aug. 2018, ISSN: 14248220. DOI: 10.3390/s18082409.

[2] R. Rondon, A. Mahmood, S. Grimaldi, and M. Gidlund, "Understanding the Performance of Bluetooth Mesh: Reliability, Delay, and Scalability Analysis," *IEEE Internet of Things Journal*, vol. 7, no. 3, pp. 2089–2101, Mar. 2020, ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2960248. [Online]. Available: https://ieeexplore.ieee.org/document/8935348/.

[3] Mesh Working Group, "Mesh Profile Bluetooth ® Specification," Bluetooth, Tech. Rep., Jan. 2019.

[4] Apple, *HMActionSet*, 2024. [Online]. Available: https://developer.apple.com/documentation/homekit/hmactionset?changes=latest_major.

[5] S. M. Das, H. Pucha, and Y. C. Hu, "Mitigating the gateway bottleneck via transparent cooperative caching in wireless mesh networks," *Ad Hoc Networks*, vol. 5, no. 6, pp. 680–703, Aug. 2007, ISSN: 15708705. DOI: 10.1016/j.adhoc.2006.11.004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1570870506001028.

[6] E. Ancillotti, R. Bruno, and M. Conti, "Load-balanced routing and gateway selection in wireless mesh networks: Design, implementation and experimentation," in *2010 IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, Montreal: IEEE, Jun. 2010, pp. 1–7, ISBN: 978-1-4244-7264-2. DOI: 10.1109/WOWMOM.2010.5534991. [Online]. Available: http://ieeexplore.ieee.org/document/5534991/.

[7] P. Pierleoni, A. Gentili, M. Mercuri, A. Belli, R. Garello, and L. Palma, "Performance Improvement on Reception Confirmation Messages in Bluetooth Mesh Networks," *IEEE Internet of Things Journal*, vol. 9, no. 3, pp. 2056–2070, Feb. 2022, ISSN: 2327-4662. DOI: 10.1109/JIOT.2021.3090656. [Online]. Available: https://ieeexplore.ieee.org/document/9459930/.

[8] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, *A survey on cache management mechanisms for real-time embedded systems*, Nov. 2015. DOI: 10.1145/2830555.

[9] J. Wang, "A Survey of Web Caching Schemes for the Internet," Tech. Rep.

[10] A. Cilfone, L. Davoli, L. Belli, and G. Ferrari, "Wireless Mesh Networking: An IoT-Oriented Perspective Survey on Relevant Technologies," *Future Internet*, vol. 11, no. 4, p. 99, Apr. 2019, ISSN: 1999-5903. DOI: 10.3390/fi11040099. [Online]. Available: https://www.mdpi.com/1999-5903/11/4/99.

[11] S. Cortesi, C. Vogt, E. Reinschmidt, and M. Magno, “Latency and Power Consumption in 2.4GHz IoT Wireless Mesh Nodes: An Experimental Evaluation of Bluetooth Mesh and Wirepas Mesh,” in *2023 19th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, vol. 2023-June, IEEE, Jun. 2023, pp. 200–205. DOI: 10.1109/WiMob58348.2023.10187799. [Online]. Available: https://ieeexplore.ieee.org/document/10187799/.

[12] a. Hernandez-Solana, D. Perez-Diaz-De-Cerio, M. Garcia-Lozano, A. V. Bardaji, and J.-L. Valenzuela, “Bluetooth Mesh Analysis, Issues, and Challenges,” *IEEE Access*, vol. 8, pp. 53 784–53 800, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2980795. [Online]. Available: https://ieeexplore.ieee.org/document/9035389/.

[13] I. Natgunanathan, N. Fernando, S. W. Loke, and C. Weerasuriya, “Bluetooth Low Energy Mesh: Applications, Considerations and Current State-of-the-Art,” *Sensors*, vol. 23, no. 4, p. 1826, Feb. 2023, ISSN: 1424-8220. DOI: 10.3390/s23041826. [Online]. Available: https://www.mdpi.com/1424-8220/23/4/1826.

[14] M. Woolley, “Understanding Reliability in Bluetooth® Technology,” Tech. Rep., Oct. 2020.

[15] Bluetooth SIG, *Bluetooth Mesh Glossary.* [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/feature-enhancements/mesh/mesh-glossary/#scene.

[16] *What is NATS*, 2022.

[17] M. ’Lewis James’ ’Fowler, *Microservices: a definition of this new architectural term*, Mar. 2014. [Online]. Available: https://www.martinfowler.com/articles/microservices.html.

[18] G. Blinowski, A. Ojdowska, and A. Przybylek, “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,” *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022, ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3152803.

[19] S. Kumar and P. K. Singh, “An overview of modern cache memory and performance analysis of replacement policies,” in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, Coimbatore: IEEE, Mar. 2016, pp. 210–214, ISBN: 978-1-4673-9916-6. DOI: 10.1109/ICETECH.2016.7569243. [Online]. Available: http://ieeexplore.ieee.org/document/7569243/.

[20] M. Zeeshan Farooqui, M. Shoaib, and M. Zunnun Khan, “A Comprehensive Survey of Page Replacement Algorithms,” Tech. Rep. 1, 2014.

[21] A. Dan and D. Towsley, “An approximate analysis of the LRU and FIFO buffer replacement schemes,” in *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems - SIGMETRICS ’90*, New York, New York, USA: ACM Press, 1990, pp. 143–152, ISBN: 0897913590. DOI: 10.1145/98457.98525. [Online]. Available: http://portal.acm.org/citation.cfm?doid=98457.98525.

[22] “nRF51822 Multiprotocol Bluetooth® low energy/2.4 GHz RF System on Chip Product Specification v3.4 Key Features,” Tech. Rep., 2023. [Online]. Available: www.nordicsemi.com..