UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Clock Synchronization between Observational Units in the Arctic Tundra

Sigurd Karlstad

INF-3990 Master thesis in Computer Science
15 May 2021

To My Gradparents.

Thanks for Everything.

“Normally, if given the choice between doing something and nothing, I’d choose to do nothing. But I will do something if it helps someone else do nothing. I’d work all night, if it meant nothing got done.”
–Ron Swanson, Parks and Recreation

“Sometimes I’ll start a sentence and I don’t even know where it’s going. I just hope I find it along the way.”
–Michael Scott, The Office

Abstract

The arctic tundra is one of the ecosystems that is most affected by climate changes. The effects of these changes on the wildlife in the arctic are therefore critical to monitor. To monitor the changes, small computing devices with sensors and cameras, known as *Observational Units*, can be used.

Using a cluster network of interconnected observational units, so that data can be reported from the most distant nodes to a homebase, introduces problems as the node's local clocks tend to skew away from each other based on the environment they are in. This thesis aims to address the problem of clock synchronization in cluster networks that are disconnected from constant power and the internet.

This thesis describes how we designed, built, and tested a prototype software solution for an interconnected Wireless Sensor Network using observational units built for arctic climates. We designed and built a two-phased system where the nodes dynamically join a network and synchronize their duty cycling and clocks with each other, creating a synchronized cluster of nodes that allows for data to be propagated, where any of the nodes can become sink nodes if they have a connection to a homebase. The nodes use a built-in clock synchronization operation to achieve synchronized clocks across the cluster network.

The prototype system can perform the operations as planned. However, the results show that the system scales poorly when introducing new sinks to the network while running in the operational phase, as the paths shared scales exponentially, in contrast to when a sink is introduced in the starting phase of the system, where the transmissions grow linearly with a degree based on the number of nodes in the network. The time synchronization experiments also showed that the network is able to remain synchronized, although transmission number is a concern when the network does not have any sink nodes.

Acknowledgements

First I would like to thank my main advisor Professor John Markus Bjørndalen, and my co-advisors Professor Otto Anshus and Postdoctoral Research Fellow Issam Raïs, for providing me with support whenever it was needed throughout this thesis.

I would also like to thank my family for supporting through every decision that I have made and encouraging me along the way.

I would like to thank my collaborator and friend, Erlend Karlstrøm for sharing his thoughts on the project, and for being a good discussion partner.

Lastly, I would like to thank my friends for all the support they have given me.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xv

List of Definitions xvii

List of Listings xix

1 Introduction 1
- 1.1 Observational Units . . . 2
- 1.2 Problem with OU Networks . . . 2
- 1.3 Limitations . . . 2
- 1.4 Overview of Contributions . . . 3
  - 1.4.1 Erlend Karlstrøm . . . 3
  - 1.4.2 Sigurd Karlstad . . . 3
  - 1.4.3 Collaboration . . . 4
- 1.5 Overview of Project . . . 4

2 Background 7
- 2.1 Motivation . . . 8
- 2.2 Aim . . . 8

3 Design Guidelines 11
- 3.1 Power Preservation and Duty Cycling . . . 11
- 3.2 Data Transmission . . . 12
- 3.3 Dynamic Topology Management . . . 12
- 3.4 Heterogeneous Communication . . . 12
- 3.5 Non-Interference . . . 12
- 3.6 Persistent Data Storage . . . 13
- 3.7 Network Clock Stability . . . 13

4 Related Work **15**
4.1 Existing Observational Node Networks 15
4.2 Routing and Communication in Node Networks 16
4.3 Score Based Routing 18
4.4 Time Synchronization 18

5 Architecture **21**
5.1 Architechtural Roles 21
5.2 Node Life-Cycle 22
5.2.1 Duty Cycling 23
5.3 Neighbor Discovery 23
5.4 Topology 24
5.5 Score Based Routing 25
5.6 Event-Driven 25
5.7 Message-Oriented Priority Mailboxes 26
5.8 Intra-Node Communication 27
5.9 Content Propagation 27
5.10 Adaptive Time Synchronization 27
5.11 Safezones In Time Synchronization 28
5.12 Forced Time Synchronization 28

6 System Design **29**
6.1 General Functionality 29
6.1.1 Operational Configuration 30
6.1.2 Path Structure 31
6.1.3 Neighbor Metadata Knowledge 31
6.1.4 Score Calculation 32
6.1.5 Path Handling 33
6.2 Starting Phase 37
6.2.1 Receiver Thread 37
6.2.2 Introduction Mode 38
6.2.3 Reactive Gossip Mode 40
6.2.4 Local Cleanup 42
6.2.5 Synchronized Sleep Schedule 42
6.3 Operational Phase 42
6.3.1 Post-Sleep Preparation 42
6.3.2 Awake-Cycle 44
6.3.3 Pre-Sleep Preparation 51
6.3.4 Low-Power Mode 51
6.4 Time Synchronization 51
6.4.1 Immediate Time Synchronization 51
6.4.2 Gradual Time Synchronization 54
6.4.3 Skewed Safe Synchronization Zone 55

7 Implementation 59
7.1 Introduction Broadcast 60
7.2 Topology Generation 60
7.3 Environmental Readings 61
7.4 Battery Drain 61
7.5 Simulated Duty Cycling 61
7.6 Simulated Skew 62
7.7 Homebase Design 62

8 Validation 65
8.1 Validation Setup 65
8.2 Validation Design 66
8.2.1 General Network validation 66
8.2.2 Time Synchronization Validation 70
8.3 Validation Results 74
8.3.1 General Network Results 74
8.3.2 Time Synchronization Results 76

9 Experiments 79
9.1 Experimental Setup 79
9.2 Experiments Design 81
9.2.1 General Network Experiments 81
9.2.2 Time Synchronization Experiments 83
9.3 Experiment results 87
9.3.1 General Network Results 87
9.3.2 Time synchronization Experiment Results 93

10 Discussion 107
10.1 General network 107
10.1.1 Node Communication 107
10.1.2 Centralized vs. Decentralized Architecture 107
10.1.3 Multi-Hop vs. Single-Hop Routing 108
10.1.4 Path Knowledge vs. Neighborhood Knowledge 108
10.1.5 Using the Prototype with Existing Systems 109
10.1.6 Restriction of Neighbor Amount 109
10.1.7 Scalability Concern 109
10.1.8 Load Balancing Concern 110
10.1.9 Score Calculation Metrics 110
10.2 Simulator 110
10.2.1 Why Create A Custom Solution 111
10.2.2 Simulated Topology Management vs. Range-Based Management 111
10.3 Time Synchronization 111
10.3.1 Why Use Two Adjustment Algorithms 111

10.3.2 Synchronization Algorithms 112
10.3.3 Using The Skewed Safe Synchronization Zones 113
10.4 Weaknesses 114
10.4.1 Dead Node Handling Problem and Path Management 114
10.4.2 Scalability 114
10.4.3 Missing CPU / Memory Performance Checks 115
10.4.4 Missing Scalability Experiment with Multiple Sinks . 115
10.4.5 Missing Skew Factor Experiment 115
10.4.6 Simulated Skew Factor 116
10.4.7 Circular Routing Problem 116
10.4.8 Late Nodes Not Included 117
10.4.9 Timer-Based to Cycle-Based 117
10.4.10 Total Transmission Amount 117

11 Future Work **119**
11.1 Real-World Implementation 119
11.2 Relay Units 120
11.3 Mini-Cluster Sinks 120
11.4 Auto Adjusting Clock Based on Previous Knowledge 120
11.5 Energy Budget 121
11.6 Retransmission Reduction Strategies 121
11.7 Improve Score Weights 121
11.8 Preemptive propagation 121
11.9 Improved Gradual Time Synchronization 122
11.10 Improved Path Sharing and Path Acceptance 122

12 Conclusion **123**

Bibliography **125**

List of Figures

2.1 Example deployment in Varager, extracted from the coat website [15] . . . 8

5.1 Figure showing the Node's Life cycle. This figure is shared with Erlend Karlstrøm . . . 22
5.2 Figure showing how the nodes use an Ad-Hoc hosting range to connect to neighboring nodes. . . . 23
5.3 Figure showing a network topology with multiple sink nodes. The blue nodes are sink nodes. . . . 24
5.4 Figure showing the same network topology as figure 5.3, and showing the sink paths from each node. . . . 25

6.1 Figure showing a network topology with all the paths from an ordinary node to a sink node present, each colored arrow is a different path . . . 30
6.2 Flowchart showing how the path selection is done, with the help of a path score. This figure is shared with Erlend Karlstrøm 35
6.3 Flowchart showing how the starting phase works. This figure is shared with Erlend Karlstrøm . . . 36
6.4 Flowchart showing how the receiver thread works. This figure is shared with Erlend Karlstrøm . . . 37
6.5 Flowchart showing how the introduction mode works. This figure is shared with Erlend Karlstrøm . . . 39
6.6 Figure showing how a node is able to connect two different nodes that are not connected together initially. . . . 40
6.7 Flowchart showing how the reactive gossip mode works. This figure is shared with Erlend Karlstrøm . . . 41
6.8 Flowchart showing how the operational phase works. This figure is shared with Erlend Karlstrøm . . . 43
6.9 Flowchart showing how the awake cycle works. This figure is shared with Erlend Karlstrøm . . . 45
6.10 Flowchart showing how the management mailbox is prioritized. This figure is shared with Erlend Karlstrøm . . . 47

6.11 Flowchart showing how the content propagation works. This figure is shared with Erlend Karlstrøm 49
6.12 Flowchart showing how the time synchronization works. This figure is **not** shared 52
6.13 Figure showing how the timestamps are recorded while using the time synchronization operation 53
6.14 Figure showing how the Latency-based algorithm offsets timer TB2 with the latency, to calculate the correct time. 53
6.15 Figure showing how the skewed safe zones work. 56

7.1 Figure showing an example of how the Homebase looks with the nodes have reported data to it. 62

8.1 Figure showing a network structure with two sinks. The sink nodes are marked as a blue cicle. This figure is shared with Erlend Karlstrøm 68
8.2 Figure showing a network structure with one sink node. The sink node is marked as a blue circle. This figure is shared with Erlend Karlstrøm 69
8.3 Figure showing how the optimal parent calculation experiment method works. This figure is shared with Erlend Karlstrøm 69
8.4 Figure showing how the temporary connection disruption and validating dead node sink path removal experiment method works. This figure is shared with Erlend Karlstrøm 70
8.5 Figure showing a network structure with one sink. 71
8.6 Figure showing the network structure shown in figure 8.1 with the estimated depth shown for each node. 72
8.7 Figure showing a network structure used for testing removal of latency in the Immediate adjustment algorithm 73
8.8 Graph showing the recorded skew, when the time synchronization operation is disabled. 77

9.1 Figure showing a set of network topologies topologies used in the experiments. 80
9.2 Figure showing a network structure used for testing accuracy of the time synchronization with an increasing depth of the nodes 86
9.3 Graph showing the total amount of sink path growth when scaling the network. This graph is shared with Erlend Karlstrøm 88
9.4 Graph showing the average length of the shortest and the longest path when scaling the network. This graph is shared with Erlend Karlstrøm 89

9.5 Graph showing the amount of transmissions during the starting phase when scaling the network. This graph is shared with Erlend Karlstrøm 90
9.6 Graph showing the amount of transmissions during the new sink operation while running in the operational phase when scaling the network. This graph is shared with Erlend Karlstrøm 91
9.7 Graph showing the amount of transmissions during 15 awake-cycles while running in the operational phase when scaling the network. This graph is shared with Erlend Karlstrøm . . 92
9.8 Graph showing the recovery of the skew with the immediate adjustment algorithm. 93
9.9 Graph showing the recovery of the skew with the gradual adjustment algorithm. 94
9.10 Graph showing the amount of transmissions using the Immediate algorithm while running for 15 cycles, and scaling over a set of scalability networks 95
9.11 Graph showing the amount of transmissions using the Gradual algorithm while running for 15 cycles, and scaling over a set of scalability networks 96
9.12 Graphs showing the amount of transmissions using sca 2, 4, and 6 networks from figure 9.1 100
9.13 Graphs showing the amount of transmissions using scalability 8, and 10 networks from figure 9.1 101
9.14 Graph showing how accurate the Immediate adjustment algorithm is when the depth of the nodes are growing 102
9.15 Overview of the graphs showing the recovery of the skew when the time synchronization has from 1 to 9 cycles between doing the operation 103
9.16 Graphs showing the recovery of the skew when the time synchronization has from 1 to 3 cycles between them, and the network fragments 104
9.17 Graphs showing the recovery of the skew when the time synchronization has from 4 to 6 cycles between them, and the network fragments 105
9.18 Graphs showing the recovery of the skew when the time synchronization has from 7 to 9 cycles between them, and the network fragments 106

10.1 Figure showing why the missing skew factor experiment is not needed 116

List of Tables

8.1 Table showing the expected Sink Paths from each node to the two sinks shown, network structure used is shown in figure 8.1. This table is shared with Erlend Karlstrøm 67
8.2 Table showing the expected calculated correct clock based on the latency between the nodes, network structure shown in figure 8.7 73
8.3 Table showing the optimal parent results. This table is shared with Erlend Karlstrøm 75
8.4 Table from temporary connection disruption validation showing the nodes' sink paths pre event. . This table is shared with Erlend Karlstrøm 76
8.5 Table from temporary connection disruption validation showing the nodes' sink paths post event. . This table is shared with Erlend Karlstrøm 76
8.6 Table from temporary connection disruption validation showing the nodes' sink paths post event fix. 76
8.7 Table showing the recorded calculated depth for the nodes . 77
8.8 Table showing the recorded time from the correctness check validation for the Immediate algorithm 78

List of Definitions

Observational Unit: An Observational Unit is a small battery-powered computer able to record data using sensors connected to the hardware itself, they are created to withstand the harsh environment in the arctic tundra. An Observational Unit can also be referred to as an OU or simply a node.

Cluster: A set of computers or nodes coupled together so that they can be viewed upon as a single system.

Homebase: A Homebase is a server that is considered as an external machine in comparison with the nodes in the cluster. The homebase can be either connected directly to a constant power source and an internet connection, or it can a mobile homebase such as a laptop. The Homebase is where the nodes try to propagate a data to, and where the nodes extract a clock synchronization from. All the nodes will try to gain access to the homebase server, but only those that are able to connect will be turned into a "sink"-node.

Sink node: A sink node is a normal Observational Unit that has been able to establish a connection to a homebase.

Neighbor-node: A Neighbor-node is the nodes that have a direct connection to the node currently being viewed upon.

Neighborhood: A Neighborhood is a set of interconnected nodes, where all are connected directly to each other.

Topological Cycle: A topological Cycle is a connection between separate parts of the network, that ties together two separate parts of the network.

Ad-Hoc network: A Ad-Hoc Network is a decentralized wireless network, that does not rely on a pre-existing infrastructure, as the nodes themselves create the infrastructure. Each node in the network forwards data from other nodes.

Single-Hop Routing: Single-Hop Routing is when nodes directly communicate with the external server.

Multi-Hop Routing: Multi-Hop Routing is where the node routes data via other nodes in the network before communicating with the external server.

List of Listings

6.1 Code showing how the re-distrubution of the acquired paths is done 34
6.2 Code showing how the safe synchronization calculation is done 55
6.3 Code showing how nodes check if they should synchronize or not. 56

1 Introduction

The arctic tundra is one of the ecosystems that is most affected by climate changes[28]. Therefore it is important to monitor the changes that are happening to be able to understand the consequences that they have and what it means for the wildlife in the arctic tundra.

To be able to monitor the effect of climate changes on the wildlife correctly and efficiently, it is important to monitor the wildlife with non-intrusive units that do not interfere with the wildlife itself so that ecologists can get a good understanding of the effects on the wildlife in their natural habitat.

The Fram Center created a long-term research project called The Climate-Ecological Observatory for Arctic Tundra, also known as *COAT*[16]. COAT is a Norwegian research group that aims to observe how climate change impacts the lives of wildlife in the arctic tundra and contributes to nature's perseverance.[17] To perform the research, the COAT group uses environmental sensors and wildlife cameras to monitor the wildlife in their natural habitat. A problem with monitoring in remote areas such as the arctic tundra is that there is no infrastructure to handle "off-the-shelf" monitoring devices, thus creating the need for specialized equipment.

1.1 Observational Units

The monitoring can be done by *Observational Units(OU's)*, which are small battery-powered computers able to record different environmental parameters such as temperature, oxygen, humidity, water levels, and take pictures using IR cameras. The OU's can store these parameters persistently so that they can be accessed at a later time. They also have networking capabilities that enable them to send and receive data to and from a remote server at the cost of increased energy consumption.

A Problem with the OU's is that since they are placed in very remote areas, they need to survive for a long time without any connection to the outside world. This means that if the OU's fail early in the deployment cycle, the researchers will not be able to notice it until the OU's are being fetched at the end of the deployment cycle. Thus the need for a connected network of OU's that can report status reports back to a homebase, by routing the data through other OU's, i.e., use multi-hop routing.

1.2 Problem with OU Networks

One of the biggest problems with the interconnected network of OU's(similarly to Wireless Sensor Networks) is that since the OU's are battery-powered, and the infrastructure for constant power and internet does not exist, the OU's does not have any place to synchronize their local clocks with. This means that the OU's internal clock will tend to *skew* away from each other at different speeds, dependant on the environmental effects on that node, one of which is temperature[33].

To be able to adhere to the non-intrusive requirement of the monitoring units, it is not possible to host a standard Wi-Fi solution with multiple hotspots spread across the monitoring area due to the Wi-Fi requiring antennas and constant power to function optimally. This means that all synchronization needs to happen between the nodes or to an on-site homebase. This creates the need for a system-integrated synchronization job that is built into the system.

1.3 Limitations

This thesis focuses mainly on the simulated node network itself and the clock synchronization between the nodes in the cluster network. As this is a simulator, the data that is recorded is not actual data but only simulated. To be able to validate and test the time synchronization algorithms, all the local clock skewing is also simulated based on a randomly generated factor within a boundary, which will be explained in further detail in chapter 7.

1.4 Overview of Contributions

This prototype has been worked on by two people: Erlend Karlstrøm, and Sigurd Karlstad, the author of this thesis. Both are master students at UiT The Arctic University of Norway.

Throughout this project, Erlend Karlstrøm and Sigurd Karlstad have cooperated closely. This is due to the requirements of the projects being very similar, which has lead to some parts of the code being written into a merged solution between the students. Therefore, it is necessary to include the contributions that Erlend Karlstrøm made to this project in order to give a clear description of the entire project and what contributions the separate theses have. In the following section, I shall describe what has been done by Erlend Karlstrøm, what has been done by Sigurd Karlstad, the author of this thesis, and what has been as cooperation between the students. The distribution is as follows:

1.4.1 Erlend Karlstrøm

Erlend Karlstrøm is writing a thesis about ’General Monitoring of Observational Units in the Arctic Tundra’[11] which means that he needed a way to handle the messages bundled together to reduce the amount of data that is transmitted. This is work that the author of this thesis(Sigurd Karlstad) does not take any credit for but uses to handle message transmitting:

- Bundling and Partial Bundle Policy
- Content Propagation and Debug Report Generation

1.4.2 Sigurd Karlstad

The parts that this thesis takes full credit for is:

- Time synchronization operations and all uses of it
- Node network clock stability

- A description of relevant observational node networks(WSNs), routing techniques, and clock synchronization techniques
- An evaluation of the time synchronization operation in the system
- A discussion of weaknesses and improvements of the current system
- Thoughts on future work of the system

1.4.3 Collaboration

The rest of the project has been worked on by both Sigurd Karlstad and Erlend Karlstrøm, and both should have shared credit for it. The following parts are considered a collaboration:

- Path Handling
- Path Score calculation
- Mailbox handling
- General System Architecture
- General Network Validation and Experiments
- Homebase Architecture
- All other support-code such as: Server Script, etc.

1.5 Overview of Project

- **Chapter 2** - Background: Describes the background, motivation, and aim of the project.
- **Chapter 3** - Design Guidelines: Describes the guidelines for designing the system.
- **Chapter 4** - Related Work: Describes the related work done in fields such as other observational node networks, path score calculation, and Time Synchronization.

- **Chapter 5** - Architecture: Describes the Software architecture of the system.
- **Chapter 6** - System Design: Describes how the architecture was built into the system.
- **Chapter 7** - Implementation: Describes the programming languages used, and different simulation aspects of the prototype.
- **Chapter 8** - Validation: Describes the validation setup, metrics used to validate the implemented system.
- **Chapter 9** - Experiments: Describes the experimental setup, metrics used to evaluate the implemented system and the results from the experiments.
- **Chapter 10** - Discussion: Discusses several main parts about the system, some in-depth discussion about the time synchronization, and addresses some weaknesses of the system as a whole.
- **Chapter 11** - Future Work: Describes some features that could be added to further improve the current system.
- **Chapter 12** - Conclusion: Concludes the thesis.

2 Background

Based on the previous research done by the COAT research group, which has been monitoring the effects of climate changes in areas such as Varager and the Svalbard islands. When monitoring the effects of climate changes, COAT has been using cameras, sensors, and such. In other environmental monitoring situations, *Wireless Sensor Networks(WSN)* has been used[21].

WSNs are networks that can consist of up to thousands of low-cost sensor nodes, usually meant to monitor and record environmental data. The deployments done by COAT do not have this number of nodes per deployment, as seen in figure 2.1. This example deployment, extracted from the coat website[15] has six camera nodes deployed. However, many of the deployments have more nodes deployed(up to around 30 nodes).

A common problem with WSNs is that since they usually are deployed remotely and are not connected to any constant source of power or internet, the internal clocks in them tend to skew away from the real-time, up to 10 seconds per day[30]. This creates the motivation for a standalone WSN system that can stay synchronized, even without an authoritative time in the network.

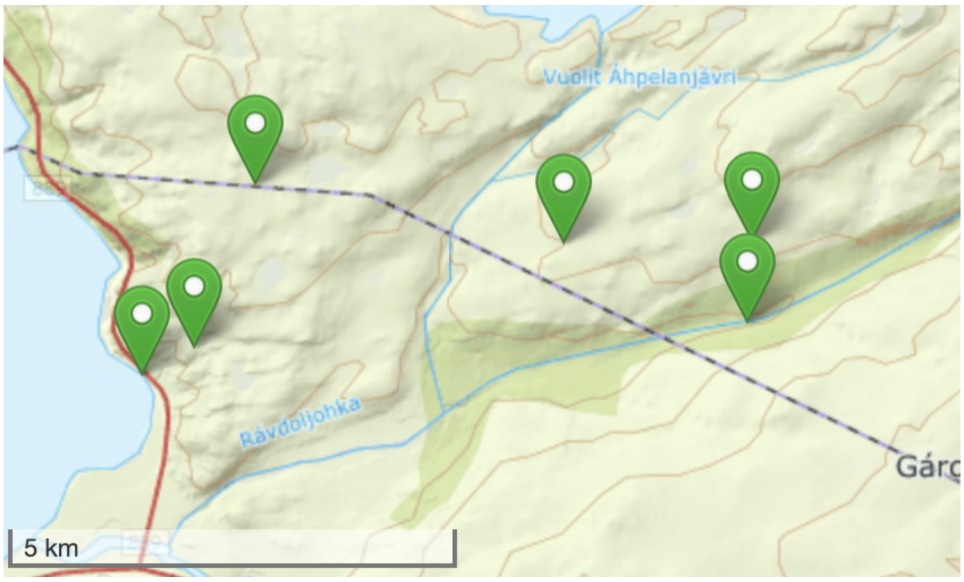


Figure 2.1: Example deployment in Varager, extracted from the coat website [15]

2.1 Motivation

The motivation behind this thesis is based upon the fact that the monitoring is performed in very remote areas(Svalbard and Varanger), and each deployment cycle can last for months[25]. This, in turn, means that the monitoring nodes are rarely visited by the ecologists – or the ones that are responsible for the nodes.

The lacking infrastructure in these remote areas, together with the consideration of the rarely visited nodes, creates a motivation for having a system that can provide remote access to the most remote monitoring nodes while remaining stable, and having a built-in local clock synchronization algorithm, which is able to remain synchronized without an internet connection.

2.2 Aim

As these monitoring devices can be spread out in uneven terrain, partly underground or fully underground, meaning that there is no guarantee that the nodes will be fully connected to the network. This means that the nodes must be able to survive without any connection and handle joining a network if a sudden node appears, tying the nodes together.

This thesis aims to create a prototype network of nodes that is stable in the sensor of synchronized sleep schedules, able to synchronize the local clocks with and without a connection to an authoritative time(homebase), and can propagate data through the network towards a homebase using multi-hop routing.

3 Design Guidelines

This thesis was inspired by the previous research done by the COAT research group, which is currently working with cameras-traps and sensors located in the arctic tundra. By following the research plan from COAT[9], which frame some design guidelines for their equipment in the field, and framing some features required for creating a stable node network in the field that enables clock synchronization internally in the network. The following sections describe these features and design guidelines.

3.1 Power Preservation and Duty Cycling

To preserve the power of the OU's so that they are able to survive in deployment for up to a year, or at least two to six months.[25] The amount of power used while deployed must remain as low as possible.

To optimize the node's battery life, they are required to periodically sleep for long periods, only waking up to record sensor readings, synchronize clocks, and propagate data towards the sink node. The sleep schedule should be synchronized with the other nodes so that they all wake up at the same time. The number of transmissions between the nodes should also remain as low as possible, and the content sent between the nodes should be optimized.

3.2 Data Transmission

To be able to propagate data between nodes so that the data recorded, even in the most distant nodes, can be sent to the researchers of the COAT group for processing, the OU's need to be able to both transmit and receive data, to and from other nodes. This connection between the nodes needs to be wireless so that the OU's can be placed wherever the researchers want to monitor.

To be able to propagate data to a sink node means that the nodes need to have an understanding of the neighboring nodes, as well as knowledge of the paths to the sink node. The knowledge shared between neighboring nodes should be shared along with normal dataflow to reduce the number of transmissions needed but increase the data size. While a wireless connection is not directly needed for the OU's, the connection between the nodes would normally be wireless as the nodes would be separated by long distances in deployment. Therefore, the time synchronization will also have to work with these conditions.

3.3 Dynamic Topology Management

To allow the OU's to be connected in a well-mannered way, it should support dynamic topology management, which means that nodes should be able to join the network without requiring a reconfiguration of the whole network. Allowing the nodes to be added into the cluster seamlessly allows for paths to gradually be shared while constructing the network. Which, in turn, reduces the overall transmissions.

3.4 Heterogeneous Communication

To allow heterogeneous nodes to communicate with each other, should the communication interface between the OU's be standardized. This will allow all nodes, even if they have different hardware, software, and sensors, to interface with each other as long as they are built using the same design model.

3.5 Non-Interference

When monitoring the wildlife in the arctic tundra, it is critical to consider the legal limitations that are required by research in the arctic tundra. Naturmangfoldsloven, §35, 2009, states "Forskriften skal [...] sikre en uforstyrret opplevelse av naturen"[20], this means that the OU's should not disturb the wildlife in their natural habitat. While this law does not directly create a guideline for the design of the software and is more relevant to the hardware, it indirectly creates criteria that the system needs to adhere to. This law enforces the use of self-contained batteries and limited internet connections, as it prohibits the use of external antennas and such, and therefore, enforces the use of multi-hop routing.

3.6 Persistent Data Storage

The OU's must be able to persistently store the data that is recorded locally, so they have a secured copy of it. While also be able to temporarily store the data that is propagated from other nodes. The size of the persistent storage must be large enough to be able to handle the recorded data that has accumulated through months of operation. While also handling temporary storing data that has been propagated through the node until delivered to the next node on the way to the homebase.

Even if the network experiences a critical failure, something that the nodes are not able to recover from, the sensor readings should still be stored persistently so that at the end of the deployment cycle, the recorded data can still be fetched directly from the OU itself. **Under no circumstances** must any of the other guidelines impact the device's ability to record and store. The device must prioritize the local persistent storage of the recorded data over anything else.

3.7 Network Clock Stability

When a homebase is in the network, all nodes in the network should try to achieve synchronization either directly or indirectly with that homebase. However, when a homebase is out of reach for the network, the nodes should still be able to remain synchronized with the rest of the network. This means discarding the "actual" clock and rather keeping the sleep schedule synchronized so that they can wake up at the same time.

4 Related Work

This chapter presents some related work in fields relevant to this thesis. Those fields include existing observational node networks, routing and communication in node networks, score-based routing, and time synchronization.

4.1 Existing Observational Node Networks

Observational node networks are not a new subject to research, and there has been a lot of research done. In the research done by Alan Mainwaring et al. in [13], they created a solution for monitoring the wildlife on The Great Duck Island. Alan et al. deployed a solution with 32 very-low-powered microcontrollers with limited battery life. The nodes use a direct connection to a gateway node, which transmits the data further on to a fixed base station. The base station had a satellite internet connection that connected to a server remotely. In contrast with the research done on The Great Duck Island, our approach does not have such low-powered nodes, and data is required to route via other nodes in the network.

In [24] Wolf-Bastian Pottner et al. presents *Data Elevators*, a delay-tolerant WSN solution where they recorded weather temperatures from the top of a 15 story building and propagated the data recorded through a node that was located in an elevator, down to the third floor. When the elevator left the third floor, the node in the elevator left that hosted network. Then, when it reached the 15th-floor, it joined the 15th-floor network. Data Elevators assumes that each node can connect to a constant power source. However, this cannot be guaranteed in our approach.

Research for monitoring volcano tomography has been presented by Dennis E. Phillips in [22]. This presents a node network that aims to study the seismic activity in volcanos, so eruptions can be caught early so that the surrounding areas can be evacuated. Sensor nodes record the seismic activity and communicate directly to an external homebase, which uses an external satellite link to send data to a remote server for processing and evaluation. In contrast with our solution, the volcano tomography uses external antennas to send the sensor readings directly to the homebase, as well as having solar panels for recharging the nodes and the homebase.

In [32] Deepak Vasisht et al. presented an IoT(Internet of Things) solution for Data-Driven Agriculture, called *Farmbeats*. In Farmbeats, they used sensors to record specific characteristics and having the sensors report their readings to an IoT Base station that hosts a WiFi network for all the sensors to connect to. This Basestation is connected to a solar panel and uses TVWS(TV White Space) to propagate data back to the Farmbeats gateway, which is connected to a constant power supply and internet. The problem with using a solution similar to Farmbeats is that on the arctic tundra, a node with a solar panel cannot be deployed, as it would go against the design guidelines and the fact that the sun is not up all year. However, using TVWS to fetch data from a node could be an interesting research topic.

4.2 Routing and Communication in Node Networks

Routing techniques in observational sensor node networks are something that has been researched extensively. Many solutions for propagating data between interconnected wireless sensor nodes have been proposed, but many of these solutions are not useful because of assumed connectivity to constant power and constant internet.

To propagate data through a Wireless Sensor Network(WSN) solutions such as *Low-Energy Adaptive Clustering Hierarchy*(*LEACH*) have been proposed. LEACH is introduced in [8] by Heinzelman et al. and uses *Cluster-heads* which are nodes that are directly connected to a fixed base-station, the nodes in the network volunteer to become a cluster-head. The nodes in LEACH organize themselves into small local clusters, where each local cluster has a cluster-head.

All the nodes in each localized cluster send their data to the cluster-head, which then propagates it to the basestation. LEACH uses a two-phased solution the first phase is to connect to the neighboring nodes, and the second is for operating normally. Using LEACH in the arctic introduces some problems such as an assumed fixed base-station that is reachable from all the local clusters, the random volunteering to become a cluster-head assumes that all nodes are able to connect to the basestation, and that all nodes can reach a cluster-head node in a localized cluster.

In [12] by Lindsey, S., and Raghavendra, C.S., they introduce *PEGASIS*, a *Power-Efficient GAthering System*, which is another solution that is an improvement on the LEACH algorithm. PEGASIS tries to improve LEACH by making the nodes in the network only communicate with close neighbors in a "chain" structure. This means routing data through neighboring nodes so that the data is closer to the cluster-head and can then be transmitted to the fixed base station. PEGASIS also improves LEACH’s cluster-head election by changing it so that the nodes in the network take turns transmitting to a fixed base station. In contrast with our approach, PEGASIS has a fixed base station, that similarly to LEACH, is assumed to be able to contact all nodes in the network so that the nodes may rotate the cluster-head role. This cannot be guaranteed with deployment in the arctic.

Due to the environmental problems that are introduced when working with monitoring wildlife in the arctic tundra, constraints such as global knowledge about the node network, fixed basestations, and a fully interconnected node network cannot be guaranteed. However, custom solutions for monitoring the arctic tundra have been proposed by other students at UiT The Arctic University of Norway. Solutions that also have considered the requirements of the COAT research group and the environment of the arctic tundra.

Camilla Stormoen proposed in her thesis [29] a solution for a WSN located in the arctic tundra. Stormoen created a Peer-To-Peer network of sensor nodes that, similarly to LEACH, uses *cluster-heads* as stationary, static nodes in localized clusters so data can be routed to a base station. However, this solution assumes that all the nodes in the network will be able to contact each other. This cannot be guaranteed in the arctic tundra. She also rotates the job of the cluster head around in the system, something that requires an election, which in turn requires extra communication.

Øystein Tveito, in his master thesis [31] created a solution that uses microcontrollers to create an event-based system that is able to record the environmental parameters that the COAT research group is interested in. Tveito deployed his microcontroller solution in the field, where he found out that he was only able to reach some of the nodes deployed. Tveitos solution uses LTE communication to communicate directly with an external server via cell towers. If a connection is established, the data will be sent to the server and deleted from the node. In contrast to our solution, Tveito's solution did not have any connections between the nodes, which means that if the nodes were deployed in areas where the LTE connection was bad, they would not be able to send data to an external server.

4.3 Score Based Routing

To reliably and efficiently propagate data through the network solutions such as score-based routing has been proposed. Score Based Reliable Routing or SBRR introduced in [34] by Yousefi, Hamed et al. have proposed a solution where each node decides on a path by rating them based on a score, which, in turn, is based on four factors: hop-count, sensor energy level, the error rate of links, and free buffer size of sensors. The solution by Yousefi et al. also contains *path-awareness(or Path Knowledge)* which is knowledge of internal parameters of every node in the path. SBRR makes sure that each node has two disjoint paths to the sink node. This means that if a node fails in one of the paths, the other pathø would most likely work. The full path-awareness, together with the full path knowledge of our system, would not be a good match for nodes in the arctic. This makes a point of only having *Neighborhood Knowledge*, to be able to perform relatively smart choices but without adding so much overhead in transmissions.

4.4 Time Synchronization

Clock synchronization between nodes in a distributed networks is a very known problem, and there has been a lot of research trying to find an optimal solution for dealing with it. How the sensors are connected and what limitations the network has frames some different strategies for dealing with the time synchronization problem.

The most known and adopted solution for synchronizing local clocks and limiting the local skew is the *Network Time Protocol(NTP)*, which is introduced by Mills, D.L. in [14]. NTP is a protocol that is created so that machines are able to fetch a correct time from an external time-server which, in turn, is connected in a set of hierarchically structured computers and timekeeping devices. These timekeeping devices are divided into sections known as *stratum*, which is directly connected to the accuracy achievable by the nodes(a higher stratum means lower accuracy). The time that is extracted can still be considered as a "correct" time. The reason that NTP is not suitable for use on the arctic tundra is that it uses the time servers to fetch the time from. This, in turn, means that to have NTP support in a system, a constant connection to the internet would be needed, or an NTP time-server must have been hosted in the local network itself(and considered correct).

The Gossiping Time Protocol(GTP) introduced by Iwanicki et al. in [10] is created for large scale decentralized systems, assumes that there is at least one node in the network that has an accurate and robust time source. In GTP, each node regularly exchanges clock settings with another node from the network, based on the quality of the clock. Introducing GTP into a WSN raises multiple issues, such as: Assuming that a node in the network has a correct and accurate clock is a problem due to making the whole network rely on a single node. GTP is created for large decentralized networks, something that is way too large scale compared to a WSN in the arctic. GTP does also not consider the energy aspect, and therefore is not suitable for deployments in the field.

I [2] Elson. Jeremy. et al. presents *Reference-Broadcast Synchronization*(or *RBS*), which is a synchronization algorithm designed for a receiver-receiver-based systems. RBS uses a broadcast message to send a message directly to the other nodes in the system, and the receiving nodes use a timestamp recorded when the message was received to calculate a reference point to use as an offset for its local time. In contrast with our solution, RBS considers the latency to be close to 0. This cannot be guaranteed when the connection between OU's is wireless and unstable.

Timing-Sync Protocol for Sensor Networks also referred to as *TPSN* is a synchronization algorithm introduced in [4] by Ganeriwal, Saurabh et al. and is an improvement upon the RBS algorithm and is designed for a sender-receiver approach in low-cost sensor networks. With the use of the sender-receiver approach – something that this system is based upon. The TPSN algorithm is able to record four different timestamps, which are used to calculate the offset of the node. These timestamps are used to create the offset between the nodes, which is used to adjust the time with. However, TPSN does not support multi-hop communication. This, in turn, means that TPSN requires that all nodes can communicate directly with an authoritative time-server.

An overview by Ill-Keun Rhee et al. introduces in [26] multiple existing techniques of handling the clock synchronization in WSNs. Techniques such as Reference Broadcast Synchronization (RBS), Timing-Sync Protocol for Sensor Networks (TPSN), Delay Measurement Time Synchronization for Wireless Sensor Networks (DMTS), Flooding Time Synchronization Protocol (FTSP), Probabilistic clock synchronization service in sensor networks, Time Diffusion Synchronization Protocol (TDP). Ill-Keun Rhee et al. also states several advantages and disadvantages with these techniques, which has helped a lot when deciding on how to handle the time synchronization in this system.

5 Architecture

This chapter describes the main building blocks of the system, which are framed based on the background of the project(chapter 2), and the design guidelines(chapter 3).

5.1 Architechtural Roles

The system frames many different roles which, are all connected in some way. These roles are: *Sink-node, Homebase, and node*. These roles will consistently be used throughout this thesis:

- A *node*(also referred to as *OU*) is one of the monitoring nodes in the network, focuses on recording data and propagating that data towards a sink node.

- A *Sink-node* is a node that has established a connection to a homebase. This enables the node to propagate data outside of the network.

- A *Homebase* is an external server, which is connected to the sink node. This external server receives the data directly from the sink nodes and is considered to be the authoritative time in the network.

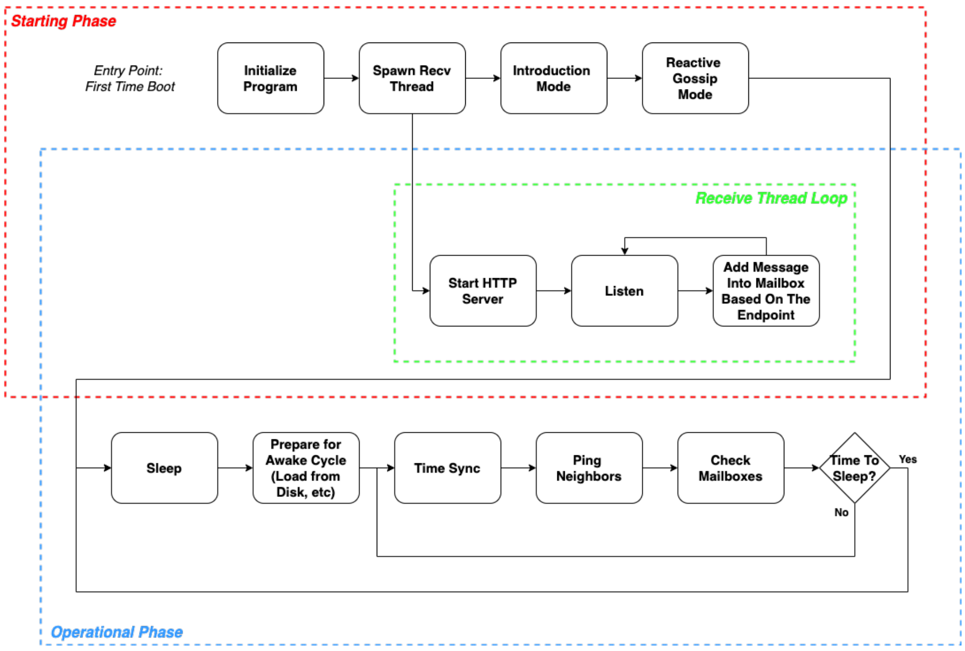


Figure 5.1: Figure showing the Node's Life cycle. This figure is shared with Erlend Karlstrøm

5.2 Node Life-Cycle

The system is split up into two phases, which are *the starting phase*, and *the operational phase*. The starting phase handles initializing variables and jobs, such as creating a separate thread for handling the receiving of messages, introducing itself to the neighboring nodes, and waiting for other nodes to join the network.

The operational phase assumes that the network is already set up after being in the starting phase. However, if there are no other nodes in the known neighborhood, it will still operate normally. After entering the operational phase, the system starts to perform the duty cycling – The periodical sleep schedule –. When the system is awake, it synchronizes with the neighboring nodes based on the known topology of the network, receives and sends data based on the content of the mailboxes, and reads/writes data to persistent storage. While the nodes are awake, they also propagate data towards other nodes based on a score calculated.

5.2.1 Duty Cycling

When nodes are running in the operational phase, as shown in the node life cycle figure 5.1, they sleep for most of the time. The nodes *duty cycle* to preserve power, only waking up to record environmental data, propagate data, and synchronize their local clocks with the other nodes. As the nodes in this thesis are all simulated processes, the system simulates the duty cycling by sleeping for a set period(explained in chapter 7). This would, however, be replaced with a separate microcontroller in a real-world implementation.

5.3 Neighbor Discovery

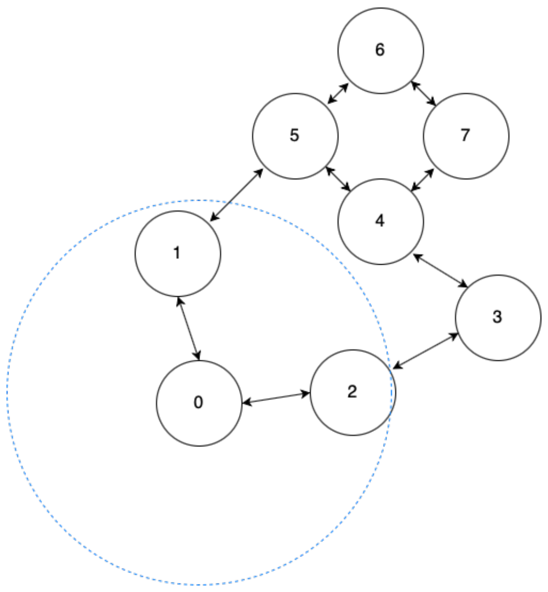


Figure 5.2: Figure showing how the nodes use an Ad-Hoc hosting range to connect to neighboring nodes.

When nodes are initialized they start by broadcasting an introduction to any neighboring nodes, within their hosted ad-hoc network distance(simulated distance, see chapter 7 for more details) – as shown in figure 5.2–. If there are any neighboring nodes active, they will introduce themselves back and, both nodes will gain a new neighbor. When the nodes are introduced, they share some local knowledge about each other, such as sharing paths to sink nodes(if any in the network), calculating a score so that a more optimal path can be chosen, and sharing local states. All the knowledge about other nodes is stored locally at each node and is continually updated along with the normal transmissions and communications between the nodes in the form of metadata.

As we see in figure 5.2, which is an example of the connections in the simulated ad-hoc network, the blue stippled line around node zero is an example of the ad-hoc hosting distance. This allows node one and node two to be tied together through node zero, which, in turn, creates another route for the propagation of data.

5.4 Topology

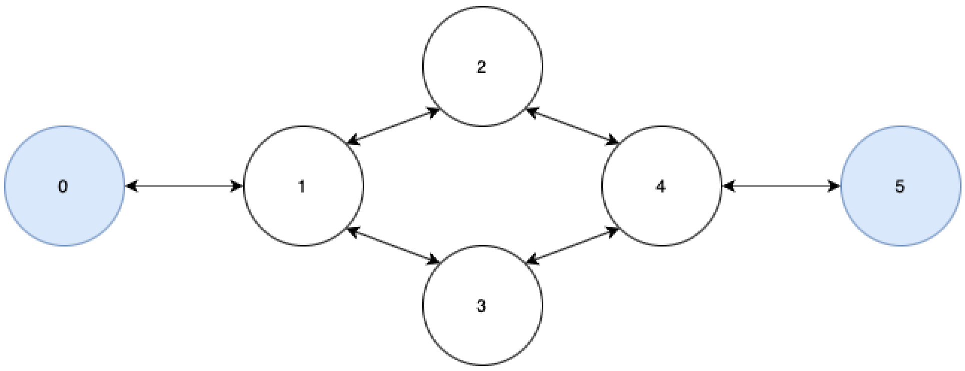


Figure 5.3: Figure showing a network topology with multiple sink nodes. The blue nodes are sink nodes.

After all the nodes have introduced themselves to each other, they form an interconnected mesh network of standalone nodes. The system is created so that each node has the ability to become a sink node, as it only needs a connection to the homebase to become a sink. This allows for multiple sink nodes in the network at once. This, in turn, allows for the nodes to propagate data in whichever direction is considered the best, based on the locally known knowledge about the node's neighbors and their score calculations.

By allowing the use of multiple sink nodes and allowing all the nodes to become sink nodes, the system also removes any requirement for consensus and elections. All the nodes decide locally and individually which node it should propagate the data to.

In the example network shown in figure 5.3, there are shown two sink nodes in the same network, node 0 and node 5. The same network is shown in figure 5.4, except this figure shows some of the local knowledge about the network. Here we can see that although the nodes don't know about the whole network, it still knows the different paths to the network sink nodes.

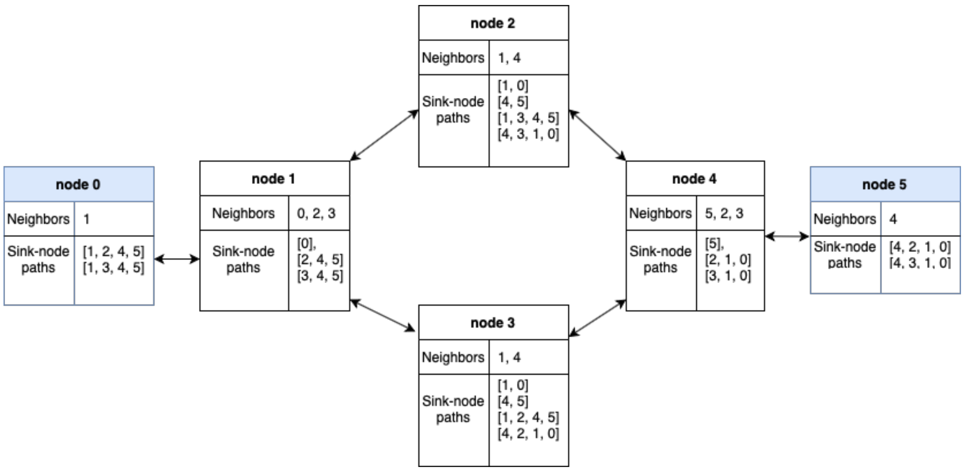

node 0
Neighbors 1
Sink-node paths [1, 2, 4, 5] [1, 3, 4, 5]
node 1
Neighbors 0, 2, 3
Sink-node paths [0], [2, 4, 5] [3, 4, 5]
node 2
Neighbors 1, 4
Sink-node paths [1, 0] [4, 5] [1, 3, 4, 5] [4, 3, 1, 0]
node 3
Neighbors 1, 4
Sink-node paths [1, 0] [4, 5] [1, 2, 4, 5] [4, 2, 1, 0]
node 4
Neighbors 5, 2, 3
Sink-node paths [5], [2, 1, 0] [3, 1, 0]
node 5
Neighbors 4
Sink-node paths [4, 2, 1, 0] [4, 3, 1, 0]

Figure 5.4: Figure showing the same network topology as figure 5.3, and showing the sink paths from each node.

5.5 Score Based Routing

To have the nodes decide on a path to propagate the data through, or decide what node to synchronize the local clock with, it uses *score-based routing* calculation based on local knowledge of the neighboring nodes. The score-based path choosing is done for load balancing purposes in the network so that the same path is not always chosen, and the battery usage of the nodes is spread across the neighboring nodes. The score-based path calculations are based on three different metrics, two directly connected to information about the neighboring node and one metric for the whole path. These metrics are Round Trip Time (RTT), Path length, and neighbor battery percentage.

The RTT metric calculates the time it takes to send and receive a message to and from the neighboring node, which is relevant and useful to limit the number of retransmissions needed when propagating data, and useful for reducing the difference in time when extracting the clock for time synchronization. The Neighbor battery metric is used for load balancing the transmissions. The path length metric is used to evaluate which path is optimal because the longer the path is, the more battery would be used in total – because the overall transmission number would increase.

5.6 Event-Driven

The nodes are *event-driven*, meaning that, to make the nodes perform an operation, an event needs to trigger it. An event can be something like a timer being reached or messages received from other nodes. As explained earlier, the nodes simulate the duty cycling by sleeping. This, in turn, means that the node is waiting for a wake-up timer-event to occur. If the wake-up timer event occurs, it starts up as it should, performing the normal operations. The simulation simulates that an IR-Camera event happens while it is sleeping so that the node simulates taking a photo and storing it persistently.

5.7 Message-Oriented Priority Mailboxes

The nodes consist of 3 mailboxes which are prioritized in a specific order based on the content of the mailboxes. The mailboxes are divided into Management, Inbox, and Outbox. The Management mailbox is used to handle the network stability and the network knowledge messages and is mostly used for intra-node communication explained in the next section.

The inbox is the mailbox that the receiver thread writes all the content propagation data to. The operational phase reads from this mailbox and does not write to the mailbox except before and after waking up(when the receiver thread is disabled), shown in Figure 5.1. The outbox is the mailbox containing the data that should be propagated to another node or the homebase, and is very similar to the inbox. The management mailbox is, as the name states, created for containing the network maintenance messages.

The priority of mailboxes is as follows:

1. Management
2. Inbox
3. Outbox

Prioritizing the different mailboxes allows the management mailbox to focused on before processing any other content messages that are to be sent to other nodes. This allows for network stability to remain one of the most important aspects of communication. The inbox is the second-most important mailbox due to it being important to temporarily store the messages that were received if it was not able to propagate all the messages further before the next sleep schedule is reached. The outbox is the least important mailbox, as it is a collection of messages that either should be sent or stored until the next cycle.

5.8 Intra-Node Communication

To maintain the network stability and be able to get updated information about the other reachable nodes, communication is needed between the nodes. Communication that would share information such as the knowledge about new nodes, dead nodes, new paths, new sink nodes, loss of sink nodes, and many other messages can be shared between the nodes.

While many different intra-node messages exist, most of them will only be used on special occasions. Some messages, such as the dead-node message is something that is only used when nodes unexpectedly die or when a node enters a Low-Power state.

5.9 Content Propagation

When a sink exists in the network and the nodes in the network propagate data through other nodes based on a score as explained in section 5.5, except if the node is a sink as it will send all the messages to the homebase. When the nodes send messages to other nodes, they use a bundling algorithm to merge the messages together, this algorithm is created by Erlend Karlstrøm and is a part of his master thesis[11], so I will not go into detail about that algorithm.

5.10 Adaptive Time Synchronization

The nodes have *adaptive time synchronization*, which means that if a sink node with a homebase connection exists in the network, it will use an *immediate adjustment algorithm* to synchronize the local clock towards that sink node. This will make the network follow a "real"-clock. When the network does not contain any sink nodes, it uses a *gradual adjustment algorithm*, which means that the nodes focus on remaining synchronized with the other nodes and disregards the "real"-clock. The nodes use the gradual adjustment algorithm because it allows a single node to skew its local clock based on the clock of other neighboring nodes without requiring a consensus algorithm to decide on a temporary leader node.

5.11 Safezones In Time Synchronization

When the network contains at least one sink node, a calculation to estimate a "depth" in the network can be done. If the node is able to calculate an estimated depth based on the paths in the network, it uses this depth to calculate *safe zones* when it should synchronize with its calculated parent. Each node gets a safe synchronization zone that is skewed with a constant based on the depth in the network. Using the safe zones allows the "parent" nodes to fetch the accurate time from its "parent" node, which, in turn, leads to a more accurate clock in the nodes in the network and reduced failed time synchronization requests.

5.12 Forced Time Synchronization

When the presence of a sink node is in the network, the nodes in the network are able to estimate what "children" they have. By using the estimated children, the nodes are able to request that child nodes synchronize with their parents if the synchronization has not happened in a while. This will enforce that the network remains stable, even if the clocks internally are not correct.

/6 System Design

This chapter describes how the building blocks of the system, which were explained in chapter 5, was used to create the design of the monitoring system framed by the background and design guidelines.

The system design is based on two main phases as stated in the Architecture section 5.2. The two phases are: The *starting phase*, and the *operational phase*. In all the flowcharts located in this chapter, the ellipses that are inside the flowcharts are links to another flowchart. I.e., An ellipse containing "To Time Synchronization" is a link to the Time Synchronization Flowchart, this is done because the flowcharts are together quite large, and some parts and functionality are shared between the different parts of the system.

All the flowcharts(except the Time Synchronization Flowchart, figure 6.12) in the following sections are shared between Erlend Karlstrøm, and the author of this thesis, as they describe the fundamental architecture and design of the system. The credit of the flowcharts should therefore be split, where both deserve 50% credit for it.

6.1 General Functionality

The following sections explain some general functionality and parts of the system that is relevant to both the starting phase and the operational phase

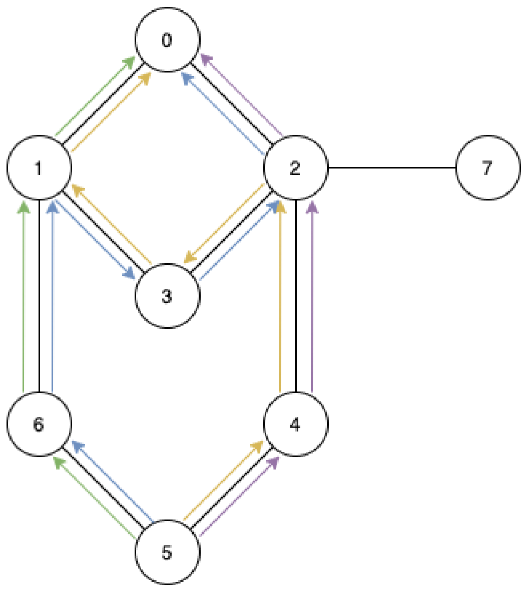


Figure 6.1: Figure showing a network topology with all the paths from an ordinary node to a sink node present, each colored arrow is a different path

6.1.1 Operational Configuration

In the starting phase, the nodes read some initial values that are set in a configuration file. This file handles the environmental running values, which the system is based upon. Since the system is currently running on a *timer-based* configuration, it means that it uses durations of time to decide on what to do rather than timestamps. The system extracts these global configurations so the system can be run and so that the environmental running options can be decided before starting a network. Some of the things that are located in the configuration file are:

- Weight configurations for score-based path calculation.
- Timer constants, such as quarantine time, thresholds, etc.
- Time synchronization settings
- Bundling settings (I will not go into detail due to this being a part of Erlend Karlstrøms thesis [11])

6.1.2 Path Structure

After a sink has entered the network, either in the starting phase or a node in the operational phase has gained sink node privileges, paths will propagate throughout the network. As the paths are created, shared, and stored in the different nodes, multiple paths routing through the same nodes may exist, although there cannot exist more than one path that is the same. As we see in figure 6.1, the routes of node 5 are all shown as arrows leading to the sink node(node 0). By allowing all the available paths to be shared between all the nodes in the network, redundancy can be assumed. However, this comes at the cost of increased transmissions and reduced network longevity. This is experimented on in chapter 9

In figure6.1 we can also see that node 0, 1, 2, and 3 form a cycle. This cycle is something we refer to as a *topological cycle*, which in turn means that it ties together two separate parts of the network, so paths are shared between the nodes, enabling routing that way.

6.1.3 Neighbor Metadata Knowledge

To be able to update the local knowledge about neighboring nodes, data has to be shared between them. All communication between nodes will contain metadata about the nodes, which allows the amount of transmissions needed to keep updated knowledge to be reduced, but increases the size of a single transmission. Some of the metadata that is shared is:

- An updated battery status
- When the node was last synchronized with
- Storing a timestamp of last communication(used to check for dead nodes)
- Which node is the preferred parent node

By storing the metadata about the other nodes locally, the nodes can perform the necessary calculations to try to keep the network stable and operational. This also allows the nodes to re-calculate the score based on the newly received metrics.

6.1.4 Score Calculation

As stated in chapter 5, section 5.2 the nodes uses a calculated score to select a path. This score consists of the normalized metrics: a *Battery percent* metric, a *Round Trip Time(RTT)* metric, and a *Path length* metric. Where two of the metrics are based on the neighboring node(the first step in a path), and the third metric is based on the whole path length.

The battery percentage knowledge metric is acquired through the use of neighbor metadata knowledge sharing, as explained in the previous section. The RTT is calculated by measuring the time it takes to send a message to the node and receiving a response. The Path score metric is found by comparing the path length to the longest known path. The metrics are normalized by using the following formulas:

$$BatteryScore = \frac{100.0 - batteryPercentage}{100.0}$$

$$RTTScore = \frac{LastKnownRTTinSeconds}{LongestRTTRecorded}$$

$$PathScore = \frac{PathLength}{LongestKnownPath}$$

By using these three normalized metrics a total path score can be calculated, this is done by following this formula:

$$TotalPathScore = \frac{A * BatteryScore + B * RTTScore + C * PathScore}{3}$$

When the total path score is calculated per viable path in the node, the system picks the path that has the lowest score, as this is deemed most optimal. The *A, B, C* that are in the TotalPathScore formula are weights that are set in the global configurations file. This is done so the calculation of the path can be weighted in a specific direction so that one of the metrics can be preferred. By default, the weights are set to A = 4, B = 1, 2 = 1. The battery weight is set to 4 so that the battery life of the nodes would be preferred over the other metrics.

By calculating a score per path, the system can select a specific path for content to be propagated through. This also means that the system might route through other nodes instead of routing directly to a sink node if the connection to the sink is bad, as it might reduce the retransmission amount. This score-based path choosing is a simple approach to score-based path selection, and it is far from optimal. To read more about the score-based pathing see chapter 11, section 11.7.

6.1.5 Path Handling

The following sections explain how paths are generated, accepted, distributed while in the starting phase, and re-distributed after receiving a message about new paths.

6.1.5.1 Path Generation

When a node gains a connection to the homebase it starts to consider itself as a sink node. This means that the node will start to propagate messages to the homebase directly and not via other nodes. Since the connection to the homebase is established, the other nodes in the network need to gain knowledge of the newly available paths so they can route to this sink(possibly load balancing other sink nodes). The node starts by sharing a path to all the neighboring nodes by sending them a new path message only containing itself. The message is handled in the same way as explained in section 6.1.5.4. This ensures full sink-path knowledge in the network, and in turn, load balances the other sinks in the network, if there are any.

6.1.5.2 Path Acceptance

When a node receives paths from other nodes, it must accept them before further propagating them to other nodes. When the node receives them, it checks if its name is contained within those paths. If the node is in one of the paths, it rejects that path. However, if it is not in those paths, it propagates them further.

6.1.5.3 Path Distribution

When a node connects two different parts of an existing network together while running in the starting phase, forming a topological cycle, the node then starts to share the paths with the other nodes it recently connected to. The newly discovered paths, which it received from the other nodes in the network, are propagated to each other, with the only condition being that the node does not send a path back to where it was received from. How a node re-distributes that path is explained in the following section.

6.1.5.4 Path Redistribution

Listing 6.1: Code showing how the re-distrubution of the acquired paths is done

```
func handleNewPathMsg(msg MBItem) {
  pathsReceived := unpackData(string(msg.content))
  var newPaths [][]string

  for _, path := range pathsReceived {
    // Check if the path is relevant for this node or if it already has
        the knowledge about that mode
    if (!listExists(node.sinkPaths, path)) && (!strExists(path, node.name)
        ) {
      // Add to our path and the newPaths
      node.sinkPaths = append(node.sinkPaths, path)
      pathThroughUs := append([]string{node.name}, path...)
      newPaths = append(newPaths, pathThroughUs)
    }
  }

  // Send the new path list that was received to the other neighbors by
      filtering out which neighbor should have what path
  for _, nei := range node.neighbors {
    var pathsToForward [][]string

    for _, path := range newPaths {
      if !strExists(path, nei.Name) {
        pathsToForward = append(pathsToForward, path)
      }
    }
    if len(pathsToForward) == 0 {
      continue
    }

    // Now we know that there is something that needs to be sent
    err := sendToRecipients(pathsToForward, "new_path", []string{nei.Name
        })
    if err != nil {
      dmError("Failed sending new paths to neigbor with error: %v", err)
      continue
    }
  }
}
```

When a node receives a new path message from the neighboring nodes, it parses the message using the code shown in listing 6.1. The first thing the node does is check if the sink path(s) are worthy of being accepted, which is done by checking if the node is in the path or if it has the path already. If none of those requirements are true, it is added to the sink-path list. Before re-distributing the newly acquired paths, the node makes sure to only propagate the paths to the node that does not exist in the path already. This will ensure full coverage of sink paths.

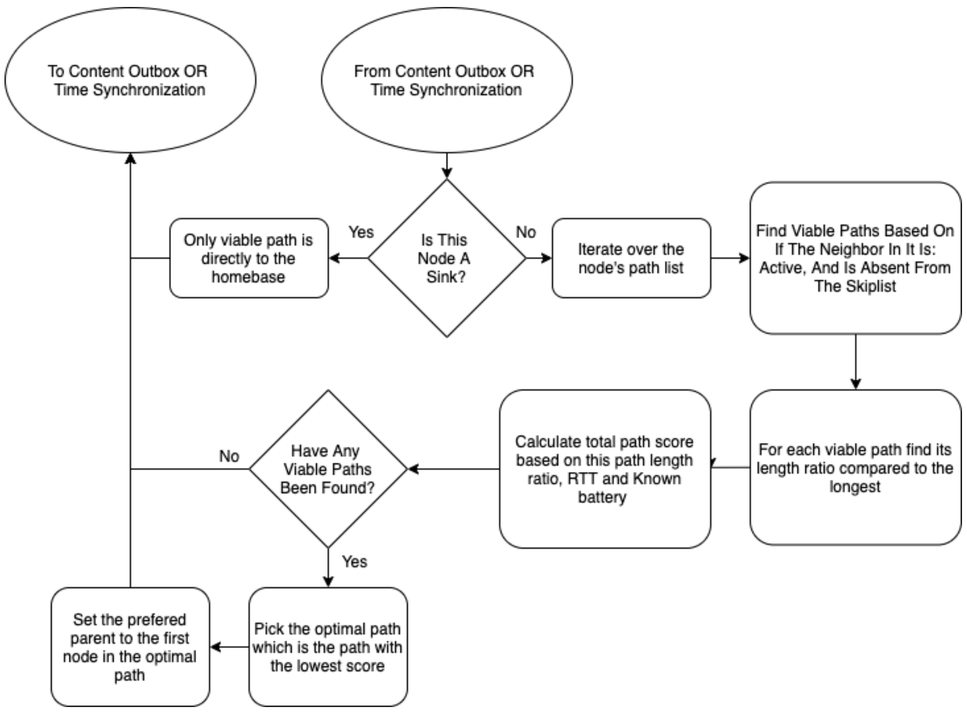

To Content Outbox OR Time Synchronization
From Content Outbox OR Time Synchronization
Is This Node A Sink?
Yes
No
Only viable path is directly to the homebase
Iterate over the node's path list
Find Viable Paths Based On If The Neighbor In It Is: Active, And Is Absent From The Skiplist
For each viable path find its length ratio compared to the longest
Calculate total path score based on this path length ratio, RTT and Known battery
Have Any Viable Paths Been Found?
No
Yes
Pick the optimal path which is the path with the lowest score
Set the prefered parent to the first node in the optimal path

Figure 6.2: Flowchart showing how the path selection is done, with the help of a path score. This figure is shared with Erlend Karlstrøm

6.1.5.5 Path Selection

Assuming there is a sink in the network(if not, the content will not be propagated). Following the flowchart shown in figure 6.2. When deciding on which path the node should propagate the content, the node first considers if itself is a sink node because if it is, all the content messages would be directly sent to the homebase. However, if the node is not a sink, it will need to path via other nodes.

Finding a path to route via is done using the score of the different paths, as explained in section 6.1.4. Here the node calculates a score for each viable path and selects the path with the lowest score. The content is then sent to the first step in the path. While selecting the path, the node also sets the preferred parent, which is done to stabilize the forced time synchronization explained in section 6.3.2.1. Read about improving the pathing in chapter 11.

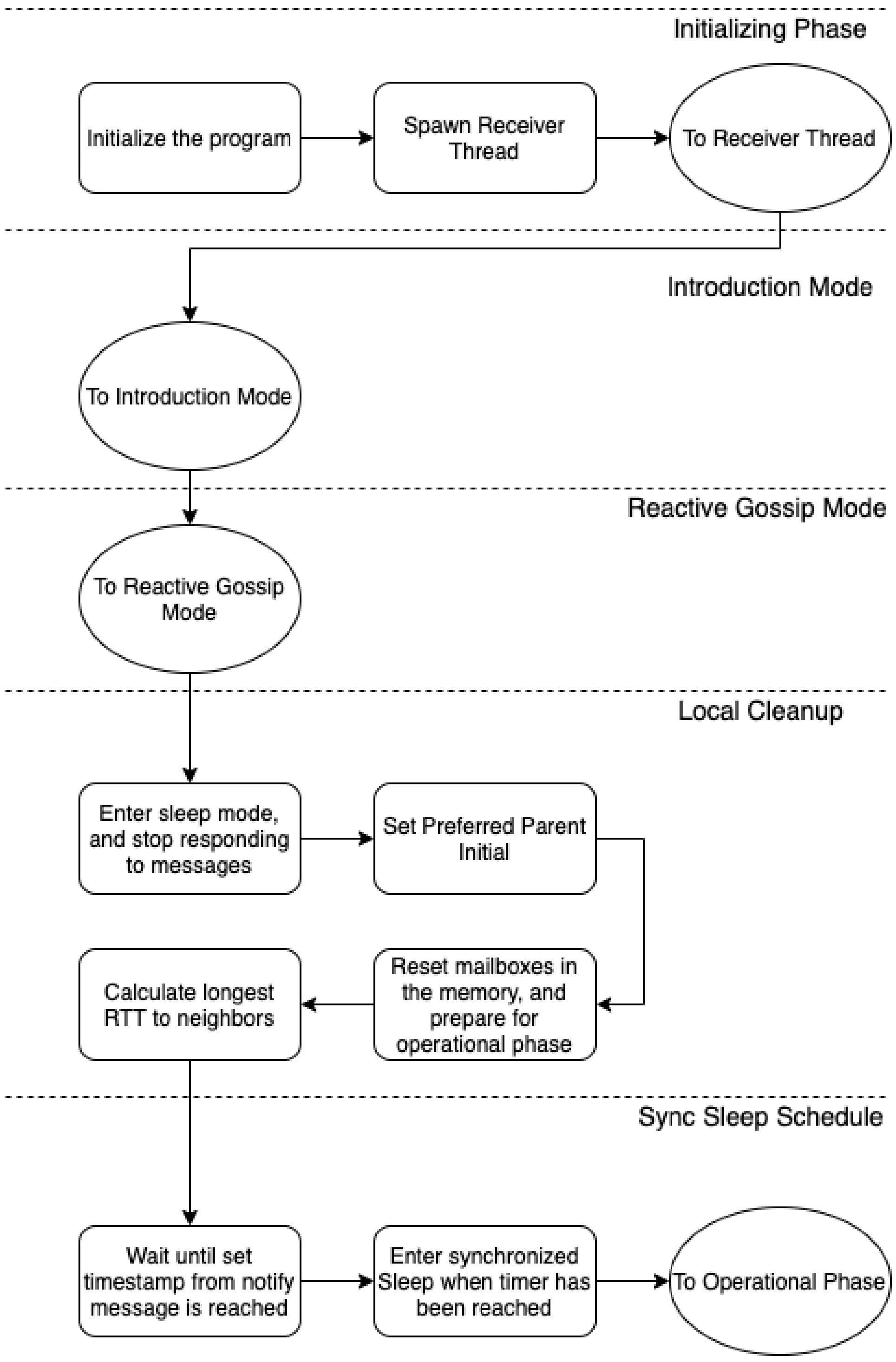

Initializing Phase
Initialize the program
Spawn Receiver Thread
To Receiver Thread
Introduction Mode
To Introduction Mode
Reactive Gossip Mode
To Reactive Gossip Mode
Local Cleanup
Enter sleep mode, and stop responding to messages
Set Preferred Parent Initial
Reset mailboxes in the memory, and prepare for operational phase
Calculate longest RTT to neighbors
Sync Sleep Schedule
Wait until set timestamp from notify message is reached
Enter synchronized Sleep when timer has been reached
To Operational Phase

Figure 6.3: Flowchart showing how the starting phase works. This figure is shared with Erlend Karlstrøm

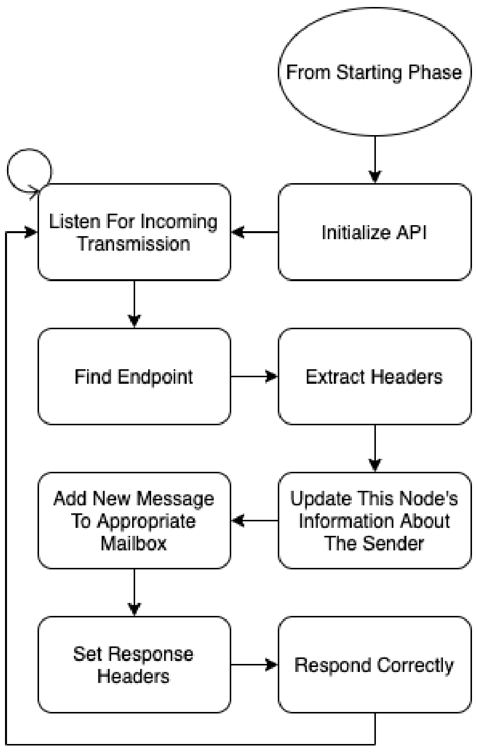

From Starting Phase
Initialize API
Listen For Incoming Transmission
Find Endpoint
Extract Headers
Update This Node's Information About The Sender
Add New Message To Appropriate Mailbox
Set Response Headers
Respond Correctly

Figure 6.4: Flowchart showing how the receiver thread works. This figure is shared with Erlend Karlstrøm

6.2 Starting Phase

The starting phase(shown in the flowchart in figure 6.3) has the job of initializing the node by extracting the initial configuration, spawning a receiver thread, trying to connect to neighbors, and synchronizing the sleep schedule for the nodes in the network. The nodes perform these tasks by using two different modes, the introduction mode, and the reactive gossip mode.

6.2.1 Receiver Thread

At the start of the starting phase, the system initializes a receiver thread(seen in figure 6.4). This receiver thread is an API that listens for incoming transmissions and handles the requests accordingly. The API extracts the message content supplied from the neighboring nodes and places the messages into the correct mailbox, which is based on the message type(content propagation in the inbox and network management messages in the management mailbox). While also extracting the metadata and updating the local knowledge about the other neighboring nodes. When responding to the requests, it also sends metadata to make sure the other nodes are updated, as explained in section 6.1.3.

6.2.2 Introduction Mode

After extracting the initial configuration and spawning the receiver thread, the node enters the introduction mode, as seen in figure 6.5, in this mode, the node broadcasts an introduction message to a range of addresses, where it tries to introduce itself to as many nodes as possible. If the node can reach any other nodes by sending the introduction message, it gets a response, and the other node is added to a neighbor list. This is a simulated event, which will be outlined in detail in chapter 7.

When the node is sending the introduction message, it first checks if it has a connection to the homebase to check if it is a sink node. If a homebase connection is established, it sends the introduction message with a body containing the path to the sink – Only containing the address to itself in this case.

If a node introduces itself to a network that already contains a sink node, it will receive paths distributed from the other nodes. As this is the first connection to the other nodes, all the paths can be accepted. After a node has introduced itself to the neighboring nodes, it tries to distribute paths between nodes that might not have been connected, assuming that there is at least one sink in the existing network. If there are no sink nodes in the system, this step will be skipped completely.

Figure 6.6 is an example of how a node connects two different nodes together. At the start, the two nodes are not connected. The blue circle is a sink node, and the white node is not. When a node introduces itself to both of the nodes and receives a path from one of them and not from the other, it distributes the path it received from one node to the other. This connects all three nodes together so that the edge node can propagate data through the newly joined(green) node to the sink node.

Introducing itself this way and gradually propagating the paths as more nodes join the network allows the nodes to reduce the number of transmissions needed when introducing multi-hop paths. However, when introducing new topological cycles, the network's total transmission amount would increase, as all paths are needed to be shared through the same newly joined node.

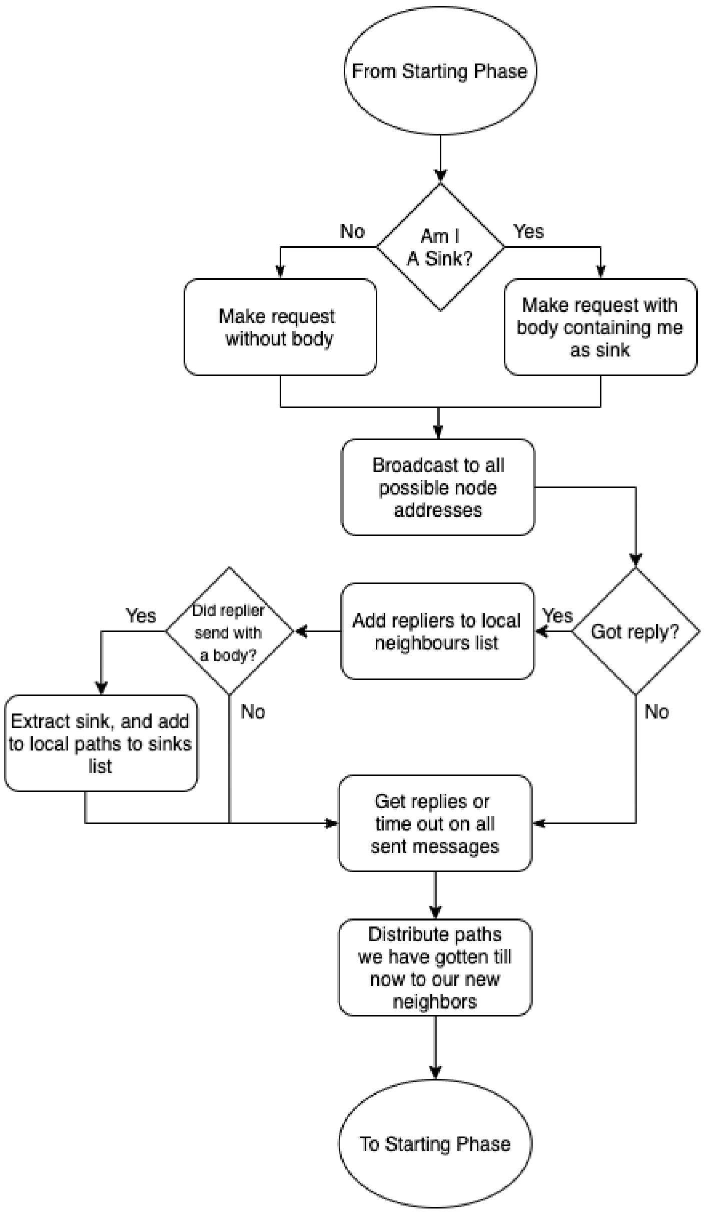

From Starting Phase
Am I A Sink?
No
Yes
Make request without body
Make request with body containing me as sink
Broadcast to all possible node addresses
Got reply?
Yes
No
Add repliers to local neighbours list
Did replier send with a body?
Yes
No
Extract sink, and add to local paths to sinks list
Get replies or time out on all sent messages
Distribute paths we have gotten till now to our new neighbors
To Starting Phase

Figure 6.5: Flowchart showing how the introduction mode works. This figure is shared with Erlend Karlstrøm

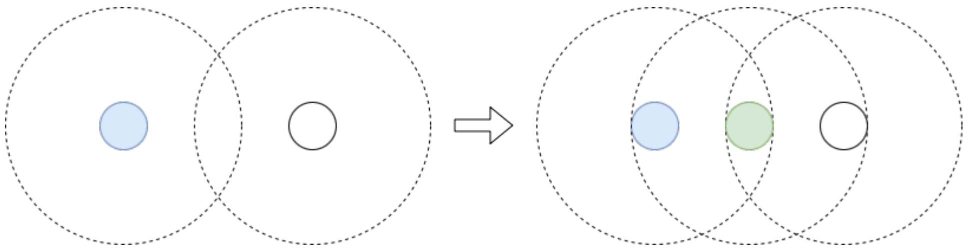


Figure 6.6: Figure showing how a node is able to connect two different nodes that are not connected together initially.

6.2.3 Reactive Gossip Mode

After the node has introduced itself to the network, it enters the reactive gossip mode. By Following the flowchart shown in figure **??**, we can see that the nodes continuously check for introduction messages from new nodes, new paths from known nodes, or a notification message from known nodes, and only reacts when actively sent something to.

When a new node introduces itself to the node running in the reactive gossip mode, it will store the information about that new node in a local knowledge-store, and try to extract a body from that message. If the message has a body containing a sink path, then it means that a sink node has joined the network. The path to the sink will be extracted from the message and directly added to the known sink-path list. After it has been added to the sink-path list, it is propagated to the other neighboring nodes, in the same way as explained in section 6.1.5.4. Allowing all paths to enter the sink-path list allows for path redundancy and will not create duplicate paths, as it is a new node.

When the node receives a new path message, it checks if it should accept the path or reject it, this happens the same way as specified in section 6.1.5.2, where it checks if itself is in the path and propagates it if it is not.

When the first node in the network reaches the timer-limit for the starting phase, which is a configurable timer set in the configuration file, it starts flooding the network with a notification message, which is a message containing a set timestamp for when the nodes should enter the *operational phase*. As soon as this message is received in a node, it extracts the timestamp and spreads the message further on. The node then proceeds to leave the reactive gossip mode while shutting down all communication.

The reactive gossip mode allows the nodes to react to other nodes joining the network, allowing them to propagate newly acquired paths and synchronize their sleep schedules.

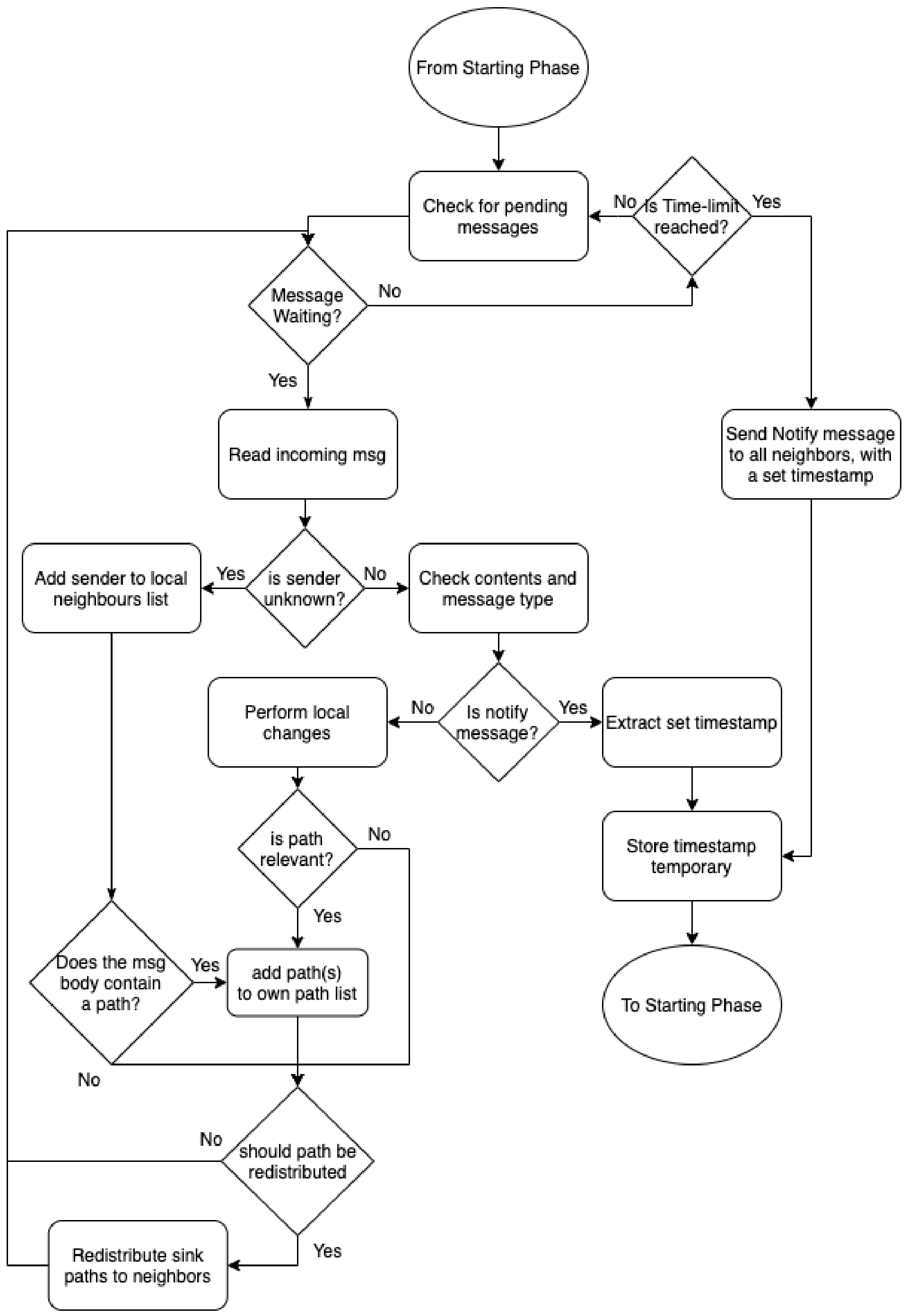

From Starting Phase
Check for pending messages
No
Is Time-limit reached?
Yes
Message Waiting?
No
Yes
Read incoming msg
Send Notify message to all neighbors, with a set timestamp
Add sender to local neighbours list
Yes
is sender unknown?
No
Check contents and message type
Perform local changes
No
Is notify message?
Yes
Extract set timestamp
is path relevant?
No
Store timestamp temporary
Yes
Does the msg body contain a path?
Yes
add path(s) to own path list
To Starting Phase
No
No
should path be redistributed
Yes
Redistribute sink paths to neighbors

Figure 6.7: Flowchart showing how the reactive gossip mode works. This figure is shared with Erlend Karlstrøm

6.2.4 Local Cleanup

After receiving a notification message, flooding the network with the same message to the other nodes, and stopping all communication, the node enters the local cleanup. This mode prepares the node for the operational phase by finding a preferred parent if the network contains a sink node(How this is done will be explained in the operational phase section 6.3). The nodes also remove all temporarily stored messages, so it is prepared to start fresh in the operational phase.

6.2.5 Synchronized Sleep Schedule

After preparing to start the operational phase, the nodes will wait until the set timestamp that was received from the notification message has been reached. This enables all the nodes to enter the operational phase at the same time. This step would not be needed in a "real-world" implementation, as the system would need to go from a timer-based system to a clock-based system, and all the nodes could just skip straight to sleeping.

6.3 Operational Phase

The operational phase(shown in the flowchart in figure 6.8) has the job of maintaining the network's health, recording environmental data(simulated, will be explained in chapter 7), propagating data, and duty cycling(also simulated, see chapter 7). The operational phase consists of 4 modes which are: Post-sleep preparation, Awake cycle, pre-sleep operation, and Low-Power mode.

Before entering the post-sleep preparation mode, the node sleeps for a while, simulating how *duty cycling* would work in a system with designated hardware. This mode also simulates recording environmental data such as oxygen, and humidity, based on simulated equipment, while also simulating taking pictures. All of which are explained in chapter 7.

6.3.1 Post-Sleep Preparation

Following the flowchart in figure 6.8 we can see that after waking up and generating a status report, the node starts to prepare itself for the awake cycle. It does this by calculating a partial send policy which is related to the bundling aspect of the system and is created by Erlend Karlstrøm. Therefore, I will not go into detail about the algorithm. However, I will state that it is used to calculate

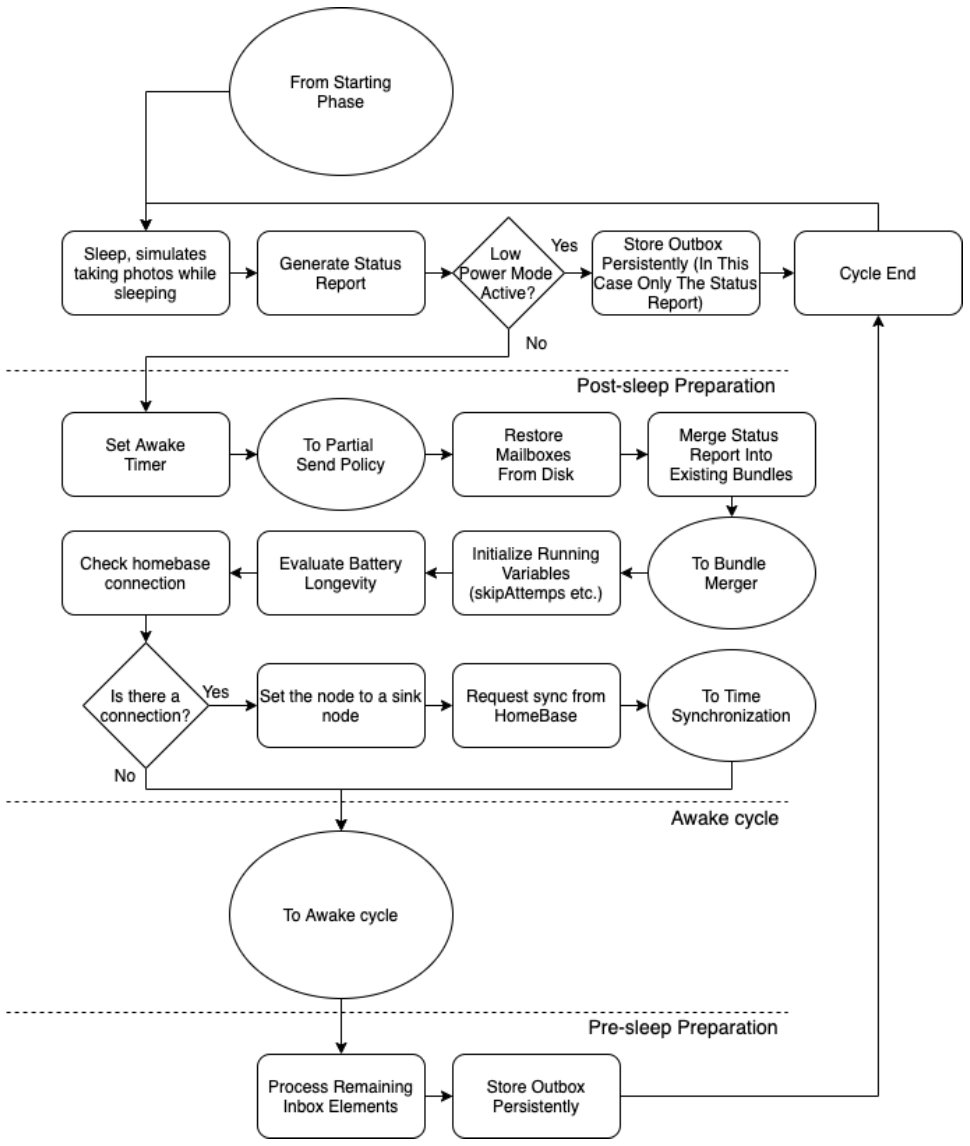

From Starting Phase
Sleep, simulates taking photos while sleeping
Generate Status Report
Low Power Mode Active?
Yes
Store Outbox Persistently (In This Case Only The Status Report)
Cycle End
No
Post-sleep Preparation
Set Awake Timer
To Partial Send Policy
Restore Mailboxes From Disk
Merge Status Report Into Existing Bundles
To Bundle Merger
Initialize Running Variables (skipAttemps etc.)
Evaluate Battery Longevity
Check homebase connection
Is there a connection?
Yes
Set the node to a sink node
Request sync from HomeBase
To Time Synchronization
No
Awake cycle
To Awake cycle
Pre-sleep Preparation
Process Remaining Inbox Elements
Store Outbox Persistently

Figure 6.8: Flowchart showing how the operational phase works. This figure is shared with Erlend Karlstrøm

which node should be allowed to send non-optimized bundles.

In this mode, the node also restores previous not sent messages from the disk so that it might be able to propagate content this cycle. While restoring the data from the disk, the node also tries to optimize new bundles by merging the data from what was restored and the newly created status report. I will also not go into detail on how the content is merged, as this is created by Erlend Karlstrøm, and is directly connected to his thesis. However, I will state that the bundles are messages that contain more data than a single status report.

When preparing for waking up, the nodes will also check if they should enter the low-power mode the next cycle. If this is the case, the node will issue a message around in the network that will trigger the other nodes to remove this node from their sink-paths list and consider the node as inactive. This will be described in the awake cycle section 6.3.2.

In this mode, the nodes also check for a connection to the homebase. If it has a connection, it spawns a message that will be propagated throughout the network to share the new sink path and perform a time synchronization immediately to the sink. This, in turn, means that as it enters the awake cycle, the node will be updated, which will make it easier for the other nodes to synchronize and fetch an updated clock. How the time synchronization is performed will be outlined in section 6.4.

6.3.2 Awake-Cycle

After preparing for the wake-up, the node enters the awake cycle as we see in figure 6.9. Here the nodes start accepting messages from other nodes but wait with sending something until a timer has been reached. This is to limit the number of retransmissions that might happen in the case of neighbors sleeping(which might happen if they have skewed a bit, making them wake up later).

The first thing a node does is to check if it should synchronize with a parent node. The time synchronization will be explained in detail in section 6.4.

After trying to synchronize with its parent, the node will check how long ago it was since its last connection with all the neighbors were (A timer can be set in the configuration file). If the node has not recently been in contact with the other nodes, it will issue a global neighborhood ping message. This, in turn, means that the node will send a ping message to all neighboring nodes even if they are marked as dead or inactive. This will exchange neighbor knowledge as explained in section 6.1.3, or if they were tagged as dead, mark them as

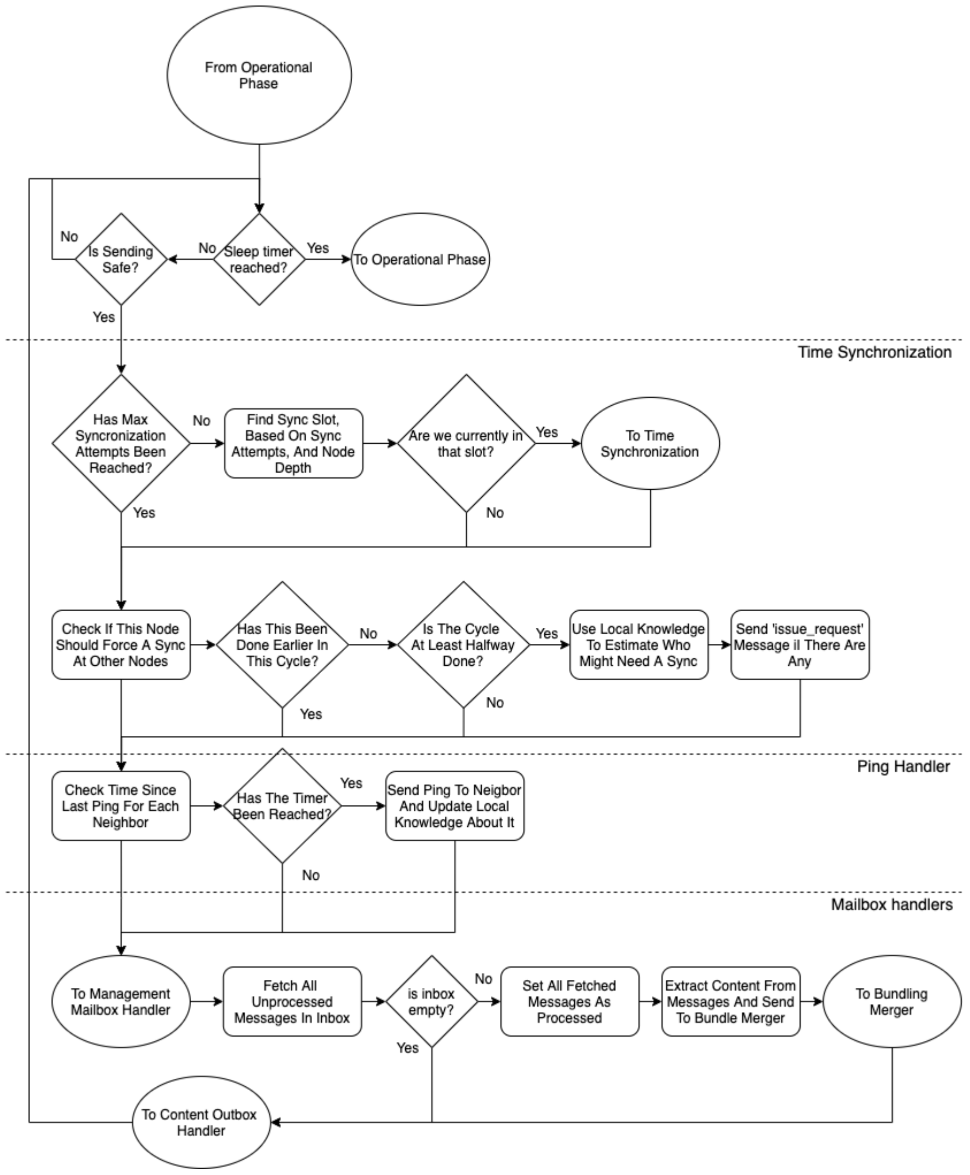

From Operational Phase
Sleep timer reached?
Yes
To Operational Phase
No
Is Sending Safe?
No
Yes
Time Synchronization
Has Max Syncronization Attempts Been Reached?
No
Find Sync Slot, Based On Sync Attempts, And Node Depth
Are we currently in that slot?
Yes
To Time Synchronization
No
Yes
Check If This Node Should Force A Sync At Other Nodes
Has This Been Done Earlier In This Cycle?
No
Is The Cycle At Least Halfway Done?
Yes
Use Local Knowledge To Estimate Who Might Need A Sync
Send 'issue_request' Message il There Are Any
Yes
No
Ping Handler
Check Time Since Last Ping For Each Neighbor
Has The Timer Been Reached?
Yes
Send Ping To Neigbor And Update Local Knowledge About It
No
Mailbox handlers
To Management Mailbox Handler
Fetch All Unprocessed Messages In Inbox
is inbox empty?
No
Set All Fetched Messages As Processed
Extract Content From Messages And Send To Bundle Merger
To Bundling Merger
Yes
To Content Outbox Handler

Figure 6.9: Flowchart showing how the awake cycle works. This figure is shared with Erlend Karlstrøm

alive again(Explained in section 6.3.2.6).

6.3.2.1 Forced Time Synchronization

Halfway through the awake cycle, the nodes will check if there any nodes that should have synchronized with it by now. It does this by estimating which nodes are children of the node, which it does by checking the neighbor knowledge and seeing if the node is set as the preferred parent of any neighbors. By estimating what nodes are children and using these nodes to see if they have contacted the node with a synchronization request within a set timer, all the estimated children which have not synchronized yet are found.

Using the newly discovered list of non-synchronized children the node can issue a force synchronization request, which will in turn trigger one of two different outcomes. Either a synchronization request by the children nodes or a notification that tells the node that it is no longer the preferred parent(as the preferred parent is calculated based on the path-score and can change quickly).

This is done to ensure that the nodes in the network remain synchronized with their parents because if a node tries to synchronize three times and fails all times, it will not be allowed to synchronize that cycle, known as *quarantined*.

This will force the children nodes to update, even if they have failed the set number of times, which might have happened due to a significant skew difference, where one node has its local clock skewed into the future and another node skewed further in the back, compared to the real-time.

6.3.2.2 Management Mailbox Priority

As stated in chapter 5, the management mailbox in the system is prioritized over the other mailboxes. This is to ensure that the network health messages are processed and propagated before handling any content.

Following the awake cycle figure 6.9 which shows that after handling the synchronization and ping, the node checks the management mailbox. Which is shown in figure 6.10.

Here the nodes try to check if it has any non-processed messages that it needs to process locally before propagating to the neighboring nodes. If there are any non-processed messages, the node extracts the data and type of message

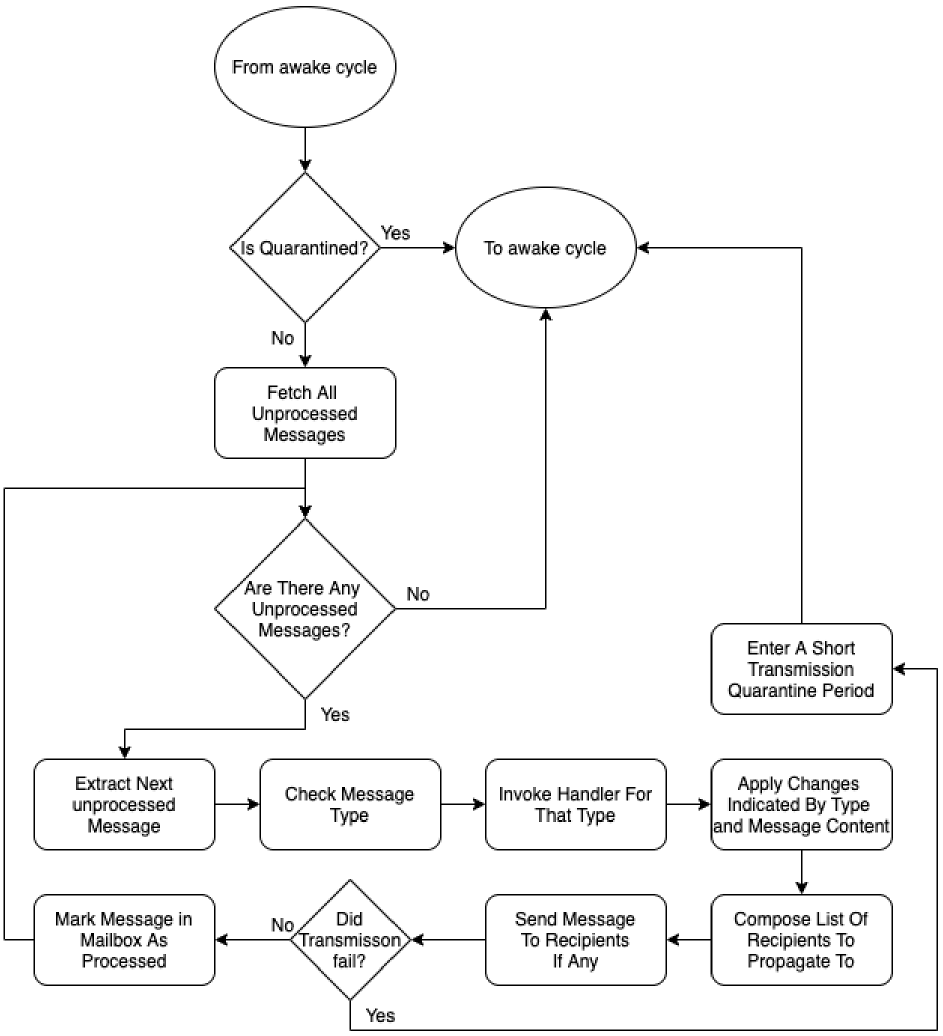

From awake cycle
Is Quarantined?
Yes
To awake cycle
No
Fetch All Unprocessed Messages
Are There Any Unprocessed Messages?
No
Yes
Enter A Short Transmission Quarantine Period
Extract Next unprocessed Message
Check Message Type
Invoke Handler For That Type
Apply Changes Indicated By Type and Message Content
Mark Message in Mailbox As Processed
No
Did Transmisson fail?
Send Message To Recipients If Any
Compose List Of Recipients To Propagate To
Yes

Figure 6.10: Flowchart showing how the management mailbox is prioritized. This figure is shared with Erlend Karlstrøm

and starts to perform the local changes, if they are required – changes such as removing dead nodes from the sink path list or start distributing messages about a newly gained homebase connection.

After the node has processed the messages locally, they start by distributing the same messages to the neighboring nodes. If the node fails to send the messages, it enters a *temporary quarantine state*. This means that the node will stop trying to communicate for a bit and resume after a while, this is to reduce the number of retransmissions needed in case of a sleeping neighbor.

6.3.2.3 Content Propagation

Following figure 6.11 we see that before sending any content, the inbox is read, and all the non-processed messages are moved over to the outbox, but first, they are optimized with the bundling algorithm. To understand how the bundling algorithm works, read Erlend Karlstrøm's master thesis[11].

After bundling the newly arrived messages trying to create optimal bundles, the node starts trying to propagate the content. Before starting to transmit the content, it makes sure that there are sinks in the network because if there are no sinks, it will not send any messages.

But if there are sinks in the network, the node will try to calculate the most optimal path towards one of those sinks. How this is done is explained in section 6.1.5.5. After fetching an optimal path, it will extract the first step in the path and start sending the optimized messages that way. If a node has been allowed to send partial bundles, as explained at the start of the operational phase section, it will do so here. If the sink node is trying to propagate content, it will send only the optimized bundles directly to the homebase and is not allowed to send partial bundles.

If the nodes fail to send, they will enter a temporary quarantined state and stop sending messages for a bit so that the connection will hopefully stabilize. However, if a node fails to send messages a set amount of times in a row (amount configurable in the configuration), it will enter a constant *quarantined* state. In the quarantined state, the node is not allowed to send any more data and will wait until the next cycle before resuming data propagation again(similarly to the time synchronization from the previous section).

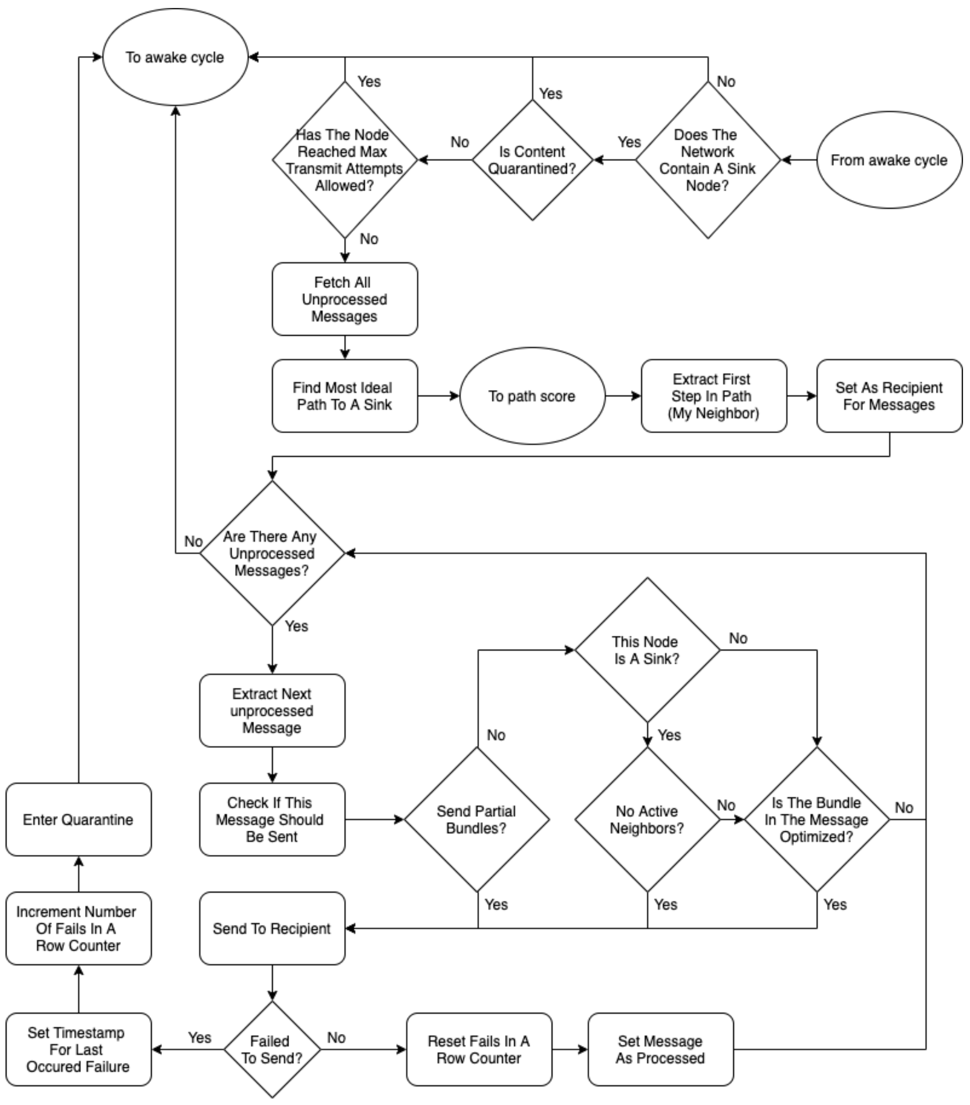

To awake cycle
Yes
Yes
No
Has The Node Reached Max Transmit Attempts Allowed?
No
Is Content Quarantined?
Yes
Does The Network Contain A Sink Node?
From awake cycle
No
Fetch All Unprocessed Messages
Find Most Ideal Path To A Sink
To path score
Extract First Step In Path (My Neighbor)
Set As Recipient For Messages
No
Are There Any Unprocessed Messages?
Yes
This Node Is A Sink?
No
Extract Next unprocessed Message
No
Yes
Enter Quarantine
Check If This Message Should Be Sent
Send Partial Bundles?
No Active Neighbors?
No
Is The Bundle In The Message Optimized?
No
Increment Number Of Fails In A Row Counter
Send To Recipient
Yes
Yes
Yes
Set Timestamp For Last Occured Failure
Yes
Failed To Send?
No
Reset Fails In A Row Counter
Set Message As Processed

Figure 6.11: Flowchart showing how the content propagation works. This figure is shared with Erlend Karlstrøm

6.3.2.4 Dead Node Discovery

When a node fails to send a message to another node, it checks its local knowledge about that targeted node to see if it should be marked as dead. This operation checks if the node has reached some threshold values(configurable), the checked values are a measure of battery life and the last contact timestamp. If any of the set thresholds are reached, the node will spawn a dead node message that will be flooded over the network.

When a node receives a message about a dead node, it removes all paths that lead through that specific node from its sink path list while also tagging the node as inactive in its local knowledge store(if it was a part of the knowledge previously). This is done to prevent the nodes from trying to route through a node that is dead. It then propagates that same message until all nodes in the network know about the dead node.

6.3.2.5 Leaving the Network When Entering Low-Power Mode

After the node has decided that it should enter the low-power mode the next cycle due to being lower than 10% battery(configurable), it creates a message that will be sent to all other nodes in the network, signing itself out of the network. This message is handled in the same way that the dead node message is handled, shown in the previous section 6.3.2.4. This enables the other nodes to stop propagating data through this node.

6.3.2.6 Wrongly Marked as Dead

Due to the environment, some connections may disappear for a while, then resurface again due to things such as snow or bad weather. This can lead to nodes being considered as dead and removed from the known sink-path lists across the network. As stated earlier, the nodes issue a global ping to all known neighboring nodes, even if they are tagged as dead. If this leads to contact between two nodes, they will share their local knowledge and update it. This will also make sure that the nodes are marked as alive. If a node was previously marked as dead, another message will be spawned, forcing the node to share its paths with the wrongly marked node. This will ensure that nodes can leave and join the network, only requiring that they joined the operational network in the starting phase.

6.3.3 Pre-Sleep Preparation

After the awake cycle(as seen in figure 6.9) has run for its designated time, the node shuts down all communication and enters the pre-sleep preparation. In this mode, the node processes the remaining messages in the inbox, transfers them to the outbox, and stores all the non-sent messages persistently(temporary as they will be deleted when sent). This is done so the other nodes do not have to retransmit their original status report when transmitting towards the sink/homebase. After storing the data and finishing the preparation for the duty cycling, the node enters a sleep mode until it reaches the designated wake-up timer again and starts the whole cycle all over again.

6.3.4 Low-Power Mode

When a node has left the network and has entered the low-power mode, all connections are stopped. The node will no longer try to remain synchronized with its neighboring nodes and will not try to send anything. This mode is designed to dedicate all the remaining battery life to recording environmental data and take photos while also storing the status reports persistently.

6.4 Time Synchronization

When the time synchronization operation is invoked, we can see by following the figure 6.12, the first thing the node does is check if there is a sink in the network. If there is a sink in the network, it will use the immediate time synchronization algorithm. If there are no sinks in the network, it will use the gradual algorithm. The immediate- and gradual algorithms will be explained in the following sections.

6.4.1 Immediate Time Synchronization

$$OneWayLatency = \frac{Abs((TA2 - TA1) - (TB2 - TB1))}{2}$$

$$CalculatedTB2Time = TB2 + OneWayLatency$$

When entering the time synchronization operation after establishing a connection to the sink, or when finding out that there was a sink in the network, the node uses the immediate time synchronization.

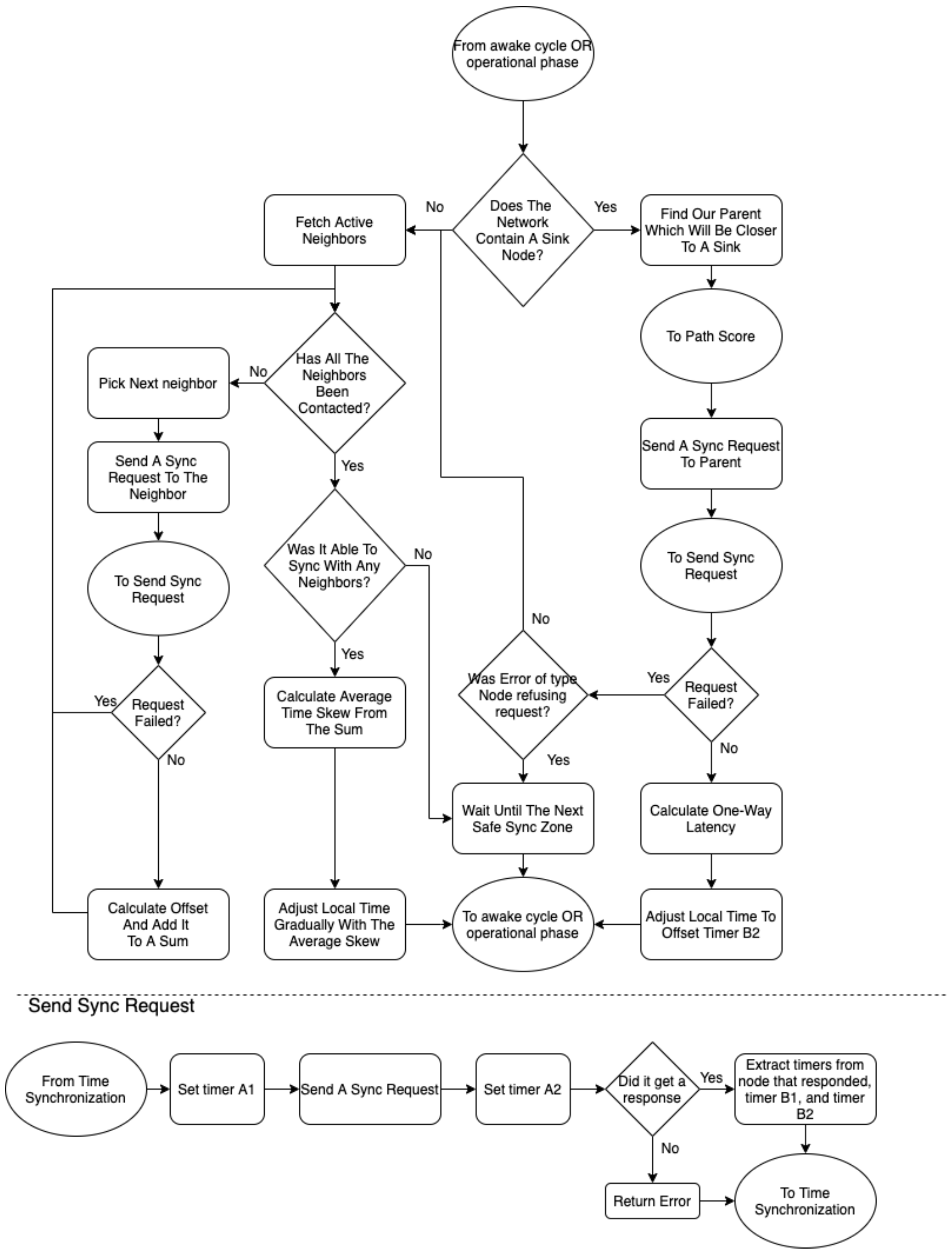

From awake cycle OR operational phase
Does The Network Contain A Sink Node?
No
Yes
Fetch Active Neighbors
Find Our Parent Which Will Be Closer To A Sink
Has All The Neighbors Been Contacted?
No
Pick Next neighbor
Yes
To Path Score
Send A Sync Request To Parent
Send A Sync Request To The Neighbor
Was It Able To Sync With Any Neighbors?
No
To Send Sync Request
To Send Sync Request
No
Was Error of type Node refusing request?
Yes
Request Failed?
Yes
Request Failed?
No
Yes
Calculate Average Time Skew From The Sum
Yes
No
Wait Until The Next Safe Sync Zone
Calculate One-Way Latency
Calculate Offset And Add It To A Sum
Adjust Local Time Gradually With The Average Skew
To awake cycle OR operational phase
Adjust Local Time To Offset Timer B2
Send Sync Request
From Time Synchronization
Set timer A1
Send A Sync Request
Set timer A2
Did it get a response
Yes
Extract timers from node that responded, timer B1, and timer B2
No
Return Error
To Time Synchronization

Figure 6.12: Flowchart showing how the time synchronization works. This figure is **not** shared

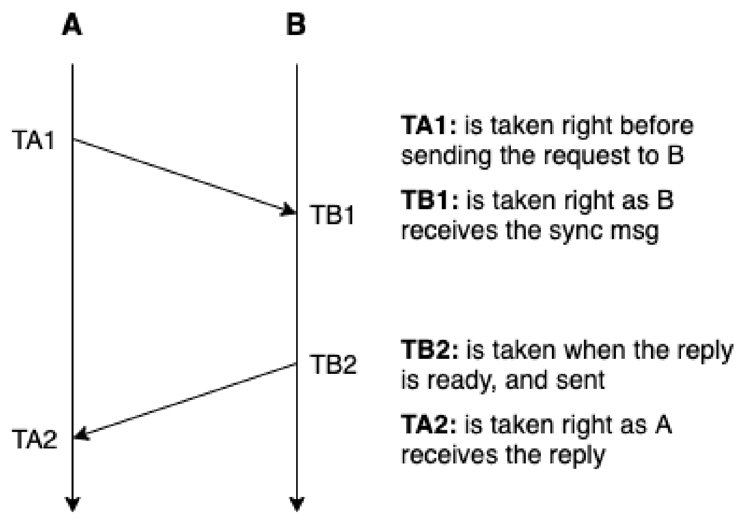

A
B
TA1
TB1
TB2
TA2
TA1: is taken right before sending the request to B
TB1: is taken right as B receives the sync msg
TB2: is taken when the reply is ready, and sent
TA2: is taken right as A receives the reply

Figure 6.13: Figure showing how the timestamps are recorded while using the time synchronization operation

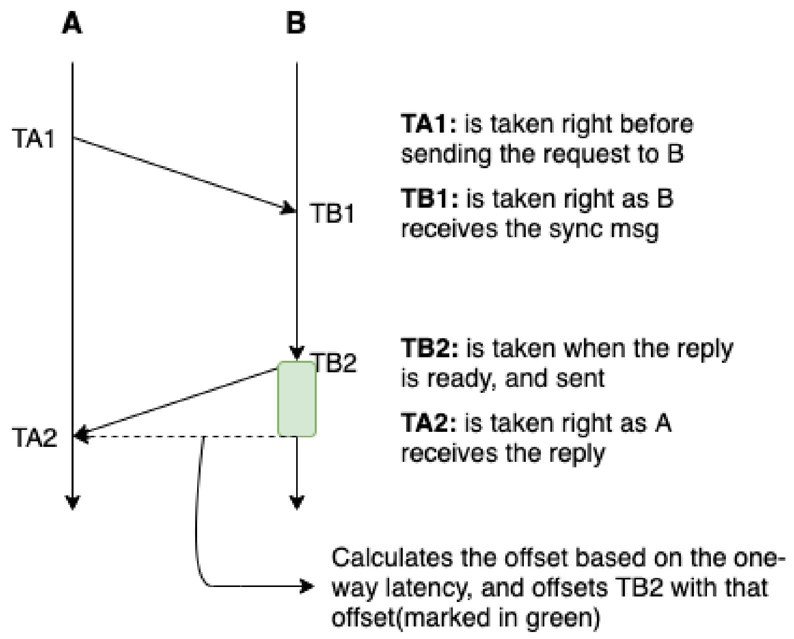

A
B
TA1
TB1
TB2
TA2
TA1: is taken right before sending the request to B
TB1: is taken right as B receives the sync msg
TB2: is taken when the reply is ready, and sent
TA2: is taken right as A receives the reply
Calculates the offset based on the one-way latency, and offsets TB2 with that offset(marked in green)

Figure 6.14: Figure showing how the Latency-based algorithm offsets timer TB2 with the latency, to calculate the correct time.

The immediate algorithm is created as a variation of the Timing-Sync Protocol for Sensor Networks(TPSN)[4] algorithm, where it uses the same timestamp fetching way(shown in figure 6.13), but rather calculates a one-way-latency from the timestamp, and uses that to offset the collected parent timestamp with. Thus, being able to calculate a correct time from the parent node. How the node uses the latency to calculate the actual parent node time can be seen in figure 6.14.

Here it calculates which node it should synchronize against using the path score to find a suitable parent node(shown in section 6.1.5.5) and starts the synchronization operation. When beginning the operation, it starts a timer known as TA1 and sends the request to the designated parent. The parent will return with two timestamps, which are set at the first contact and the end of the processing time, known as TB1 and TB2. After returning, the node sets the last timestamp, known as TA2. The timestamps readings are shown in figure 6.13.

Using the formula shown above, node A is able to calculate the correct time for node B(shown as CalculatedTB2Time in the formula), offsetting the recorded time in the parent node with the latency between them. After finding the actual clock of node B, node A adopts this time and changes its local clock to that timestamp.

The immediate operation can fail in two different ways. The first way is having the nodes refuse the synchronization, which happens when the parent node is not yet updated, which the skewed safe synchronization zones are trying to fix(see section 6.15). It can also fail if the calculated parent is not responding to the messages.

If the immediate operation fails, with the parent responding by refusing the node to synchronize(as the parent node is not updated), it will wait until its next skewed safe zone and try again. This will happen three times, and if they all fail, the node will fall back on synchronizing with their neighbors as a backup, using the gradual algorithm. However, if the immediate operation fails due to no response, it will immediately use the backup solution(gradual), as shown in the next section.

6.4.2 Gradual Time Synchronization

$$Offset = \frac{((TB2 - TA1) + (TB1 - TA2))}{2}, fromTPNS[4]$$

When entering the time synchronization and finding out that the network does not contain any sink nodes or entering the time synchronization after failing the immediate algorithm, the node proceeds to use the gradual time synchronization algorithm.

The gradual time synchronization algorithm is also created as a variation of the TPSN algorithm[4], where it calculates the offset in the same way, but instead of using the offset from an authoritative time source directly, it calculates an average neighborhood offset and uses that average neighborhood offset to gradually skew the local clock towards the center of the network, this will ensure that all nodes move in the same direction, and remain synchronized.

First, the node finds all active neighbors and starts issuing requests to each of them. The timestamp recording is done similar to the immediate time synchronization operation and the TPSN algorithm, which is shown in figure 6.13. The node calculates its offset compared to each node and adds all of the offsets into a temporary offset list. The node calculates a neighborhood average offset based on that list. The formula above shows how the offset is calculated for each node. The node then uses this neighborhood offset average to skew the local time in that direction.

If the node fails to send a request to a single node, it simply skips that node. However, if it fails to send a message to all nodes, it will not synchronize this awake cycle, as it most likely has already skewed out of sync with the other nodes.

6.4.3 Skewed Safe Synchronization Zone

Listing 6.2: Code showing how the safe synchronization calculation is done

```
func calculateSafeSyncZones() []time.Duration {
  safeZones := []time.Duration{}
  safeArea := ALIVEDURATION - (2 * SAFESKEWOFFSET)
  safeZoneOffset := float64(safeArea) / float64(SYNCATTEMPTS)

  for i := 0; i < SYNCATTEMPTS; i++ {
    zone := float64(SAFESKEWOFFSET) + (float64(i) * safeZoneOffset)
    safeZones = append(safeZones, time.Duration(zone))
  }
  safeZones = append(safeZones, time.Duration(ALIVEDURATION-SAFESKEWOFFSET
      ))
  return safeZones
}
```

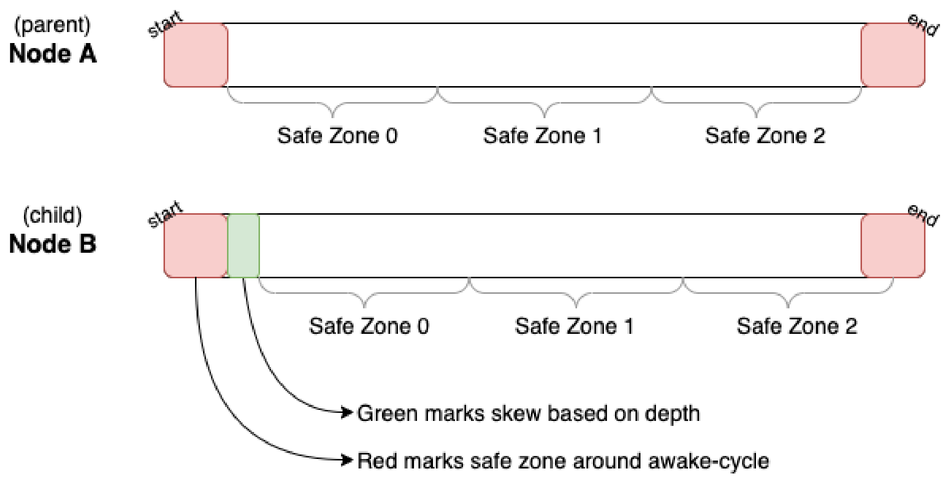


Figure 6.15: Figure showing how the skewed safe zones work.

Listing 6.3: Code showing how nodes check if they should synchronize or not.

```
func syncOrNot(ts time.Time) bool {
  if node.timeSyncAttempts >= SYNCATTEMPTS || node.synchronizedThisCycle {
    return false
  }

  myDepth := findMyDepth()
  for i := node.timeSyncAttempts; i < SYNCATTEMPTS; i++ {
    nextSync := skewDuration(node.safeSyncZones[i], myDepth)
    afterNextSync := skewDuration(node.safeSyncZones[i+1], myDepth)
    if timerReached(ts, nextSync) && !timerReached(ts, afterNextSync) {
      node.timeSyncAttempts = i + 1
      return true
    }
  }
  return false
}
```

Before entering the time synchronization operation from the awake cycle, the node checks if it has reached its calculated synchronization zone. The safe zones are calculated in the initializing phase of the node, as they are based on a few configurable settings. By using the code shown in listing 6.2, the nodes are able to calculate a set of zones within the awake cycle timer. These zones are timer marks that divide the awake cycle into a set amount of chunks, where the nodes are allowed to try to synchronize once in each chunk. This is to reduce the number of transmissions, and so the node fetches the updated clock from its calculated parent.

To ensure that the nodes are able to fetch an updated clock from their parents, the safe synchronization zones are skewed based on the node's estimated depth in the network, as seen in listing 6.3, where it calculates its depth and skews the calculated chunks from the safe zone based on a fixed timer(configurable). By offsetting each zone, based on the estimated depth, the nodes can synchronize out of order with each other, which leads to a bigger chance of fetching the updated and correct clock. How the skewed safe zone works is also shown in figure 6.15.

If the node can fetch a zone where it is allowed to synchronize, it will try to synchronize with a parent node. However, if it has failed a set number of times(configurable), it will be quarantined for that cycle and only be allowed to synchronize if the parent node forces a synchronization request.

7 Implementation

This chapter is about the implementation of the prototype system, which the author, in collaboration with Erlend Karlstrøm has developed in accordance with the proposed system architecture and design.

The content of this chapter is shared between both authors, where both deserve 50% credit for it. However, the Homebase Design(section **??**) is created only by the author of this thesis.

Programming Languages:

- Golang v1.16
- Python v3.9.4

Golang Packages:

- Termui v3.1.0[5]

The code for the prototype nodes was implemented in Golang[6], which is an open-source programming language designed at Google. Here, Golang's stock HTTP library is used for functionality such as hosting an API on each node and handling the sending of data in the form of HTTP requests using the TCP(Transmission Control Protocol) protocol. The homebase is also implemented in Golang and uses the HTTP library in much the same way. The homebase also uses the *termui*-package from Gizak on GitHub to provide a GUI(Graphical User Interface) in the terminal, which has been helpful in presenting the data that the homebase has received from the nodes.

The Python[23] language was used to make a script, which automated the process of launching multiple nodes in different configurations. The subprocess module from Python's standard library was used to start and manage the nodes, and provide the executables with the correct arguments to form the network topology that was needed.

To execute the prototype, follow the steps located in the README's, which are located in the source folder, and the following sub-folders.

7.1 Introduction Broadcast

When the nodes introduce themselves to the other nodes in the network, the nodes send messages to a range of node addresses. These receiving nodes will then respond with a message, which indicates whether or not they accept or reject the handshake. They make this decision based on their internal list of expected neighbors(will be explained in section 7.2). This range of node addresses is supposed to simulate a radio frequency range dedicated for broadcasting.

7.2 Topology Generation

A node is initialized with a list of node addresses which tells it which neighbors it is *supposed* to have. This is used by the node to decide which introduction messages from new nodes it is supposed to reject or accept. This list of nodes is either supplied from the python script, which has many predefined networks stored, or it may be supplied by hand, when starting a single node. This is intended to simulate how nodes will not get a response when attempting to contact potential neighbors which are out of radio range. This approach makes it so that the network is always guaranteed to generate itself in the same way every time, given the same configuration. While this is not representative of how this would work in a real-world deployment scenario, this enables us to reproduce network behavior and specific scenarios multiple times, making the network characteristics easier to validate and experiment on.

7.3 Environmental Readings

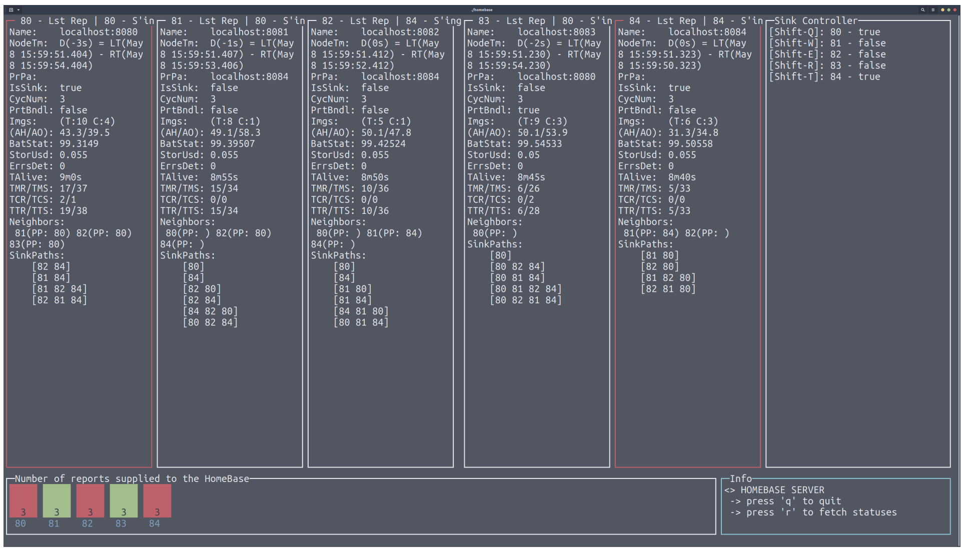
While this section is more relevant to Erlend Karlstrøm's thesis, it is still relevant for Sigurd Karlstad, as the main focus of the OU's is to record environmental readings. OU's observe the surrounding physical environment, and produce data which reports on the state of it. In this project, this functionality is simulated by using random generation, in order to populate the reports that the nodes produce. Events are produced randomly during node sleep to simulate an IR-sensor detecting movement and taking a picture with its IR-camera. The humidity and oxygen counts uses random number generation to produce averages within a certain range.

7.4 Battery Drain

As environmental readings from sensor components are simulated, so are those components' drain on the battery. The battery levels are represented via a variable in the node that is initialized to 99.9, intended to represent a percentage value. As the node goes through its cycles, a new value for the battery is calculated based on two factors: the time since the node was first initialized, and the number of transmissions it has sent and received. The calculation itself is very simple, and is not based on any kind of estimation of battery drain in a real-world scenario. While the battery drain itself is not relevant in a simulated system, it is very relevant in order to demonstrate the score calculation and path selection features.

7.5 Simulated Duty Cycling

Between each awake cycle the nodes sleep to simulate the duty cycling in the system. The nodes use timestamps to keep track of when they last awoke and slept, and checks them periodically during execution to keep time. While the node is sleeping, it will reject any incoming transmission, as an OU which has turned off its antenna would also be unavailable for communication. This is implemented as the receiver-thread checking the awake flag before processing the message, and if the flag says that the node is supposed to be sleeping, then the response will be a custom error code. The error will be handled by the other node as if the node did not respond at all, i.e., a timeout error.



```
80 - Lst Rep | 80 - S'in
Name:     localhost:8080
NodeTm:   D(-3s) = LT(May
8 15:59:51.404) - RT(May
8 15:59:54.404)
PrPa:
IsSink:   true
CycNum:   3
PrtBndl: false
Imgs:     (T:10 C:4)
(AH/AO): 43.3/39.5
BatStat: 99.3149
StorUsd: 0.055
ErrsDet: 0
TAlive:   9m0s
TMR/TMS: 17/37
TCR/TCS: 2/1
TTR/TTS: 19/38
Neighbors:
 81(PP: 80) 82(PP: 80)
83(PP: 80)
SinkPaths:
    [82 84]
    [81 84]
    [81 82 84]
    [82 81 84]
```

```
81 - Lst Rep | 80 - S'in
Name:     localhost:8081
NodeTm:   D(-1s) = LT(May
8 15:59:51.407) - RT(May
8 15:59:53.406)
PrPa:     localhost:8084
IsSink:   false
CycNum:   3
PrtBndl: false
Imgs:     (T:8 C:1)
(AH/AO): 49.1/58.3
BatStat: 99.39507
StorUsd: 0.055
ErrsDet: 0
TAlive:   8m55s
TMR/TMS: 15/34
TCR/TCS: 0/0
TTR/TTS: 15/34
Neighbors:
 80(PP: ) 82(PP: 80)
84(PP: )
SinkPaths:
    [80]
    [84]
    [82 80]
    [82 84]
    [84 82 80]
    [80 82 84]
```

```
82 - Lst Rep | 84 - S'ing
Name:     localhost:8082
NodeTm:   D(0s) = LT(May
8 15:59:51.412) - RT(May
8 15:59:52.412)
PrPa:     localhost:8084
IsSink:   false
CycNum:   3
PrtBndl: false
Imgs:     (T:5 C:1)
(AH/AO): 50.1/47.8
BatStat: 99.42524
StorUsd: 0.055
ErrsDet: 0
TAlive:   8m50s
TMR/TMS: 10/36
TCR/TCS: 0/0
TTR/TTS: 10/36
Neighbors:
 80(PP: ) 81(PP: 84)
84(PP: )
SinkPaths:
    [80]
    [84]
    [81 80]
    [81 84]
    [84 81 80]
    [80 81 84]
```

```
83 - Lst Rep | 80 - S'in
Name:     localhost:8083
NodeTm:   D(-2s) = LT(May
8 15:59:51.230) - RT(May
8 15:59:54.230)
PrPa:     localhost:8080
IsSink:   false
CycNum:   3
PrtBndl: true
Imgs:     (T:9 C:3)
(AH/AO): 50.1/53.9
BatStat: 99.54533
StorUsd: 0.05
ErrsDet: 0
TAlive:   8m45s
TMR/TMS: 6/26
TCR/TCS: 0/2
TTR/TTS: 6/28
Neighbors:
 80(PP: )
SinkPaths:
    [80]
    [80 82 84]
    [80 81 84]
    [80 81 82 84]
    [80 82 81 84]
```

```
84 - Lst Rep | 84 - S'in
Name:     localhost:8084
NodeTm:   D(0s) = LT(May
8 15:59:51.323) - RT(May
8 15:59:50.323)
PrPa:
IsSink:   true
CycNum:   3
PrtBndl: false
Imgs:     (T:6 C:3)
(AH/AO): 31.3/34.8
BatStat: 99.50558
StorUsd: 0.055
ErrsDet: 0
TAlive:   8m40s
TMR/TMS: 5/33
TCR/TCS: 0/0
TTR/TTS: 5/33
Neighbors:
 81(PP: 84) 82(PP: )
SinkPaths:
    [81 80]
    [82 80]
    [81 82 80]
    [82 81 80]
```

```
Sink Controller
[Shift-Q]: 80 - true
[Shift-W]: 81 - false
[Shift-E]: 82 - false
[Shift-R]: 83 - false
[Shift-T]: 84 - true
```

```
Number of reports supplied to the HomeBase
 3    3    3    3    3
 80   81   82   83   84
```

```
Info
<> HOMEBASE SERVER
 -> press 'q' to quit
 -> press 'r' to fetch statuses
```

Figure 7.1: Figure showing an example of how the Homebase looks with the nodes have reported data to it.

7.6 Simulated Skew

While this section is more relevant to Sigurd Karlstads thesis, it is still relevant to Erlend Karlstrøm as the local clock skew is a common problem for WSNs. When nodes are disconnected from services like NTP, a computer's RTC clock will tend to skew away from the authoritative time. This skew is simulated in the prototype by maintaining a variable in the node which is added to real clock timestamps whenever the node needs to record the current time. This variable is incremented by a factor after the sleep cycle ends, which simulates clock drift during duty cycling. The factor is generated by the nodes upon intialization, and is a random number in a certain range specified by the configuration file. The randomness is meant to intentionally create differences in the nodes' clocks, as the clock drift of computing devices located in the tundra may be influenced by a set of different factors. In [33], by Yik-Chung Wu et al. they state that "In the long term, clock parameters are subject to changes due to environmental or other external effects such as temperature, atmospheric pressure, voltage changes, and hardware aging". This difference is what the skew is supposed to simulate.

7.7 Homebase Design

While the homebase is not viewed upon as one of the most important parts of this thesis as a whole, it is still relevant because the nodes would not be able to propagate data without it, and the system would be forced to use the gradual time synchronization algorithm, as there would be no sink nodes in the network. The homebase consists of three parts, the report receiver, the Sink Controller, and the content viewer.

The report receiver is an API that handles receiving the status reports from the sink nodes and answering time synchronization requests from sink nodes. The sink controller is a way to turn nodes from ordinary nodes to sink nodes. As this is a simulator, the sink controller(top right in figure 7.1) can contact all the nodes and turn all of them to sink nodes. The content viewer is a section of the screen that displays how many reports have been received and also displays the last report supplied from each node, which is the section to the left in figure 7.1.

/8 Validation

In this chapter, we will aim to uncover the characteristics of the implemented solution by measuring various aspects of it.

The General Network Validation, General Network Validation Results, Validation Setup, and the relevant data are all shared between the author of this thesis, and Erlend Karlstrøm, where both deserve 50% credit for it.

However, the Time Synchronizing Validation is created only by the author of this thesis. The author, therefore, deserves full credit for it.

8.1 Validation Setup

All the following characteristic validations were done on a HP Z4 G4 Workstation with the following specifications:

- **CPU:** Intel Xeon W-2123 8-core @ 3.900 GHz
- **GPU:** NVIDIA Quadro RTX 4000
- **RAM:** 32 GiB DDR4 Memory
- **OS:** Ubuntu 20.04.1 LTS 64-bit

- **Kernel Version:** 5.4.0-56-generic

While doing the validation described in the subsequent sections, we will be using the following settings if not specified otherwise:

- **Time Synchronization:** Enabled
- **Simulated skew:** In the range of -3 to 3 seconds (except 0).
- **Cycles per Sync:** Synchronization is performed once per awake cycle.
- **Bundling behavior:** Optimal bundle size set to 3, with Partial Bundle Policy.
- **Score Weights:** Path weight set to 2.0, Battery weight set to 4.0, Latency weight set to 1.0.
- **Battery Discharging:** Enabled

8.2 Validation Design

8.2.1 General Network validation

These network validations are aimed to validate the general architecture of the system and that it operates as was outlined in the architecture and design sections. Parts we are going to investigate include: path generation, sharing, acceptance, selection and removal, optimal parent calculation, and rejoining after a network disruption.

8.2.1.1 Path Discovery and Path Acceptance during Starting Phase

To validate how accurate and correct the path generation and path acceptance is, we need to test how the paths are shared and stored in the starting phase. These tests are done because path generation, acceptance, and sharing are core components of the system solution. It is also important because without extended network knowledge(Outside the local "neighborhood"), data propagation and time synchronization would be not possible.

Method: Using the network structure shown in figure 8.1. We start the network in the starting phase, and read the paths off the reports to check if they are

Node Name	Sink Paths To sink node 0	Sink Paths to sink node 6
0		[[1, 6], [2, 4, 5, 6], [2, 3, 1, 6], [1, 3, 2, 4, 5, 6]]
1	[[0], [3, 2, 0], [6, 5, 4, 2, 0]]	[[6], [3, 2, 4, 5, 6], [0, 2, 4, 5, 6]]
2	[[0], [3, 1, 0], [4, 5, 6, 1, 0]]	[[0, 1, 6], [4, 5, 6], [3, 1, 6]]
3	[[1, 0], [2, 0], [1, 6, 5, 4, 2, 0], [2, 4, 5, 6, 1, 0]]	[[1, 6], [2, 0, 1, 6], [2, 4, 5, 6], [1, 0, 2, 4, 5, 6]]
4	[[2, 0], [2, 3, 1, 0], [5, 6, 1, 0], [5, 6, 1, 3, 2, 0]]	[[5, 6], [2, 0, 1, 6]]
5	[[4, 2, 0], [6, 1, 0], [4, 2, 3, 1, 0], [6, 1, 3, 2, 0]]	[[6], [4, 2, 0, 1, 6], [4, 2, 3, 1, 6]]
6	[[1, 0], [1, 3, 2, 0], [5, 4, 2, 0], [5, 4, 2, 3, 1, 0]]	
7	[[2, 0], [2, 3, 1, 0], [2, 4, 5, 6, 1, 0]]	[[2, 0, 1, 6], [2, 4, 5, 6], [2, 3, 1, 6]]

Table 8.1: Table showing the expected Sink Paths from each node to the two sinks shown, network structure used is shown in figure 8.1. This table is shared with Erlend Karlstrøm

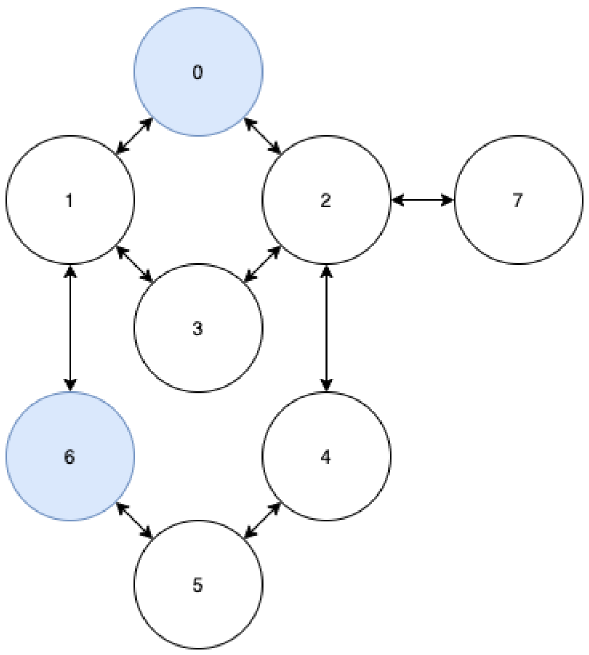


Figure 8.1: Figure showing a network structure with two sinks. The sink nodes are marked as a blue cicle. This figure is shared with Erlend Karlstrøm

correct. After running the test we expect that the generated and accepted paths will look similar in both cases to the paths shown in table 8.1.

8.2.1.2 Path Discovery and Path Acceptance during Operational Phase

To validate how accurate and correct the path generation and path acceptance is, we need to test how the paths are shared and stored in the operational phase. These tests are done because path generation, acceptance, and sharing are core components of the system solution. It is also important because without extended network knowledge(Outside the local "neighborhood"), data propagation and time synchronization would be not possible.

Method: Using the network structure shown in figure 8.1, with the modification of having no sinks initially. Let the network run for 2 cycles, then promote the same node that was a sink in the previous experiment to a sink node – Same sink as shown in the figure, node 80. After running the test we expect that the generated and accepted paths will look similar in both cases to the paths shown in 8.1.

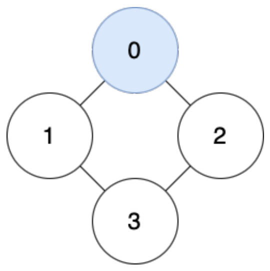


Figure 8.2: Figure showing a network structure with one sink node. The sink node is marked as a blue circle. This figure is shared with Erlend Karlstrøm

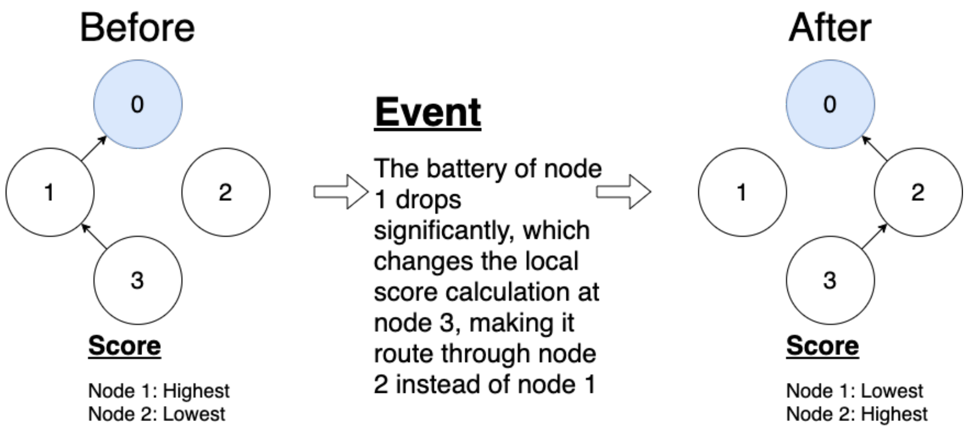

Before
0
1
2
3
Score
Node 1: Highest
Node 2: Lowest
Event
The battery of node 1 drops significantly, which changes the local score calculation at node 3, making it route through node 2 instead of node 1
After
0
1
2
3
Score
Node 1: Lowest
Node 2: Highest

Figure 8.3: Figure showing how the optimal parent calculation experiment method works. This figure is shared with Erlend Karlstrøm

8.2.1.3 Optimal Parent Calculation

To validate if a node is able to calculate and select an optimal parent from one of the first steps in the known path list, and calculate this based on the local knowledge of the neighboring nodes. These tests are important because it also demonstrates a node's ability to calculate a score for potential parents, and act on the result in a correct manner. Switching between optimal parents as their batteries drain, network-links decay or best paths become longer, is the way in which the system implements decentralized load-balancing.

Method: Using the network structure shown in figure 8.2. Start a network with bundling disabled and, generation of data continually so that the child node is always sending data to its parent. Record the path score values for nodes the current parent and the other potential one. A message is sent to the current parent which will artificially adjust its battery. The score is scored with regular intervals until the child node changes its parent node.

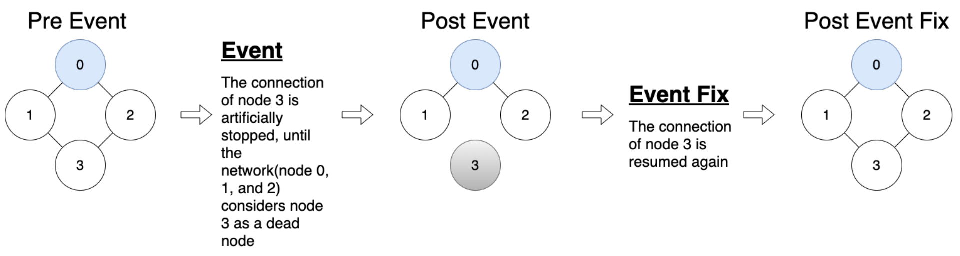

Pre Event
0
1
2
3
Event
The connection of node 3 is artificially stopped, until the network(node 0, 1, and 2) considers node 3 as a dead node
Post Event
0
1
2
3
Event Fix
The connection of node 3 is resumed again
Post Event Fix
0
1
2
3

Figure 8.4: Figure showing how the temporary connection disruption and validating dead node sink path removal experiment method works. This figure is shared with Erlend Karlstrøm

8.2.1.4 Temporary Connection Disruption and Validating Dead Node Sink Path Removal

To validate if: neighboring nodes are able to detect if a node is "dead", all paths in the network that uses this node as a step are removed, and the "dead" node is successfully re-integrated when it starts communicating again. This test is important because it demonstrates a node's ability to join a network after a disruption in communication, and the network's ability to recalibrate when previous members of the network join once again after being considered dead. These are capabilities that contribute to the overall robustness of the distributed system.

Method: Using the network structure shown in figure 8.2. Start a network that is able to start and connect in a normal manner, and let it run. After a few cycles, select one node to appear as though it is dead by not sending or answering requests, so that it will be considered dead by the rest of the network. Once the rest of the nodes are done with the recalibration process, resume the connections and check if it is able to rejoin the network again. To verify that the rejoining process has been successful, we record all paths: prior to stopping the selected node's communications, after it has been considered dead by all neighboring nodes, and after it has joined the network again. Finally, the paths in all three instances are checked for correctness.

8.2.2 Time Synchronization Validation

To validate how the network is able to remain stable by using the time synchronization algorithms, we need to check that the local clocks in the nodes in the system will skew away from each other before we can check if the implementation of the time synchronization is working correctly.

Method: Using the network shown in figure 8.5. Start the network with the

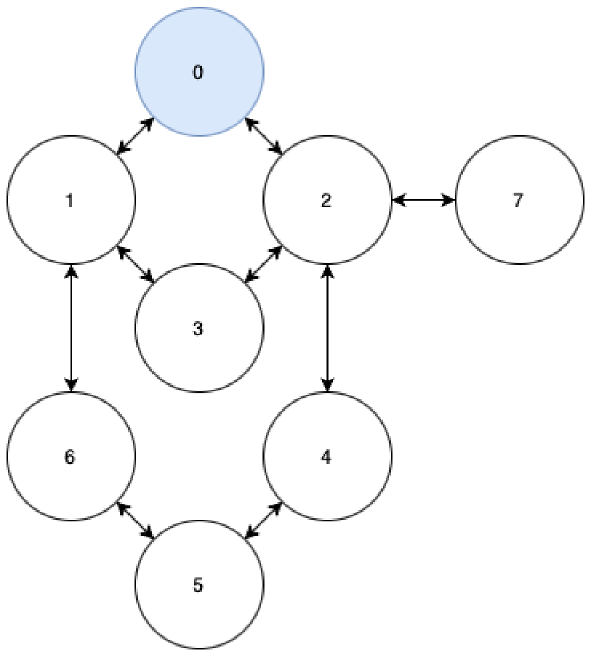


Figure 8.5: Figure showing a network structure with one sink.

time synchronization algorithms **disabled**, and let the system run for 20 cycles. At the start of each awake cycle record the node's local time, which is compared with the real-time. This will result in the time-skew. Finally, compare the time-skew and see if the nodes will skew away from each other.

8.2.2.1 Depth Calculation

To validate if the calculation of the estimated depth is correct, we need to check the calculated depth of each node in a system with a multi-sink topology to see if it can choose the correct depth, as it should always pick the shortest depth. This is important to investigate because the calculation of the estimated depth is used to skew the safe zones so the nodes synchronize their local clocks at a more appropriate time so that the "parent"-node should already be synchronized with its parent.

Method: Using the network structure shown in figure 8.1. We start the network in the starting phase. After the paths have been generated and the knowledge has been shared, we record the estimated depth at the start of the operational phase. The recorded estimated depth can then be compared with the figure 8.6, where the nodes should pick the depth that is closest to the sink node, i.e., the lowest depth.

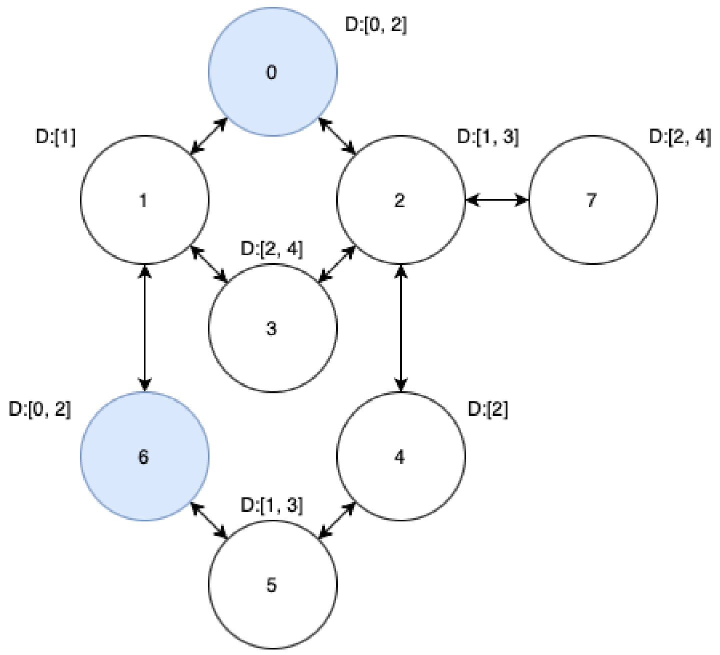


Figure 8.6: Figure showing the network structure shown in figure 8.1 with the estimated depth shown for each node.

8.2.2.2 Correctness Check for Immediate Adjustment Algorithm

To validate the correctness and accuracy of the immediate adjustment algorithm, which is based on the latency between the nodes, we need to perform calculations with a varying latency to see if the algorithm is able to handle longer latencies between the nodes and still calculate a correct clock. This is important to investigate because it checks the correctness of the immediate adjustment algorithm, which is important for network stability.

To perform the validation, we need to validate the results from the node calculation against a table that uses the algorithm. By implementing the algorithm into a spreadsheet where the nodes have similar static clocks(where the clocks do not change), we can increase the latency between the nodes. This creates a table showing what clock the node should be able to calculate the other actual clock to. This table is shown in table 8.2.

Method: Using the network structure shown in figure 8.7. We set a static time in both nodes, using the static clocks as specified in the table 8.2. Then we let the system run for 1 awake cycle and read the local time of a node after a successful time synchronization. To emulate an increasing amount of latency, we artificially sleep in node TB(parent node) after it has received a time synchronization request.

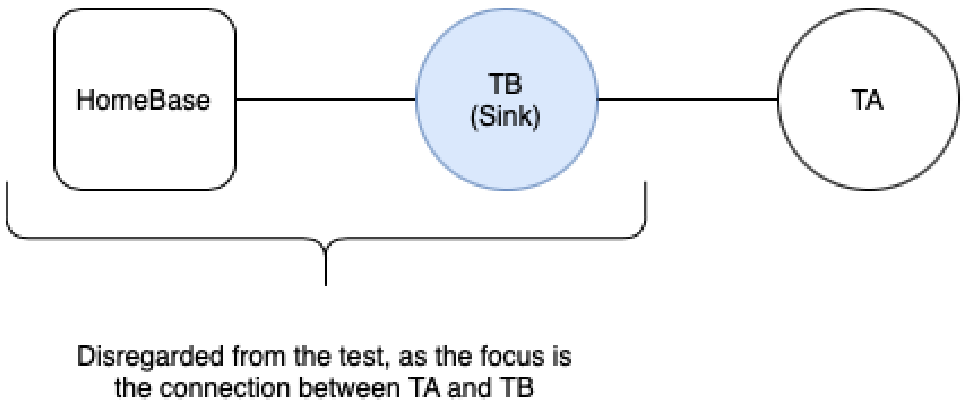

HomeBase
TB (Sink)
TA
Disregarded from the test, as the focus is the connection between TA and TB

Figure 8.7: Figure showing a network structure used for testing removal of latency in the Immediate adjustment algorithm

TA1	Total Latency in Sec	TB1	TB2	TA2(offset with total latency and processing time in TB)	Calculated Actual TB2(after total latency)
12:00:00	00:00:02	12:05:00	12:05:04	12:00:06	12:05:05
12:00:00	00:00:04	12:05:00	12:05:04	12:00:08	12:05:06
12:00:00	00:00:06	12:05:00	12:05:04	12:00:10	12:05:07
12:00:00	00:00:08	12:05:00	12:05:04	12:00:12	12:05:08
12:00:00	00:00:10	12:05:00	12:05:04	12:00:14	12:05:09
12:00:00	00:00:12	12:05:00	12:05:04	12:00:16	12:05:10
12:00:00	00:00:14	12:05:00	12:05:04	12:00:18	12:05:11
12:00:00	00:00:16	12:05:00	12:05:04	12:00:20	12:05:12
12:00:00	00:00:18	12:05:00	12:05:04	12:00:22	12:05:13
12:00:00	00:00:20	12:05:00	12:05:04	12:00:24	12:05:14
12:00:00	00:00:22	12:05:00	12:05:04	12:00:26	12:05:15
12:00:00	00:00:24	12:05:00	12:05:04	12:00:28	12:05:16

Table 8.2: Table showing the expected calculated correct clock based on the latency between the nodes, network structure shown in figure 8.7

8.3 Validation Results

8.3.1 General Network Results

8.3.1.1 Path Discovery and Path Acceptance during Starting Phase

After running the tests and comparing the results with the sink paths we expected to see, shown in table 8.1, we then saw that although the paths were not in the same order, the paths that were generated while in the starting phase of the system's life-cycle were themselves identical.

Based on this result we can conclude that the path discovery, path acceptance, and path sharing is working as intended for this particular configuration, when starting the network with a pre-configured sink. Although not shown here, while developing and debugging the simulator we have also tested all the configurations that we have made for it, and have found the same to be true for all of them.

8.3.1.2 Path Discovery and Path Acceptance During Operational Phase

After running the tests and comparing the results with the sink paths we expected to see, shown in table 8.1, we then saw that although the paths were not in the same order, the paths that were generated while in the operational phase of the system's life-cycle were themselves identical.

Based on this result we can conclude that the path discovery, path acceptance, and path sharing is working as intended for this particular configuration, when promoting a node to a sink while in the operational phase. While developing and debugging the simulator we have also tested all the configurations that are available in the simulator, and while not shown here, have found the same to be true for all of them.

8.3.1.3 Optimal Parent Calculation

By running the test and observing the transmissions and debug updates from the nodes, we were able to create table 8.3 to show the sequence of events and what the nodes estimated the potential parent's scores to be at the given times. Please remember that a lower score is better than a larger one, and that these nodes are in the configuration shown in 8.2. The table shows that node 3 estimated the scores of its potential parents to be 0.39-0.43 for node 1 and

Transmission	Transmission Reason	Parent	Score Node 1	Score Node 2
1	Time-Synchronization	Node 1	0.390813	0.668333
2	Content Propegation	Node 1	0.423120	0.669333
3	Time-Synchronization	Node 1	0.423120	0.702830
4	Content Propegation	Node 1	0.430342	0.702830
EVENT OCCURED SETTING NODE'S BATTERY TO 30%				
5	Time-Synchronization	Node 1	0.430342	0.702830
6	Content Propegation	Node 2	1.494231	0.702830
7	Time-Synchronization	Node 2	1.494231	0.708385
8	Content Propegation	Node 2	1.494231	0.715608

Table 8.3: Table showing the optimal parent results. This table is shared with Erlend Karlstrøm

0.69-0.70 for node 2. After the reduction in battery caused node 1's score to spike to 1.49, node 3 was able to calculate that node 2 – with its 0.70-0.71 – was the better option and select it as its preferred parent instead.

This shows that the nodes are able to incorporate changes into their score estimates – changes which they extract from the metadata of requests that they have exchanged with their parents. In this experiment we saw that node 3 updated its information one cycle after the change, and was able to choose a new parent when the scores skewed in another node's favor.

8.3.1.4 Temporary Connection Disruption and Validating Dead Node Sink Path Removal

After running the tests and recording the sink path lists in the 3 different stages of the network, we were able to create the 3 tables as shown in table 8.4, 8.5, and 8.6.

The first, table 8.4, shows that before inducing the artificial disruption the nodes had all the sink paths that is expected from the configuration shown in

Node	Sink Paths
0	SINK NODE
1	[[0], [3, 2, 0]]
2	[[0], [3, 1, 0]]
3	[[2, 0], [1, 0]]

Table 8.4: Table from temporary connection disruption validation showing the nodes' sink paths pre event. . This table is shared with Erlend Karlstrøm

Node	Sink Paths
0	SINK NODE
1	[[0]]
2	[[0]]
3	[]

Table 8.5: Table from temporary connection disruption validation showing the nodes' sink paths post event. . This table is shared with Erlend Karlstrøm

Node	Sink Paths
0	SINK NODE
1	[[0], [3, 2, 0]]
2	[[0], [3, 1, 0]]
3	[[2, 0], [1, 0]]

Table 8.6: Table from temporary connection disruption validation showing the nodes' sink paths post event fix.
. This table is shared with Erlend Karlstrøm

figure 8.2. The second, table 8.5, shows that after the disruption, and having node 3 being considered dead by nodes 0, 1 and 2, and node 3 considering the others dead from its perspective, the sink paths were reduced to just node 1 and 2 having a single path directly to node 0. The third, table 8.6, shows that after node 3 stopped being disrupted, it was able rejoin the network, and the paths that were present prior to the event were fully restored – for node 3 which lost all its paths, and nodes 1 and 2 who lost their paths through node 3.

From looking at the first table 8.4, and the second table 8.5 we are able to conclude that the system is able to remove the reduntant paths of "known" dead nodes, so that it will not route through them. This means that the system's sink node removal functionality works as intended.

From looking at the second table 8.5, and the third table 8.6 we can also conclude that nodes once believed to be dead, are given the paths that they previously deleted upon rejoining, and that the other nodes can accurately recognize this node as a routing option once again.

8.3.2 Time Synchronization Results

After running the network shown in figure 8.5 for 20 cycles, we were able to create a graph showing how nodes slowly grow from each other, based on the simulated skew(as explained in chapter 7). The graph is shown in figure 8.8

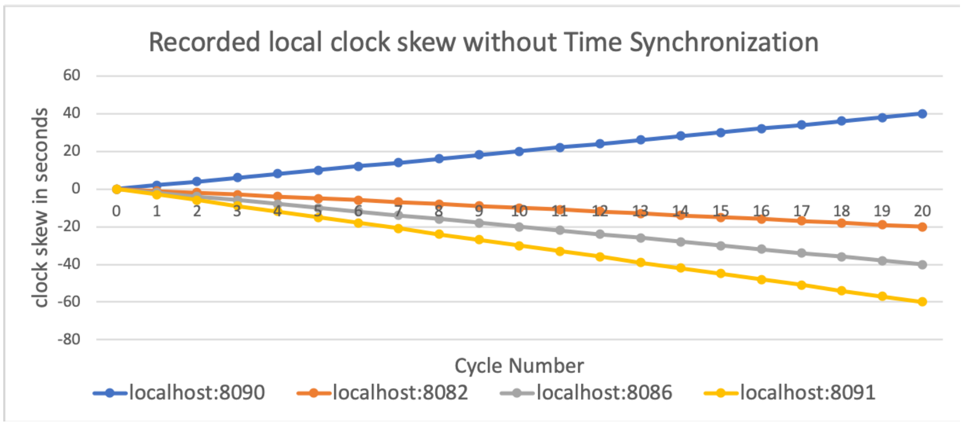


Figure 8.8: Graph showing the recorded skew, when the time synchronization operation is disabled.

Node	Recorded Depth
0	0
1	1
2	1
3	2
4	2
5	1
6	0
7	2

Table 8.7: Table showing the recorded calculated depth for the nodes

By looking at the graph in figure 8.8 we can see that the nodes skew away from each other based on a factor per cycle. This means that the local clock-skew will grow so far from each other so that the nodes will lose connection with each other.

In conclusion, running the system with the time synchronization **disabled** makes the nodes skew away from each other based on the skew factor per cycle, which, in turn, means that the system requires time synchronization to work.

8.3.2.1 Depth Calculation

After running the network shown in figure **??**, and recording the estimated depth of the nodes, we were able to create the table shown in table 8.7.

Calculated Actual TB2	Recorded Actual TB2
12:05:05	12:05:05
12:05:06	12:05:06
12:05:07	12:05:07
12:05:08	12:05:08
12:05:09	12:05:09
12:05:10	12:05:10
12:05:11	12:05:11
12:05:12	12:05:12
12:05:13	12:05:13
12:05:14	12:05:14
12:05:15	12:05:15
12:05:16	12:05:16

Table 8.8: Table showing the recorded time from the correctness check validation for the Immediate algorithm

By inspecting the recorded data in the table shown in 8.7, and comparing it with the network structure containing the node depths in figure 8.6. We see that the nodes can estimate a depth that is considered correct as the nodes are picking the shortest depth possible. Thus, in conclusion, the node's ability to calculate an estimated depth is correct, which, in turn, makes the skewed safe zones correct but necessarily optimal.

8.3.2.2 Correctness Check for Immediate Adjustment Algorithm

After running the network shown in figure 8.7, and recording the calculated time of the A-node based on the different latencies, and static clocks, shown in table 8.2, we were able to create a table of recorded clocks and calculated clocks as shown in table 8.8.

By inspecting the table shown in table 8.8, we can see that the calculated time and the recorded time were the same. This means that even in the case of latencies for up to 22 seconds – Something that is way more than expected – the algorithm was able to remove it.

In conclusion, the immediate adjustment algorithm is able to remove the latency aspect of the requests and able calculate a correct clock, assuming that the latency is divided equally between the transmissions to and from.

9 Experiments

The General Network Experiments, General Network Results, Experiments Setup, and the relevant data are all shared between the author of this thesis, and Erlend Karlstrøm, where both deserve 50% credit for it.

However, the Time Synchronizing Experiment is created only by the author of this thesis. The author, therefore, deserves full credit for it.

9.1 Experimental Setup

All the following experiments were done with the same setup as in chapter 8, section 8.1.

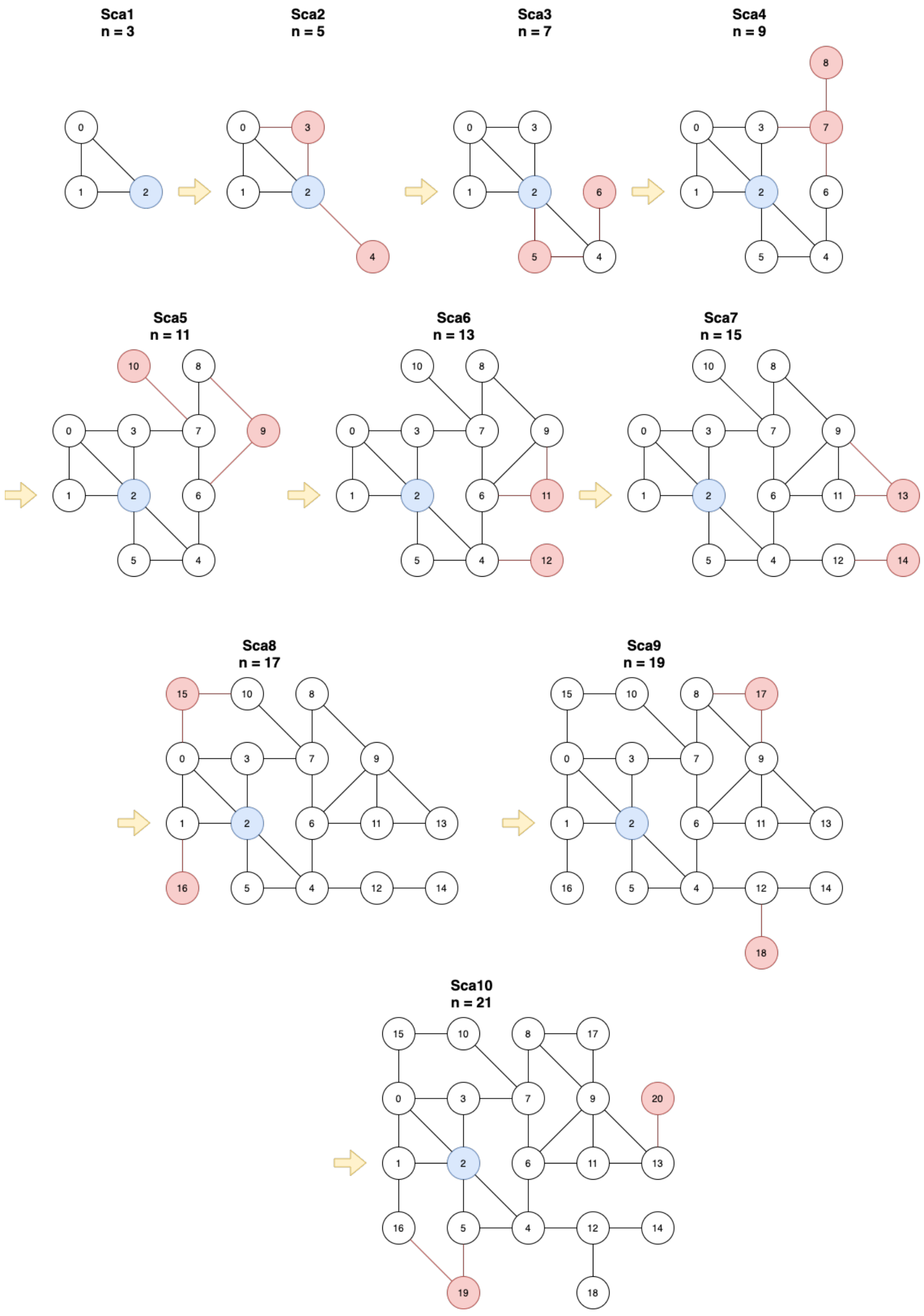


Figure 9.1: Figure showing a set of network topologies topologies used in the experiments.

9.2 Experiments Design

For the experiments, we will be using a set of network configurations which can be seen in 9.1. Each configuration has been procedurally generated by following a set of rules, which are as follows:

- Start with a triangle (This is Sca1)
- Add two nodes, one with two links to the pre-existing nodes, and the other with one link to any node.
  - Repeat this step as needed

This set of configurations will represent a single network that scales progressively.

9.2.1 General Network Experiments

These experiments aim to test the network's stability and capacity to deal with a growing number of nodes in different aspects i.e., the systems' scalability.

9.2.1.1 Sink Path Scaling

This experiment aims to measure how much the number of sink paths grows as the number of nodes increases. More specifically, measure by what degree the volume of local knowledge grows, based on the number of nodes present in the network. This is important to investigate because the list of sink paths is not only added to or removed from, but also frequently iterated over to inform decisions that takes the node's placement in the network into account – such as selecting which path to a sink is the best.

Method: This experiment runs through all the different network configurations shown in figure 9.1, from sca1-sca10. We start the networks in the starting phase, and have all the nodes count the number of paths they have.

9.2.1.2 Path Length Scaling

This experiment aims to measure how much the lengths of paths grow as the number of nodes increases. More specifically, it will find the average length of the shortest paths for all the nodes in a network, and the average length of the longest paths for the same nodes, and show the difference between them. This is important to investigate because the shortest paths are often the most favorable routing options, and seeing how much they scale with the network size is interesting. The longest paths are the least used because of their length, but they take up the most space in the path list, and require the most communication to propagate through the network, as every step in a path essentially equals one transmission.

Method: This experiment runs through all the different network configurations shown in figure 9.1, from sca1-sca10. We start the networks in the starting phase, and have all the nodes report back the length of their shortest path, and the length of their longest path. These values are then averaged into a representation for the network as a whole. When this has been done for all the network configurations, they are then compared against each other.

9.2.1.3 Transmissions During Starting Phase

This experiment aims to measure how many transmissions were sent and received during the starting phase, with a growing number of nodes. More specifically, measure how many transmissions were needed to establish and stabilize the network, from initialization to full operation, and how does this amount grow for the different network configurations. This is important to investigate because the starting phase will be the space of time where the network experiences the most activity in terms of transmissions in the least amount of time. Since high energy consumption has been so closely linked with wireless transmissions, in this case, poor scalability may be a contributing factor to high energy usage during the starting phase.

Method: This experiment runs through all the different network configurations shown in figure 9.1, from sca1-sca10. We start the networks in the starting phase, and have all the nodes record all the transmits done, which include all sends, and all successful receives. These transmit numbers are added together, which gives the total amount of transmissions during the starting phase across all nodes.

9.2.1.4 Transmissions During New Sink Operation

This experiment aims to measure how many transmissions were sent and received during the operation of adding a sink into the network and sharing the paths to that sink, as the number of nodes in the network grows. This is important to investigate, because the act of adding a new sink, generates a lot of messages which needs to be propagated across the whole network, and checking how the amount of transmissions grows based on a network size will then show how the sink path sharing scales on its own.

Method: This experiment runs through all the different network configurations shown in figure 9.1, from sca1-sca10. However, with one modification: the networks does not contain a sink initially. After the starting phase has finished, we send a message to a node, converting it into a sink. We then record how many transmissions this event caused to be generated for each network configuration.

9.2.1.5 Transmissions Normal Operation

This experiment aims to measure how many transmissions were sent and received during the normal operations done in the operational phase. This is important to investigate because it will show how the starting phase – characterized by stabilization and path sharing – contrasts with the communication pattern in the operational phase, such as time synchronization and data propagation. It also shows that as long as the network remains stable, transmissions will remain close-to linear and will not show any spikes – which would be indicative of moments when the network needs to recalibrate, such as when a new sink is added and new paths needs to be propagated.

Method: This experiment runs through a selection of the different network configurations shown in figure 9.1, using sca2, sca4, sca6, sca8, and sca10. We start the networks in the starting phase, and record all transmissions done for 15 awake cycles(including the starting phase as the 0th cycle). No special actions are performed, and the network is allowed to run normally.

9.2.2 Time Synchronization Experiments

These experiments aim to test the recoverability using the time synchronization operations and check how the algorithms deal with a growing number of nodes, i.e., the time synchronizations' scalability.

9.2.2.1 Skew with Immediate Adjustment Algorithm

This experiment aims to measure how the immediate time synchronization algorithm is able to correct the node's local clock-skew back towards a close-to-zero skew after the simulated time-skew has happened. This is important to investigate because, as explained in the design chapter 6, the immediate adjustment needs to "adopt" the parent's local time. The aspect of remaining synchronized with the other nodes is also one of the most important contributions of this system as it allows for data to be propagated and sleep schedules to be synchronized.

Method: Using the network shown in figure 8.5. Start the network and let the system run for 20 cycles. At the start of each awake cycle, we record the node's local time to get the skew after the duty cycling, then record the node's local time again after the awake cycle to get the skew correction. By recording two timestamps per cycle, it is possible to see how much the node skewed and how much it recovered from the skew.

9.2.2.2 Skew with Gradual Adjustment Algorithm

This experiment aims to measure how the gradual time synchronization algorithm is able to correct the node's local clock-skew back towards the average time of the network. This is important to investigate because, as explained in the design section, the gradual adjustment algorithm calculates the global average of the neighborhood and gradually skews the clock toward that time, disregarding the "real"-time. It is also important to investigate due to the aspect of remaining synchronized even though no sinks exist in the network.

Method: Using a variation of the network shown in figure 8.5, except the network does not contain any sinks. Start the network with no sinks, and let the system run for 20 cycles. At the start of each awake cycle, record the node's local time to get the skew after the duty cycling, then record the node's local time again after the awake cycle to get the skew correction. By recording two timestamps per cycle, it is possible to see how much the node skewed and how much it was able to recover from the skew.

9.2.2.3 Immediate- and Gradual Adjustment Transmission Amount

This experiment aims to measure how the immediate algorithm and gradual algorithm compare to each other based on the amount of transmission overhead it adds to the network while also testing the consequences of having the number of nodes scale up. This is important to investigate because the difference in the number of transmissions between the algorithms is relevant to the longevity of the network.

Since this is a two-part experiment that uses two different adjustment algorithms, they also require two different networks. While the structures of the two networks remain the same, the gradual algorithm requires that there are no sinks in the network.

Method – Test 1 Immediate algorithm: This experiment runs through a selection of the different network configurations shown in figure 9.1, using sca2, sca4, sca6, sca8, and sca10. We start the networks in the starting phase and record all transmissions marked as time synchronization for 20 awake cycles(including the starting phase as the 0th cycle).

Method – Test 2 Gradual algorithm: This experiment runs through a selection of the different network configurations shown in figure 9.1, using sca2, sca4, sca6, sca8, and sca10, except the configurations does not contain any sink nodes. We start the networks in the starting phase and record all transmissions marked as time synchronization for 20 awake cycles(including the starting phase as the 0th cycle).

9.2.2.4 Total Increase of Transmissions with Time Synchronization Algorithms

This experiment aims to measure what impact that the different time synchronization algorithms have on the total amount of the transmissions in the network while running in their intended environment, i.e., measuring content propagation while running the immediate algorithm, but not when running the gradual algorithm. While also testing what sort of impact they have when the network is scaling up with an increasing number of nodes. This is important to investigate because measuring the total amount of transmissions while having the different algorithms run in their intended environment, we get to see what impact that the different algorithms have upon the system as a whole.

Similar to the previous experiment, this experiment tests the two different algorithms in their intended environment. Therefore the tests require two different network structures. While the structures of the two networks remain the same, the gradual algorithm requires that there are no sinks in the network. The data that can be read is also different as their intended environment has different data propagating through the network.

When testing the gradual adjustment algorithm, the network cannot contain a sink node. This means that the content propagation transmissions will not exist, but only time synchronization messages and management messages will.– management messages is a group term, which contains time synchronization messages and other messages.

When testing the Immediate algorithm, the network is then required to contain at least one sink node. This means that the network will also contain content messages along with management and time synchronization messages.

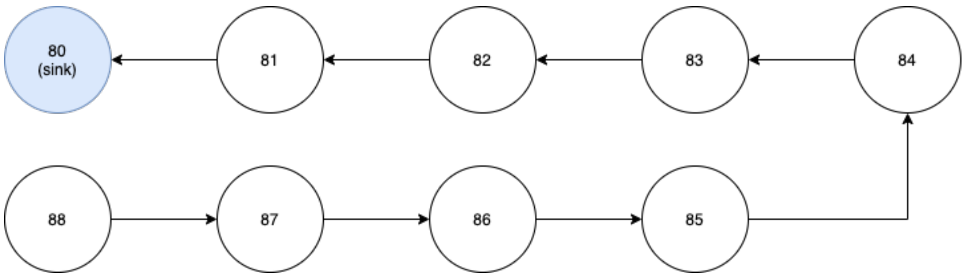


Figure 9.2: Figure showing a network structure used for testing accuracy of the time synchronization with an increasing depth of the nodes

Method – Test 1 Immediate algorithm: This experiment runs through a selection of the different network configurations shown in figure 9.1, using sca2, sca4, sca6, sca8, and sca10. We start the networks in the starting phase and record all transmissions marked as time synchronization, management, and content for 20 awake cycles(including the starting phase as the 0th cycle). These transmissions are all added together, based on their category, dependant on the cycle that the nodes were at.

Method – Test 2 Gradual algorithm: This experiment runs through a selection of the different network configurations shown in figure 9.1, using sca2, sca4, sca6, sca8, and sca10, except the configurations does not contain any sink nodes. We start the networks in the starting phase and record all transmissions marked as time synchronization and management for 20 awake cycles(including the starting phase as the 0th cycle). These transmissions are all added together based on their category, dependant on the cycle that the nodes were at.

9.2.2.5 Accuracy with Immediate Adjustment

This experiment aims to measure how accurately the immediate adjustment algorithm is to adjust the node's clock towards the calculated parent's clock based on an increasing depth. We need to form a network that has only one sink and a lot of nodes in different calculated depths. This is important to investigate because a direct connection to a sink cannot be guaranteed, which means that fetching an updated time needs to be done through other nodes, and checking how accurately it is, is critical for network longevity.

By measuring the accuracy of the immediate algorithm, we can also see how well the safe zone calculation is working, as the main focus of it is trying to fetch the clock after it has already been updated at the parent node.

Method: Using the network structure shown in figure 9.2. We start the network and let the system run for 20 cycles. At the start of each awake cycle, we record the node's local time to get the skew, then record the node's local time again after the awake cycle to get the skew correction. By recording two timestamps per cycle, it is possible to see how much the node skewed and how much it was able to recover from the skew.

9.2.2.6 Reduced Immediate Time Synchronization Operation

This experiment aims to measure how many cycles between the immediate time synchronization operation that the system is able to handle. This is important to investigate because checking how much reduction of the time synchronization operation that the system is able to handle could lead to increased network longevity as it is possible to decrease the number of synchronization transmissions needed.

Method: Using the network network shown in figure 8.5. We start the network and let the system run for 20 cycles. At the start of each awake cycle, we record the node's local time to get the skew, then record the node's local time again after the awake cycle to get the skew correction. By recording two timestamps per cycle, it is possible to see how much the node skewed and how much it was able to recover from the skew. We run the same test multiple times while reducing the amount of the time synchronization operation, starting at one synchronization per cycle – which is the default amount – and reduce it incrementally until one synchronization per 9th cycle.

9.3 Experiment results

9.3.1 General Network Results

9.3.1.1 Sink Path Scaling

After running the experiments while using the scalability testing configurations shown in figure 9.1, we were able to create a graph containing the total amount of sinks in the network at those specific configurations, shown in figure 9.3.

By looking at the graph in figure 9.3, we can see that by increasing the number of nodes in the system, which creates new possible paths, they end up growing exponentially. This means that while the network size is small, the path-sharing and path acceptance algorithm works well, but when the network grows larger, the number of paths will grow exponentially, rather than run proportionally to

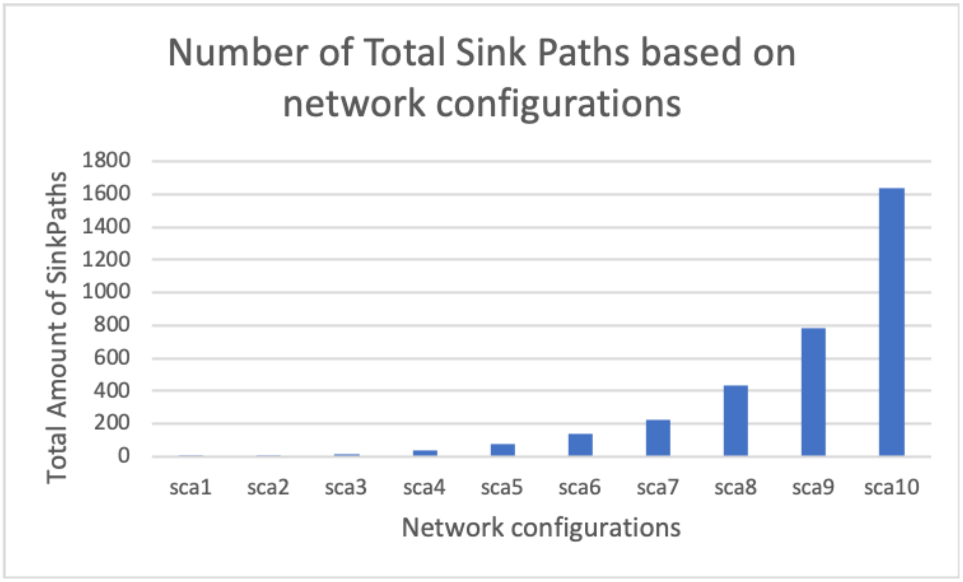


Figure 9.3: Graph showing the total amount of sink path growth when scaling the network. This graph is shared with Erlend Karlstrøm

the number of nodes as it grows linearly.

In conclusion, the path-sharing and path acceptance works well while the network remains small or has few to no cycles in the topology. However, it scales poorly to larger networks, containing multiple cycles. Considering that WSNs often scale to thousands of nodes, this poor scaling in the space of just 3 to 21 nodes is rather severe.

9.3.1.2 Path Length Scaling

After running the experiments while scaling through the scalability testing configurations shown in figure 9.1, we were able to create a graph containing the averages of the longest paths and the shortest paths for all the network configurations. The graph is shown in figure 9.4.

By looking at the graph in figure 9.4, we can see that while increasing the number of nodes in the system, the average shortest path remains at approximately the same length, with a minimal linear growth, while the average longest path is growing more dramatically, but also linearly. This means that while the network size is growing, the shortest available path will increase far less on average than the ones at the other end of the spectrum, which in turn means that the paths that are most relevant for routing are growing the least, while the ones least relevant are growing the most.

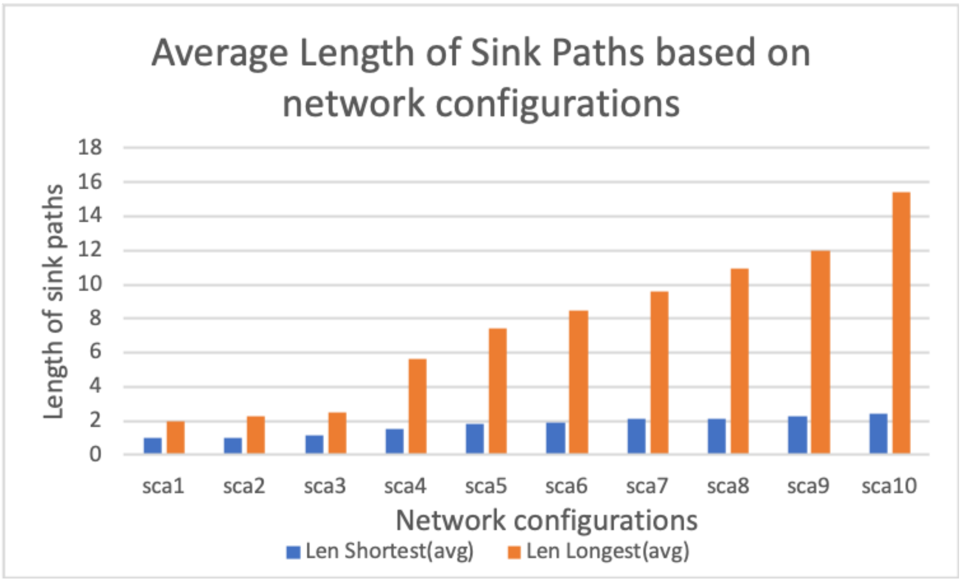


Figure 9.4: Graph showing the average length of the shortest and the longest path when scaling the network. This graph is shared with Erlend Karlstrøm

[Possible disc topic: Jumps between sca3 to sca4 and from sca9 to sca10. Possible reason: Topological cycles]

In conclusion, the shortest paths will, on average, grow minimally for each node – at least according to the experiment. We may also note on the other hand, that the paths shared between the nodes may contain so many steps – and keeping in mind that these are the longest ones, and thus least favored for routing – that they amount to little more than simply taking up a significant amount of space and processing time. While their presence does provide a guarantee of absolute redundancy – in that all paths are maintained as long as they are not deemed useless, which means that as long as a theoretical path exist in the network the nodes will know about it – it is not so obvious from the authors' point view whether or not this is a worthwhile trade-off.

9.3.1.3 Transmissions During Starting Phase

After running the experiments while scaling through each of the scalability testing configurations shown in figure 9.1, we were able to create a graph containing all the transmissions between nodes during the starting phase. The graph is shown in figure 9.5.

By looking at the graph in figure 9.5 we can see that while increasing the number of nodes in the system, the total amount of transmissions grows linearly with

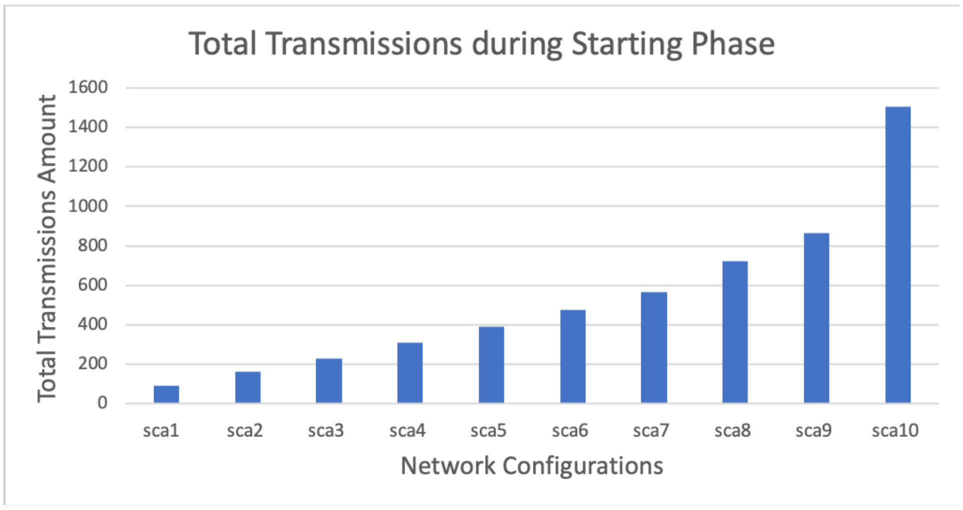


Figure 9.5: Graph showing the amount of transmissions during the starting phase when scaling the network. This graph is shared with Erlend Karlstrøm

small increase between each configuration, except a big jump between sca9 and sca10, which we suspect is because of node 19 forming another topological cycle with node 16 and node 5. The pathing opportunity which is subsequently created needs to be propagated to the rest of nodes, and will be propagated to all of them, because it is on the periphery of the sink and can therefore form a detour for several existing paths. This will therefore result in several new longer paths which uses node 19 as a step.

In conclusion, increasing the size of the network does not scale in exactly the same way as the number of transmissions needed during the starting phase. It will mostly scale linearly with the network size and the number of nodes itself is not necessarily indicative of how many transmissions are needed to complete the initial handshakes and path sharing. However, when topological cycles are added into the network, like in sca10's case, the transmissions needed for sharing the paths will increase significantly, because it creates a new route through an "older" part of the network.

9.3.1.4 Transmissions During New Sink Operation

After running the experiments while scaling through each of the scalability testing configurations shown in figure 9.1 we were able to create a graph containing all the transmissions between nodes during a new sink operation while running in the operational phase. The graph is shown in figure 9.6.

By looking at the graph in figure 9.6 we can see that while increasing the

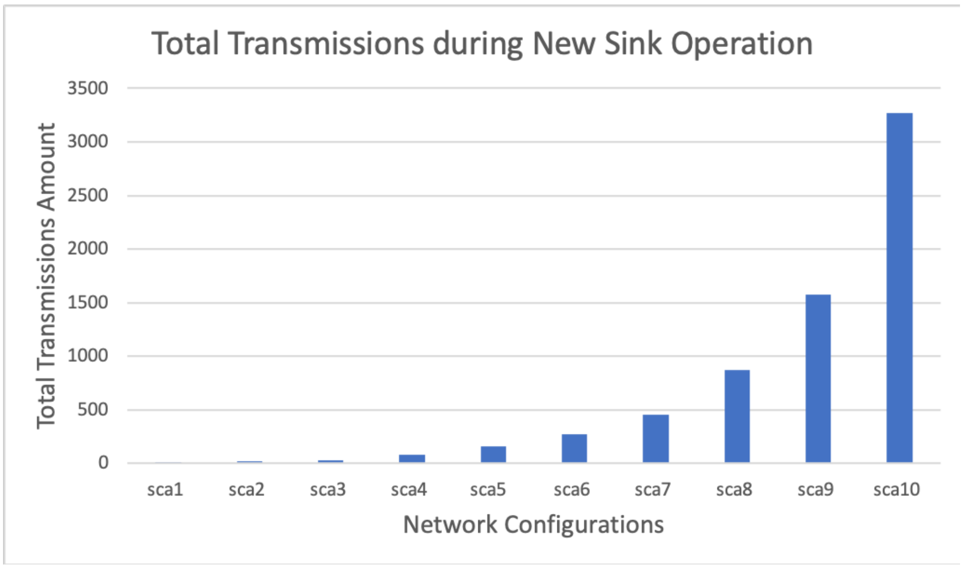


Figure 9.6: Graph showing the amount of transmissions during the new sink operation while running in the operational phase when scaling the network. This graph is shared with Erlend Karlstrøm

number of nodes in the system, the total amount of transmissions required when adding a new sink to the network, while running in the operational phase, grows exponentially with the number of nodes. The reason for this exponential growth is likely because of the way in which paths are shared in the operational phase. Contrary to the starting phase where paths are shared along with handshakes, and not their own individual events as in the operational phase . We can see that this growth is consistent with the earlier experiment in section 9.3.1.2 where, the number of sink paths generated in the network was investigated – the growth there was also exponential.

In conclusion, introducing a new sink node in the operational phase, will scale exponentially with the number of nodes. Thus, introducing a sink node in the operational phase instead of the starting phase will reduce the network's overall longevity because of the increase in transmissions. Hence, introducing a sink node in the starting phase(like in the previous experiment) and holding it connected for the duration of the networks deployment cycle, would be more beneficial compared to introducing it while in the operational phase.

9.3.1.5 Transmissions Normal Operation

After running the experiments, for 15 awake cycles, while scaling through a selection of the scalability testing configurations shown in figure 9.1, we were

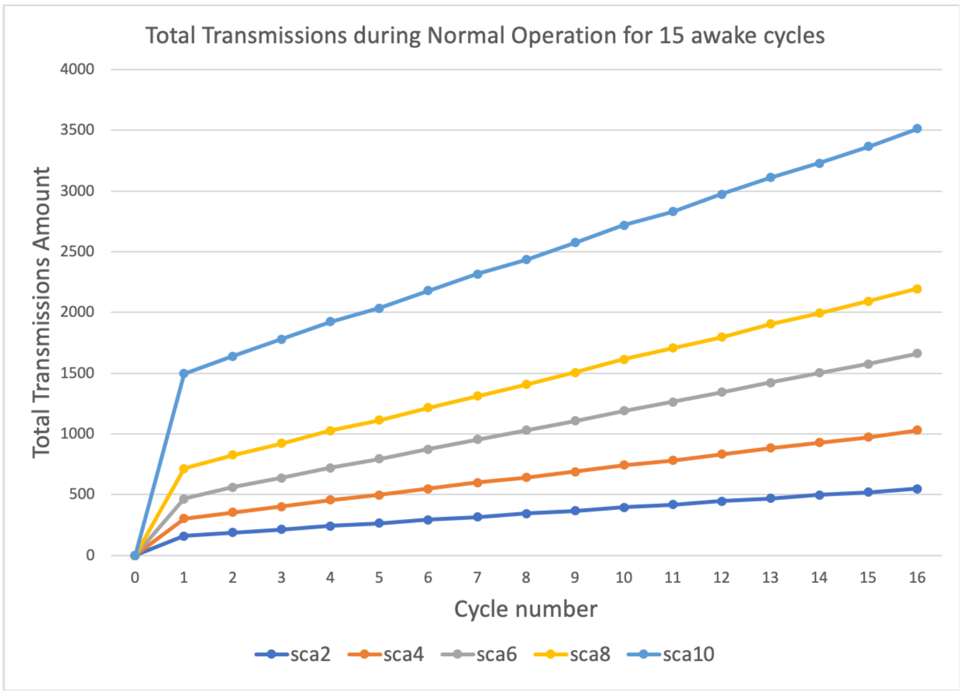


Figure 9.7: Graph showing the amount of transmissions during 15 awake-cycles while running in the operational phase when scaling the network. This graph is shared with Erlend Karlstrøm

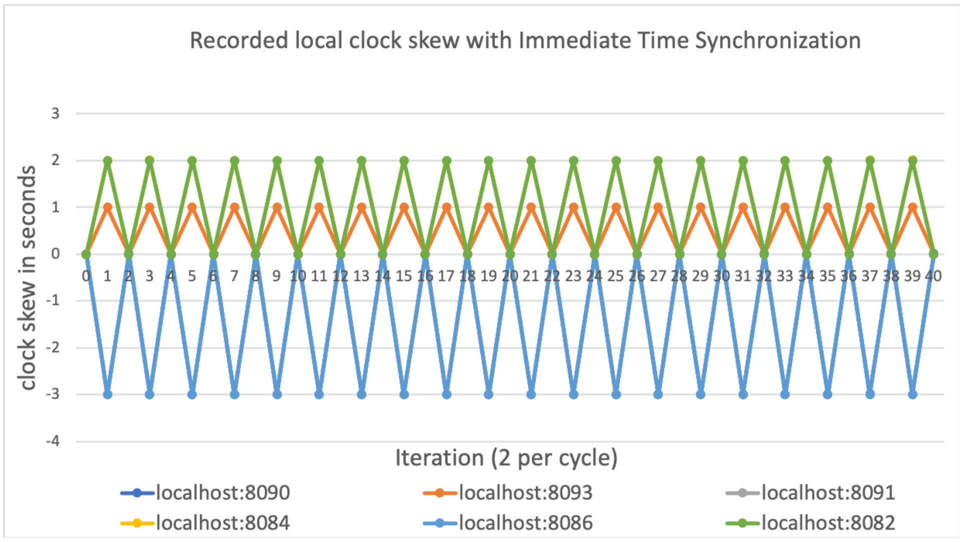


Figure 9.8: Graph showing the recovery of the skew with the immediate adjustment algorithm.

able to create a graph containing all the transmissions between nodes during the normal operations in the operational phase. The graph is shown in figure 9.7.

By looking at the graph in figure 9.7 we can see that while increasing the amount of nodes in the system, the transmissions from starting phase (between cycle 0 and 1), the transmissions increased sharply with the number of nodes, particularly with sca10. Further, we can see that the transmission amount increases steadily in a linear fashion, but also that the growth factor seems to be somewhat influenced by the number of nodes, sca6 increases more across time than sca4 for example, but not by much.

In conclusion, running the system after the starting phase, shows that the growth in transmissions is linear and the factor growth is higher the more nodes are present. The is to be expected, as naturally, more nodes means more activity in the network.

9.3.2 Time synchronization Experiment Results

9.3.2.1 Skew with Immediate Adjustment Algorithm

After running the network shown in figure 8.5 for 20 cycles, we were able to create a graph showing how the nodes are able to recover from the skew that happens between each awake cycle. The graph is shown in figure 9.8.

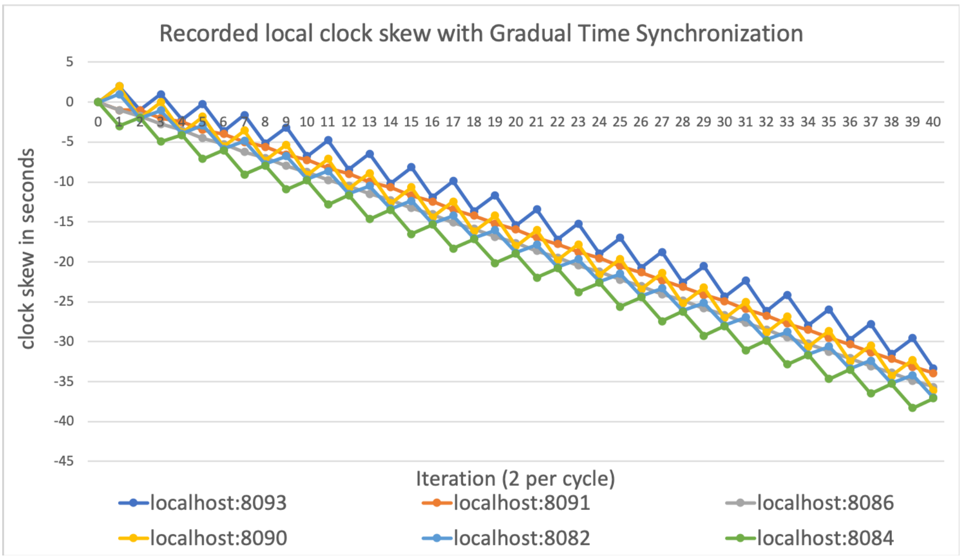


Figure 9.9: Graph showing the recovery of the skew with the gradual adjustment algorithm.

By looking at the graph in figure 9.8 we can see that while running the system normally in a configuration with a sink, the nodes are able to recover from each skew. Further, we can see that each odd number in the iteration(x-axis) is right before starting the awake cycle, which is recording the skew, and the even number is recording the skew correction after the awake cycle. We can also see that the system is able to correct the skew with near perfection as it is able to set it as close to 0 as possible. In conclusion, using the Immediate adjustment algorithm in a system enables an accurate time synchronization to an authoritative clock, using the homebase's clock as the authoritative time.

9.3.2.2 Skew with Gradual Adjustment Algorithm

After running the network shown in figure 8.5, without the sink node which is shown in the figure, for 20 cycles, we were able to create a graph showing how the nodes are able to recover from the skew that happens between each awake cycle, while not having a sink in the network, and therefore no authoritative time. The graph is shown in figure 9.9.

By looking at the graph shown in figure 9.9, we can see that over time the nodes are not able to maintain the correct clock while staying synchronized with each other. I suspect that this is because of how the gradual algorithm is trying to calculate an offset average based on the neighboring nodes, in which

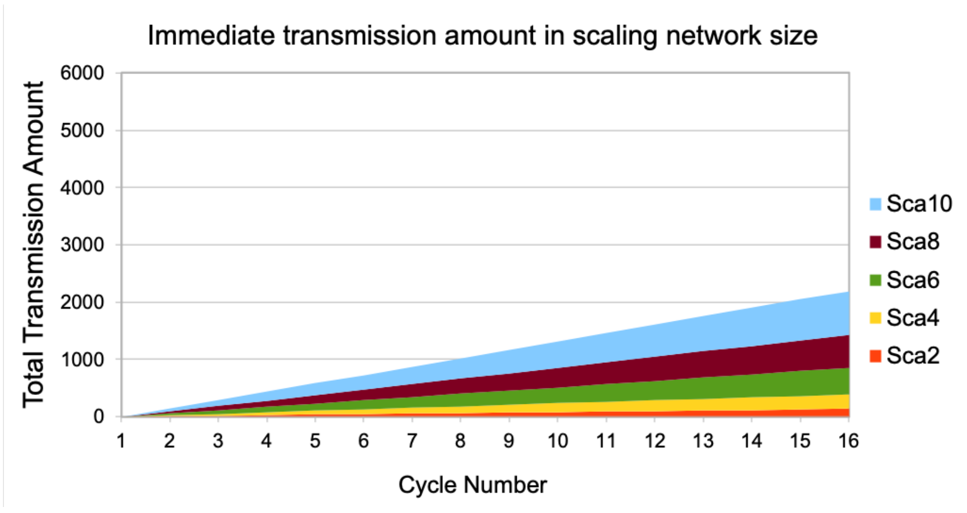


Figure 9.10: Graph showing the amount of transmissions using the Immediate algorithm while running for 15 cycles, and scaling over a set of scalability networks

the majority of the nodes has a negative skew factor, this makes all the nodes calculate a neighborhood average that is negative, and adjusts towards that time. However, the network as a whole is still able to maintain stability in the form of synchronized sleep schedules. We can also see that nodes seem to skew simultaneously in a linear fashion. I believe this is because of the static simulated skew factor and would not necessarily be the same in a real-world scenario.

In conclusion, using the Gradual adjustment algorithm in a system without a sink is **not** able to maintain an accurate clock, just as expected. However, the nodes in the system remain synchronized with each other in the form of a synchronized sleep and wake-up schedule.

9.3.2.3 Immediate- and Gradual Adjustment Transmission Amount

After running the experiments, for 15 awake cycles(Between cycle 1 and 2 are all transmissions for cycle 1, and so on.), while scaling through a selection of the scalability testing configurations shown in figure 9.1, we were able to create two graphs containing all the time synchronization transmissions between nodes during the normal operations in the operational phase. The graph showing the number of transmissions using the immediate adjustment algorithm is shown in fig 9.10, and the graph showing the number of transmissions using the gradual adjustment algorithm is shown in fig 9.11.

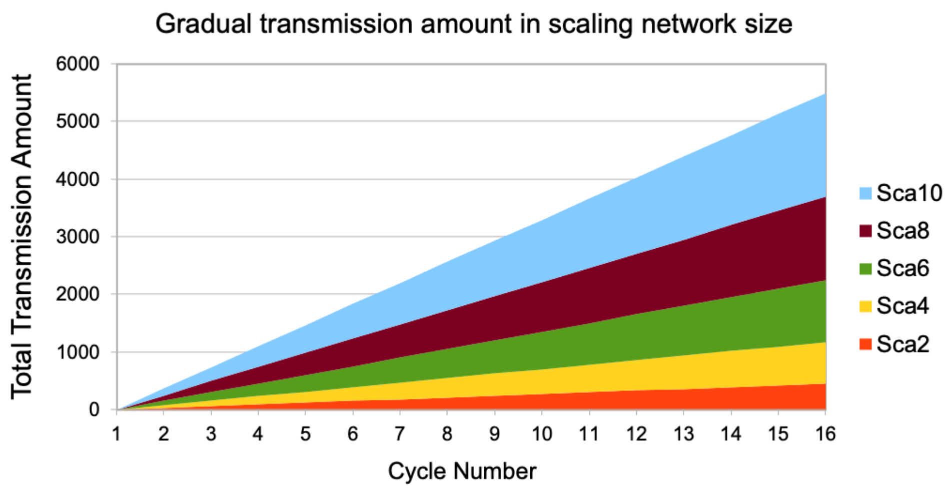


Figure 9.11: Graph showing the amount of transmissions using the Gradual algorithm while running for 15 cycles, and scaling over a set of scalability networks

By looking at the graph showing the immediate adjustment transmissions shown in figure 9.10, we can see that the number of transmissions grows linearly based on the number of nodes in the network, with the transmission amount growing higher with more nodes. I suspect this is due to the skewed safe zone not being entirely perfect, as it is refusing "child" nodes of fetching a non-updated time, making them do it at a later time in the awake cycle. However, this function ensures that the "child" nodes are able to fetch a correct and updated clock.

By looking at the graph showing the gradual adjustment transmissions shown in figure 9.11, we can see that the number of transmissions grows linearly based on the number of nodes in the network. We can also see that it grows a significant amount per cycle, especially when the number of nodes grows. I suspect that this is due to the topological structure shown in the different topologies in figure 9.1. When the number of neighboring nodes grows, so will the transmission amount do, as it is a factor of the number of neighbors per node.

By comparing the two graphs, we can see that the amount of transmissions needed to maintain the network stability is very different. This means that using the gradual adjustment algorithm will significantly decrease the longevity of the network. Thus, in conclusion, using the immediate synchronization algorithm will have the least amount of reduction to the longevity of the network, especially with such interconnected nodes, as shown in the scalability topologies in figure 9.1.

9.3.2.4 Total Increase of Transmissions with Time Synchronization Algorithms

After running the experiments, for 15 awake cycles(between cycle 1 and 2 are all transmissions for cycle 1, and so on.), while scaling through a selection of the scalability testing configurations shown in figure 9.1, we were able to create five graphs(1 for each scalability configuration tested) showing the number of transmissions the time synchronization adds while running in its intended environment. The graphs are shown in figure 9.12 and, figure 9.13.

By looking at the graphs shown in figure 9.12 and, figure 9.13, we can see that all the transmissions grow linearly, with the only difference being the factor that they grow at(linearly after the starting phase, which is between cycle number 0 and 1). We can see that the gradual algorithm has a significantly higher cost at the start of a smaller network but, as the size of the networks grows, the starting phase cost of the gradual algorithm remains similar due to not requiring sink path sharing. However, the gradual algorithm will be more expensive in the long term, even with content propagation being used with the Immediate algorithm.

In conclusion, while using the gradual algorithm ensures that the network is able to remain synchronized, the amount of transmissions added makes a case for not using it. Using the immediate algorithm would ensure a synchronized network(with a correct time) and a smaller footprint of added transmissions. Thus, increasing the longevity of the network and enabling content propagation.

9.3.2.5 Accuracy with Immediate Adjustment

After running the network shown in figure 9.2 for 30 cycles(2 points along the x-axis per cycle), we were able to create a graph showing how a selection of the nodes skew and how they are able to recover from the skew that happens between each awake cycle. The graph is shown in figure 9.14

By looking at the graph shown in figure 9.14, we can see that the nodes are able to recover from the skew up to the maximum tested depth of 8(Where the sink node is depth 0). Since the even number of iterations(on the x-axis) are so close to the 0 – meaning that the time synchronization is accurate. This, in turn, means that the skewed safe synchronizing zones are working as designed but not necessarily optimally, as seen in the previous experiments.

In conclusion, the immediate adjustment algorithm is very accurate up to the tested depth of 8. This, in turn, means that the skewed safe synchronization zones are working as designed, although increasing the transmission amount.

9.3.2.6 Reduced Immediate Time Synchronization Operation

After running the experiments, for 20 awake cycles(2 points along the x-axis per cycle), while increasing the cycle number distance between each time synchronization, from 1 synchronization per cycle to 1 synchronization per 9th cycle. By doing this, we were able to create 9 different graphs showing the skew after sleep and the recovery of that skew. The graphs are shown in figure 9.15.

By looking at the overview of the graphs in figure 9.15 we can see that there is a large difference in the graphs, where the graphs grow higher(or lower, depending on the skew factor), each time before centering at the middle. This is due to the increasing distance of the time synchronization operation. We can also see in figure 9.16, and figure 9.17, that the nodes are able to recover from the skew up to 1 synchronization per 5th cycle. However, after that point, the nodes start to disconnect from the network in the form of losing connection, as the wake-up cycles become too skewed to recover, as seen in figure 9.17 and figure 9.18.

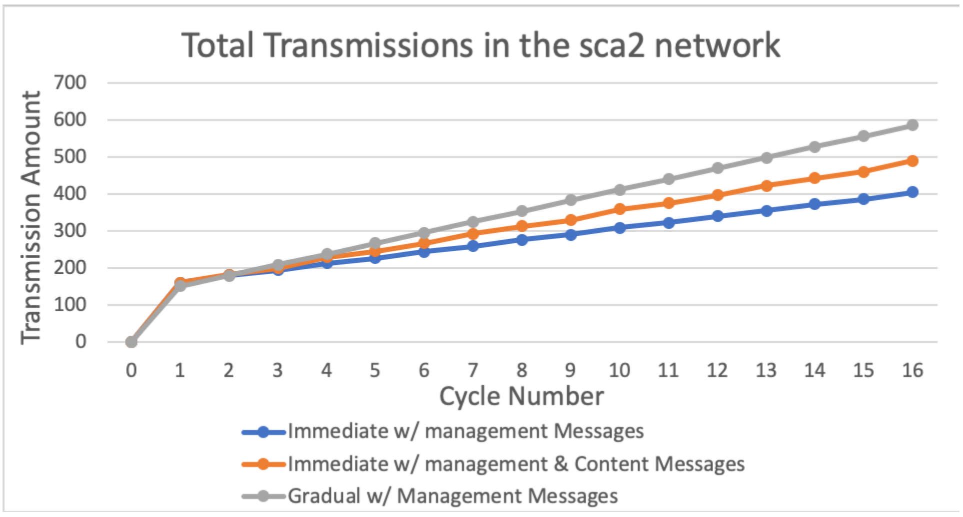
By looking at the graph in figure 9.18, showing 1 synchronization per 7th cycle. We can see that the green and yellow line are moving in a certain direction away from the rest of the nodes together, and we can see that the yellow line tries to synchronize, but as it most likely has lost its connection with its parent, it uses its backup synchronizing method, the gradual adjustment(with its neighbors, which the only neighbor it can reach is the green node). This means that the network has fragmented into multiple smaller networks.

In figure 9.18 we can see the graphs showing 1 synchronization per 8th cycle and 1 synchronization per 9th cycle. In these graphs, we can see that many of the nodes disconnect from the network and are therefore considered dead to the other nodes. This, we believe, is due to the neighboring nodes skewing in different directions compared to the disconnected node.

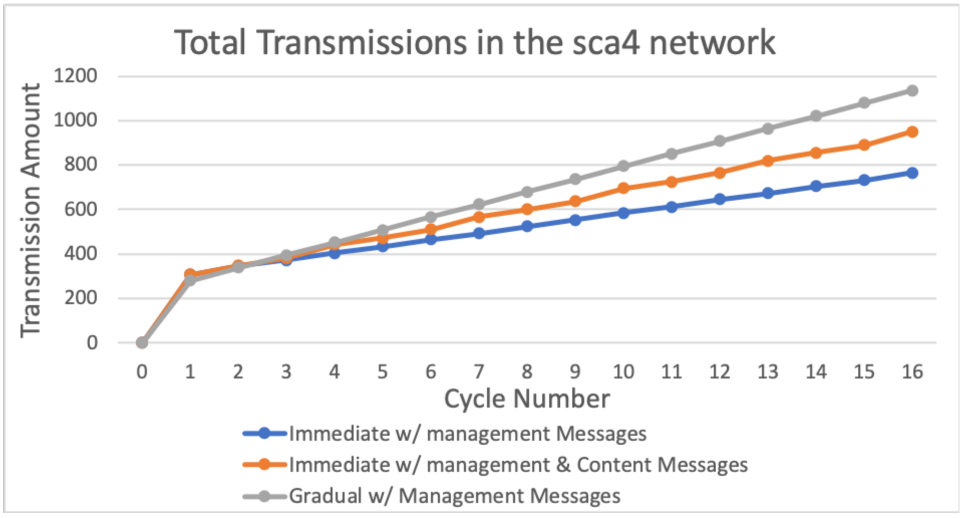
This, in turn, means that the amount of time synchronization operation done can be reduced, up to once per 5th cycle. Thus, allowing us to reduce the footprint of the time synchronization operations by a factor of up to 5 when it comes to transmissions needed to remain synchronized.

In conclusion, the time synchronization is able to handle up to 1 synchronization per 5th cycle, but after that, the nodes will start to disconnect from the network.

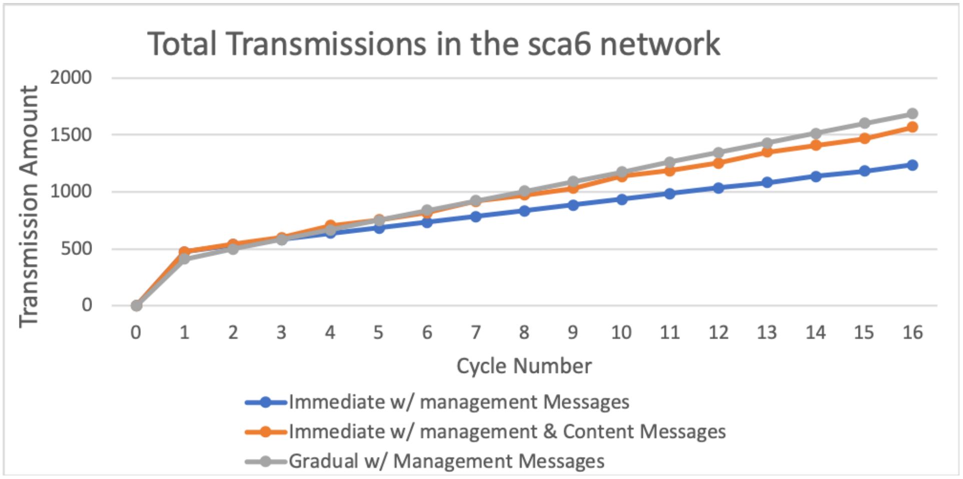
This, in turn, means that the amount of time synchronization can be reduced by a factor of up to 5, which would increase the network longevity.



(a) Graph showing the amount of transmissions using scalability 2 network

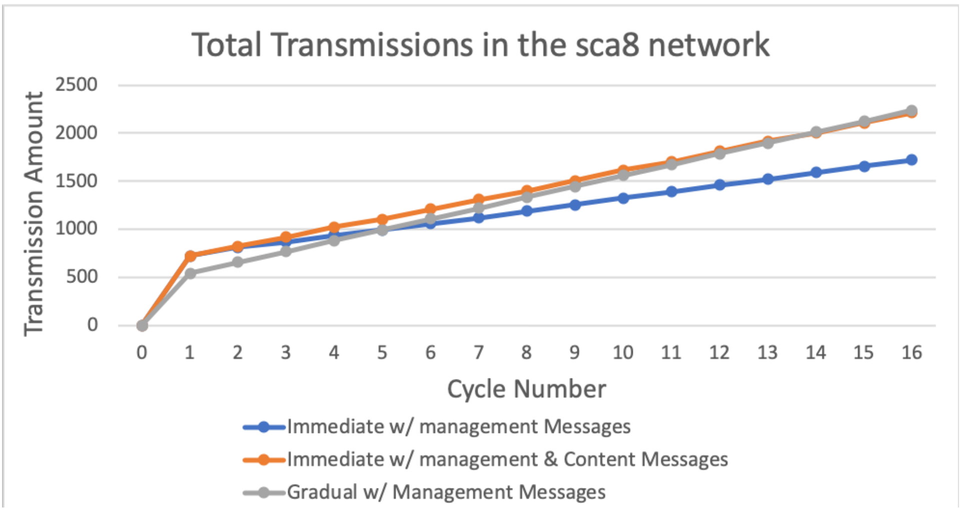


(b) Graph showing the amount of transmissions using scalability 4 network

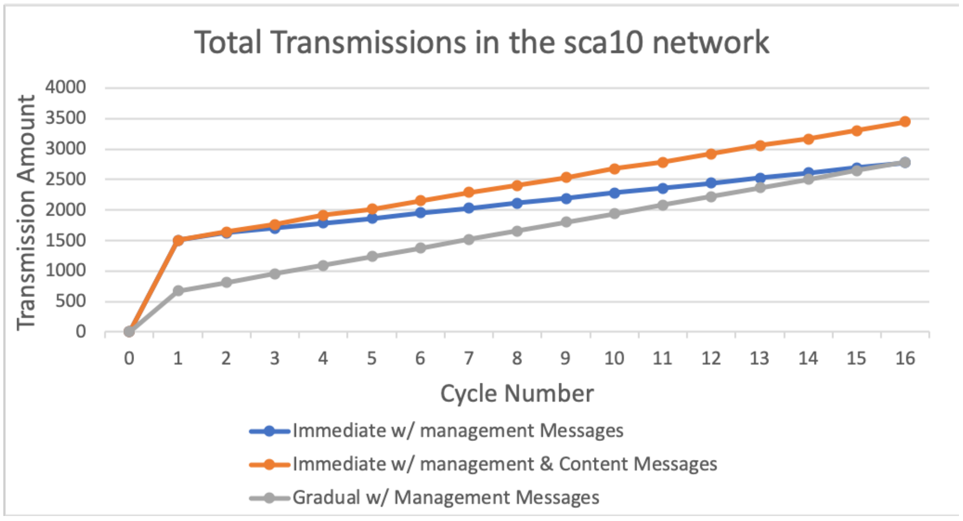


(c) Graph showing the amount of transmissions using scalability 6 network

Figure 9.12: Graphs showing the amount of transmissions using sca 2, 4, and 6 networks from figure 9.1



(a) Graph showing the amount of transmissions using scalability 8 network



(b) Graph showing the amount of transmissions using scalability 10 network

Figure 9.13: Graphs showing the amount of transmissions using scalability 8, and 10 networks from figure 9.1

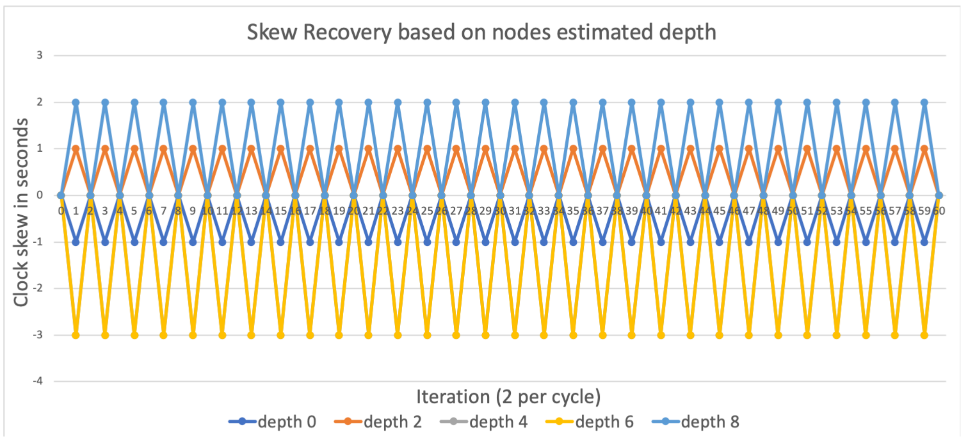
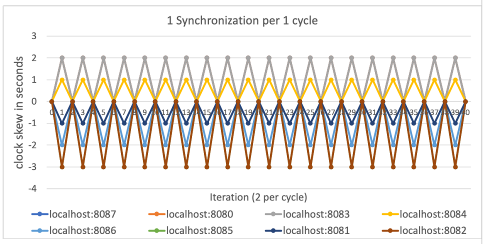
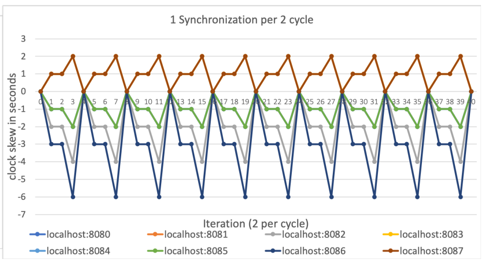


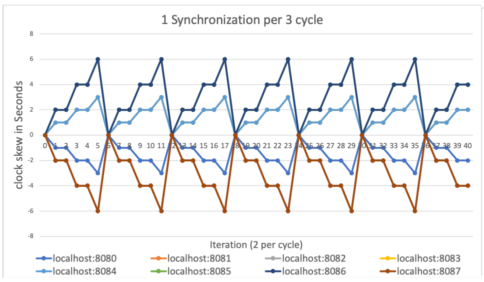
Figure 9.14: Graph showing how accurate the Immediate adjustment algorithm is when the depth of the nodes are growing



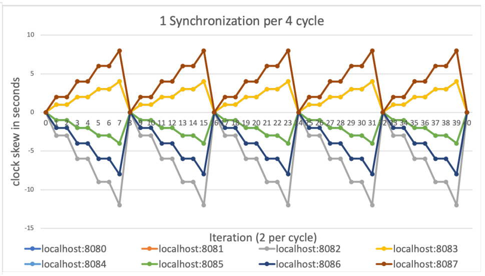
(a) 1 synchronization per 1 cycle



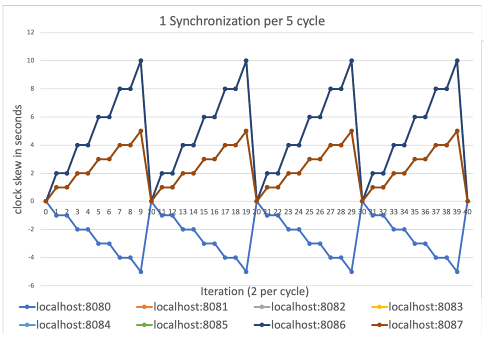
(b) 1 synchronization per 2 cycle



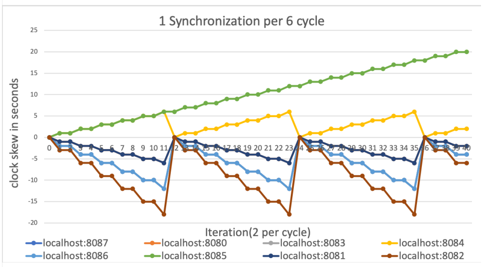
(c) 1 synchronization per 3 cycle



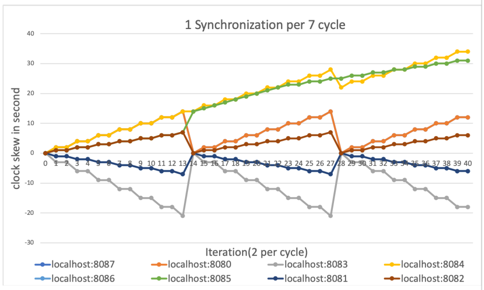
(d) 1 synchronization per 4 cycle



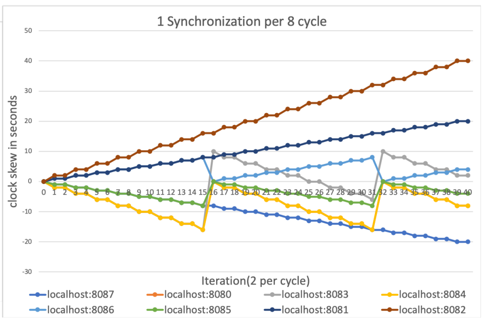
(e) 1 synchronization per 5 cycle



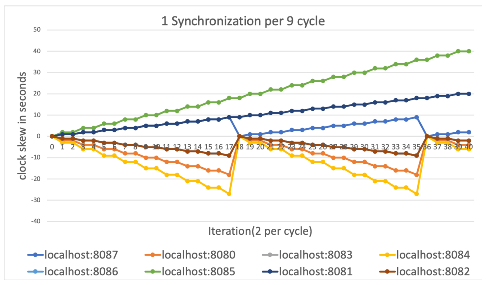
(f) 1 synchronization per 6 cycle



(g) 1 synchronization per 7 cycle

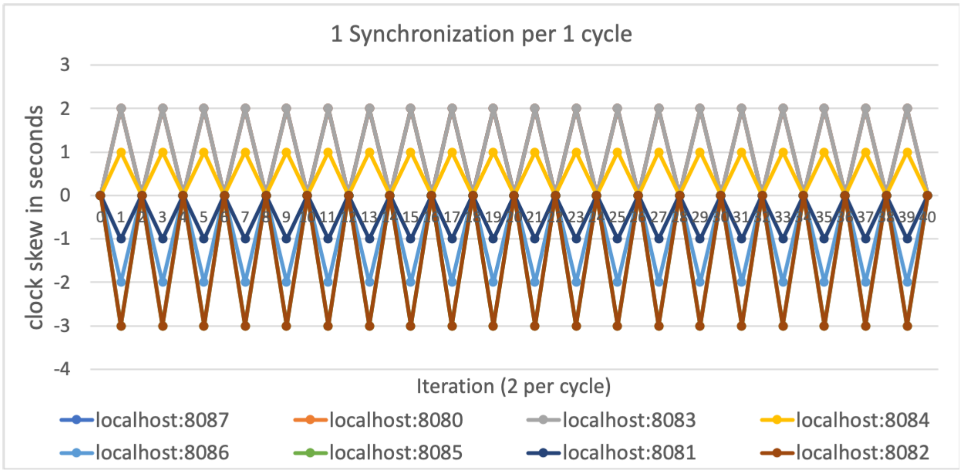


(h) 1 synchronization per 8 cycle

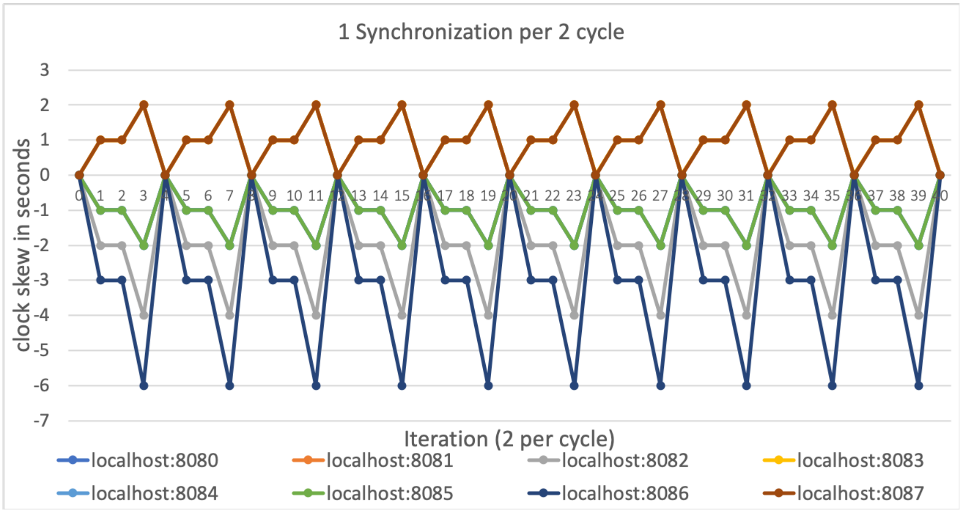


(i) 1 synchronization per 9 cycle

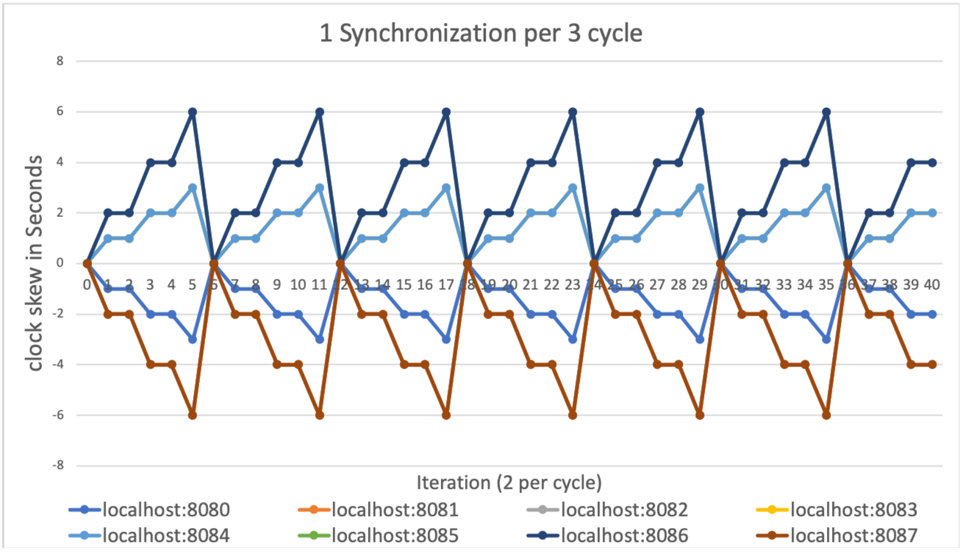
Figure 9.15: Overview of the graphs showing the recovery of the skew when the time synchronization has from 1 to 9 cycles between doing the operation



(a) 1 synchronization per 1 cycle

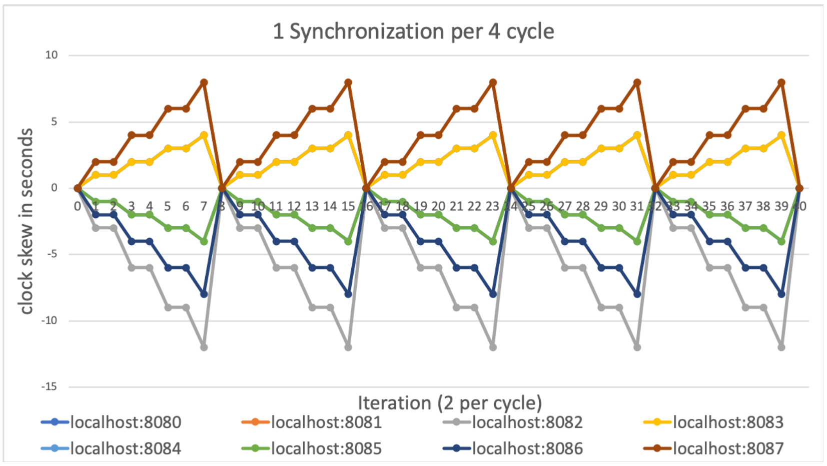


(b) 1 synchronization per 2 cycle

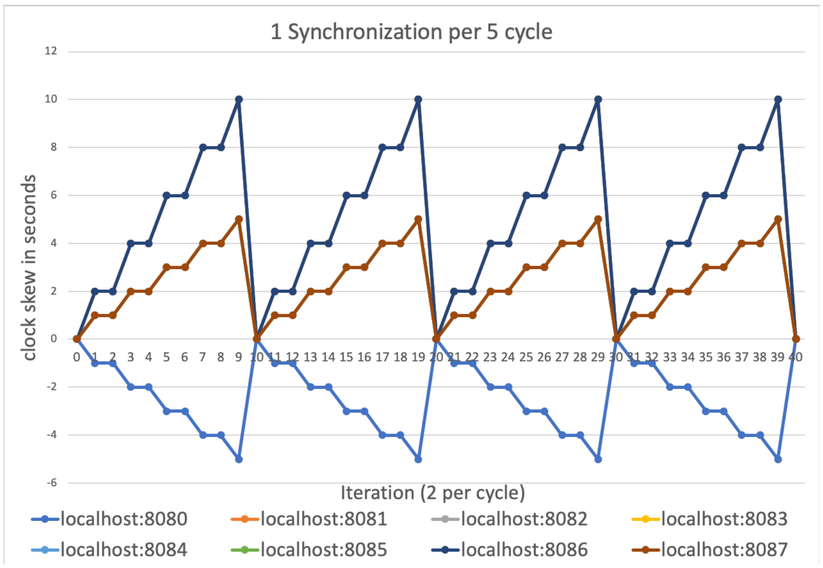


(c) 1 synchronization per 3 cycle

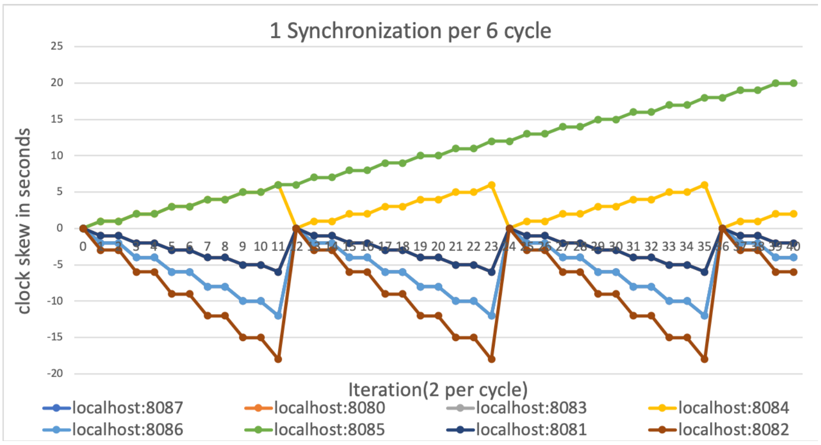
Figure 9.16: Graphs showing the recovery of the skew when the time synchronization has from 1 to 3 cycles between them, and the network fragments



(a) 1 synchronization per 4 cycle

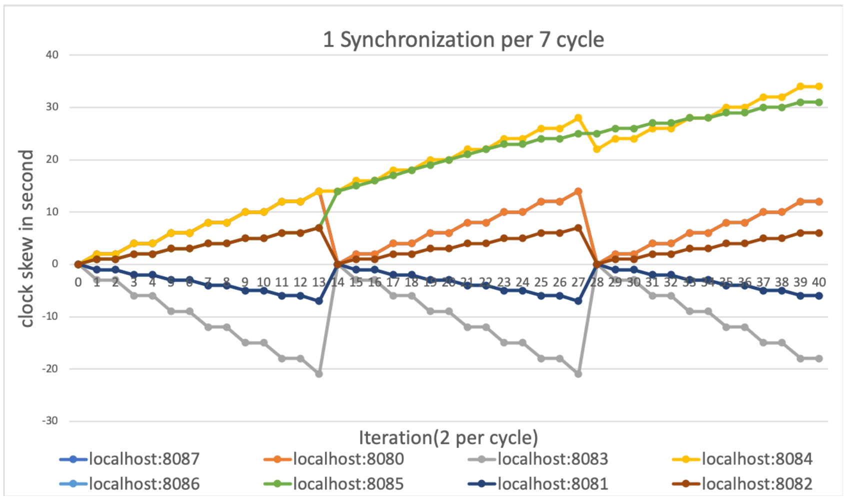


(b) 1 synchronization per 5 cycle

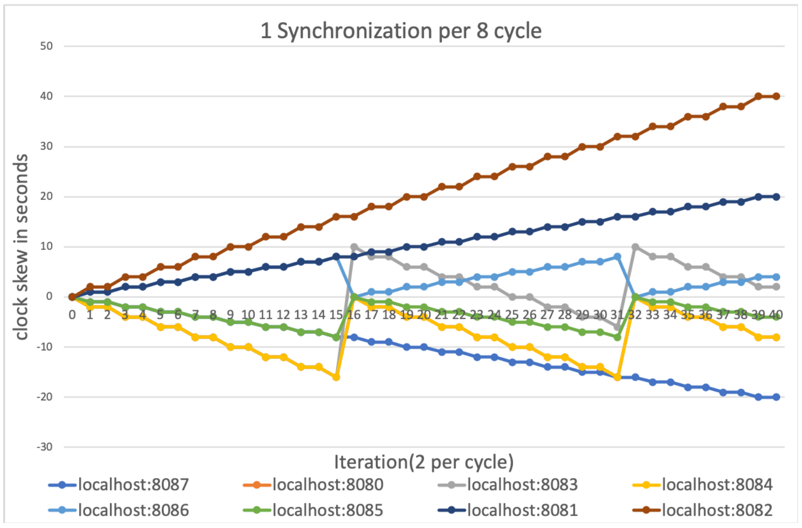


(c) 1 synchronization per 6 cycle

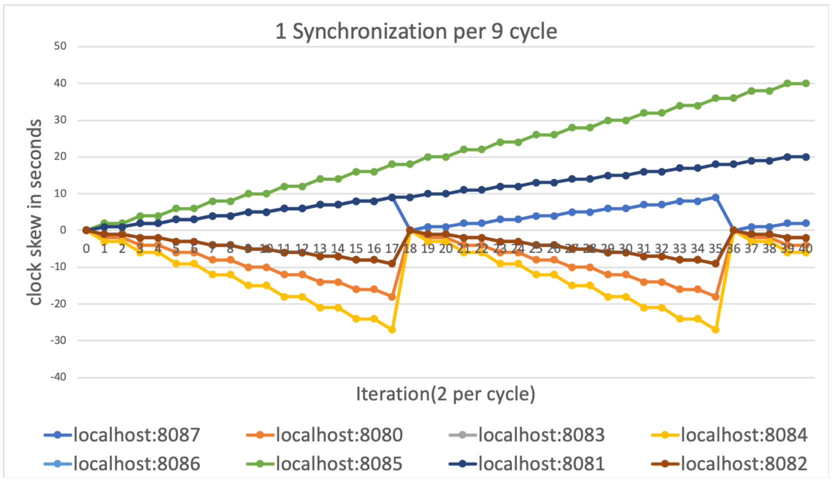
Figure 9.17: Graphs showing the recovery of the skew when the time synchronization has from 4 to 6 cycles between them, and the network fragments



(a) 1 synchronization per 7 cycle



(b) 1 synchronization per 8 cycle



(c) 1 synchronization per 9 cycle

Figure 9.18: Graphs showing the recovery of the skew when the time synchronization has from 7 to 9 cycles between them, and the network fragments

10 Discussion

This chapter discusses some approaches, experiences, how we solved problems, and why we chose the solution that we did.

10.1 General network

10.1.1 Node Communication

Since this system is a simulation and a prototype, the communication technology that is used is WiFi. Using WiFi for development and prototyping purposes works very good, but if this system would be implemented into a "real-world" test-deployment phase and have the system run on specialized hardware, a better choice would be to implement it using LoRa, as it is specialized to fix IoT application challenges such as energy management[27], or a combination of multiple technologies.

10.1.2 Centralized vs. Decentralized Architecture

Currently, the system is implemented as a decentralized system. This means that there are no nodes designated to be a leader of the network. This removes the need for an election algorithm to decide what nodes are allowed to become sink nodes. Having a decentralized system means that we are able to remove overhead transmissions added by an election algorithm and that the system has no single point of failure. However, enabling all the nodes to become sink nodes also increases the overall transmission amount due to the number of paths that need to be shared.

As we saw from chapter 9, the scaling of the system was bad. This means that to be able to say that the decentralized architecture was a good choice would require more work in the form of a better path-sharing algorithm. As it stands now, the system is not scalable in that the nodes are very interconnected, but this will be more outlined in section 10.1.7.

10.1.3 Multi-Hop vs. Single-Hop Routing

When deciding on multi-hop routing or single-hop routing, the choice was easy. As the nodes are deployed in the arctic, they may be so distant that they are not able to contact each other(or at least without spending a lot of energy). This means that a single-hop routing algorithm would not be suitable. – Using a single-hop algorithm would also increase the energy usage for the most distant nodes if they were even able to connect.

Using Multi-hop routing up to a certain point is good, as it load-balances the transmission energy consumption in the network[3]. But when the multi-hop routing ends up routing via a significant amount of nodes, it may end up reducing the total longevity in the network. To enforce the use of multi-hop routing, it might be a good solution to limit the length of the paths, as this will have a reduction in the total amount of hops in a path, and also improve the score-based path choosing, so it chooses more optimal paths.

10.1.4 Path Knowledge vs. Neighborhood Knowledge

Currently, this prototype system is created with neighborhood knowledge. That means that each node has some in-depth knowledge about its neighbors, which, in turn, allows the nodes to propagate to the neighbor that is calculated as the most optimal. Another choice of knowledge sharing is path knowledge(such as presented in the related work chapter, in [34]), which means that the node would know about all the nodes in its paths. However, performing path knowledge sharing will require a lot more overhead communication but will allow the nodes to perform better choices when deciding where to route the content.

Having path knowledge together with a better path-sharing algorithm will allow the nodes to perform better routing choices, which would lead to better load balancing and increased longevity of the network, especially if the path knowledge sharing could be gossiped as metadata along with normal requests.

10.1.5 Using the Prototype with Existing Systems

In the paper, [25] by Raïs et al. they presented some use-cases of a UAV in use with OU's in the arctic tundra, one of which were: the UAV could be used to fly into the deployment area, connect to an OU, and download the data that it has. Using the prototype system presented in this thesis, together with a UAV as a homebase, it could enable nodes to propagate data to a sink node, and directly to the homebase(UAV), especially with the help of adding a data flush routine, to fetch all recorded data, within the same cycle.

10.1.6 Restriction of Neighbor Amount

Currently, each node tries to connect itself with as many neighbors as possible this means that when nodes become sink nodes, if the nodes are very interconnected, they will share their paths to many neighbors, creating a lot of topological cycles. This reduces the scalability of the system significantly.

A way to solve this could be to restrict the number of neighbors that each node is allowed to have. This requires a good algorithm that is able to have the network remain stable and not fragment into smaller networks. This would reduce the transmissions overall.

10.1.7 Scalability Concern

As we saw from chapter 9, the network scaled exponentially, up to 21 nodes(known as sca10). We also saw from the experiments that when the topology changed, creating more topological cycles, the transmission number increased by a significant degree. If we were to be able to create a better solution for path-sharing and be able to reduce the number of paths that each node has, we could create a system that is a lot more scalable. As stated, the number of transmissions is dependant on how many topological cycles that the network has. This, in turn, means that if the network has few topological cycles, it can scale far better as the amount of shared paths would not be increased by a factor dependant on how many nodes the topological cycle tied together.

Based on the background of the project presented in chapter 2, one of the deployment zones had six nodes(seen in figure 2.1) deployed, and from that, we also saw that the nodes were somewhat spread out(not creating topological cycles). However, the number of nodes deployed can be varied by a lot, which means that the system can scale as long as there are a limited amount of topological cycles.

10.1.8 Load Balancing Concern

Currently, the system does not use a built-in load balancing operation, such as LEACH or PEGASIS from chapter 4. However, due to the decentralized aspect of the system, it can rotate the sink node job between the nodes, dependant on which node got a homebase connection. While this does not ensure that the nodes in the network are load-balanced properly, we believe that this together with the multi-hop routing enables the nodes to stay load-balanced, especially if the network uses the same sink node throughout the deployment and therefore bases most of the routing on the battery life of the neighboring nodes.

10.1.9 Score Calculation Metrics

Currently, the system uses the metrics: Battery percentage, RTT, and Path length to calculate the path score. By using these three metrics, the node can perform somewhat smart choices for deciding what direction it should route the data. The reason why the battery metric is important is so the nodes can try to distribute the load onto the nodes that have not been as much. The reason that the RTT is important is that a high RTT would result in increased failed transmissions, which would reduce the longevity of the network as a whole due to useless transmission costs. The reason for the path length metric is to route a way that does not have a lot of nodes in the path, as this would also reduce the longevity of the network as a whole. However, the score calculation is not without its problems, which will be explained in further detail in section 10.4.7.

10.2 Simulator

As the system is implemented as a simulator, there are a few important discussion subjects, which decided how the system was created.

10.2.1 Why Create A Custom Solution

Currently, the simulator is implemented as a custom-made simulator. This was chosen instead of using ns-3[1]. NS-3 is an "out-of-the-box" network simulator designed for research and education. Using a custom solution allowed us to control every part of the simulator. While using the ns3 might be better for an actual simulation, it requires research and knowledge about how it works to get it to work well.

10.2.2 Simulated Topology Management vs. Range-Based Management

This simulator also handles the dynamic topology management, as stated in chapter 7, i.e., the simulator creates the networks based on a predefined structure. When deciding on how the topology would be generated, we also discussed adding simulated range-based management, where the nodes would be given random coordinates and talk to other nodes within a set range. Using the custom predefined solution was easier to work with and is constant in the way that it works, meaning that the system has reproducibility. This makes the system easier to test with and easier to experiment on due to the non-randomness of the system.

10.3 Time Synchronization

As time synchronization is one of the most important aspects of the system(and one of the main aspects of this thesis), the choice of finding a good way to handle synchronization that adhered to the design guidelines was important. When deciding on what time synchronization technology was going to be used, many existing options were considered.

10.3.1 Why Use Two Adjustment Algorithms

Due to the dynamic topology management, which means that all nodes can become sink nodes, and the decentralized aspect of the system, which means that there can also be no leader nodes in the system, the nodes do not have a leader where they can fetch an authoritative time from. Therefore, the use of two algorithms arose: One that can handle synchronizing with an authoritative time source and one that allows for having the nodes synchronized with each other while disregarding the authoritative and correct time.

10.3.2 Synchronization Algorithms

There are many different time synchronization algorithms, and many of them are created for specialized scenarios. NTP is the time synchronization algorithm that is considered the golden standard for time synchronization(see chapter 4). Based on the environment stated for this thesis, the NTP would require significant modification to become relevant. Therefore, the use of a standalone synchronization algorithm tailored for the synchronization of nodes in the arctic arose.

10.3.2.1 Choice of the Two Algorithms

When choosing the algorithms for this system, the most obvious choice was to pick the TPSN algorithm with a variation. This is due to it having many of the important aspects already considered. Using two different variations of TPSN-algorithm allowed the nodes to fetch from an authoritative time source and offset their time based on the global neighborhood offset. Another reason for basing the adjustment algorithms on TPSN is that both require to record the timestamps in the same way as shown in figure 6.13.

Because of their ability to use the same code, it allows nodes that are not updated – in the sense of knowing if there are sinks in the network or not – to use the wrong synchronization algorithm and still synchronize with the neighboring nodes or a parent node. Therefore, ensuring that the nodes can synchronize as long as they are awake at the same time, even if their local knowledge is different.

10.3.2.2 Adapting the Timing-Sync Protocol for Sensor Networks

As explained in chapter 4, there were problems with adopting the TPSN algorithm to fit in this system. Therefore, an adaption of the TPSN has been done, which uses the same timestamp readings(as shown in figure 6.13) and calculations. Except the adaption calculates a neighborhood average offset and skews the clock gradually towards the center of the neighborhood. Thus, the algorithm disregards the aspect of the correct time but allows for the network to remain synchronized. However, adapting the TPSN algorithm to work with the gradual adjustment introduced its own problems. These will be outlined in section 10.3.2.3.

10.3.2.3 Problem with Adopted TPSN

Based on the experiments shown in chapter 9 we saw that the number of transmissions required by the gradual time synchronization to remain synchronized was very high. This means that the gradual time synchronization is not optimal and should therefore be optimized, so it has a smaller footprint.

10.3.2.4 Adapting TPSN to a Latency-Based Adjustment

Latency-Based adjustment is also a variation on the TPSN algorithm and is also created for a low-cost sensor network with a sender-receiver approach. The Latency-Based algorithm uses the same four timestamps that the TPSN algorithm uses, as shown in figure 6.13. However, Latency-Based calculates the latency between the nodes as shown in section 6.4.1, and uses the One-way latency to offset the clock of the parent node with, shown in figure 6.14. This allows a node to adopt the time of their parents. However, adapting the TPSN algorithm to calculate a latency also creates problems on its own. These will be outlined in section 10.3.2.5.

10.3.2.5 Problem with Latency-Based Algorithm

By default, would the Latency-Based Algorithm suffer from the same problems as the TPSN as shown in chapter 4. However, when using the Latency-based algorithm, the network would have a sink node in it. Using the skewed safe zone together with the latency-based adjustment allows the nodes to fetch a correct time via multi-hop.

But, another problem with the adoption of time arises. When the One-Way latency, is skewed the calculations would be wrong – Skewed in this sense means that it takes longer to send to the other node rather than to respond back to it. In other words, this assumes that the RTT is equally divided into two one-way latencies.

10.3.3 Using The Skewed Safe Synchronization Zones

Currently, the system uses skewed safe synchronization zones when there is a sink in the network, as explained in the design chapter. This is to enforce the children to try to synchronize with their parents a bit skewed into the awake cycle. This allows the parent nodes to refuse a synchronization request based on the fact that it has not been updated yet. If nodes fail a request, it adds additional overhead transmissions, which is not good. But, it makes sure the nodes fetch the updated time, especially when the depth of the nodes grows. By configuring the settings connected to the skewed safe zone, the number of fails would be reduced. How the skewed safe synchronization zones work is shown in figure 6.15

10.4 Weaknesses

The implemented prototype system also has a few weaknesses, which will be presented in the following sections.

10.4.1 Dead Node Handling Problem and Path Management

When a node is found to be dead by another node in the network, it starts issuing a message around the network making, the other nodes tag it as inactive and remove all paths that lead through that node. However, if the node was not dead and the connection to it was just poor, a message telling the other nodes in the network that the "dead" node is alive would start to propagate throughout the network. This would lead to the "dead" node joining the network again with the next communication, and paths would need to be shared again, increasing the number of transmissions significantly. This could be repeated multiple times until the connection between the two nodes stabilizes.

When a node goes from being a sink node to being an ordinary node, it notifies the other nodes in the network about it. Which, in turn, makes the other nodes in the network remove the paths that lead to that specific node. This could be improved so that the paths are stored in all nodes after first being discovered and tagged as inactive when not leading to a sink node anymore. Doing this would reduce the number of transmissions needed when exchanging sink-paths, as a simple gossiping algorithm only gossiping the name of the new sink could be used to exchange the information about which node is a sink, rather than re-introducing the sink again to the network, which we saw from the experiments chapter took a lot of transmissions.

10.4.2 Scalability

As we can see from the experiments in chapter 9, the network has a problem with scaling with a number of nodes as the number of transmissions grows exponentially.

The biggest problem with scalability is the path sharing algorithm, which happens in both the starting phase and the operational phase. As we saw from the experiments, the amount of transmissions is larger when a node is introduced while in the operational phase compared to the starting phase(due to the gradual path-sharing done in the starting phase). Therefore, the system would benefit from an optimized path-sharing algorithm.

10.4.3 Missing CPU / Memory Performance Checks

As we can see in the experiments in chapter 9, there were no experiments that measured the systems CPU and memory footprint. When performing the experiments, we tried to perform some CPU and memory experiments using the pprof package for golang[7]. However, when we were using it, we were not able to get any useful data with it, as it mostly showed background processes that go used. We also decided not to measure the memory footprint of the nodes using the used memory of the computer because it would not show the entire picture of the single node but rather the whole simulator of the network.

10.4.4 Missing Scalability Experiment with Multiple Sinks

As we also can see from the experiment chapter 9, there are no experiments that measure the scalability of the system with multiple sinks. We believe that this would not show us something that was not expected, as adding more sinks would increase the transmissions based on a factor similar to the number of sink nodes in the network.

10.4.5 Missing Skew Factor Experiment

In chapter 9, an experiment testing the system's ability to handle larger skew factors are missing. All the experiments and results were performed with the skew factor settings set to +3 to -3 seconds. This means that each cycle, the nodes would skew somewhere between +3 seconds and -3 seconds. However, this test would not be useful, as we believe the system would be able to function the same, up until the point where the nodes would be started(after the starting phase) completely separated from each other. Figure 10.1 explains how the system would work up until the nodes are initialized so far apart that they would not wake up at the same time at the first cycle, thus, rendering the experiment useless.

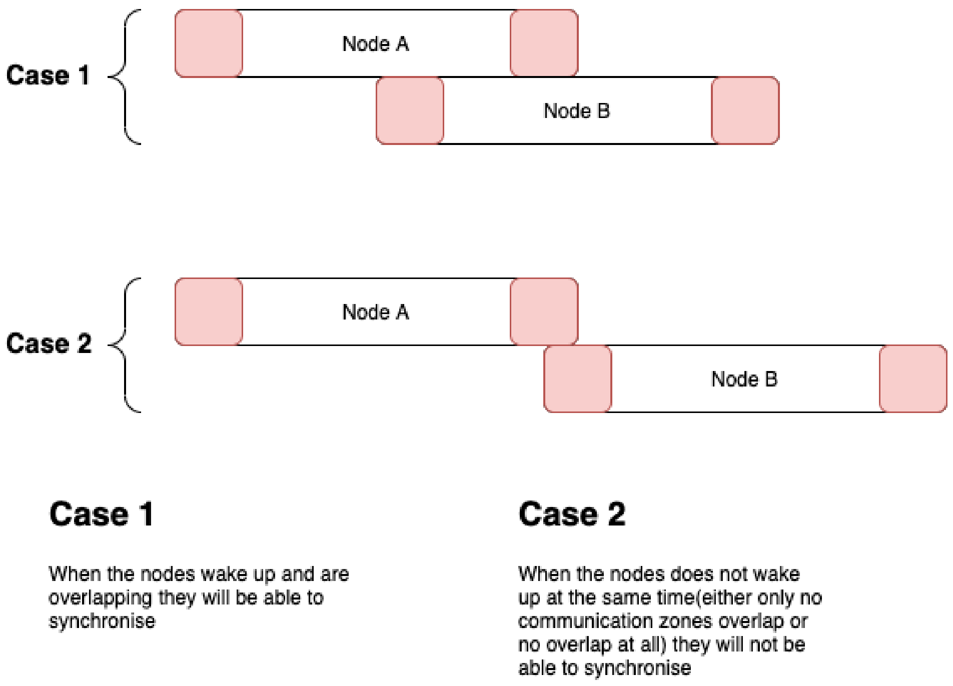

Case 1
Node A
Node B
Case 2
Node A
Node B
Case 1
When the nodes wake up and are overlapping they will be able to synchronise
Case 2
When the nodes does not wake up at the same time(either only no communication zones overlap or no overlap at all) they will not be able to synchronise

Figure 10.1: Figure showing why the missing skew factor experiment is not needed

10.4.6 Simulated Skew Factor

As the time synchronization is based on a simulated skew in each node, the simulated skew factor could have been created better. Currently, the skew factor is a constant value that skews the local node time each time it sleeps(as explained in chapter 7). In the early versions of the system, the skew factor used to be a random value that was added each time it slept. The problem with this was that the nodes ended up skewing the clocks back to the correct time. While this might simulate the actual clock better, it made it harder to work with. To create a better version of the simulated skew factor, looking into some research about what factors that decide what direction the nodes should skew in would be useful.

10.4.7 Circular Routing Problem

Currently, the score-based routing is not optimal as the algorithm normalizes the values of the different metrics and sends the content(via Erlend Karlstrøm's bundling algorithm) to the first step in the calculated optimal path. This could lead to having nodes send data back and forth between each other, which would drain the remaining battery percentage of the nodes. However, this could be fixed by adding more metadata to the propagated content, so each content message has a designated path selected for them. This would ensure a form of load balancing, as the circular routing problem would not exist anymore

10.4.8 Late Nodes Not Included

Currently, nodes are included in the network when they introduce themselves in the starting phase. That way, the paths are shared(if there are any active sinks), and the sleep schedules are synchronized. However, if a node is too late to introduce itself and introduces itself while the other nodes are running in the operational phase, the network will allow it to become a neighboring node, but the newly joined node would not have a synchronized sleep schedule.

This could, however, be fixed by adding some data into the introduction response message, stating that the other nodes are already running in the operational phase, and having the newly joined node forced into the operational phase with a synchronized sleep schedule.

10.4.9 Timer-Based to Cycle-Based

The system currently uses timers to ensure that the network maintenance operations – such as the ping operation and dead-node checking – are well spaced out. These timers could cause problems for the network health, as they would skew with the local clocks.

The timer-based system should therefore be changed to a cycle-based system, except for the timers which, ensure the time in the awake cycle remains the same. The cycle-based system performs these operations based on a specific iteration of cycles, which would, in turn, mean that they are equally spread out over the deployment cycle, rather than having them dependant on a timer that may be skewed due to the clock skew, and make the system easier to handle.

10.4.10 Total Transmission Amount

Based on the experiments shown in chapter 9 we saw that the number of transmissions required by the gradual time synchronization to remain synchronized was very high. This means that the gradual time synchronization is not optimal and should therefore be optimized, so it has a smaller footprint.

We can also see that the total amount of paths in the network is very high and that the amount grows exponentially with a growing number of nodes.

This requires a large number of transmissions, which is at the cost of network longevity.

11 Future Work

Based on the experiments shown in chapter 9, and what is described in the architecture and design chapters, some future work of the system is outlined. These improvements will not necessarily work as intended but are something to at least consider when trying to improve the general system.

11.1 Real-World Implementation

Currently, the system has not been tested in a "Real-World" test deployment. This means that we do not know for certain if this system would work when deployed. Thus, a real-world implementation could be useful to figure out if the system could be used for long deployment cycles in the arctic.

Before deploying the system some work would be needed to find out what sort of hardware that the system could use. The software would also need to be fixed and optimized to reduce the number of transmissions that it currently uses, while also moving from a timer-based system to a clock-based system.

11.2 Relay Units

As the deployment location areas can be large, spanning over multiple kilometers.[18][19] Since the distances between the OU's can be long, the use of what we decide to call *Relay Units* arise. Relay units(or *RUs*) are nodes that do not have the capabilities to record environmental data or take photos. However, we propose that using the same enclosures that housed the internals for the OU's, except adding more batteries where the internals that controlled the environmental recording devices were, will allow for a node with significantly higher longevity.

Placing these RUs between nodes that have a bad connection or not able to reach each other, can increase the coverage making it easier to propagate data throughout the network. Using RUs between nodes that are close to each other can also allow for increased longevity in the network(assuming that the path sharing algorithm is improved), as they can close the distance between nodes so that the normal nodes can reduce the transmitting power and therefore retain more battery power in the long run.

Using RUs might not be possible due to naturmangfoldloven §35[20], also stated in the design guidelines section **??**, where they state that the animals should not be disturbed.

11.3 Mini-Cluster Sinks

Since the probability of having the previous sinks becoming sinks again is high, the use of "mini-cluster sinks" arises. Using a small cluster of nodes viewed as a single sink, and spreading the load round-robin style between the nodes could increase the network's longevity. It would require some development effort to get it to work properly but it might be worth it, as the sink nodes are the ones that use the most amount of transmissions.

11.4 Auto Adjusting Clock Based on Previous Knowledge

When nodes are placed in the arctic, the environment they are located in mostly stays the same, which means that the clocks might skew similarly between each awake cycle. The system could build up a knowledge store that contains the amount that the node had to adjust its clock with, and use that knowledge to calculate the amount skew it would require after running a while. The system could automatically adjust the clock itself without sending a synchronization transmission, and basing the automatic skew on these previous calculations and thus reducing the number of transmissions overall.

11.5 Energy Budget

After putting together some hardware to make the system run, one could perform some tests on the hardware to see how much power each operation will draw, and by using the results create a solution for budgeting a set amount of power each day, so it limits how much it should be able to use. This will ensure that power is being spent correctly and efficiently.

11.6 Retransmission Reduction Strategies

As the distances between the nodes in the arctic can be quite large, which could lead to the connection between them being poor. The nodes should therefore have a solution for optimizing the communication messages between the nodes in such a fashion that they can reduce the number of failed transmissions. This means that implementing a Forward Error Correction(FEC) algorithm or another solution that will be able to reduce the number of retransmissions.

11.7 Improve Score Weights

As the score-based calculation right now is very simple(the calculations currently just normalizes the data). This could be improved significantly in ways such as finding a good way to handle the weighting of the different parameters in the calculation. However, this requires extensive research.

11.8 Preemptive propagation

After having a sink in the network and then losing it, the nodes could store that sink node, together with the path to that sink. This could be used for preemptive propagation of data while having no sinks in the network. This means that the data could be stored on the previous sink node, which would leave it very close to the homebase(assuming the same sink would be used as a sink node multiple times).

11.9 Improved Gradual Time Synchronization

As we saw in the time synchronization experiments(chapter 9), the gradual adjustment algorithm used a significantly larger amount of transmissions than the immediate adjustment algorithm, to achieve node network stability. By improving this, the network could reduce the number of transmissions used by a significant degree.

There are a few ways one could go about improving the solution. The gradual algorithm could be implemented with a limited amount of neighbors that are contacted for synchronizing, this would require some development effort to find a solution that can ensure that all nodes get updated. Another way is by using the immediate algorithm instead. When there is no sink node in the network, it could either appoint a random node to become a temporary sink node(or a node with authoritative time) or use the previously known sink as a node to synchronize against(similarly to the previous section). As the gradual algorithm discards the correct time anyway, following the skew of a single node can be acceptable, while also reducing the transmissions significantly.

11.10 Improved Path Sharing and Path Acceptance

As seen in the experiments chapter 9, the transmission amount of the path sharing is very high. When designing the path sharing and acceptance algorithms, we decided that we wanted redundancy paths at all nodes to ensure that no nodes were disconnected from the network. This, in hindsight, was not the best decision as the number of transmissions required just to share the paths are so high and non-scalable, especially when the amount of topological cycles increases.

Therefore, improving the path sharing and acceptance algorithms in a way so that all the nodes are able to propagate data towards all sink nodes in the network, and still having redundancy paths in the case of node failure could be useful, and necessary before eventual "real world" deployment.

12 Conclusion

In this thesis, we implemented and described a prototype system of a Wireless Sensor Network where the nodes were designed to observe environmental changes by reading data from sensors. The nodes can dynamically form a network by introducing themselves to each other within a broadcasting distance – which in this thesis is simulated. This enabled all the nodes to become sink nodes, so data could be propagated throughout the network, from the most distant nodes to the homebase. The nodes are also able to remain synchronized, but the whole network will skew together away from the "real" clock when there are no sinks in the network.

The general network experiments showed that the path sharing transmissions grew exponentially, when scaling the number of nodes up in the network while still having the shortest path remain short, indicating that the number of paths in each node is redundant and should therefore be reduced significantly. We also saw that the transmissions after introducing a sink in the network while in the starting phase, and keeping that sink operational, scaled linearly after introducing the sink, with the number of transmissions being dependant on the number of nodes in the network.

The time synchronization experiments showed that the implemented algorithms were able to perform clock synchronization accurately and precisely, with the only problem being that the transmissions needed for synchronization without a sink grew significantly larger than the synchronization with a sink in the network. The experiments also showed that the amount of time synchronization operation could be reduced up to 1 synchronization operation per 5th cycle – Which, in turn, means that the transmissions could be reduced by a factor of up to 5. –, after that the network started to fragment(breaking into smaller networks)

While the system was able to do as planned, it did not perform optimally. This is because the transmissions needed for connecting the nodes, sharing the paths to a sink, and synchronizing the clocks while not having a sink in the network grew very large. However, with a better path sharing algorithm, and an improvement of the synchronization algorithm(without sinks), the system could be used to propagate data from distant nodes located on the arctic tundra to a homebase, located outside of the deployment area.

Bibliography

[1] Network Simulator 3. Ns-3. `https://www.nsnam.org/`. Accessed: 2020-05-10.

[2] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, December 2003.

[3] Szymon Fedor and Martin Collier. On the problem of energy efficiency of multi-hop vs one-hop routing in wireless sensor networks. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, volume 2, pages 380–385, 2007.

[4] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, page 138–149, New York, NY, USA, 2003. Association for Computing Machinery.

[5] Gizak. Golang terminal ui. `https://github.com/gizak/termui`. Accessed: 2020-05-03.

[6] Golang. Golang programming language. `https://golang.org`. Accessed: 2020-05-03.

[7] Golang. Package pprof. `https://golang.org/pkg/net/http/pprof/`. Accessed: 2020-05-03.

[8] W.R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 10 pp. vol.2–, 2000.

[9] Stien A. Ims R.A., Jepsen J.U. and Yoccoz N.G. Science plan for coat: Climate-ecological observatory for arctic tundra. *Fram Centre Report Series 1*, pages 0–177, 2013.

[10] Konrad Iwanicki, Maarten van Steen, and Spyros Voulgaris. Gossip-based clock synchronization for large decentralized systems. In Alexander Keller and Jean-Philippe Martin-Flatin, editors, *Self-Managed Networks, Systems, and Services*, pages 28–42, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] Erlend Karlstrøm. General monitoring of observational units in the arctic tundra. Master's thesis, UiT The Arctic University of Norway, 2021.

[12] S. Lindsey and C.S. Raghavendra. Pegasis: Power-efficient gathering in sensor information systems. In *Proceedings, IEEE Aerospace Conference*, volume 3, pages 3–3, 2002.

[13] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, page 88–97, New York, NY, USA, 2002. Association for Computing Machinery.

[14] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[15] Climate-Ecological Observatory. Coat viltkamera. `https://www.coat.no/en/Research/Data/COAT-Camera-traps`. Accessed: 2020-05-04.

[16] Climate-Ecological Observatory. Klimaøkologisk observasjonssystem for arktisk tundra (coat). `https://www.coat.no/`. Accessed: 2020-05-03.

[17] Climate-Ecological Observatory. Metode. `https://www.coat.no/Metode`. Accessed: 2020-05-03.

[18] Climate-Ecological Observatory. Svalbard. `https://www.coat.no/en/Svalbard`. Accessed: 2020-05-03.

[19] Climate-Ecological Observatory. Varanger. `https://www.coat.no/en/Varanger`. Accessed: 2020-05-03.

[20] Klima og miljødepartementet. Lov om forvaltning av naturens mangfold (naturmangfoldloven). `https://lovdata.no/dokument/NL/lov/2009-06-19-100`, 2009.

[21] Luis Oliveira and Joel Rodrigues. Wireless sensor networks: A survey on environmental monitoring. *JCM*, 6:143–151, 04 2011.

[22] Dennis E. Phillips, Mohammad-Mahdi Moazzami, Guoliang Xing, and Jonathan M. Lees. A sensor network for real-time volcano tomography: System design and deployment. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2017.

[23] Python. Python programming language. `https://python.org`. Accessed: 2020-05-03.

[24] Wolf-Bastian Pöttner, Felix Büsching, Georg von Zengen, and Lars Wolf. Data elevators: Applying the bundle protocol in delay tolerant wireless sensor networks. In *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*, pages 218–226, 2012.

[25] Issam Raïs, John Markus Bjørndalen, Phuong Hoai Ha, Ken-Arne Jensen, Lukasz Sergiusz Michalik, Håvard Mjøen, Øystein Tveito, and Otto Anshus. Uavs as a leverage to provide energy and network for cyber-physical observation units on the arctic tundra. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 625–632, 2019.

[26] Ill-Keun Rhee, Jaehan Lee, Jangsub Kim, Erchin Serpedin, and Yik-Chung Wu. Clock synchronization in wireless sensor networks: An overview. *Sensors (Basel, Switzerland)*, 9:56–85, 01 2009.

[27] Samtech. What is lora®? `https://www.semtech.com/lora/what-is-lora`. Accessed: 2020-05-03.

[28] Eeva Soininen, Åshild Ø Pedersen // Norwegian Polar Institute Rolf A Ims // UiT The Arctic University of Norway, and Audun Stien // NINA – Norwegian Institute for Nature Research. Climate-ecological observatory for arctic tundra—status 2020. `https://framsenteret.no/forum/2020/climate-ecological-observatory-for-arctic-tundra-status-2020/`. Accessed: 2020-05-03.

[29] Camilla Stormoen. Peer observations of observation units. Master's thesis, UiT The Arctic University of Norway, 2018.

[30] Francisco Tirado-Andrés and Alvaro Araujo. Performance of clock sources and their influence on time synchronization in wireless sensor networks. *International Journal of Distributed Sensor Networks*, 15:155014771987937, 09 2019.

[31] Øystein Tveito. Beneath the snow – developing a wireless sensor node for remote locations in the arctic. Master's thesis, UiT The Arctic University

of Norway, 2020.

[32] Deepak Vasisht, Zerina Kapetanovic, Jong-ho Won, Xinxin Jin, Ranveer Chandra, Ashish Kapoor, Sudipta N. Sinha, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 515–528, USA, 2017. USENIX Association.

[33] Yik-Chung Wu, Qasim Chaudhari, and Erchin Serpedin. Clock synchronization of wireless sensor networks. *IEEE Signal Processing Magazine*, 28(1):124–138, 2011.

[34] Hamed Yousefi, Ali Dabirmoghaddam, Kambiz Mizanian, and Amir Hossein Jahangir. Score based reliable routing in wireless sensor networks. In *2009 International Conference on Information Networking*, pages 1–5, 2009.