UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

General Monitoring of Observational Units in the Arctic Tundra

Erlend Melum Karlstrøm

INF-3990: Master's Thesis in Computer Science
May 15, 2021

“"All right, I’ve been thinking. When life gives you lemons? Don’t make lemonade. Make life take the lemons back! Get mad! ’I don’t want your damn lemons! What am I supposed to do with these?’
Demand to see life’s manager! Make life rue the day it thought it could give Cave Johnson lemons! Do you know who I am? I’m the man who’s going to burn your house down! With the lemons! I’m going to get my engineers to invent a combustible lemon that burns your house down!"”
– Cave Johnson - Portal 2 (2011)

Abstract

Climate change is going to change what we know about the arctic tundra. Patterns in the behavior of the wildlife that lives there are predicted to undergo a shift, and it will therefore be important to have reliable sources of empirical data, so that we can understand how these developments are playing out. The arctic tundra is remote and difficult to deploy sensing instruments on, and signal coverage is unreliable.

Finding a way to monitor them reliably from a distance is needed.

This thesis describes how a prototype for a Wireless Sensor Network was designed, implemented, and tested, with the aim of connecting Observational Units together in a local cluster, and cooperate amongst themselves to propagate monitoring data to external servers.

The system was designed so that nodes can dynamically discover neighboring nodes within their range, and gossip knowledge about where sinks are in the network. Sinks are nodes which have managed to establish a link with an external server, and the paths to these sinks are spread across the network. Such that if only node in the entire cluster is a sink, then data from every node has a path outside of the cluster.

Results from running validation shows that the implemented prototype functions as intended, but experiments have revealed apparent weaknesses. The number of paths which are shared in gossiping shows an exponential growth when the number of nodes in a cluster grows linearly. The experiments into bundling and monitoring-data propagation shows that combining data together causes a reduction in these types of transmissions by a factor equal to that of the number of data fragments which are combined, however the Partial Bundle Policy measure to increase throughput for fringe nodes has unexpected consequences.

The prototype system works as intended per the design. We have found however that the system is not scalable due to the extent of the accumulated path knowledge. Suggestions for avenues to address this has been outlined in the discussion chapter. There is a need to explore how something similar to this prototype would look and perform in a real-life deployment on the arctic tundra.

Acknowledgements

First I would like to thank my friend and collaborator during this project, Sigurd Karlstad. Having someone to argue with, and point out when I'm being an idiot has been tremendously useful.

I would also like thank my main advisor Professor John Markus Bjørndalen, and my co-advisors Professor Otto Anshus, and Post-doc Issam Raïs as well, for providing guidance and support whenever I needed it.

Lastly, I would like to thank my family for being supportive during this period, and their ever-present encouragement towards me to get this master's degree.

Contents

List of Figures

List of Tables

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. In this thesis, Observational Units and the more general term 'node' will be used interchangably.

Homebase: A Homebase is a server / machine that is considered as an external machine in comparison with the nodes in a cluster. The Homebase is where the nodes tries to propagate data to, and is the source of authoritative time. 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.

Sinks: A sink-node is an OU that is directly connected to a Homebase, this means that this is the node that all the other nodes in the network will try to propagate messages to.

Neighbor: From the perspective of a given OU, a neighbor is another OU which is close enough to do short-range communication reliably.

/1 Introduction

Climate change and radical ecological shifts has been a topic of ever-increasing concern in recent times. The arctic tundra – the earth's northernmost terrestrial biome – is predicted to be especially impacted in this regard, with average temperatures increasing by as much as 10*C by the next century. Such a rapid shift is likely to have far-reaching consequences and result in a new status quo which current scientific models will be unable to accurately predict. [8]

With these concerns in mind, the Fram Centre proposed in 2013 a plan for the establishment of a Climate-ecological Observatory for Arctic Tundra (COAT) to monitor how these changes impact ecosystems on the tundra in the long term. Implementation of this plan has involved deploying several on-the-ground sensing instruments – like camera traps with food as bait, and temperature sensors – to harvest data on activity in the surrouding area.

This is easier said than done however. The arctic tundra is harsh and remote; varied weather, below-freezing temperatures, infrastructure scarcity, unreliable signal coverage, and long distances are all present factors which make the tundra a challenging deployment environment.

This thesis presents the architecture and design of a decentralized overlay system for Observational Units (OU) which enables them to use each other as relays to external servers when uniform signal coverage cannot be assumed. Along with this is a propagation scheme for debugging data which combines fragments into larger pieces in order to reduce transmissions – a significant source of energy expenditure.

1.1 Motivation

The motivation for this project is the need for a scheme for delivering debugging data and status updates from Observational Units. Propagating data should be energy-efficient, reliable, and maintain a level of throughput to the degree that the data is still useful by the time it arrives to users.

1.2 Contributions

The prototype and system design presented in this thesis has been worked on by two people: the author, and Sigurd Karlstad, another master student here at UiT. Throughout this project, the author has cooperated closely with Karlstad, due to the subject of each thesis' being quite similar. It is therefore necessary to give a precise description of what each person takes credit for.

This thesis contains the following contributions:

1.2.1 Author's Contribution:

- Bundling and Partial Bundle Policy
- Content Propagation and Debug Report Generation
- An description of technology and techniques in WSNs and Habitat Monitoring
- An analysis of the difficulties inherent in designing habitat monitoring systems for the arctic tundra, based on previous experiences
- Bundling Validation and Experiments
- A discussion of weaknesses, improvements, and pros and cons.

1.2.2 Sigurd Karlstad's Contribution:

- Time Synchronization Operations and all uses of it.

- Node network clock stability.

1.2.3 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.

1.3 Limitations

This thesis will not be presenting a real-world deployment of Observation Units. When OUs are spoken about and how they gather data about their surroundings, the actual data is fake. The OUs that are discussed are purely virtual and are meant to demonstrate how an overlay network would act with this kind of data. Details about how the system simulates the OUs is explained in Chapter 7.

1.4 Thesis Outline

This thesis is structured into 12 chapters, and excluding the introduction they are:

- Chapter 2: Background, describes the circumstances which one ought to be familiar with when designing computer systems towards the tundra, by examining previous experiences.
- Chapter 3: Design Principles, an attempt to formulate principles to guide a concrete design on the back of what we learned in the background.

- Chapter 4: Related Work, a presentation of relevant work done in various fields, along with a discussion on their applicability to our circumstances.
- Chapter 5: Architecture, a description of the proposed system's main functionality.
- Chapter 6: Design, a deeper dive into the workings of the proposed system.
- Chapter 7: Implementation, a description of the attempt to simulate the restrictions and behavior an OU will find itself in.
- Chapter 8: Validation, an investigation into the correctness of the implemented prototype.
- Chapter 9: Experiments, an investigation into the characteristics of the implemented prototype.
- Chapter 10: Discussion, an evaluation of what has been produced, exploring problems, weaknesses and areas of improvement.
- Chapter 11: Further Work, a discussion about the potential paths forward.
- Chapter 12: Conclusion, a summary about our aim, experiences and lessons learned.

2 Background

In this section, the background of this thesis will be presented. This includes an introduction to Habitat Monitoring and WSNs in Computer Science, and an outline of connected projects that serve as the underlying context for this thesis.

The most common approach in monitoring habitats and natural environments – according to Oliveira et al in a survey from 2011 [14] – is in deployment of a Wireless Sensor Network(WSN) consisting of hundreds up to thousands of individual devices, which organize themselves into a network, and cooperate towards accomplishing some task. These devices are usually small in dimensions and cheaper to acquire than compared to, for example a personal computer. On the other hand, they also have less resources to work with when it comes to computational capacity, energy resources and transmission capabilities.

Common problems addressed in WSN research can be summarized as finding ways to use this constrained sum of resources efficiently. Nodes in this type of system are often powered by batteries and have small communication ranges – as well as being limited in their processing and storage capacity due to the hardware solutions usually emphasizing low-power, low-cost devices with small physical dimensions.

This set of difficulties is carried over into habitat monitoring, but the problem of conserving power becomes even more pronounced in this case. Monitoring natural habitats for research purposes often carries with it a requirement for any

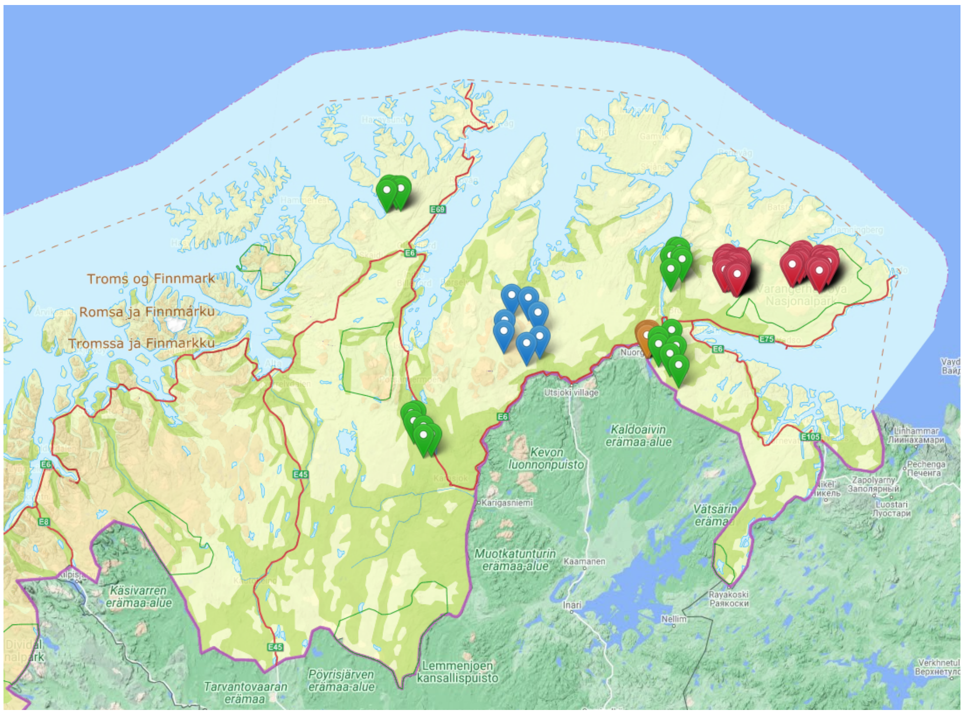

Figure 2.1: Map of Troms and Finnmark fylke, Norway showing the COAT instrument placements. Image taken from their website at [2]

sort of human activity to be kept to a minimum, either in order to reduce the influence the monitoring itself has on the observed, or minimize the necessity for maintenance trips as the deployment site may be quite remote. With this in consideration, ensuring that a sensor network can stay alive for extended periods of time (at least collectively) without needing maintenance/interference from humans, becomes a vital quality of any solution which seeks to be a viable option when potential users are considering which system to choose for their deployment.

The COAT Project is one such potential user. Their efforts thus far in monitoring the Varanger Region in northern Norway and on Svalbard has consisted in placing camera traps in these areas to monitor the movement and presence of different animal species. A map from their website [2] shows the placement of measuring instruments that they have placed and pictures they have accumulated of different animal species. In their Science Plan [8], a document describing how they are going to implement their ecological tundra monitoring experiments, COAT affirms their intent to have a minimal environmental footprint, both when conducting their measurements and associated activities like collecting data from instruments.

In conjunction with COAT, the Computer Science Department at UiT has also taken an interest in the measuring instruments that are placed on the tundra. The DAO (Distributed Arctic Observatory) Project uses instruments called OUs (Observation Units), that are distributed in the tundra and monitors state variables before reporting back somewhere for further analysis [17].

In a paper by Raïs et al [17] where they experiment with using a UAV drone to carry in fresh batteries and function as a temporary network access point, deploying in the arctic tundra is described as "expensive, time-consuming and dangerous" and asserts the need for automation such that physical visitations by humans can be reduced to a strict minimum. The size of batteries is also mentioned, in that large batteries are generally not practical, because of environmental concerns and regulations.

In an earlier master's thesis from the same institution by Øystein Tveito, the author describes experiences with building and deploying hardware on the tundra in detail, describing a lengthy trip to deploy the OUs and then finding that the OUs were having problems reporting back after being set up. In his thesis [21], he outlines several deployment cycles. In the first, no OUs had managed to establish a connection to the back-haul network. These problems were hypothesized to be the result of erroneous mounting of the antenna. This was adjusted for the next attempt. In the second, some OUs managed to report back, but not all. The hypothesis here was that the local environment was most likely obstructing the signal. In addition, the coverage map which had been used to determine whether or not a connection was possible, were not generated from in-the-field test data but rather made by an algorithm, most likely based on approximations. These experiences shows that building a system around the assumption that every OU is going to be able to establish a direct connection with a cell tower, and by extension with the back-haul network, to be an unreasonable one.

Tveito also mentions considerations he had to make when designing containers for the hardware modules. Since the locations that they were going to deploy to where situated in national parks, some regulations about visibility and interfering with the local fauna had to be observed. His interpretation was that the environment "... must appear to be unspoiled nature. ... they must be camouflaged to be as unnoticable as reasonably possible."[21].

With these experiences in mind, the next chapter will try to summarize what seems to be required of a design which seeks to viable in a arctic tundra deployment environment.

3 Design Principles

This section seeks to investigate which general principles can serve to guide the design of systems of networked observation instruments on the arctic tundra.

3.1 Power Scarcity

The tundra consists mostly of large and remote areas of wilderness. This is not a place teeming with human activity, settlements, or power lines. The point is that providing the nodes with electricity for as long as several months to a year is quite a challenge.

Sensing equipment will be relying on battery reserves, and using methods like solar panels to recharge is not feasible in the winter because of the extended dark period so far north. Sending maintainence crews regurlarly is impractical due to the distances involved, and the winter will again further complicate things. It is therefore necessary to make use of what is there from initial deployment.

This also excludes the use of base stations in the local area which can be larger than the nodes and has persistent power, as there is not anywhere to get it from.

3.2 Energy-efficiency

Generally, transmission is an expensive operation, and much more so than computation.

Wireless Data Transmission is more expensive than data processing. Making it preferable to process whatever data is needed at the node rather than send it. [14]

The power consumed when the radio is in receive mode is almost equal to that consumed when transmitting. So the radio should be turned off when not necessary. [14]

Therefore an important consideration when making design choices for this system will be whether or not it contributes to the overall number of transmissions.

3.3 Unreliable Connectivity - Internal

Nodes are thought to be spread over a wide area, and therefore every node may not be in range of any other. However, planning placements in the field based on this alone may not be sufficient, as there may be obstructions to the signal in the physical environment. A dynamic, and unstructured mesh topology is therefore necessary to support.

3.4 Unreliable Connectivity - External

As previous experiences have shown, whether or not the OUs are able to get a signal to external parties is uncertain, and we must therefore make two considerations: we cannot know how many OUs have this connection, but it is likely to be few, and we cannot know which have it.

3.5 Habitat Non-interference

There is a general need for OUs and other equipment to not interfere with the natural environment in which they are placed.

This might mean that they have to be camouflaged to a degree, to not influence the fauna's behavior, and also not invite curious critters or humans even to potentially damage the device. This places restrictions on how the physical make-up of the OU. The size of the OU itself plays a part here, as it seems reasonable to assume that the bigger it is, the harder it is going to be to adequately hide it. This has consequences for what kind of hardware we can expect to have available.

3.6 Persistent Data Storage

Sensor readings must be stored persistently on the OU. Research data is important to keep secure, and in COAT's case their tundra monitoring is an on-going experiment over a long timeframe, so whatever overlay network or experimental node hardware is tested in the field, cannot interfere or potentially lose raw data from the sensing equipment. Providing for this also enables us to extract the raw data directly from the OU when they are fetched.

4 Related Work

In this section we will review a selection of earlier works in the fields of Wireless Sensor Networks (WSNs), Habitat Monitoring, Delay-Tolerant Networking (DTN), and System Monitoring.

4.1 Wireless Sensor Networks

One of the best known works in WSN research [19] is the Low Energy Adaptive Clustering Hierarchy protocol, published in the year 2000 by Heinzelman et al.

LEACH [7] is an energy-efficient communication protocol in which nodes are dynamically clustered into groups where they take turns gathering data in the cluster and sending it to an external receiver. Coordinating amongst themselves, the nodes select one of their number to be a Cluster Head (CH). The CH acts as a localized base station which collects bits of data from the other nodes in the cluster, and then sends it towards a high-capacity sink further away. After a period, another node relieves the previous one of this responsibility, and acts as Cluster Head until it in turn is relieved. Thus, the cost of acting as Cluster Head is spread evenly among the nodes present in the local cluster.

A paper by Lindsey et al from 2002 presented PEGASIS [11]. An improvement on the LEACH protocol, the network forms a chain composed of the nodes –

as opposed to LEACH's star-like clustering with the Cluster Head at the center. By following the chain, the nodes propagate their data forward where on each step it is fused together bit by bit, and eventually arrives at the last node, who is responsible for transmitting the final package to a base station. As in LEACH, this task of transmitting to the base station is rotated to another node after completing it, thus spreading out this cost in energy.

The problem with using a similar solution in our circumstances, is that LEACH and PEGASIS assumes that it is possible for every node to reach a base station, or every other node, and as we have outlined in the previous chapters, this is not a reasonable assumption in our case.

LEACH and PEGASIS both use something called an "Advertisement Phase" [7] [11] in which the nodes coordinate between each other to decide on which one of them should be the one to contact the base station. We will be doing something similar in our design, where a initial, high-activity period where duty cycling is disabled will serve as a window where the OUs can discover each other and do an introduction.

A master's thesis by Camilla Stormoen from 2018 from UiT [20] uses some ideas from LEACH and some of its derivatives as a basis for developing an approach where the nodes form links in a mesh network based upon if they are in radio-transmission range of each other. This approach uses a bully algorithm to elect a CH, where nodes gossip a score to their neighbors to decide who should be elected. A path to the potential CH is gossiped along with the score. When the nodes have decided a CH, they will wait for it request data from them, and then send data along the path so that it can be collected at the CH and be stored.

This is a design which addresses the need for an unstructured mesh network where no assumptions can be made in regards to what links exists between the OUs. It also uses a path gossiping mechanism so that nodes several degrees of separation away can have knowledge about certain notable nodes. However, our design will not be as concerned with collecting data from the nodes on a single node where it can be accessed, and instead takes advantage of the fact that multiple sinks can be present in the network at once, and nodes should route their data to the one that is most suitable according to the path selection algorithm.

4.2 Delay-Tolerant Networking

Delay-Tolerant Networking, or Disruption-Tolerant Networking as its sometimes called, is an area in networking research which is chiefly concerned with designing systems which are tolerant of long delays and interruptions in end-to-end transmissions[3]. The term was coined in 2003 by a team of the Internet Research Task Force with the publishing of RFC 4838 (Delay-Tolerant Networking Architecture). At this time the focus in DTN research was on the question of how to facilitate communication in deep space over long distances, which had borne an architecture for the Interplanetary Internet (IPN) project, with an aim to provide Internet-like services across the gap that separates celestial objects. The RFC [1] that was published suggested an extension of this architecture to apply these concepts to terrestrial networks – as they also suffer from delays and disruption. The DTN perspective was broadened to include networks and connection patterns of other types, an example of which is opportunistic mobile ad-hoc networks, where nodes can be unavailable for extended periods of time.

A review from 2008 by Fall and Farrell [3] discusses the current state of research in DTNs, and provides an useful outline of the design elements and principles that has been associated with this kind of network since its inception. Some of which are:

- Accommodation of long network delays and possible disruption

- Operation in heterogeneous networks. As a result of being able to support a variety of protocols at the same time by using convergence layer adapters, which enables translation of data units specific to each protocol into a DTN data unit and back. Another point for heterogeneity is a pluralism in supported naming formats.

- Data bundling, in the sense of combining all the data that is needed to complete a transaction in a protocol into a single data unit, in order to minimize the amount of exchanges necessary to complete it.

- Routing protocols and technology are ambiguous. In order to be applicable in a number of different operating environments, a DTN node may have to be able to use a set of different routing strategies and protocols to propagate what it wants and to where. Routing may involve a number of factors, such as managing multiple copies, using network knowledge of various resolutions (including knowledge of topology and ongoing traffic),

- Custody transfer. A custodian of a bundle is the stated "responsible entity" for making sure that the bundle is not lost in the network, as an alternative to the originating node keeping track of whether or not its data arrived successfully at the intended destination. Being the custodian of a data bundle entails keeping it safely in persistent memory until it can be further forwarded and stored in the same manner at the next node along the path. Custodianship is thereby transferred to this node.

While the DTN concepts were formulated some time ago and where designed to accommodate a Internet-like network architecture, a newer paper from 2012 by Pöttner et al. [16] demonstrates an implementation of the bundle protocol towards WSNs. They conduct an experiment revolving around a temperature micro-sensor which is placed on a rooftop, and is intermittently connected to a relay node placed inside an elevator. The elevator moves from the rooftop floor to a another node placed on a lower floor which then forwards the data to a back-end computer where it is processed and further analyzed. This chain forms a relay from the temperature sensor on the rooftop to the back-end computer where the node in the elevator functions as a data mule for the data readings.

This is facilitated by an implementation of a bundle protocol convergence layer called µDTN which can carry data to intermittently connected destinations, and provides interoperability between heterogeneous hardware platforms. In their conclusion, they state that a bundle-like protocol is a suitable choice for heterogeneous solutions which seeks to integrate low-power WSN networks with back-end systems possessing more substantial network and computational resources. They argue that a form of bundle protocol should be the first choice for WSNs where delay-tolerant communication and back-end connectivity is a requirement.

Overall, this focus on introducing features that enable a network to compensate for long delays and disruptions, appers like a good fit for our circumstances. Bundling of transaction data and custody transfer, are features which enable a sort of decoupling between the originator of a request and the final recipient, which would allow a transaction to be completed successfully without the need for both of them to be connected and available at the same time.

We will be using the custody transfer concept in our design, in the sense that once bundles are propagated from the node which generated it, it will rescind its responsibility that it arrives at the homebase. This responsibility will instead fall on the next node it was propagated to, until it propagates it further.

We will also be using a bundle concept, but not in the sense of packaging data which are needed to complete a multi-step interation into a singel piece, but rather as an object consisting of potentially numerous pieces of separate data units which may or may not be from the same node.

4.3 Environmental Monitoring

Monitoring natural processes is a problem-space which has not garnered the same kind of attention in Computer Science as fields like Artificial Intelligence or Cloud Computing have, but there are some related works in this field.

The survey by Oliveira et al [14] – which was mentioned in the Chapter 2 chapter – provides an outline for some of the challenges which this area of research must address. We are not going through all of them because the list is quite long and not all of them are relevant, but here are some that are:

- Power Management: An essential feature for long-term operation, especially for remote locations where visitation is impractical.
- Remote Management: Some form of remote configuration and operation is necessary, again due to how remote and isolated some locations may be.
- Mesh Routing Support: mesh network topologies can both provide multi-hop and path diversity, so routing protocols which can support this is crucial
- Size: reducing the size is essential for many use-applications.

This survey also goes through some notable applications of environmental monitoring. Among which is the first WSN that was implemented with habitat monitoring in mind [14]. Published in 2002 by Mainwaring et al [12] The goal was to monitor certain bird populations on some islands off the U.S east coast, specifically their nesting behavior given some environmental variables – like temperature. Their approach consists of deployment of some hundred micro-sensors, which are organized into a tiered hierarchical network, where the nodes will use multi-hop forwarding and coordinate between each other.

Our design uses multi-hop forwarding, but Mainwaring et al uses more substantial hardware for gateways which are locally present to the sensing instruments, and the base station that is connected to the internet via a satellite link (I was unable to find if these use batteries or wired power).

5 Architecture

In this section we will be looking at the architecture and primary functionality of the proposed system.

5.1 Topology and Communication

The proposed system is thought to function in a mesh topology, where the links between nodes are formed via being within a certain proximity to each other, and are not determined by the need to conform to a star-like structure for example.

Communication will be facilitated through wireless ad-hoc communication, and OUs will therefore have a limited range of communication (fig 5.2). The exact range will depend on the technology that they use, and the surronding physical environment. A system design will have to allow for the fact that any one-to-one communication between entities cannot be presumed.

5.2 Duty Cycling

In order to conserve their battery, nodes will shut down most of their systems, including transceivers, and will be unavailable for an extended period of time.

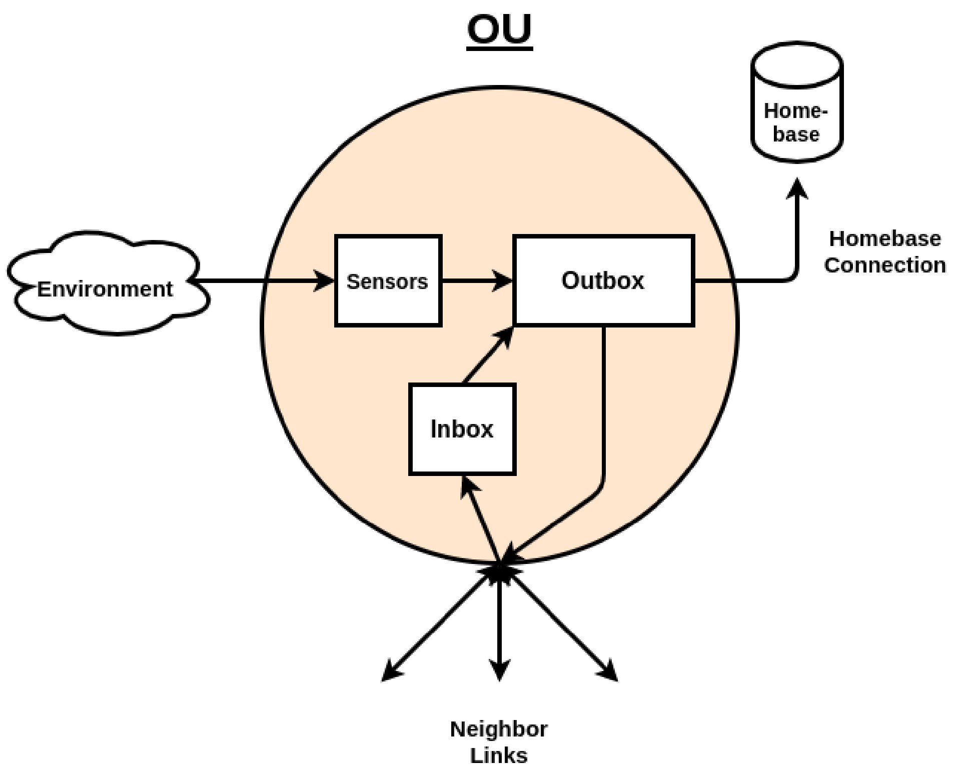

Figure 5.1: A simplified data flow diagram of a single OU.

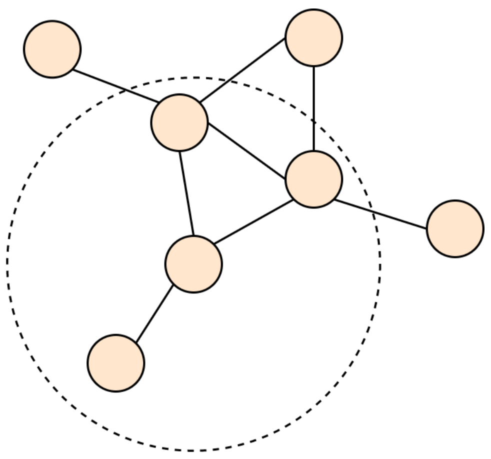

Figure 5.2: Mesh network showing links, and the communication range of one node.

This sleep phase is then followed by a time interval where they are awake and available, and is therefore the only time in which they will attempt to communicate. A sleeping phase followed by an awake phase is referred to as a cycle, and forms the basis for how nodes will try to schedule their activity. The time intervals for sleeping and being awake is uniform in the network, and so the nodes' schedules will overlap if their measures of time are similar.

5.3 Neighbors

Nodes that are in range of each other, and can maintain stable connections, will be referred to in this thesis as a *Neighbor*, and by extension, the neighbors of a given node will collectively be referred to as that node's *Neighborhood*. Every node will keep a record of their Neighbors and characteristics about them, like contact information, battery levels and link latency.

Periodically, the nodes will check up on each other through pings, and exchange data about their status. If a Neighbor is found to be unresponsive, the transmitting node will update this neighbor's record, and won't communicate with it for the rest of that cycle. If the Neighbor is unresponsive for a given number of cycles in a row, it will be tagged as inactive, and the transmitting node will stop trying to communicate with them.

5.4 Sinks

Outside of the local cluster, is the homebase: a static server in the background which is the final destination for propagated data and a source for authoritative time. Every cycle the OUs will test their connection with the homebase, and try to establish a link (fig 5.3). If this is successful, they will become a sink: a relay for fellow nodes who have not managed to establish this link themselves.

5.5 Paths

Overlaid on the mesh network is a path routing scheme which gives the nodes knowledge about who which sinks exist in the local network, and where they are in relation to themselves. The paths describe which sink they lead to and which intermediate steps are between the sink and the particular node. By using these, nodes can propagate their data to the homebase without requiring a direct connection themselves.

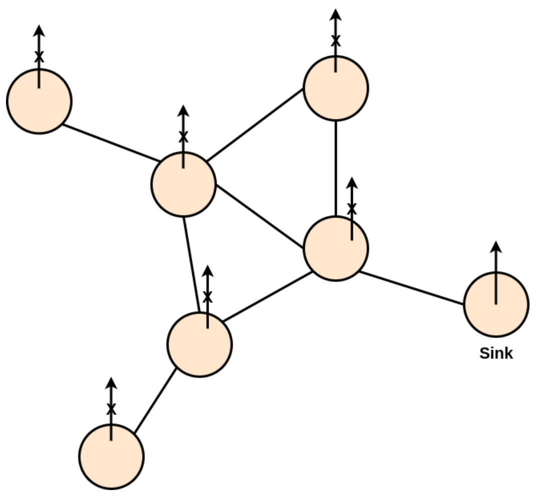

Figure 5.3: Mesh network showing which nodes have managed to establish a link to the homebase. Up-arrows with an 'X' in them, means they have not.

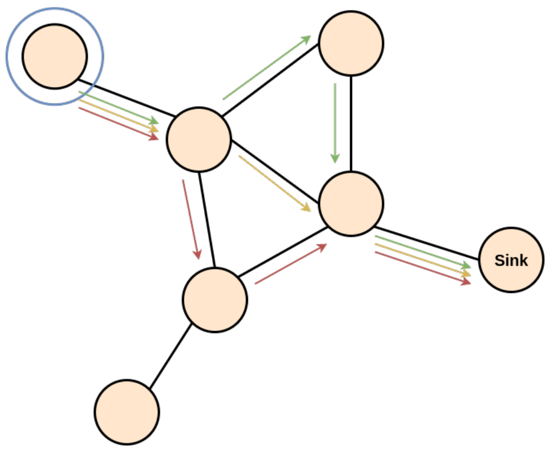

Figure 5.4: The useful paths for the circled node on the left, given the present sinks shown in fig 5.3.

5.6 Data Generation: Reports

Once every cycle, the OUs will generate a report with information pertaining to their status and activity. This report is arranged as a JSON-formatted string, which is then inserted into a Bundle Object. This object is a simple structure which contains: a list of string-objects, a length counter for this list, and a flag which indicates whether or not the list has reached an optimal length.

5.7 Data Propagation

The bundle object is then inserted into the node's Outbox, where it will be propagated towards a sink or to the Homebase – if certain conditions are in place (this is explained in section x.xx). Before propagating the bundle, the node will check its list of paths and select the best one (this process is described in section x.x.xx). The first step in this path will be one of this node's Neighbors, which will be the first recipient of the bundle on its journey to the homebase.

5.8 Time Synchronization

To compensate for clock drift, the nodes will periodically request time synchronization with their neighbors, so that they can have sleep schedules with as much overlap as possible. How they do this depends on whether or not there are sinks present in the network, if there are then they will use the path selection algorithm to figure out which neighbor is closest to the sink, as its presumed that they would have a more accurate clock. If not then the node will gather timestamps from all of its neighbors and aggregate them to get an average.

The details of this and how the nodes calculate new clock settings will not be described in this thesis, this is something that has been worked on by Sigurd in his thesis [9], and more can be found there.

6 Design

This chapter will cover important aspects of the design of the proposed solution.

6.1 Starting Phase and Operational Phase

The network as a whole goes through two phases of operation. The first is the *Starting Phase*, which is entered when a node is booted up for the first time, and ends when one of the OUs detects that a specified amount of time has passed since boot, and then floods a message to all the other nodes.

The purpose of this phase is for new OUs to do a broadcast upon boot, and form neighbor relationships based on the responses it received, and do an initial exchange of paths – if any sinks are discovered right from the get-go.

This phase is characterized by high activity and is meant to have the nodes remain awake for an extended period of time so that they can establish the network without worrying about divergent awake cycles. The message that is flooded after one node detects that a certain amount of time has passed since it booted (this will probably be the first node to boot) also serves as a mechanism for synchronizing the nodes' sleep schedules so that they can overlap from the start.

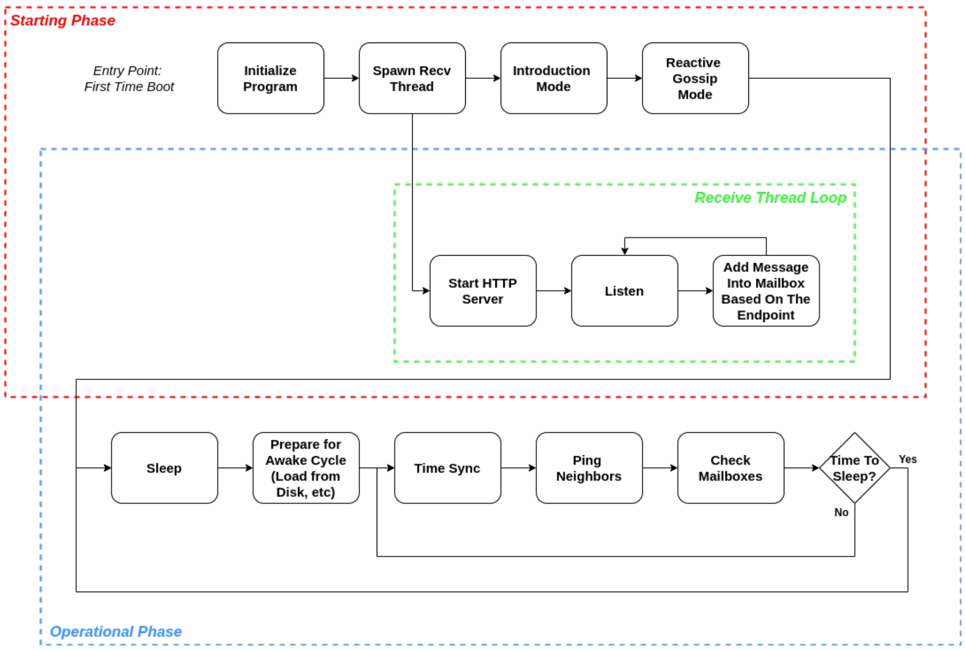

Figure 6.1: Simplified diagram of a node's life cycle. Figure is shared with Sigurd Karlstad.

The second is the *Operational Phase*, which is entered immediately after the starting phase is concluded, and goes on as along as there are nodes in the network. The Operational Phase is characterised by a sleeping period and a awake period, the former in order for the node to save some power by partially shutting down for the majority of a cycle, and the latter in order to perform tasks with other nodes – like transmit data to (or towards) the homebase, perform clock synchronization with other nodes, or adjust their knowledge of the network when changes occur.

The Operational Phase has 4 logically separate stages:

- Sleeping: the sensors will be read from as events happen and this data will be stored in memory, but nothing else is happening otherwise.
- Post-sleep: when sleep is exited, the node will load its state from disk, and turn on the receiver.
- Awake: perform tasks: propagate bundles, ping neighbors, and update the clock.
- Pre-sleep: save state and sensor buffer to disk, and turn off the receiver.

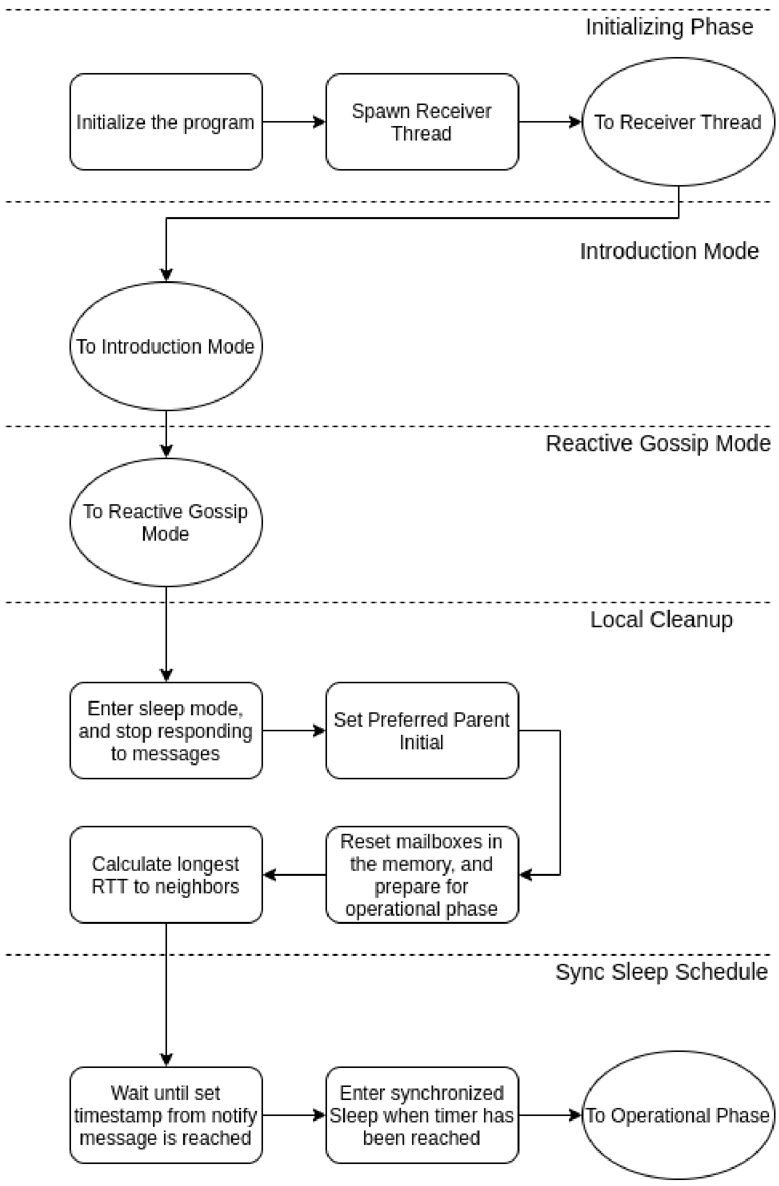
Initializing Phase
Initialize the program
Spawn Receiver Thread
To Receiver Thread
Introduction Mode
To Introduction Mode
Reactive Gossip Mode
To Reactive Gossip Mode
Local Cleanup
Enter sleep mode, and stop responding to messages
Set Preferred Parent Initial
Calculate longest RTT to neighbors
Reset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

Figure 6.2: Flow chart describing the starting phase. Figure is shared with Sigurd Karlstad.

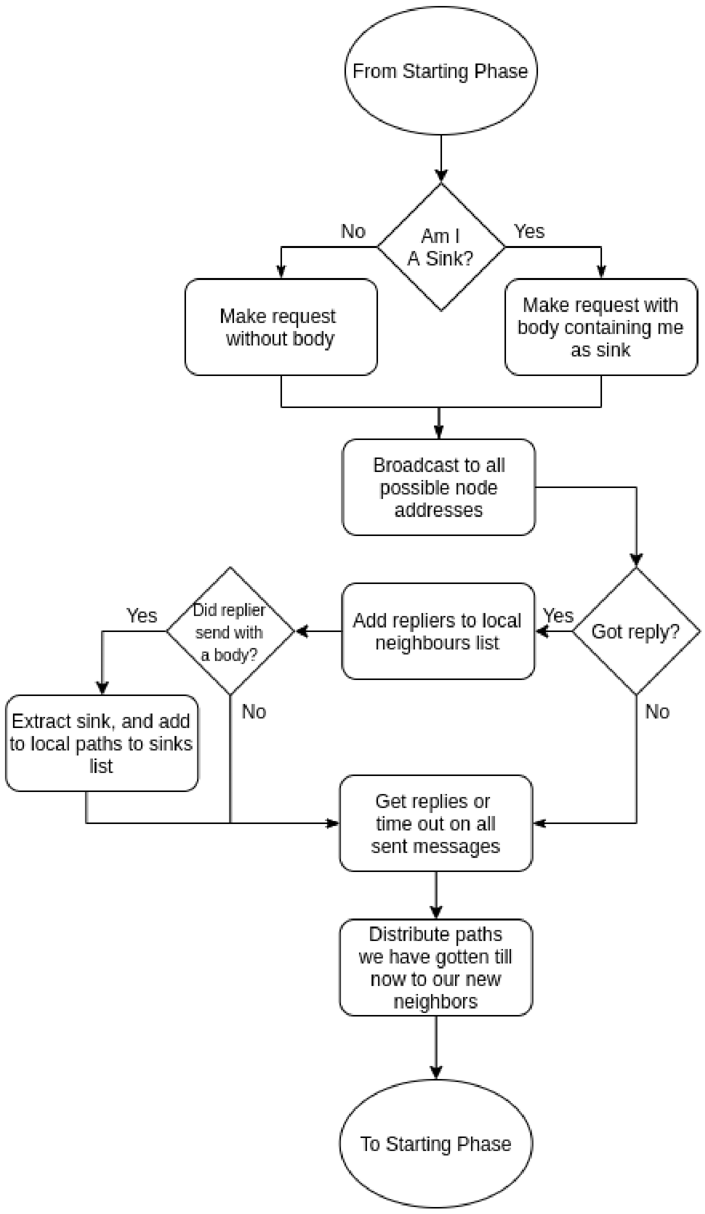
From Starting Phase
Am I A Sink?
No
Yes
Make request without body
Make request with body containing me as sink
Broadcast to all possible node addresses
Got reply?
Yes
No
Add repliers to local neighbours list
Did replier send with a body?
Yes
No
Extract sink, and add to local paths to sinks list
Get replies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

Figure 6.3: Flow chart describing the introduction mode. Figure is shared with Sigurd Karlstad.

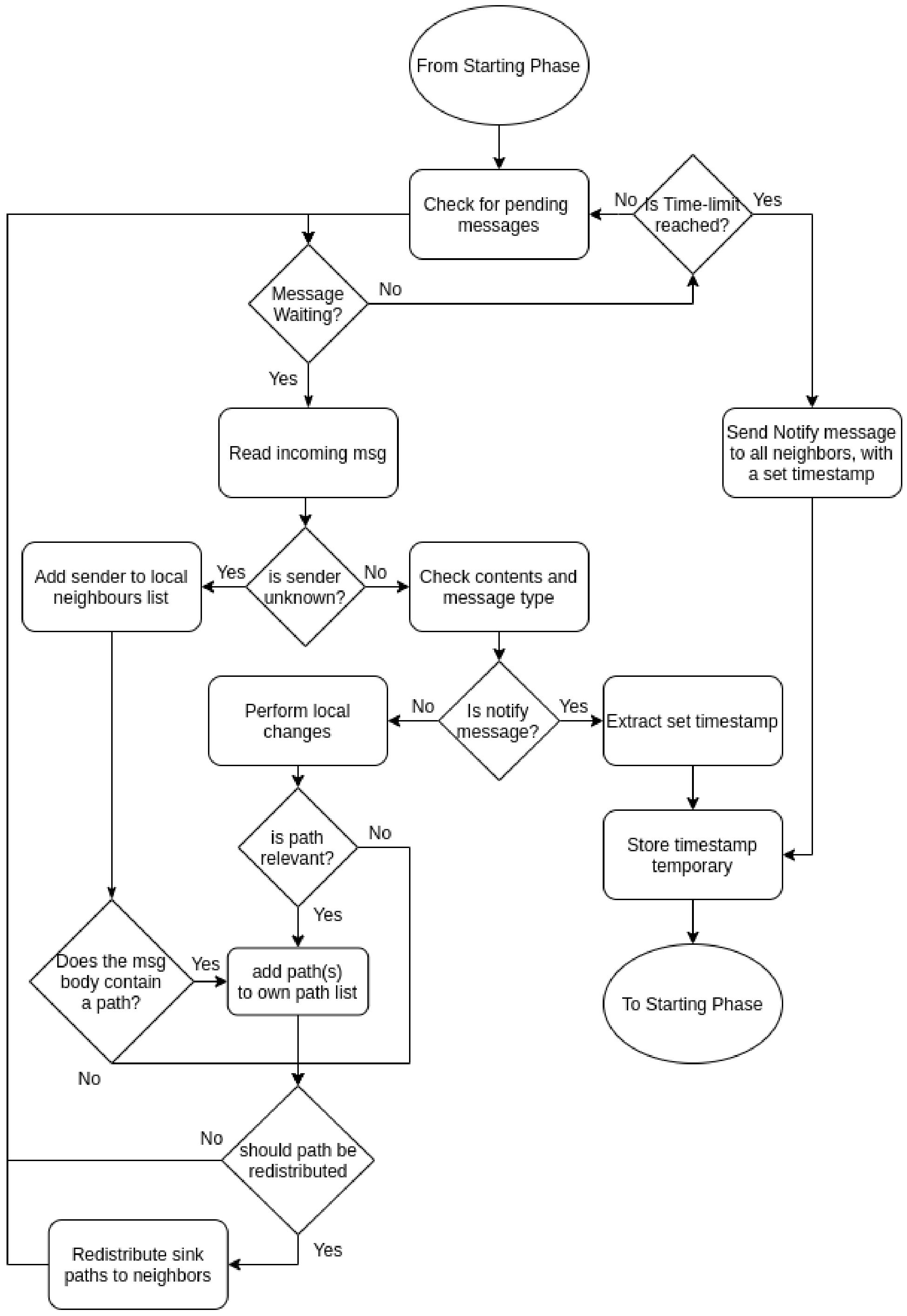
From Starting Phase
Check for pending messages
No
Is Time-limit reached?
Yes
Message Waiting?
No
Yes
Read incoming msg
Send Notify message to all neighbors, with a set timestamp
Add sender to local neighbours list
Yes
is sender unknown?
No
Check contents and message type
Perform local changes
No
Is notify 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 6.4: Flow chart describing the reactive gossip mode. Figure is shared with Sigurd Karlstad.

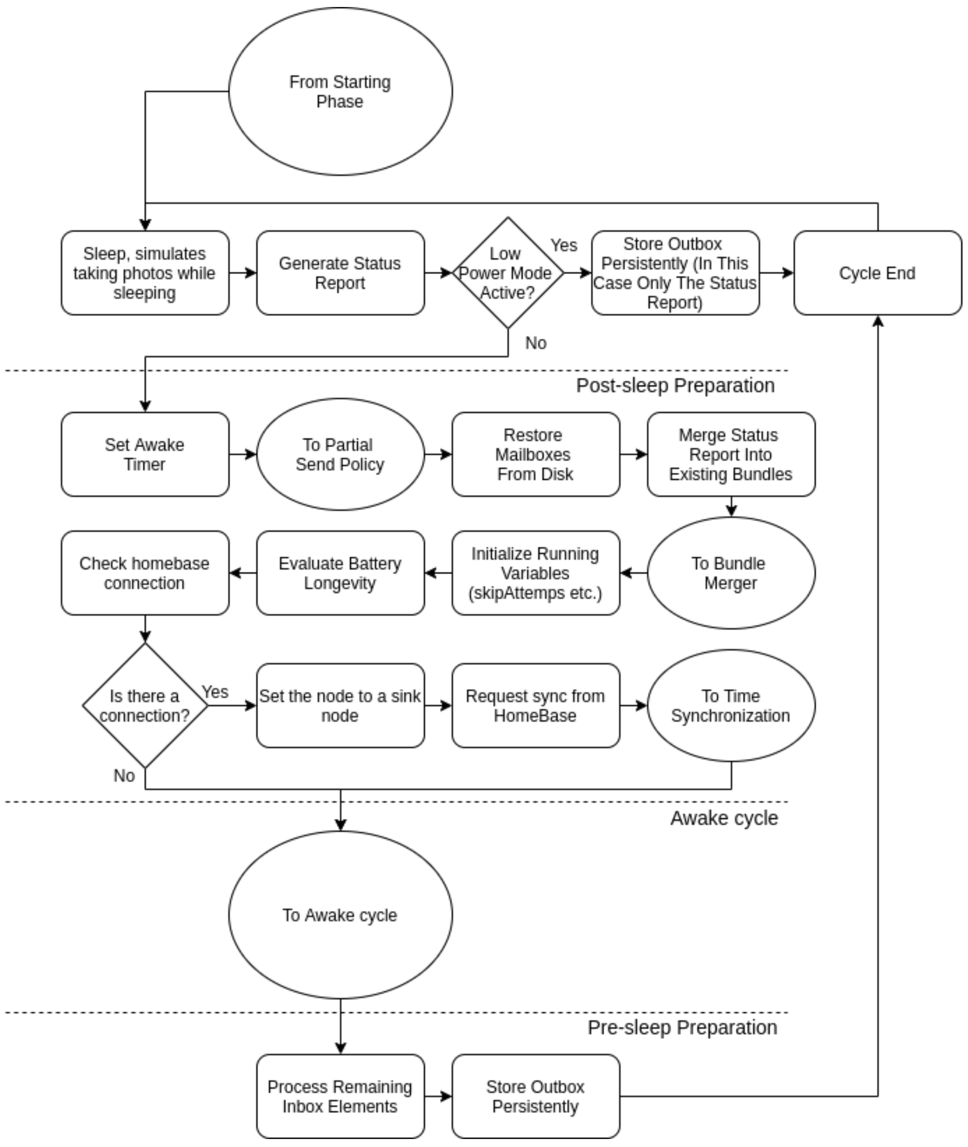
From Starting Phase
Sleep, simulates taking photos while sleeping
Generate Status Report
Low Power Mode Active?
Yes
Store Outbox Persistently (In This Case Only The Status Report)
Cycle End
No
Post-sleep Preparation
Set Awake Timer
To Partial Send Policy
Restore Mailboxes From Disk
Merge Status Report 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 6.5: Flow chart describing the operational phase. Figure is shared with Sigurd Karlstad.

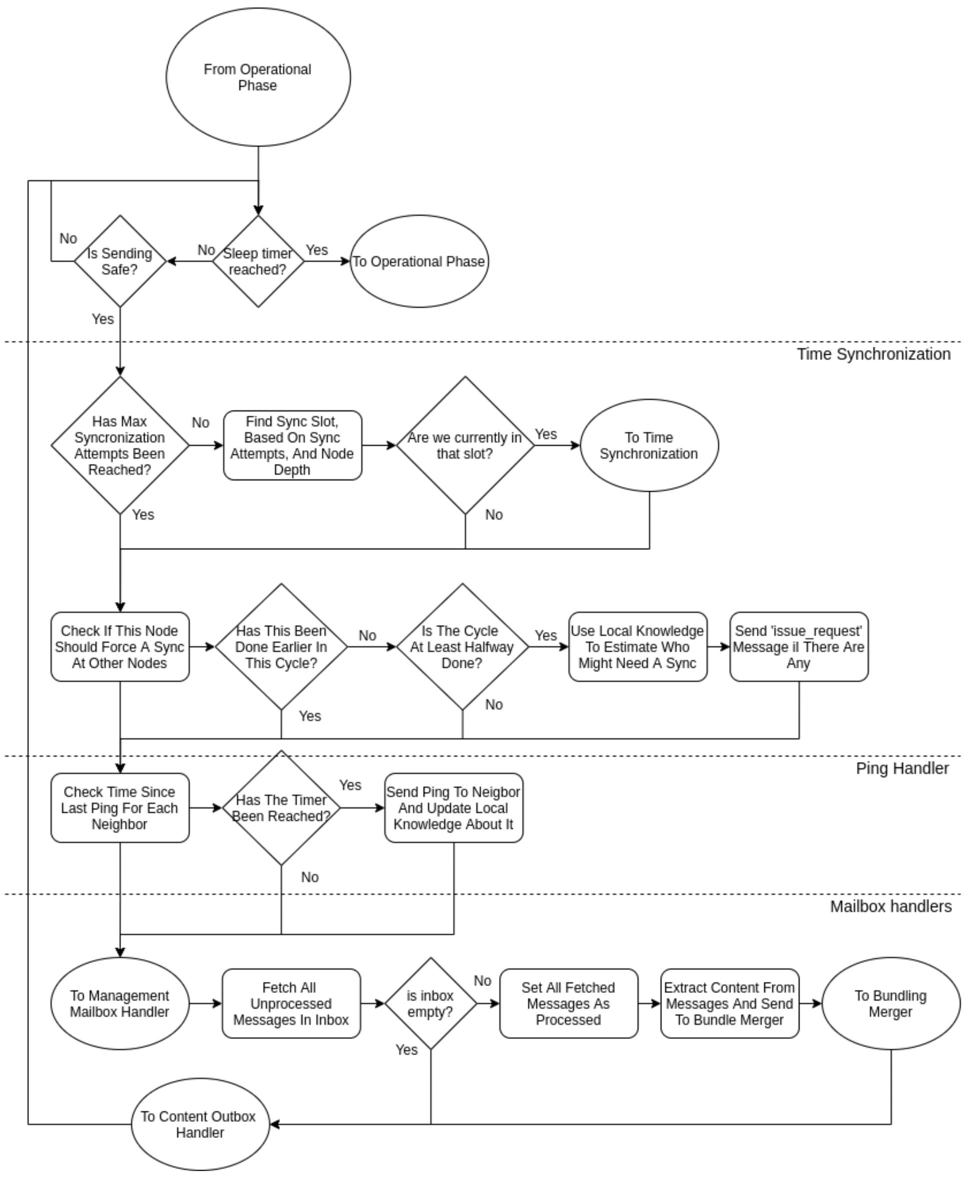
From Operational Phase
Sleep timer reached?
Yes
To Operational Phase
No
Is Sending Safe?
No
Yes
Time Synchronization
Has Max Syncronization Attempts Been Reached?
No
Find Sync Slot, Based On Sync Attempts, And Node Depth
Are we currently in that slot?
Yes
To Time Synchronization
No
Yes
Check If This 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
No
Yes
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 6.6: Flow chart describing the awake cycle. Figure is shared with Sigurd Karlstad.

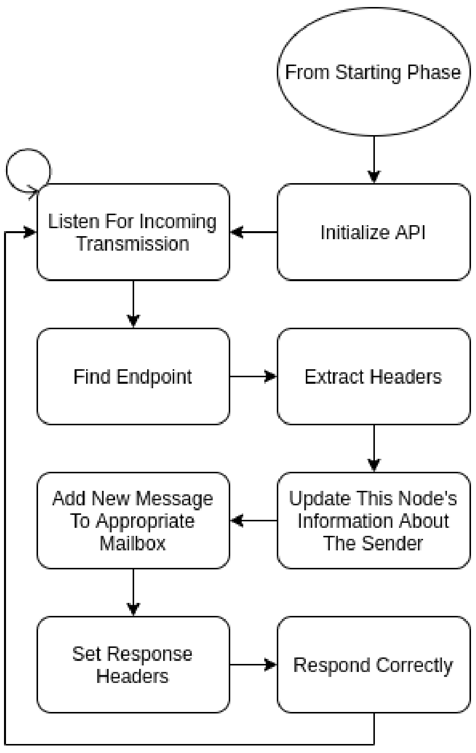
From Starting Phase
Initialize API
Listen For Incoming Transmission
Find Endpoint
Extract Headers
Update This Node's Information About The Sender
Add New Message To Appropriate Mailbox
Set Response Headers
Respond Correctly

Figure 6.7: Flow chart of the receiver thread. Figure is shared with Sigurd Karlstad.

The last stage is not included in fig 6.1, but it is the inverse stage of "Prepare for Awake Cycle", which would be after the "Time to Sleep?" conditional says Yes.

Additionally, if the node discovers that its battery has reached a certain threshold, it will stop communicating with its neighbors all together, and effectively exit the network. This is called the Low-Power Mode and is meant to get as much environmental sensing out of the current energy levels as possible, before it dies.

6.2 Generating Paths (How do we find paths?)

Every path leads to a sink, which is a node that is able to connect to the homebase and can then function as a gateway for nearby nodes which do not.

Paths are generated, either in the starting phase when the nodes are performing their initial introductions, or further down the line when a node manages to establish a connection, after previously being unable to.

Paths are discarded, either when a sink loses its connection – which prompts the nodes in the network to remove their paths to this sink–, or when a node which served as a intermediate step in any path becomes unresponsive, and has to be removed as a result.

Sinks will advertise themselves to the rest of the network by sending messages to their neighbors, who will themselves further propagate the message to their own neighbors and so on. This message contains a path which shows the receiving node which intermediate steps are between it and a sink. These messages are epidemically spread in the network from the sink once they establish a connection, but nodes will only spread the path further if they choose to accept it, which will prevent the messages from being spread over and over again.

Whether or not a path is accepted is dependent on some criteria. Incoming paths must:

- Not be already present in this node's path list (Duplication)
- Not contain this node as a step (Self-Reference)

Additionally, any paths that contain a neighbor as a step, will not be shared to that neighbor, as it will fail the Duplication criteria anyway.

6.3 Path Selection (How do we select paths?)

Of the hopefully numerous paths which has been collected by the node, one has to be selected for use in propagation. The nodes use a score to evaluate which path is the best to route through, which is based on 3 metrics:

- Path Length: the amount of intermediate nodes in a path.
- Neighbor Latency: the round trip time latency between the neighbor and this node.
- Neighbor Battery Reserves: percentage of power battery left on the neighbor

The numbers from each measurement is normalized so that they are balanced against each other. In addition, a weight is added to each metric so that one

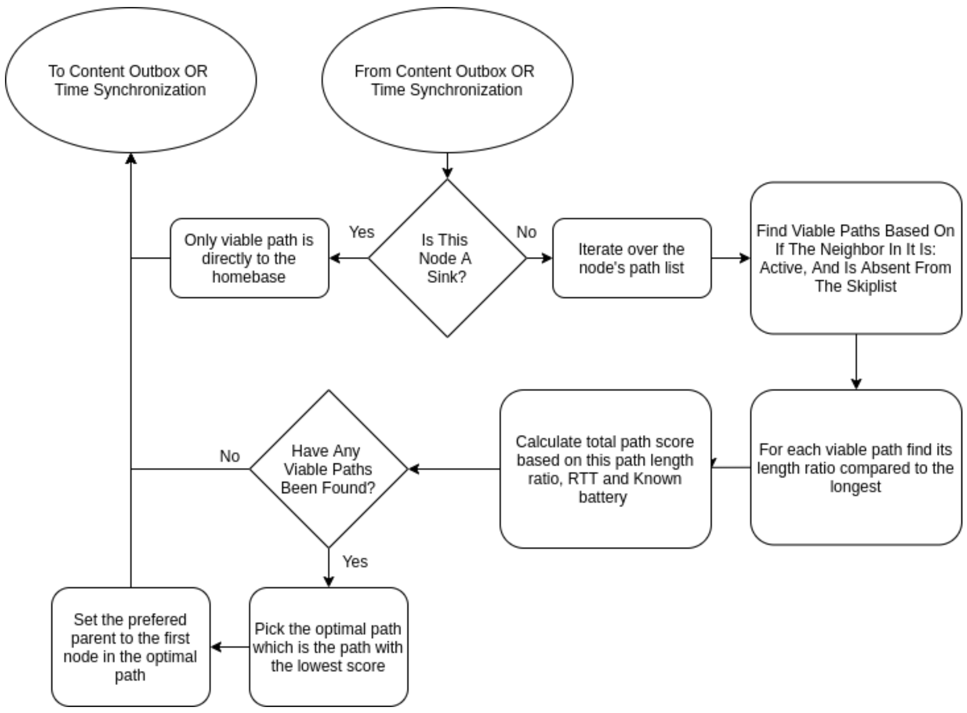
To Content Outbox OR Time Synchronization
From Content Outbox OR Time Synchronization
Is This Node A Sink?
Yes
No
Only viable path is directly to the homebase
Iterate over the node's path list
Find Viable Paths Based On If The Neighbor In It Is: Active, And Is Absent From The Skiplist
For each viable 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 6.8: Flow chart of the path score calculation algorithm. Figure is shared with Sigurd Karlstad.

aspect can be emphasized over another if needed – If battery reserves are thought to be more important than RTT latency for example. Whichever path has the smallest score, will be considered the best for routing by the node.

The formulas for calculating the scores can be seen here:
Partial Battery Score:

$$Bs = (100 - RemainingBatteryPercentage)/100$$

Partial RTT Score:

$$Rs = MostRecentRTT/HighestRecordedRTT$$

Partial Path Length Score:

$$Ps = PathLength/LongestKnownPath$$

Total Score

$$((A * Bs) + (B * Rs) + (C * Ps))/3$$

Where A, B, and C are the weights.

6.4 Monitoring Data and Reports

Monitoring and delivering debug data to interested parties, is accomplished through the generation and propagation of reports which the node's main loop is responsible for carrying out once per cycle.

The report consists of information about the following aspects:

- Operational Status: characteristics about the OU's current state, e.g.: battery levels, sink connection, and errors logged.
- Sensor Instrument Status: e.g.: readings since last report, sample value.
- Transmission Record: e.g.: transmissions since last report, transmissions in total, transmissions along different categories.

The implementation generates a report like the one seen in fig [example report]. When the node wakes up after sleep, the report is generated before starting the cycle, inserted into a new bundle, and then added to the node's Outbox.

6.5 Mailboxes

Message passing is the mode of communication in the proposed system.

Events, data, and requests are all propagated in the form of a message. Messages have headers which describes an operation or intent, and they also have a payload.

Each node has 3 mailboxes, each for a different purpose:

- Management: handles messages related to network management, and events that may trigger a recalibration – like being notified of: new paths, new sinks, lost sinks, lost non-sink nodes, etc.
- Inbox: contains newly-received messages with the type of 'content'.
- Outbox: contains messages with the type of 'content' that are ready for transmission.

'content' is a universal descriptor for anything that is meant to be propagated towards and received by the homebase. The payload is presently a JSON-formatted string (report), but it is never processed by intermediate nodes, so

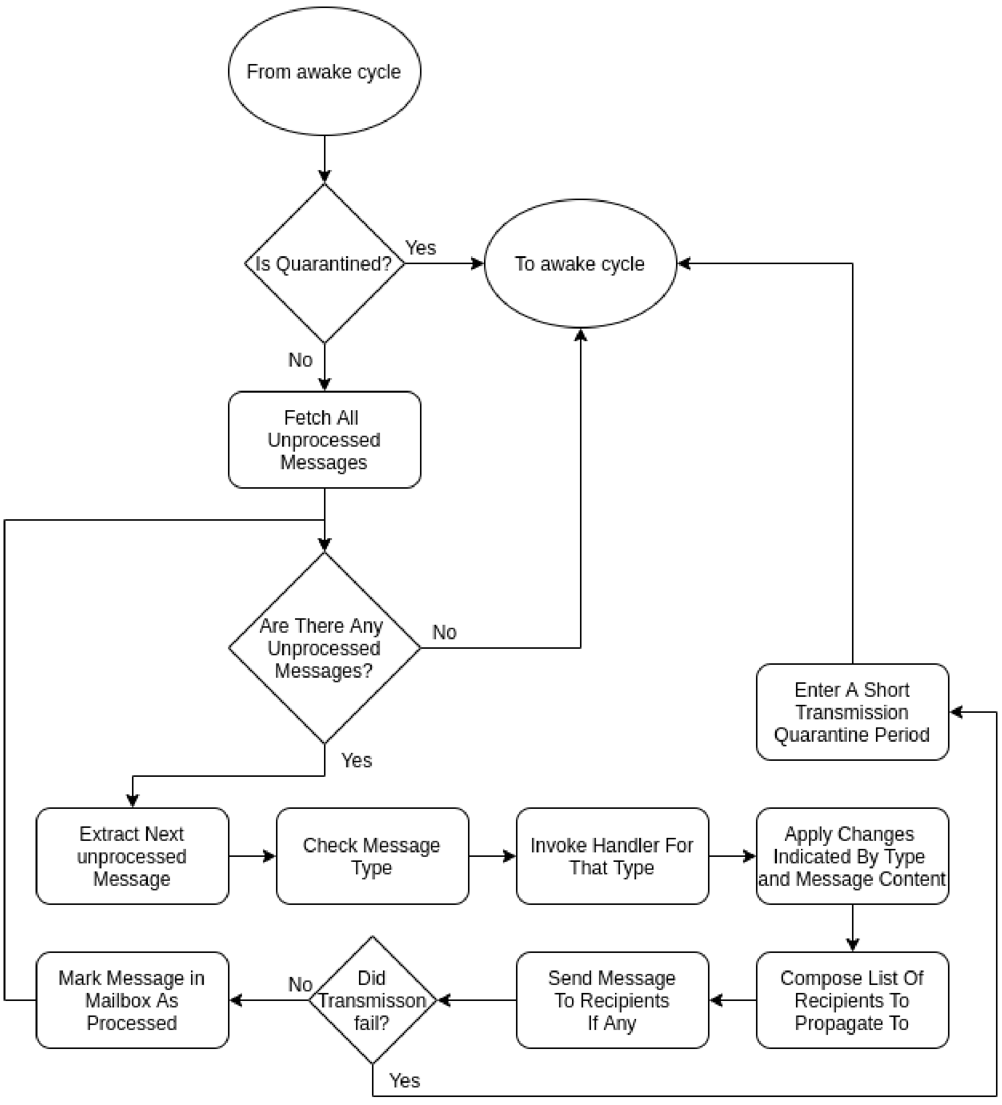
From awake cycle
Is Quarantined?
Yes
To awake cycle
No
Fetch All Unprocessed Messages
Are There Any Unprocessed Messages?
No
Yes
Enter A Short Transmission Quarantine Period
Extract Next unprocessed Message
Check Message Type
Invoke Handler For That Type
Apply Changes Indicated By Type and Message Content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

Figure 6.9: Flow chart describing how management messages are handled. Figure is shared with Sigurd Karlstad.

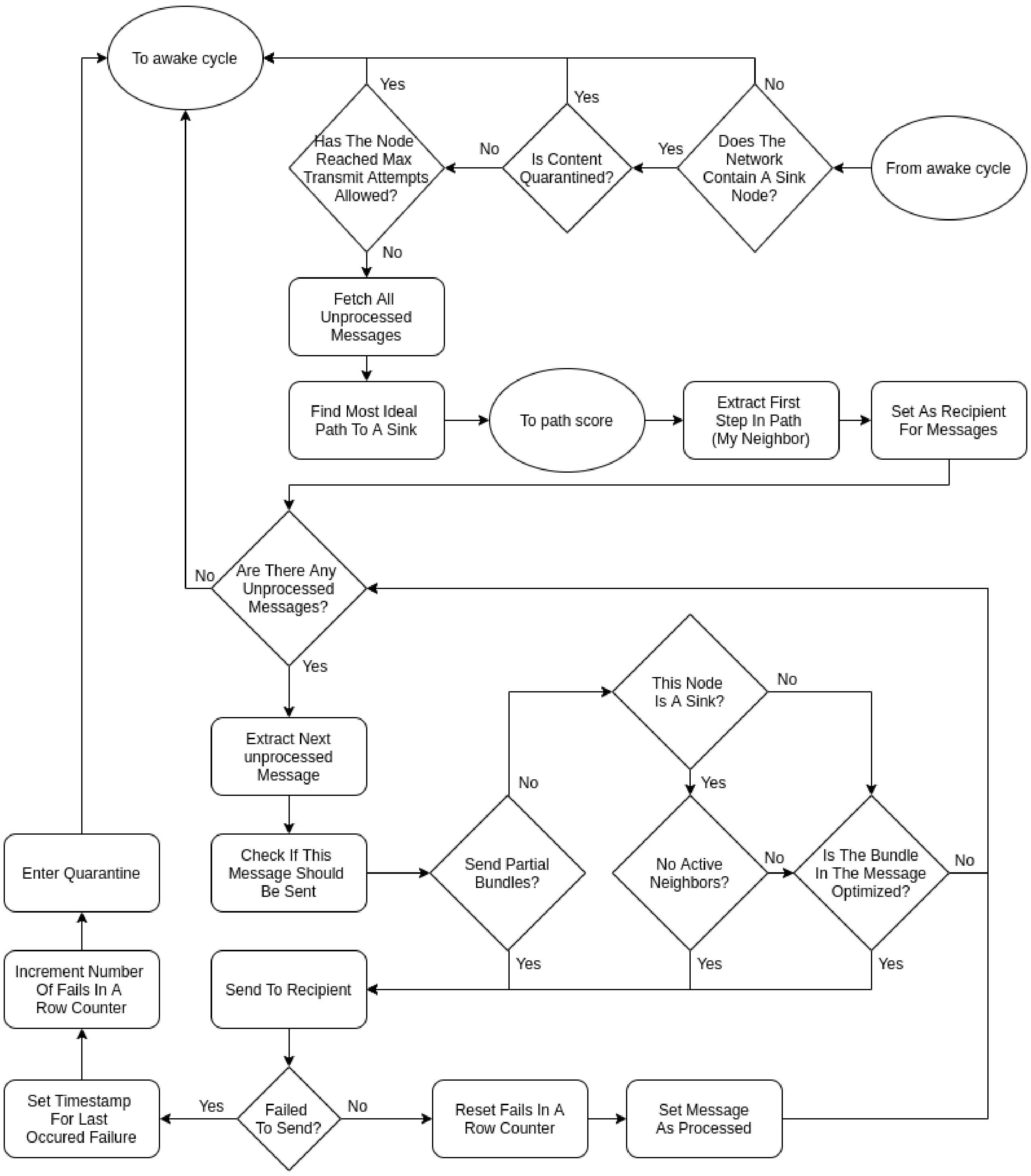
To awake cycle
Yes
Yes
No
Has The Node Reached Max Transmit Attempts Allowed?
No
Is Content Quarantined?
Yes
Does The Network Contain A Sink Node?
From awake cycle
No
Fetch All Unprocessed Messages
Find Most Ideal Path To A Sink
To path score
Extract First Step In Path (My Neighbor)
Set As Recipient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

Figure 6.10: Flow chart describing how content messages are handled. Figure is shared with Sigurd Karlstad.

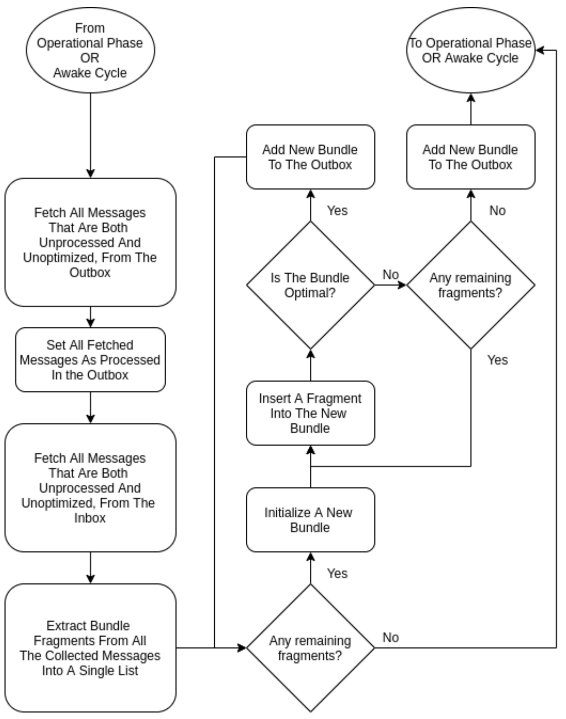
From Operational Phase OR Awake Cycle
Fetch All Messages That Are Both Unprocessed And Unoptimized, From The Outbox
Set All Fetched Messages As Processed In the Outbox
Fetch All Messages That Are Both Unprocessed And Unoptimized, From The Inbox
Extract Bundle Fragments From All The Collected Messages Into A Single List
Any remaining fragments?
Yes
No
Initialize A New Bundle
Insert A Fragment Into The New Bundle
Is The Bundle Optimal?
Yes
No
Add New Bundle To The Outbox
Any remaining fragments?
Yes
No
Add New Bundle To The Outbox
To Operational Phase OR Awake Cycle

Figure 6.11: Flow chart describing the algorithm for merging bundles.

it could in theory be anything of interest.

All other messages are processed by the management mailbox, and has a variety of descriptors such as "new_path", "ping", etc.

6.6 Constructing Bundles / Data Combination

Data fusion or aggregation is a common method for reducing data volumes – PEGASIS and LEACH are examples here. The reasons for doing so are plenty: save space on devices, reduce message payload sizes, reduce processing times, etc. In this thesis, we will be concerning ourselves with another reason: reducing necessary transmissions.

From the author's point of view, it is not obvious how to aggregate or fuse diagnostic/debug/status data into a form that does not disentangle it from the node it is about, or lose important information in general.

So, instead of reducing the volume of data, the proposed solution instead combines reports into larger volumes called bundles, so that they at least can be transmitted at the same time. Initiating a transmission, turning on a transceiver and so on, is a more expensive operation than sending data itself. By this measure then, performing 3 separate transmissions would be more expensive in total, than performing a single one with 3 times the data.

6.7 Sending Partial Bundles

Data is propagated towards the sink or the homebase by determining checking the sink-path list and selecting the best path for propagation. Data is sent in the form of bundles containing multiple reports. In order to get a reduction in the total amount of transmissions that is accumulated in the network, bundles cannot be sent unless either of the following conditions hold true:

- the bundle has an optimal length,
- or the node is allowed to send "partial" bundles, meaning bundles which has not reached its length.

The first condition is in place to reduce the overall amount of transmissions which take place during propagation of these reports, and only forward the bundle if it has accumulated a certain amount of reports, upon which it will be declared as *optimal*. The second condition is a mechanism which allows certain nodes to forward bundles, whether or not they are optimal. This is intended to be a way for the network to get a higher degree of throughput, and to avoid having to accumulate reports on a single node before it can be sent.

This mechanism has been named as the node's *Partial Bundle Policy*, and is calculated based on the likelihood of other nodes using this one as a relay towards a sink. Without this, the nodes would simply have to wait for a number of cycles equal to the optimal length of a bundle before anything arrives at the homebase.

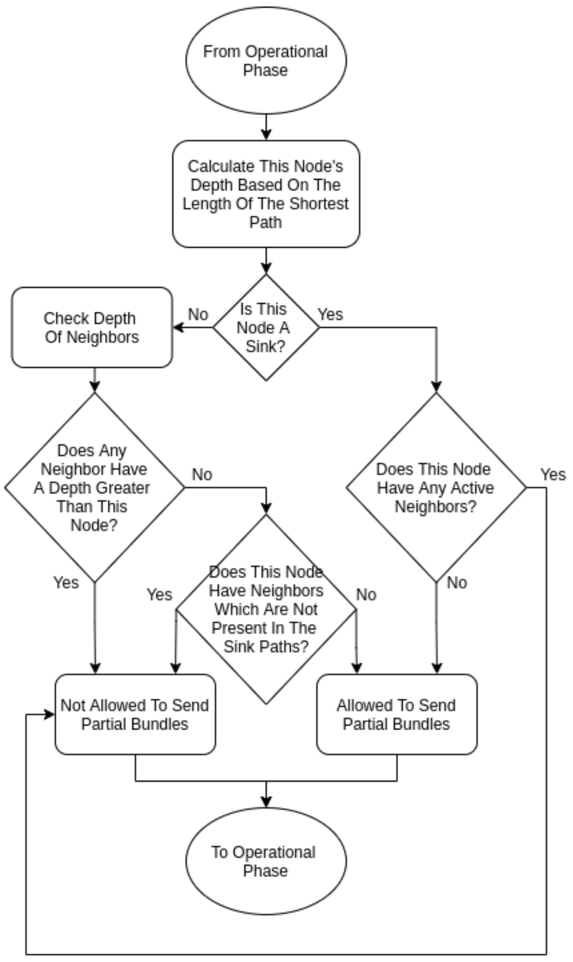
From Operational Phase
Calculate This Node's Depth Based On The Length Of The Shortest Path
Is This Node A Sink?
No
Yes
Check Depth Of Neighbors
Does Any Neighbor Have A Depth Greater Than This Node?
No
Yes
Does This Node Have Any Active Neighbors?
Yes
No
Does This Node Have Neighbors Which Are Not Present In The Sink Paths?
Yes
No
Not Allowed To Send Partial Bundles
Allowed To Send Partial Bundles
To Operational Phase

Figure 6.12: Flow chart describing the algorithm for deciding the Partial Bundle Policy

7 Implementation

This chapter is about the implementation of the prototype system which the author in collaboration with Sigurd Karlstad 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.

- Programming Languages:
  - Golang v1.16 [5]
  - Python v3.9.4 [15]
- Golang Packages:
  - Termui v3.1.0 [4]

The code for the prototype nodes was implemented in Golang [5], which is a 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 [15] 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.

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 (this is 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 Karlstrøms thesis, it is still relevent for Sigurd Karlstad, as the main focus of the OUs is to record environmental readings. OUs 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 amount 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.

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 [22], 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.

8 Validation

In this chapter we will aim to confirm the correctness of selected parts of the system's functionality as they were outlined in the architecture and design chapters.

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

However the Bundling Validation is created only by the author of this thesis, and deserve full credit for it.

8.1 Validation Setup

All the following 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:

- **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 test and 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 network disruption.

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 correct. After running the test we expect that the generated and accepted paths

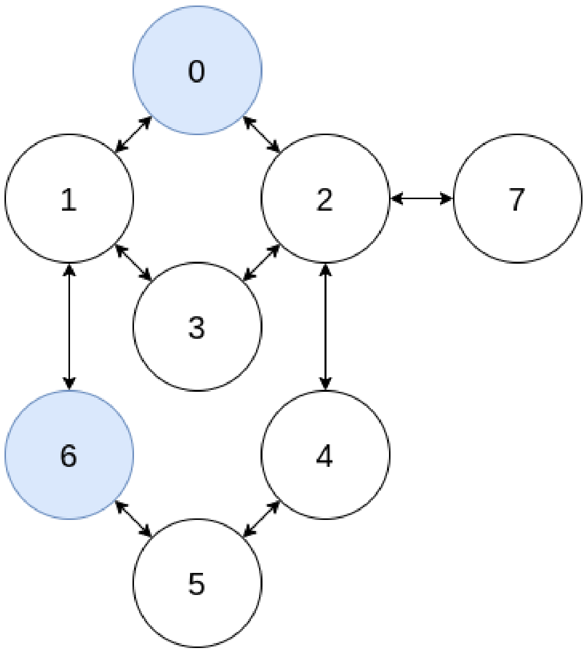

Figure 8.1: A network with two sinks. Figure shared with Sigurd Karlstad

Node	Paths to Sink Node 0	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 of the paths which we expect to be generated in the stated configuration. Table shared with Sigurd Karlstad

will look similar in both cases to the paths shown in table 8.1.

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 table 8.1.

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. Figure 8.3 describes the method with figures.

Connection Disruption Recovery

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

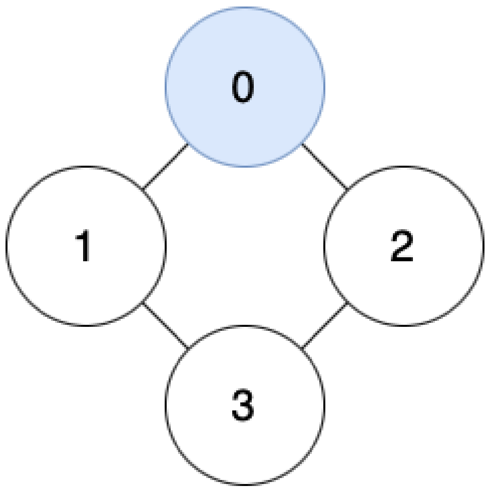

Figure 8.2: A network with one sink. Figure shared with Sigurd Karlstad.

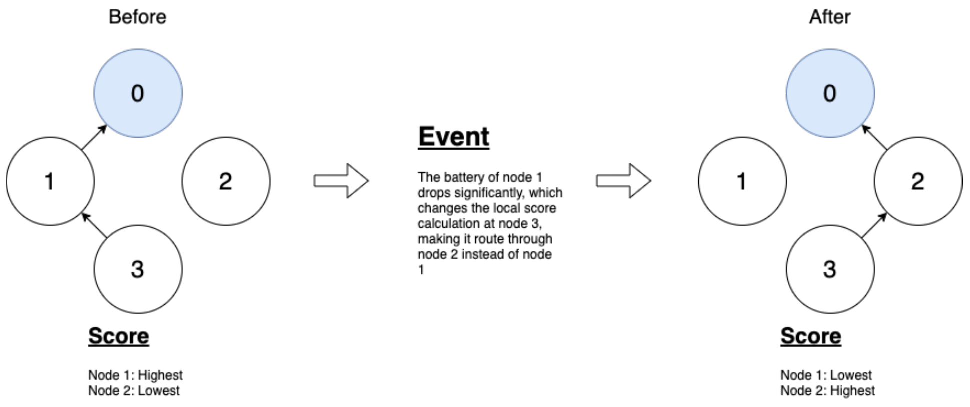
Before
0
1
2
3
Score
Node 1: Highest
Node 2: Lowest
Event
The battery of node 1 drops significantly, which changes the local score calculation at node 3, making it route through node 2 instead of node 1
After
0
1
2
3
Score
Node 1: Lowest
Node 2: Highest

Figure 8.3: Figure describing the method for the optimal parent calculation validation. Figure shared with Sigurd Karlstad.

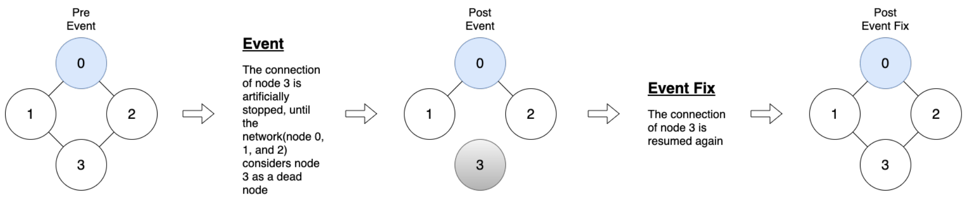
Pre Event
0
1
2
3
Event
The connection of node 3 is artificially stopped, until the network(node 0, 1, and 2) considers node 3 as a dead node
Post Event
0
1
2
3
Event Fix
The connection of node 3 is resumed again
Post Event Fix
0
1
2
3

Figure 8.4: Figure describing the method for the connection disruption recovery validation. Figure shared with Sigurd Karlstad.

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. Figure 8.4 describes the method with figures.

8.2.2 Bundling Validation

These bundling validations are aimed to test and validate the bundling and report mechanisms of the system and that it operates as was outlined in the architecture and design sections.

Eventual Arrival For Reports

To validate if: reports which are generated on the nodes all eventually arrive at the homebase. This test will show that reports are not lost due to some algorithmic error in the propagation mechanism, or the bundle merge mechanism.

Method: Using the network structure Sca8 from figure 9.1, run the system normally. Track the reports which are generated in between 0-60 cycles. Allow the system to run until cycle 70 so that stragglers get a chance to arrive. Do this for all of these settings: Bundling with Partial Bundle Policy, Bundling without Partial Bundle Policy, and Bundling disabled. If there are no gaps in the cycle numbers in the reports, then we will know that this validation was successful.

8.3 Validation Results

8.3.1 General Network Results

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.

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.

TRANSMISSION #	TRMS REASON	PARENT	N 1 SCORE	N 2 SCORE
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 OCC.)				
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 8.2: Table describing the sequence of events where the node changed its parent. The event mentiond involves setting NODE 1's battery to 30%. Table shared with Sigurd Karlstad.

Optimal Parent Calculation

By running the test and observing the transmissions and debug updates from the nodes, we were able to create table 8.2 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 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.

Connection Disruption Recovery

N	PATHS
0	
1	[[0], [3, 2, 0]]
2	[[0], [3, 1, 0]]
3	[[2, 0], [1, 0]]

Table 8.3: Paths before disruption. Figure shared with Sigurd Karlstad.

N	PATHS
0	
1	[[0]]
2	[[0]]
3	[]

Table 8.4: Paths after disruption. Figure shared with Sigurd Karlstad.

N	PATHS
0	
1	[[0], [3, 2, 0]]
2	[[0], [3, 1, 0]]
3	[[2, 0], [1, 0]]

Table 8.5: Paths after rejoin. Figure shared with Sigurd Karlstad.

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 8.3, 8.4, and 8.5.

The first, table 8.3, shows that before inducing the artificial disruption the nodes had all the sink paths that is expected from the configuration shown in fig 8.2. The second, 8.4, 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 8.5, 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 8.3, and the second table 8.4 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 8.4, and the third table 8.5 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 Bundling Results

Eventual Arrival For Reports

This validation was done in parallel with the experiment that measured the delay in packages arriving in section 9.2.2. Looking at the homebase's counter for how many reports it has received from each node, it was observed that there was no gap in the reports' cycle numbers. We can therefore conclude the no reports are lost during propagation or merging.

/9 Experiments

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

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

However the Bundling Experiments is created only by the author of this thesis, and deserve 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.

9.2 Experiment Design

For the experiments, we will be using a set of network configurations which can be seen in fig 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 system's scalability.

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 amount of paths they have. We add these counts together and obtain a total amount of paths for each network configuration.

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 favourable 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

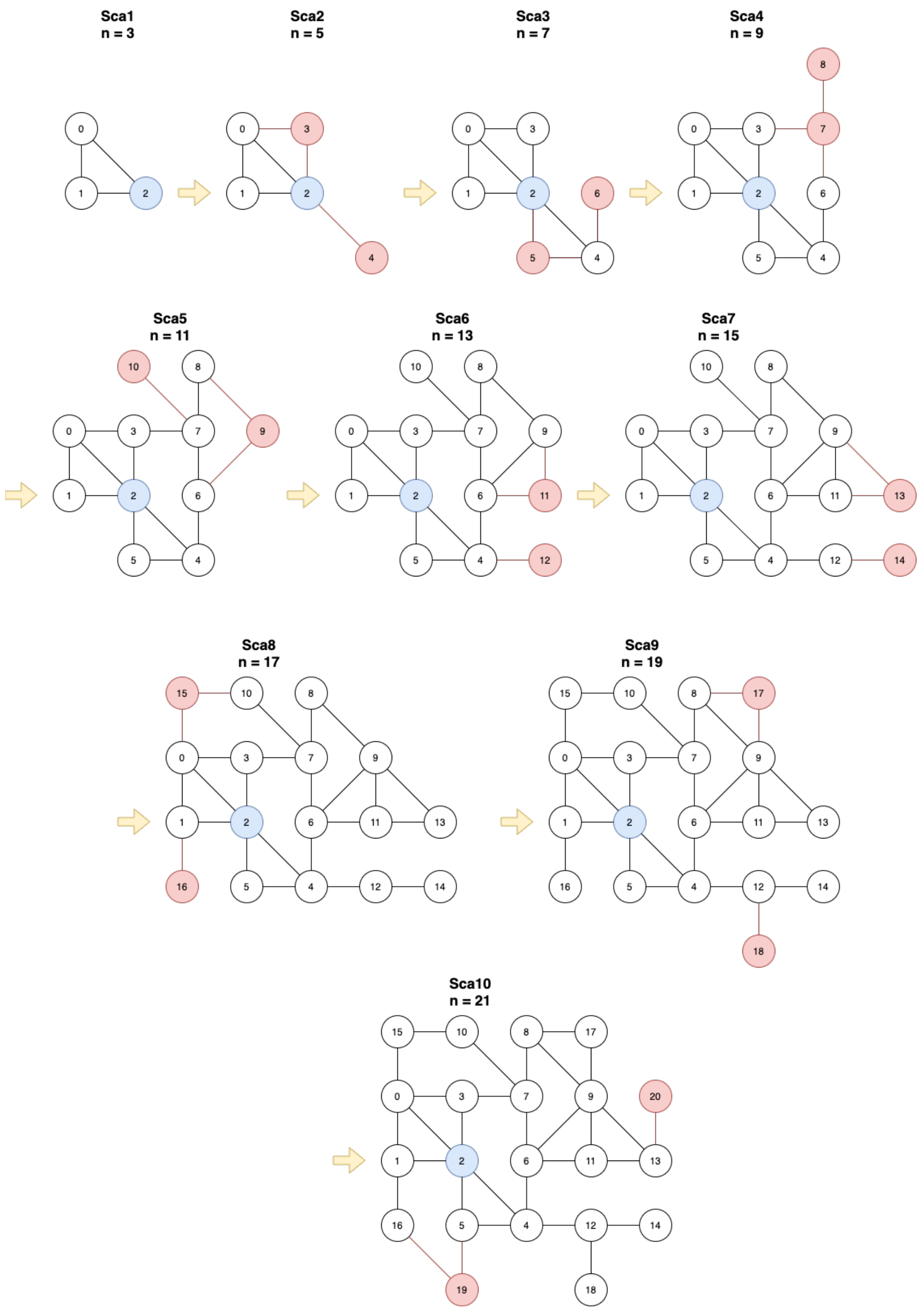

Figure 9.1: A set of network configurations which are named Sca1-Sca10. Figure shared with Sigurd Karlstad.

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.

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.

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 do not contain a sink initially. After the starting phase has finished, we send a message to node converting it into a sink. We then record how many transmissions this event caused to be generated for each network configuration.

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 shows 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 sca 10. 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 Bundling Experiments

This section presents the method and reasoning behind the experiments which aims to measure various aspects of the bundling and data propagation implementation.

Transmissions with Bundling, Bundling with Partial Bundle Policy, and without bundling

This experiment aims to measure the difference in content transmissions when the nodes have: Bundling enabled with Partial Bundle Policy enabled, Bundling enabled with Partial Bundle Policy, and Bundling disabled all together. This is important to investigate as it will show how much of an impact bundling as a feature has on the nodes' transmission count when forwarding debug information.

Method: This experiment runs through a selection of the different network configurations shown figure 9.1, using sca2, sca4, sca6, sca8, and sca10. We start the networks in the starting phase, and record the content transmissions done for 15 awake cycles(including the starting phase as the 0th cycle). No special actions are performed during execution, and the network is allowed to run normally. This setup is run 3 times, and each time with the following variation: 1. bundling enabled, and Partial Bundle Policy enabled; 2. bundling enabled, but with Partial Bundle Policy disabled; 3. bundling disabled and Partial Bundle Policy also disabled as a consequence. In this experiment, the bundling size is set to 3.

Transmissions with different bundling sizes

This experiment aims to measure the impact on content transmissions when the nodes are using different sizes for optimal bundles. Partial Bundle Policy is going to be enabled, so the behavior will not be as predictable as if every node simply saved up their fragments and propagated their optimal bundle on the same cycle, thus making this an interesting case to investigate.

Method: In this experiment we will be using the following settings: network configuration is Sca10 (fig 9.1), battery decrease turned off, bundling enabled with Partial Bundle Policy enabled as well, the timespan will be from the starting phase to 30 completed awake cycles. These settings will be used in 5 runs, where each run will be initialized with a different optimal bundle size – the span of which will be: 1, 3, 5, 7 and 9.

Delay in packages arriving with bundling partial policy, and without bundling

This experiment aims to measure the difference between two points in time: when reports are generated, and when they reach the homebase. Varying between bundling with partial policy, bundling without partial policy, and no bundling, will allow us to see how the throughput is affected by these modes in comparison with each other.

Method: In this experiment we will be using the following settings: network configuration is Sca8 (fig 9.1), battery decrease turned off, optimal bundle size set to 3 where applicable, the timespan will be from the starting phase to 70 completed awake cycles. As an addendum to that last point, we will only be tracking reports that were generated up until the 60th cycle. The 10-cycle gap is to give in-transit reports time to arrive to the homebase, but we are not counting reports generated after the 60th. To calculate the differences, we will be extracting the cycle number from each report and compare it to the cycle number that the sink registered when it sent the report directly to the

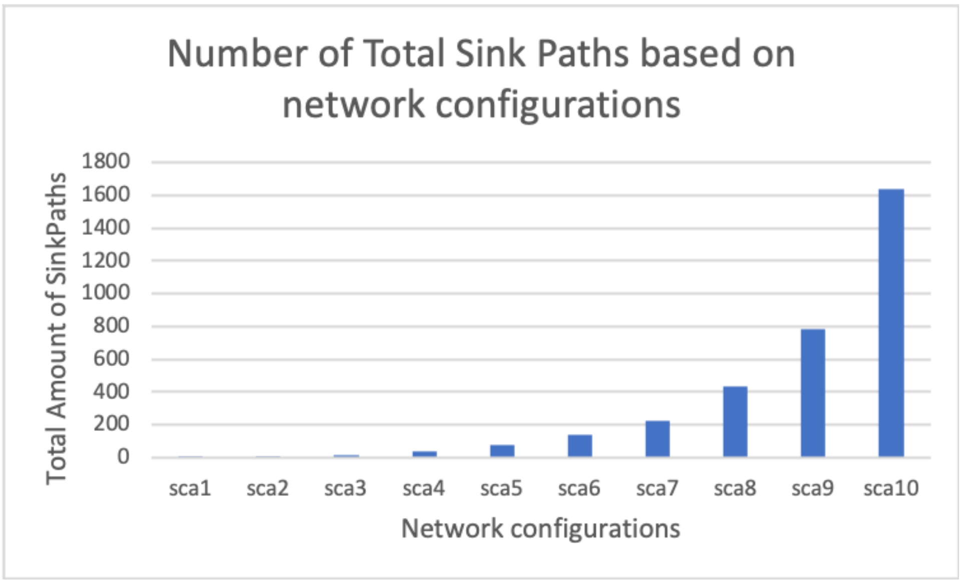

Figure 9.2: Graph showing how the number of total sink-paths increase with network size. Figure shared with Sigurd Karlstad.

homebase.

9.3 Experiment Results

This section will present the results from conducting the experiments outlined in the section 9.2.

9.3.1 General Network Results

This section present the results from conducting the general network experiments outlined in section 9.2.1

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 fig 9.2.

By looking at the graph in figure 9.2 we can see that by increasing the amount of nodes in the system, which creates new possible paths, they end up grow-

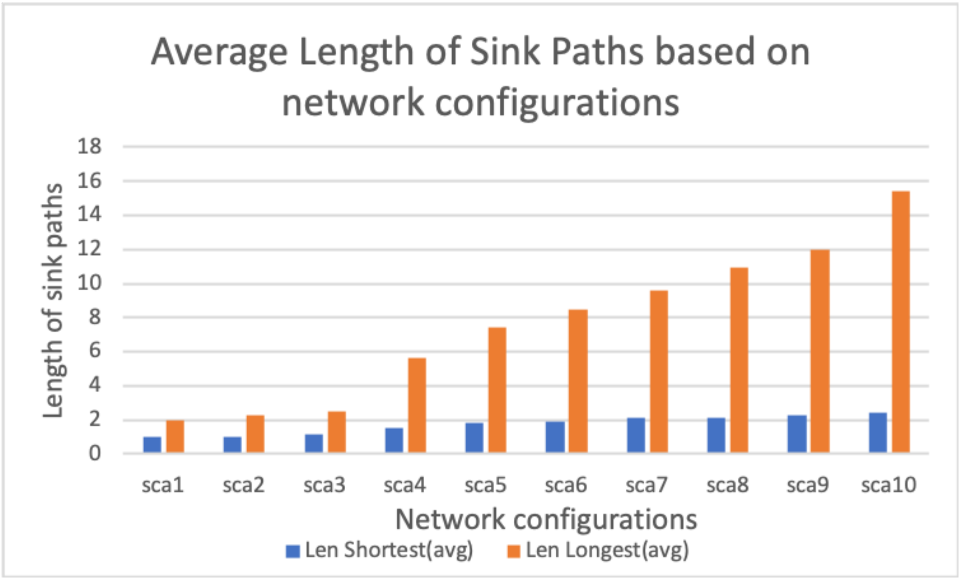

Figure 9.3: Graph showing how the length of the shortest path on each node, and the longest path on each node grows with the network size. Figure shared with Sigurd Karlstad.

ing exponationally. This means that while the network size is low the path sharing, and path acceptance algorithm works well, but when as the network grows larger, the amount of paths will grow exponentially rather than run proportionally to 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.

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 fig 9.3.

By looking at the graph in figure 9.3 we can see that while increasing the amount 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

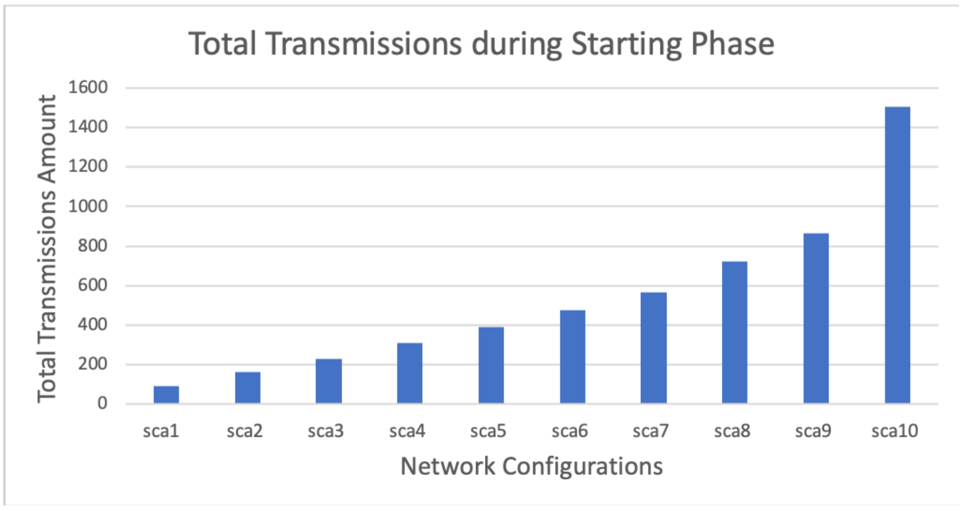

Figure 9.4: Graph showing how the total number of transmissions in the starting phase grows with the network size. Figure shared with Sigurd Karlstad.

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.

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.

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 fig 9.4.

By looking at the graph in figure 9.4 we can see that while increasing the amount of nodes in the system, the total amount of transmissions grows linearly with

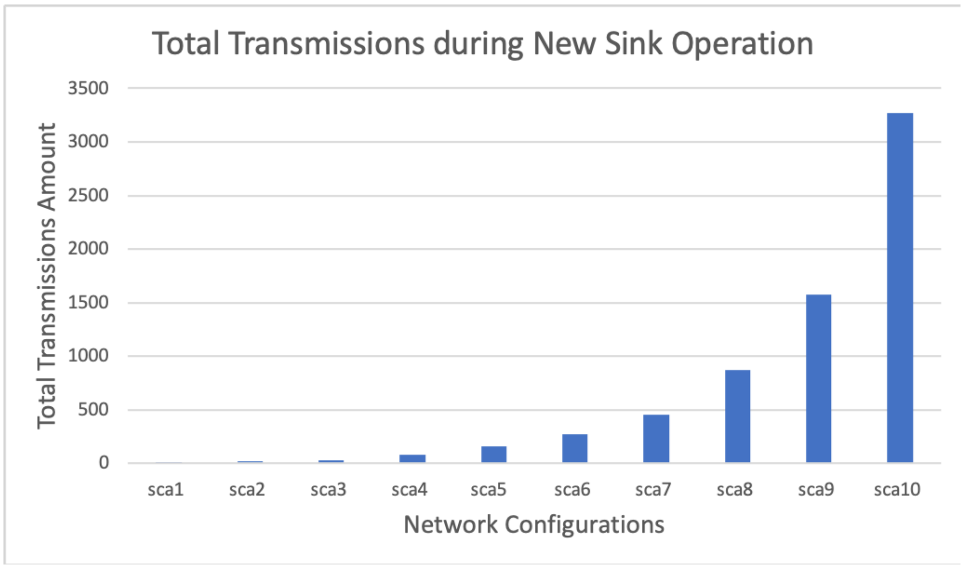

Figure 9.5: Graph showing how the total number of transmissions during a new sink operation grows with the network size. Figure shared with Sigurd Karlstad.

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 amount 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.

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 fig 9.5.

By looking at the graph in figure 9.5 we can see that while increasing the amount 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 x.x.x[ref to Sink Path Scaling secion] were 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.

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 able to create a graph containing all the transmissions between nodes during the normal operations in the operational phase. The graph is shown in fig 9.6.

By looking at the graph in figure 9.6 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.

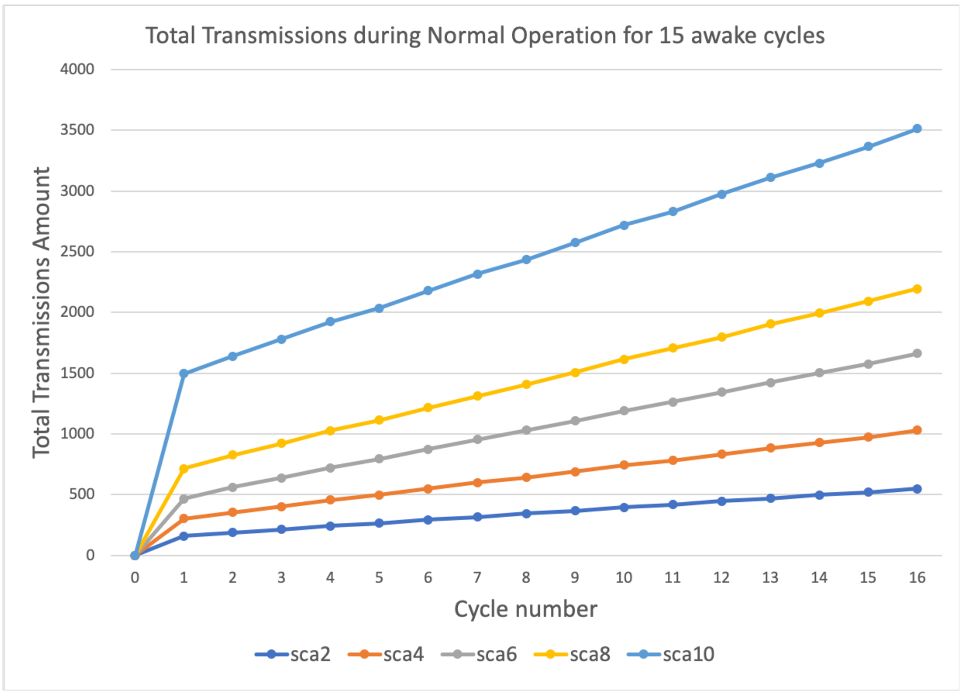

Figure 9.6: Graph showing how the total number of transmissions grows with the network size when the system is allowed to start in the starting phase and run over 15 cycles. Figure shared with Sigurd Karlstad.

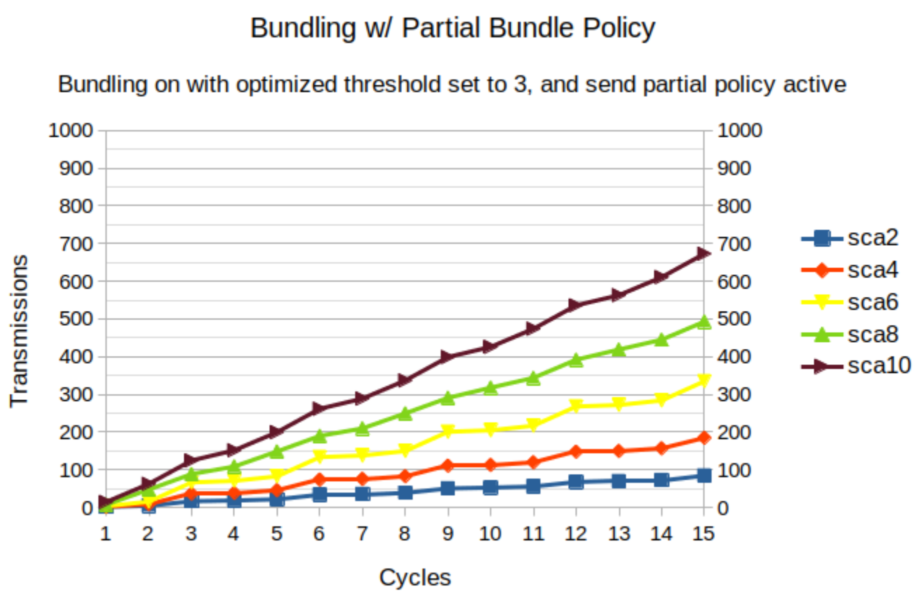

Figure 9.7: Graph showing how the total content transmissions grow when Bundling is enabled with Partial Bundle Policy.

9.3.2 Bundling Results

This section present the results from conducting the bundling experiments outlined in section 9.2.2

Transmissions with bundling with partial bundle policy, bundling without partial bundle policy, and no bundling

After running the experiments while varying between the different propagation modes – bundling with partial bundle policy, bundling without partial bundle policy, and bundling disabled – the graphs in fig 9.7, fig 9.8, and fig 9.9 were produced from adding together all the transmissions accumulated by the nodes from sending content on the given cycle indicated by the x-axis. Note that the transmissions are cumulative, so the number shown for cycle 4 for example, is equal to the amount of transmissions which occurred in that specific cycle, added on top of the amount from the previous cycle 3.

By looking at the graphs, we can see that there is a notable contrast between having bundling enabled (fig 9.7 and fig 9.8), versus not (fig 9.9). It would seem that for all the network configurations, the transmission amount was reduced by a factor of 2-3, which is best seen on the far right 15th cycle. As an

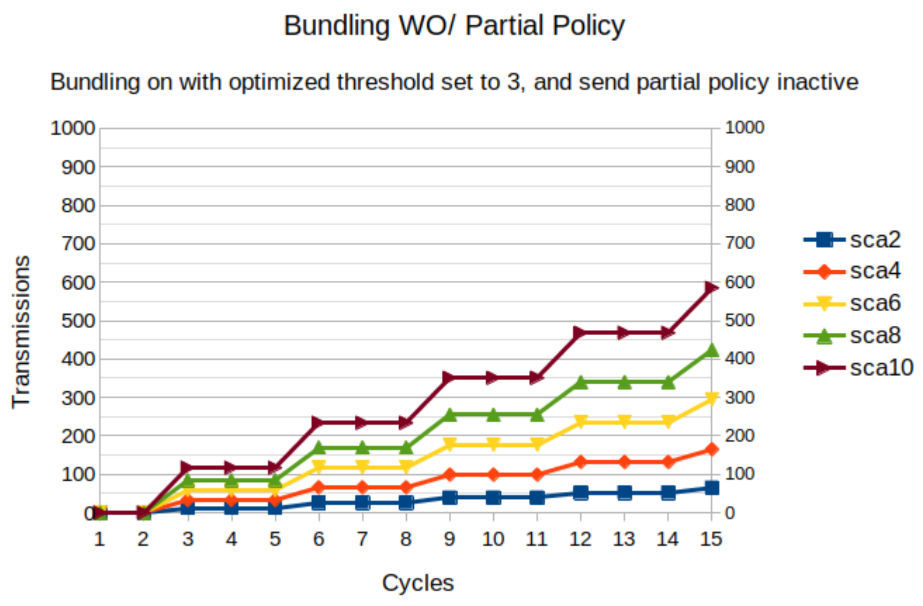

Figure 9.8: Graph showing how the total content transmissions grow when Bundling is enabled without Partial Bundle Policy.

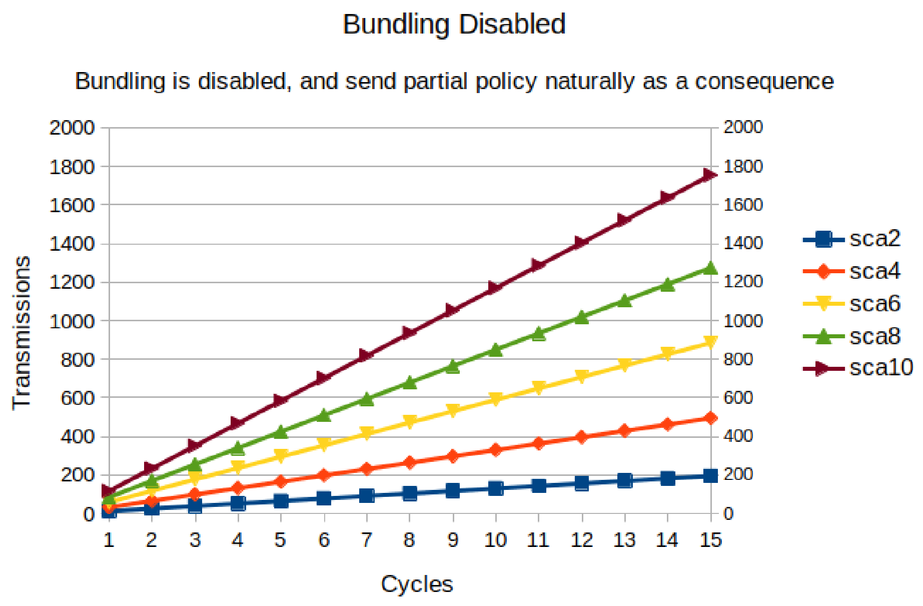

Figure 9.9: Graph showing how the total content transmissions grow when Bundling is disabled.

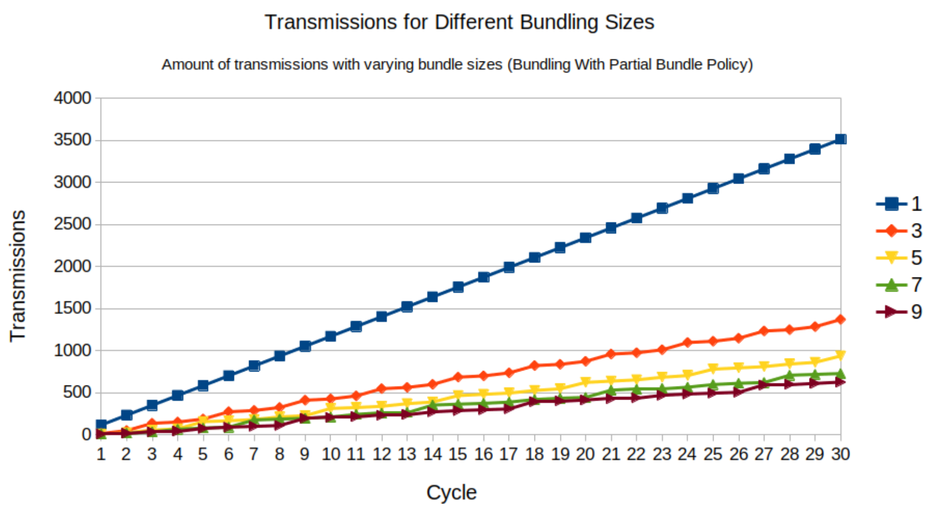

Figure 9.10: Graph showing how content transmissions grow with different bundle sizes.

example, the exact numbers for Sca10 on the 15th cycle is: 673 for 9.7, 585 for 9.8, and 1755 for fig 9.9, i.e. the number for fig 9.9 is 2.61 times higher for that in 9.7, and exactly 3 times higher for that in 9.8.

In conclusion, it would appear that bundling has a significant impact on the amount of transmissions that are performed by the nodes in the system. Bundling with the Partial Bundle Policy is slightly worse than without, but this is a feature that aims towards striking a balance between the extremes of bundling and no bundling, and looking at how the lines are much smoother in 9.7 than in 9.8 where the lines have these intermittent plateaus, shows that reports are received more regularly, while not sacrificing that many savings in transmissions.

Transmissions with different bundling sizes

After running the experiments while varying the optimal bundle size that nodes are initialized with, the graph in 9.10 was produced by adding together all the transmissions accumulated by the nodes from sending content on the given cycle indicated by the x-axis. Note that the transmissions are cumulative, so the number shown for cycle for example, is equal to the amount of transmissions which occurred in that specific cycle, added on top of the amount from the previous cycle 3.

By looking at the graphs, we can see that there is a large jump in the reduction

Node	BWP	BWOP	BDIS
8080	0.350	1	0
8081	0.333	1	0
8082	0.117	1	0
8083	0.317	1	0
8084	0.333	1	0
8085	1.000	1	0
8086	0.883	1	0
8087	0.433	1	0
8088	1.467	1	0
8089	1.000	1	0
8090	1.567	1	0
8091	1.000	1	0
8092	0.333	1	0
8093	1.000	1	0
8094	0.883	1	0
8095	1.383	1	0
8096	0.667	1	0

Table 9.1: Table showing the average delay between reports being generated and arriving at the homebase for every node in network. The unit of measurement is cycles of sleeping and being awake. Label explanation: Node - ID of a particular node, BWP - Bundling enabled with Partial Bundle Policy enabled, BWOP - Bundling enabled with Partial Bundle Policy disabled, BDIS - Bundling disabled.

of transmissions when moving from 1 to 3 in optimal bundle size. Interestingly, it would seem that the difference between each size becomes lesser while increasing it. An explanation for this might be the Partial Bundle Policy, which causes the nodes to ignore the restriction on sending content if it is not optimized. This might indicate that while less optimal bundles are sent because of the optimal bundle size being high, the number of transmissions caused by partial bundles remains the same since they are independent of this, thus leading to the numbers in 9.10, where increasing the optimal bundle size will yield diminishing benefits.

In conclusion, it would appear that increasing the optimal bundle size with Partial Bundle Policy enabled, yields a large reduction in the number of transmissions at the start, but this reduction gets increasingly smaller for higher sizes. This is likely because partial bundles are not affected by the optimal bundle size, and simply becomes a larger part of the whole, while the number of optimal bundles decreases as the optimal bundle size gets larger.

Delay in packages arriving with bundling partial policy, and without bundling

After running the experiments while varying between the different propagation modes – bundling with partial bundle policy, bundling without partial bundle policy, and bundling disabled – table 9.1 was produced by averaging the differences across the 60 cycles into a single value for each node.

What is immediately apparent is the variation in numbers for different nodes when Partial Bundle Policy is enabled, as opposed to the other two where the delay is uniform. To explain these numbers: when Partial Bundle Policy was turned off, the unaggragated series had this pattern: 2, 1, 0, 2, 1,... which indicates that reports were accumulated on the originating node until the third was generated, which optimized the bundle and allowed it to be sent. The average of a [2,1,0] series is 1, and so this was the case for all the nodes. When bundling was disabled, every report had a delay of 0, because it would always reach the homebase on the same cycle that it was generated. When Partial Bundle Policy was on variation started appearing between the nodes, most likely because the topology, Partial Bundle Policy and path selection suddenly became relevant as factors, as these can be quite varied between nodes.

In conclusion, it would appear that the Partial Bundle Policy introduces some inequality between the nodes when it comes to when their reports reach the homebase – the range was 0.117 to 1.567 in average delay – while this was not the case for the other two cases. Another interesting point is that the delay was for some nodes higher on average than if bundling without Partial Bundling Policy has been used, which is ironic considering that it was supposed to be a mechanism for increasing throughput.

10 Discussion

This section contains a discussion about central topics pertaining to the system presented in this thesis. The first sections under this paragraph will talk about some of the problems, weaknesses, and trade-offs which are present in the design.

10.1 Decentralized Architecture

The proposed solution is decentralized in the sense that no elections are performed, and no nodes are appointed for special roles in the network. The Sink role is not one for coordinating the other nodes, but rather just a gateway for data transmission.

This approach avoids the difficulties and overhead inherent in election algorithms, and lets the nodes maintain some autonomy, as they are the ones deciding where to send data based on the knowledge that has been shared in the network. This equality and autonomy is also advantageous for fault-tolerance, as losing a node only impacts the routing options of the others, and won't cripple the network in a fundamental way (such as losing the coordinator in a star-topology).

We may also ask why a coordinator, or a "dictator"-role is necessary in the first place. Reports are unique – no copies exists in the overlay network, and custody is transferred as reports are propagated, so there are no consistency issues to resolve. The data is not aggregated either, so keeping track of which data pieces have been processed, and who should be responsible for doing it, is not a relevant question.

That being said, the routing scheme does produce a lot of transmissions by itself. Whenever a change occurs (a sink loses connection, a node runs out of power), every node in the system has to made aware of this, so that they can adjust their knowledge base. The advantage for this design as opposed to, for example Stormoen's design in [20], is that paths are always shared and kept, and only need to be updated if a change does happen in the network. The nodes do not need to come to a consensus about which to choose, they simply share knowledge when needed, and then make their own decisions.

Despite this, there are other downsides which will be discussed in the next sections.

10.2 Load-Balancing

Load-balancing was a central issue for LEACH [7] and PEGASIS [11], but in the proposed solution this aspect has been less prioritized. While Low-Power Mode and the pathing metrics in concert may balance the load somewhat on their own, some more extensive guarantees may be necessary to introduce.

Incorporating battery charge as a metric into the Path Selection algorithm was a conscious decision to compensate for some of the benefits lost by not using a LEACH-esque architecture. If this metric is weighted enough it will allow nodes to eventually avoid paths to neighbors which are struggling with their power reserves, supposing that they have other routing options available.

This might not be good enough, as nodes are only aware of their Neighborhood's battery reserves and not the other intermediate nodes further down a path. An improvement that may be made here is to have the nodes keep information about the battery levels of every unique node, and then gossip this further through the network such that every node has a complete picture of other nodes' status.

10.3 Scalability

As the experiments from Chapter 9 show, the apparent exponential growth for the number of paths, and messages needed to spread paths in the network, does not bode well for the network's ability to scale with more populated networks.

However, something to note about the network topologies which were used in those experiments (9.1), is that they are quite interconnected and contain a decent number of cycles. It may be that if we had used something more resembling a tree – with no or few cycles – that we wouldn't have been able to pick up on that trend with under 25 nodes.

What it does show at least is that there is potential for this kind of growth. What amount of paths would Sca12, Sca15, or Sca30 have generated?

This result is not surprising, as every change in the network triggers a flood of messages, and every usable path is spread around to every node. So, what are our options?

10.3.1 Suggestions for Improvement

Reducing the number of paths that are necessary to store seems like a relatively simple issue to handle. Introducing some form of rule that a node can only keep a static number of paths per sink would drastically reduce the total number, at least if we assume that the number of sinks is quite low. We could also expand our filter for paths by introducing some more criteria. We can for example judge that if two paths have a certain amount of the same intermediate nodes then they are similar enough to warrant a 'one or the other, but not both' decision.

A problem with filtering paths that may be useful, would be if some important nodes check out of the network, and this causes some other nodes to be completely without paths, how are they supposed to get new ones? A possible answer to this issue, may be to have the node petition its neighbors to share their list, and if they too have none, then they again would ask their neighbors. Not keeping a complete list would remove the guarantees that a node will always be aware of the best paths given the network's structure at any given time, but it may be necessary to sacrifice this aspect of the routing scheme, if scaling to the same sizes as traditional WSNs is to be a feasible prospect.

Another measure we could employ is to limit how many neighbors a node can communicate with, despite being in proximity of each other. This would result in a smaller number of path options overall, and fewer neighbors to keep track of. If this were to be implemented, the choice of neighbors could somehow be done strategically, to for example negotiate a link based on the paths that would be formed via this connection. The nodes would select neighbors based on what they had to offer, and limit how many they chose based on some arbitrary number. A pitfall here would be that fringe nodes in range of only one neighbor are excluded and, thus fragment from the network, because they could not offer any routing options.

10.4 Node Monitoring

The current approach to monitoring of the nodes' health and activity can described as a self-reporting, and self-diagnostic mechanism. This may be enough for reporting on: their place and activity in the overlay network (transmissions performed, paths, neighbors, etc), and the integrity of the sensing instruments (assuming that the device has a self-diagnostic feature), but is it enough for detecting other, more critical failures? Complete node software crashes or Byzantine failures are examples of issues which the current system will not be able to pick up on.

Complete crashes is something that a node's neighbors may able to pick up on, as they will be attempting to send pings, but resolving the nuance between a temporary failed link and complete failure, is not so obvious if potentially unreliable connections are taken as a given.

10.4.1 Fault Inference from Analysis and Trends

Analysis of the reports and the nodes' behavior may make it possible to infer certain things. Having access to all of the reports, the homebase may be able to detect failures or abnormal behavior which is not explicitly apparent in the reports themselves. For example, the homebase may be able to infer that something is wrong by keeping track of the delay in which reports arrive to it – if the homebase has not received a report from a particular node for 10 cycles, and the average delay is around 2, then this may worthy of logging for inspection.

10.4.2 Alternative: Falcon-esque Approach

An alternative way of monitoring these nodes may to introduce a separate entity which can spy on the primary OU program, and report on its status, while remaining disentangled and independent from the OU. Something conceptually similar to Falcon [10] spies could be used here to observe the OUs inner goings-on as a separate entity, and report back for analysis.

This sort of suggestion raises some key questions:

- First, at what level would this entity operate? It could be implemented as a separate hardware module in a micro-controller architecture which could observe the other components – verify that the sleep schedule is followed, sensors are operational, and so on. It could also be implemented as a

- Second, regardless of how it is implemented, how would it communicate its observations? It would need to use the same equipment as the OU to connect with external servers, and is also under the same limitations that we have discussed in chapter 3. In order to deliver data reliably, it would essentially have to implement the same architecture as the OU's overlay network. It would also draw a significant amount of power transmitting everything.

10.5 Is Partial Bundle Policy Worth It?

In the experiments (Chapter 9) section 9.3.2, we saw that the difference in transmissions performed when using Partial Bundle Policy versus not, was negligible (for Sca10 there was a 15% difference). When we measured for the delay in when reports arrived at the homebase compared to when they were generated, we found that having the PBP enabled, reduced the average delay for some nodes, and increased it for others. For the nodes which had a reduction, some of them had quite a drastic decrease, and they were more numerous by a small margin than the ones which experienced an increase.

What can we conclude from this? It would seem that in terms of transmissions, the PBP created a noticeable, but not substantial increase. Considering that the PBP was meant to strategically allow certain nodes to send more frequently, the fact that the increase is not higher is a positive result. Comparing this to our delay experiments, it would seem that in exchange for these transmissions, what we are getting is a reduction in average delay for the majority of our nodes, and a quite substantial reduction at that for some of them.

Whether or not this is a good trade-off really comes down to how important we consider throughput to be, versus keeping a low transmission count. And if we do, by how much?

One way of resolving this would be to investigate how it would affect the system's ability to function. If this increase in transmissions would mean that the nodes run out of power before we need them to, we may have to sacrifice this increase in throughput. On the other hand, if removing the PBP means that the reports are arriving at the homebase too late to be useful, then we may have to spend those transmissions.

Another issue is the apparent inequality that arises between nodes. Is this a useful feature if it seemingly hinders some nodes from reporting?

10.6 Simulator

This section will address some issues with the simulator/implementation which are appropriate to mention.

Why Not a Out-of-the-box Solution?

At some point in the project, using a out-of-the-box network simulator, like for example *ns-3* [13], to develop the system was suggested as a way to test the logic and algorithms without having go through the work of implementing a simulator from scratch. However, it was advised that these types of simulators are quite complicated to use and takes some time and experience to get used to. More familiar tools like Golang, Python and the HTTP protocol were chosen instead.

Automating Multiple Consecutive Runs

This is done to some degree already as the python script allows for running a full network launch and sustain it over several cycles, before waiting for all the nodes to quit, and then start up a new network.

Getting multiple runs like this with a different set of parameters automatically is what is currently not present. The current implementation can't run through all of the listed experiments with one push of a button, but instead requires manual modification to the source files in order to get the required settings.

Keeping Time by Cycles, Rather Than Timestamps

The current implemenation uses timestamps and static time durations to keep track of how long the sleep and awake cycles are supposed to be. This is a bit specific to the Golang language, but the point is that it would have been better if the nodes scheduled these things around specific clock times, like "sleep from now till 12:02:30", and "remain awake until 12:03:30", instead of simply making a timestamp when it awakes, and then checks when the clocks has gone past that timestamp plus a given duration like now. By doing this we can statically align a cycle to a specific timeframe, and be sure that a node keeps to it.

10.7 Missing Validations/Experiments

This section will mention some experiments that would be appropriate to conduct, but were not due to a lack of time.

10.7.1 Sink-paths while varying the number of sinks

Measuring how the number of sink-paths grow with different numbers of active sinks, may be interesting to investigate as the number of total paths will be directly linked with the number of sinks. The question will be by precisely how much it increases with more sinks. In the author's estimation, this would most likely scale linearly with the number of sinks in the same network topology.

10.7.2 Measuring CPU and memory performance

It was attempted to measure how the nodes performed in terms of CPU performance and memory utilization with a golang library called *pprof* [6]. This would allow for measuring how certain aspects of the node source code performed and provide some analysis.

In the attempt, the CPU profiliing was difficult to understand and did not reveal anything useful about the application. During some runs, it would not show anything at all even. The memory profiling was not much better – the only useful piece of information we were able to draw from it was that procedures from golang's http library used the majority of the memory.

Due to these difficulties, and a lack of time, this endeavour was ultimately abandoned.

10.7.3 Influence of bundle size on delays

Measuring how varying the bundle size would influences delays, may be interesting to investigate as the bundle size will have a direct influence on the delay between when reports are generated, versus when they arrive at the homebase. This would be especially true if the Partial Bundle Policy was not a factor, as reports would simply accumulate on their originators until the bundle reaches its optimal size. However, how things would play out with Partial Bundle Policy enabled is more muddled, as this introduces a variable which is different for each node.

10.7.4 Validating Partial Bundle Policy

The lack of a validation of the Partial Bundle Policy is quite a serious one, given how much attention has been given to it as a feature. While doing test-runs in the development phase, it seemed to be functional when it was implemented – however, a structured approach where its correctness is investigated with different parameters and some formal criteria, should have been included in this thesis.

10.8 Improvements for Bundling and Debug Data

This section discusses some changes or features which may be added to improve the debug data generation, and bundling construction and propagation functionality.

10.8.1 Bundle Timeout

Adding a timeout to bundles is something that is already mentioned in the original bundle specification document [18]. Adding this is as a feature for our design may look something like this: attach a timestamp to every constructed bundle, or bundle fragment. This will then be used during certain instances (perhaps when nodes are about to send a bundle) to determine how long this bundle has been circulating in the network. If it exceeds a certain arbitrary limit, it will be tagged as processed, not be further propagated, and then ultimately removed, without having reached the homebase.

While this would leave holes in the homebase's record, and prevent a complete history of debug data from being put together, it might be a necessary precautionary measure to prevent old bundles from circulating perpetually due to a bug of some kind. If these bundles are allowed to accumulate it could eventually fill up the storage if whichever node is stuck with them, and be tatamount to a critical failure as the node is enable get rid of them.

10.8.2 Flush

Flushing a node's system may provide the same kind of precautionary measure against old data slowly building up and causing failures as Bundle Timeouts. Flushing caches, refreshing the memory and checking the persistent storage for old fragments which no longer serve a purpose are actions which fall under this kind of "housekeeping".

10.8.3 More Extensive Debugging

The current method of generating a summarizing report about the state of a node and its activity is a relatively simplistic one. The way that report generation is implemented, they will be relatively small in terms of size in bytes – the only elements which are not single fields of integers or short strings, are the sink-path and neighbor lists.

However, this is by no means exhaustive. A more extensive system of error, and event logging may be implemented. Notable events may be, for example, when a node deems another as 'dead' after failing to contact it, and it could then generate a log entry with some details. Core dumps, and raw data from sensing instrument self-diagnostics are also potential elements to consider here.

11 Further Work

This chapter will discuss some future paths for this project, along with some more radical changes to the design.

11.1 Implementation with Micro-controllers

In the survey on Habitat Monitoring by Oliveira et al [14], the use of micro-controllers and similar architectures are put forward as a more energy-efficient alternative to using miniaturized computers like Raspberry PIs.

By following this line of reasoning, fig 11.1 shows what a micro-architecture implementing the OUs described in thesis may look like. The efficacy of this design may be limited due to the author's lack of experience working with this kind of hardware.

Suggesting an architecture based on micro-controllers for use on the tundra may be a moot point, as Raïs et al [17] state in their paper that such architectures have, at best, limited support for more advanced network functionalities, which may be required for our purposes.

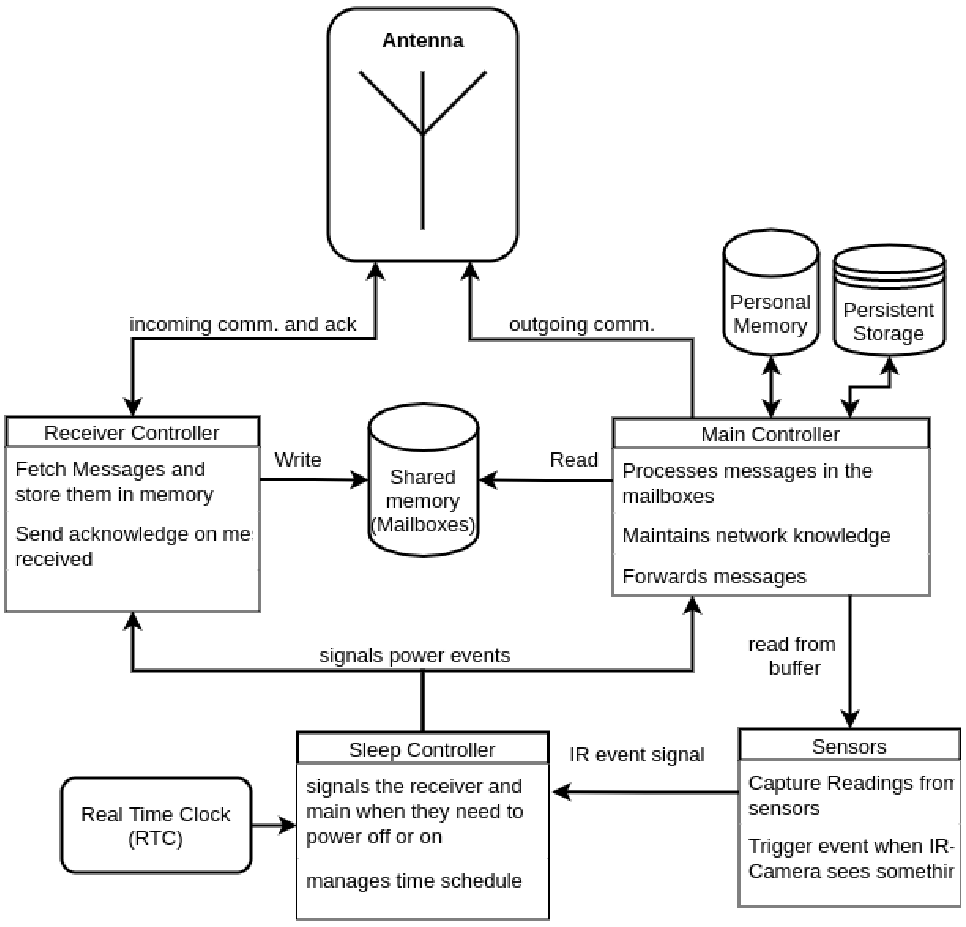

Figure 11.1: A diagram showing a possible adaptation into a micro-controller based hardware architecture

11.2 Relay Nodes

The OUs are placed according to where users believe they may get the best/-most sensor readings in the deployment environment. However, this set of placements does not necessarily coincide with the set of placements where the best antenna reception is. Relay nodes could be introduced as non-sensing OUs which are placed strategically in the environment to maximize the antenna reception, patch holes in a fragmented local cluster, or serve as optimal neighbors in some way. While they are not strictly sensing instrument, they could conform to the size, and non-obtrusion requirements that the OUs would be under.

11.3 Virtual Nodes

Some parts of a network may experience a rather high degree of traffic and activity. Most likely, these are nodes which are either sinks, or centrally placed such that they are frequently chosen for routing by other nodes. Virtual nodes may provide a way to offload some stress off these OU by building horizontally. A couple of relay nodes could be placed in the same location as a sensing node and form a sub-cluster where each member appears the same to outside nodes. They would then distribute work and responsibilities between each other using a separate scheme.

11.4 Content-Delivery Protocol Adaptation

The current debug data propagation scheme is in principle a content-delivery protocol. It could also be used for propagating sensor data itself like images, and raw data from readings, since there are no hard requirements for what the message payloads should contain or how large they should be.

11.5 Ideas from DTN: Heterogeneous Support

The earlier work in DTNs talks about providing support for interoperability between heterogeneous networks. This topic was not focused on in this thesis, but it is relevant considering that WSNs are often implemented using different protocols and architectures from the ones used in more substantial computers. Gateways are often used as hubs to mediate between these WSNs networks, and more common types like LAN and the internet. Implementing a convergence layer on the OUs would enable a future OU-network to consists of heterogeneous nodes which are doing different things, while communicate effectively with a variety of external entities.

12 Conclusion

In this thesis, we have described how a prototype for a Wireless Sensor Network was designed and implemented. Nodes, or Observational Units, in the system are designed to generate data about the surrounding physical environment, store this data persistently, generate a report describing its state and activity, and then propagate this along a path to a suitable sink, which will relay it to an external server. The nodes can dynamically discover neighboring nodes within their range, and gossip knowledge about where sinks are in the network. This enables each OU to send their data outside the local cluster, by relying on other nodes to forward their data. For this to function, it is presumed that at least one OU is a sink. While propagating data, the OUs will attempt to combine multiple pieces into a bundle, up to a certain limit, which will serve to reduce the total amount of transmissions, in a bid to conserve energy.

Results from running validation shows that the implemented prototype functions as intended, but experiments have revealed apparent weaknesses. The number of paths which are shared in gossiping shows an exponential growth when the number of nodes in a cluster grows linearly. The experiments into bundling and monitoring-data propagation shows that combining data together causes a reduction in these types of transmissions by a factor equal to that of the number of data fragments which are combined, however the Partial Bundle Policy measure to increase throughput for fringe nodes has unexpected consequences.

While the prototype system worked as planned in the design, there are several avenues for improvement which could be explored to tackle the problems and weaknesses which has been revealed. The most pertinent of which is the scalability problem, which can potentially be alleviated by imposing restrictions on how many paths a node can maintain at once, and using additional criteria in path acceptance which limits how similar paths can be. In addition to these improvements, a future system could explore how this system may be implemented as a micro-architecture, as this seems to be the most realistic choice of hardware given the constraints that are imposed upon deployed devices on the tundra.

Bibliography

[1] Vinton Cerf, Scott Burleigh, Adrian Hooke, Leigh Torgerson, Robert Durst, Keith Scott, Kevin Fall, and Howard Weiss. Delay-tolerant networking architecture. 2007.

[2] COAT. Coat camera traps image. `https://coat.no/Research/Data/COAT-Camera-traps`. Accessed: 05.05.2021.

[3] Kevin Fall and Stephen Farrell. Dtn: an architectural retrospective. *IEEE Journal on Selected Areas in Communications*, 26(5):828–836, 2008.

[4] Zack 'gizak' Guo. termui - golang terminal dashboard. `https://github.com/gizak/termui`.

[5] Golang. Golang official website. `https://golang.org/`. Accessed: 05.05.2021.

[6] Golang. Golang profiling tool. `https://golang.org/pkg/runtime/pprof/`. Accessed: 13.05.2021.

[7] 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.

[8] 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.

[9] Sigurd Karlstad. "clock synchronization between observational units in the arctic tundra". Master's thesis in computer science, UiT: Arctic University of Norway, 2021.

[10] Joshua B Leners, Hao Wu, Wei-Lun Hung, Marcos K Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy

network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 279–294, 2011.

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

[12] 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*, pages 88–97, 2002.

[13] NS3. ns-3, network simulator. `https://www.nsnam.org/`. Accessed: 05.05.2021.

[14] Luís ML Oliveira and Joel JPC Rodrigues. Wireless sensor networks: A survey on environmental monitoring. *JCM*, 6(2):143–151, 2011.

[15] Python. Python official website. `https://www.python.org/`. Accessed: 05.05.2021.

[16] 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.

[17] 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.

[18] Keith Scott and Scott Burleigh. Bundle protocol specification. 2007.

[19] Sunil Kumar Singh, Prabhat Kumar, and Jyoti Prakash Singh. A survey on successors of leach protocol. *IEEE Access*, 5:4298–4328, 2017.

[20] Camilla Stormoen. "peer observations of observation units". Master's thesis in computer science, UiT: Arctic University of Norway, 2018.

[21] Øystein Tveito. "beneath the snow – developing a wireless sensor node for remote locations in the arctic". Master's thesis in computer science, UiT: Arctic University of Norway, 2020.

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