DESIGN OF HIERARCHICAL, MOBILE, MULTI-SINK ROUTING FOR LOW-POWER AND LOSSY NETWORKS

by

Kevin Andrea

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Science

Committee:

Dr. Robert Simon, Thesis Director

Dr. Hakan Aydin, Committee Member

Dr. Sean Luke, Committee Member

Dr. Sanjeev Setia, Department Chair

Dr. Kenneth S. Ball, Dean,
Volgenau School of Engineering

Date: December 5, 2014

Fall Semester 2014
George Mason University
Fairfax, VA

Design of Hierarchical, Mobile, Multi-Sink Routing for Low-Power and Lossy Networks

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at George Mason University

By

Kevin Andrea
Bachelor of Science
George Mason University, 2012

Director: Dr. Robert Simon, Professor
Department of Computer Science

Fall Semester 2014
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my parents for supporting me in my decision to return to finish my degree and to the friends who told me I was a fool for working so many long hours for so many years on this pursuit.

Acknowledgments

Over the past many years of my studies at George Mason University, several faculty members have both aided and influenced me on my path towards this work. I would like to thank Prof. Simon, for introducing me to this topic, directing me in my research for these past two years, encouraging me to begin this work, and for serving as the chairman of my committee. I want to thank Prof. Luke and Prof. Aydin for their encouragements as I began on this pursuit, and for agreeing to join me on this journey as members of this committee. I also wish to thank the National Science Foundation for their support of this line of research. I would also like to recognize Prof. Duric and Prof. White for their guidance and their encouragements towards my pursuing a graduate degree.

I extend my deepest appreciation to my colleagues for assisting with myriad aspects of this work, particularly James Pope, Arda Gumusalan, Chris Vo, and Raven Russell. I would also like to thank KB2995388 for encouraging me to take some time to sit back and reflect on the work of the coming day; for helping me to slow down during my deadline rush.

Table of Contents

Page

List of Tables . . . viii
List of Figures . . . ix
Abstract . . . xi
1 Introduction . . . 1
1.1 Contribution to the Community . . . 4
1.2 Roadmap . . . 5
2 Background and Related Work . . . 7
2.1 The Internet of Things . . . 7
2.2 Wireless Sensor Networks . . . 8
2.3 WSN Hardware . . . 9
2.3.1 Hardware Considerations for Deployment . . . 10
2.4 IEEE 802.15.4 . . . 11
2.5 IPv6 . . . 12
2.6 Operating System (Contiki) . . . 13
2.7 IPv6 Transmission over IEEE 802.15.4 (6LowPAN) . . . 15
2.7.1 Transmission Concerns . . . 16
2.8 Network Layer Routing (RPL) . . . 17
2.8.1 Routing within RPL (DODAG) . . . 18
2.8.2 Assessment of RPL . . . 20
2.8.3 Overview of Design Goals . . . 21
2.9 Motivating Implementation . . . 23
2.10 Related Work . . . 24
3 HOIST Architecture . . . 28
3.1 Design Considerations . . . 29
3.1.1 Increasing the Size of the Deployment . . . 30
3.2 Analysis of the Common WSN Architecture . . . 32
3.3 HOIST Architecture . . . 40
3.3.1 Design . . . 40

3.3.2 Demonstration of the Architecture . . . 44
3.3.3 Network Communications . . . 47
4 Implementation . . . 51
4.1 Assessment of the Routing Protocol . . . 51
4.1.1 Problems with the Contiki 2.7 RPL Implementation . . . 53
4.2 Analysis of Contiki 2.7 Multiple Instance Routing . . . 54
4.2.1 Base Contiki 2.7 Multiple Instance Routing Analysis . . . 55
4.3 Completed Modifications to Contiki 2.7 for Multi-Instance Routing . . . 60
4.3.1 Optimizing the Environment . . . 60
4.3.2 RPL Parents Table Modifications . . . 61
4.3.3 RPL Header Modifications . . . 65
4.3.4 RPL Routing Decision Modifications . . . 70
4.3.5 RPL Instance Whitelist Modification . . . 72
4.3.6 RPL Instance Lifetime Modification . . . 73
5 Validation of Modifications and Architecture . . . 77
5.1 Initial Validation . . . 78
5.1.1 Unmanned Aerial System Evaluation . . . 78
5.1.2 Initial Assessment . . . 93
5.2 Validation Phases . . . 93
5.2.1 Phase One: Bridge and Messenger . . . 94
5.2.2 Phase Two: Collector and Bridge . . . 95
5.2.3 Phase Three: Collector, Bridge, and Observer . . . 96
5.2.4 Phase Four: Collector, Bridge, and Messenger . . . 98
5.2.5 Phase Five: Collector, Bridge, Observer, and Messenger . . . 100
5.3 Validation through Simulation . . . 101
5.3.1 Validation Setup . . . 101
5.3.2 Mobility Definitions . . . 102
5.3.3 Validation of Implementation . . . 104
5.4 Analysis of Validation . . . 111
5.4.1 Necessary Adjustments for Implementation . . . 112
6 Evaluation . . . 114
6.1 Evaluation of Architecture . . . 114
6.1.1 Evaluation 1: Twenty Collectors . . . 114
6.1.2 Evaluation 2: Fifteen Collector Comparison . . . 117
7 Conclusion . . . 120
7.1 Accomplishments . . . 120

7.2 Limitations of this Technology 121
7.3 Future Work 121
Bibliography 123

List of Tables

Table		Page
3.1	Downward Routing Table for Collector **8** in figure 3.3	39
4.1	Routing of Messages using the Multiple Instance Scenario	57
4.2	Summary of Changes for Multiple Instance Routing	60
4.3	Firmware Size Comparison of Optimizations	62
5.1	Initial Range-Check Flight Configuration Data	82
5.2	Second Range-Check Flight Configuration Data	85
5.3	Initial Phase One Implementation Test Flight Configuration Data	86
5.4	Phase One Implementation UAS Test Data	90
5.5	DIO Reception	91
5.6	Test Completion Times	92
5.7	Validation Initial DIO Messages	104
5.8	Initial DAO Messages	105
5.9	Initial Data Collection Prior to Messenger	105
5.10	Data Transmissions Following Observer Arrival	106
5.11	Data Transmissions Following Messenger Arrival	107
6.1	Twenty Collector Data Arrival Times	117

List of Figures

Figure		Page
1.1	A Representation of the HOIST Architecture	3
2.1	Zolertia Z1 WSN Mote[1]	9
2.2	TelosB WSN Mote[2]	11
2.3	Comparison between the canonical Open Systems Interconnection model (OSI) and the IEEE 802.15.4 Implementation	12
2.4	IPv6 Packet Header (40 octets)	14
2.5	ContikiMAC Operation[3]	14
2.6	Comparison between the IEEE 802.15.4 stack and the Contiki Stack	16
2.7	Sample RPL DODAG	18
2.8	WSN Packet Routing Per Send Period	21
2.9	Hierarchical DODAG	22
2.10	Vegetative Index Sample from St. Francis Winery and Vineyards[4]	24
3.1	The HOIST Architecture	29
3.2	Basic RPL Architecture	33
3.3	Basic RPL Architecture Following Movement	35
3.4	Multi-Sink Tree Topology in a Simulated RPL Network	36
3.5	Multi-Sink Star Topology in a Simulated RPL Network	37
3.6	Analysis of the Tree Topology	38
3.7	Analysis of the Star Topology	38
3.8	The Designed Architecture	41
3.9	The HOIST Architecture	45
3.10	HOIST Network after Mobility	46
3.11	Start of Messenger Data Transmission	48
3.12	End of Messenger Data Transmission	48
3.13	Start of Observer Data Transmission	49
3.14	End of Observer	50
4.1	Multitest.csc Topology	55
4.2	Full DIO Reception Showing Preferred Parents	56

4.3 Showing the Default Instances in the Scenario 58
4.4 The Fatal Flaw of a Global Parents Table 59
5.1 The Full Architecture 77
5.2 Implementation for the Unmanned Aerial System 78
5.3 Payload Bay with WSN Device 79
5.4 Control Table - WSN Devices along with the Payload Bay and Skate 80
5.5 Placement of WSN Device 7 in the Field 81
5.6 Planned Placement of WSN Device in the Field 83
5.7 Launch of the Skate UAS 84
5.8 Placement of WSN Device in the Field 85
5.9 Placement of WSN Devices in the Field 87
5.10 Loading the Devices for Test 2 88
5.11 Placement of WSN Devices in the Field for Test 2 89
5.12 Phase One Validation 94
5.13 Phase Two 95
5.14 Phase Three 97
5.15 Phase Four 98
5.16 Phase Five 100
5.17 Cooja Simulation 102
5.18 Start of Observer Data Transmission 106
5.19 Start of Messenger Data Transmission 108
5.20 End of Messenger Data Transmission 110
6.1 Twenty Device Implementation in Cooja 115
6.2 Twenty Collector RPL Logical DODAGs 116
6.3 Fifteen Collector Comparison - Traditional Architecture 118
6.4 Fifteen Collector Comparison - HOIST Architecture 119

Abstract

DESIGN OF HIERARCHICAL, MOBILE, MULTI-SINK ROUTING FOR LOW-POWER AND LOSSY NETWORKS

Kevin Andrea, M.S.

George Mason University, 2014

Thesis Director: Dr. Robert Simon

Wireless Sensor Network (WSN) devices are small, low-powered sensors that are deployable for long-periods of time to provide information on the local evironment to the world at large. These devices are designed to be deployed and operated for extended periods of time with a minimum amount of maintenance. These inexpensive devices open up a new field in computing, providing the ability for small groups to gather data on their environment for use in myriad tasks.

This work presents the Hierarchical network of Observable devices with Itinerant Sinks Transporting data (HOIST), a WSN architecture I designed to alleviate the effects of the disadvantages that lie within a WSN deployment. I began this design by dividing the traditional sender-sink model into a three-tier hierarchial model. This model separates the senders and the ultimate destination sink into tiers joined together by a series of bridging sinks. These bridging devices each control a smaller fragment of the collectors, enabling downward routing to a larger degree than otherwise possible with a traditional single DAG, while also coordinating the transmission of data between the collectors and the destination sink.

Furthermore, HOIST adds in mobility to the model, allowing the highest tier sink to move about the deployment area, minimizing any set of bridging sinks from being the sole route burdened with relaying the messages of all of the devices on the network to the sink. This architecture further defines the role of the bridges to coordinate their local collectors to send data one device at a time, enabling the expansion in network traffic to increase logarithmically with respect to the increase in the number of collectors.

HOIST also allows further scaling by using the first mobile sink, designated as the messenger, to travel between geographically segregated deployment areas to enable each cluster to send their data without having to create unnecessary lines of devices to connect the fields together. Furthermore each of the collectors is designed to locally store all of their data until the messenger next arrives, when it will receive a dump of all collected data before leaving. In this manner, a central data processing facility would be able to send the messenger to each of the remote deployment areas and receive back all of the collected data for processing.

Further extending this design, I have added the ability to use a second mobile sink, designated as an observer, to perform live collection on the local cluster it is in, allowing a responsible party to perform a spot-check of the environment. The architectural design required significant modifications to the open-source, commonly used Contiki operating system. These modifications fall within the guidelines of the routing protocol specifications and represent unfinished aspects of Contiki's implementation. Additional modfications were also made beyond the protocol specification to further allow the operating system the capabilities to properly manage general, multiple-sink routing networks.

The objective of this work is to assess, design, and implement modifications for the Contiki operating system to enable the validation and implementation of the designed HOIST architectural framework. The operating system modifications serve as a standalone component of this work to enable generalized multiple sink routing applications.

Chapter 1: Introduction

Wireless Sensor Network (WSN) devices are small, low-powered sensors that are deployable for long-periods of time to provide information on the local environment to the world at large[5]. These devices are designed to be deployed and operated for extended periods of time with a minimum amount of maintenance[6]. These inexpensive devices open up a new field in computing, providing the ability for small groups to gather data on their environment for use in myriad tasks.

These devices, by way of example, are currently being used throughout private residences to provide information to both the occupants and to other devices, enabling them to govern the temperature, manipulate the lights, and provide quality of life adjustments autonomously [7]. Beyond consumer service, these devices are also used in both industrial [8] and emergency response situations [9], among many other fields. One of the principle advantages WSN devices have, which allows such a versatile range of deployments, is that they are able to create ad-hoc networks without any prior information about the deployment area[10]. WSN devices are able to assess their surroundings and communicate with each other to build large networks autonomously[5]. Once the destination for the data is identified, typically through a special information message, the devices are able to assess a suitable route and begin sending their data[11]. WSN devices not only operate in this manner for a static networked environment, but they can adjust to changing conditions through continual reassessment and maintain the network despite positional changes or device outages[12].

Furthermore, as these devices typically operate under very low power conditions, they can survive in an environment for months, or even years at a time with little to no maintenance required on the power available in a pair of consumer batteries[1]. This resilience

is what has driven this technology to the forefront of a larger movement; the Internet of Things (IoT)[13].

This thesis works to leverage these advantages and build on them by developing an architecture to enable these devices to form this autonomous network in a more stable and efficient manner than is typically implemented. The commonly used model [14], [15], [16], [17], [18], [11] for WSN deployment involves the emplacement of devices in two principle roles: sender and sink. This is the standard multipoint-to-point routing model in which all of the devices are collectors that send their data to a single destination device, known as the sink. This flat model presents problems related to the sending of replies back down through the network from the sink to the original sender[19], in addition to a significant energy drain problem for the senders closest to the sink[20].

This work presents a Hierarchical network of Observable devices with Itinerant Sinks Transporting data (HOIST), an architecture I designed to alleviate the effects of these disadvantages that lie within a WSN deployment. This architecture breaks up the sender-sink model into a three-tier hierarchical model; which separates the senders and the ultimate destination sink into tiers joined together by a series of bridging sinks. These bridging devices each control a smaller fragment of the collectors and coordinate the transmission of data between the collectors and the highest, destination sink. Furthermore, this architecture adds in mobility to the model, allowing the highest tier sink to move about the deployment area, minimizing the prevalence of any set of bridging sinks from being the sole route burdened with relaying the messages of all of the devices on the network to the sink.

The HOIST architecture allows the increase in the number of collectors for a deployment area by subordinating them under the bridging sinks. When the number of collectors is no longer supported by the present bridges, due to the memory-constraints placed on the routing tables, a new bridging sink can be added to increase the deployment size. This architecture further defines the role of the bridges to coordinate their local collectors to send data one device at a time, enabling the growth of network traffic to increase logarithmically with respect to the increase in the number of collectors.

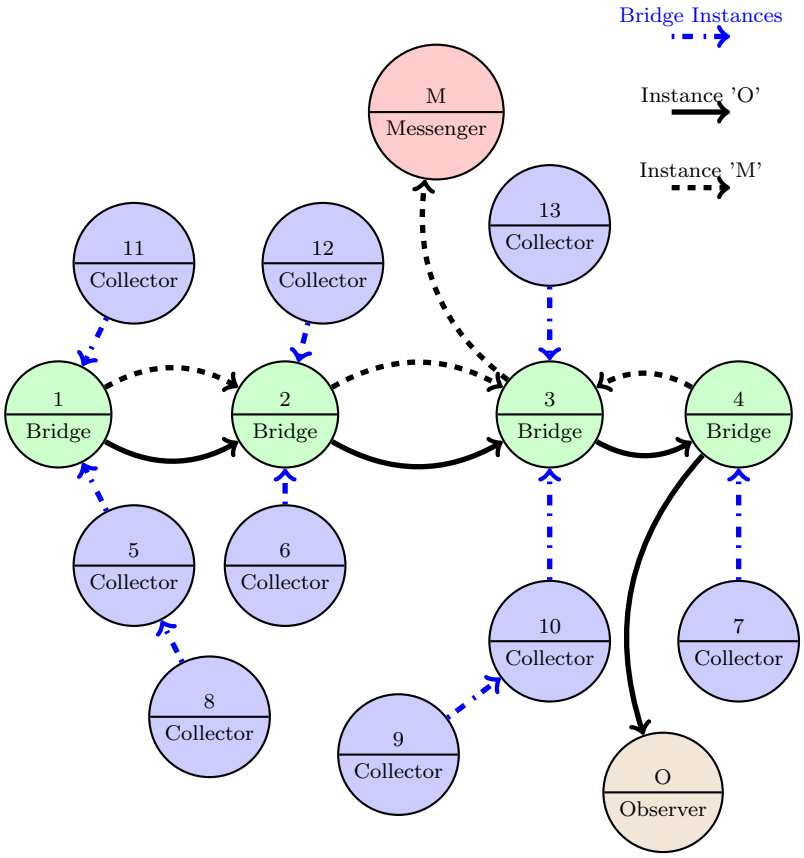


Figure 1.1: A Representation of the HOIST Architecture

This architectural design also allows further scaling by using the first mobile sink, designated as the messenger, to travel between geographically segregated deployment areas to enable each cluster to send their data without having to create unnecessary lines of devices to connect the fields together. Furthermore each of the collectors is designed to locally store all of their data until the messenger next arrives, when it will receive a dump of all collected data before leaving. In this manner, a central data processing facility would be able to send the messenger to each of the remote deployment areas and receive back all of the collected data for processing.

Further extending this design, I have added the ability to use a second mobile sink, designated as an observer, to perform live collection on the local cluster it is in, allowing a responsible party to view data as it is collected. This observer enables quick assessments of

the local area without disrupting any ongoing network actions. If each bridge is presently coordinating the transfer of data from one of its collectors to the messenger, the remaining collectors continue to collect and send their live data to the observer.

While this architecture complies with the standards for the standard WSN routing protocol, RPL[21], Contiki, the popular operating system running these devices[22], precludes its implementation at present. This limitation on the implementation of HOIST is due to areas of RPL which have not yet been fully implemented in that operating system. This work responds to this situation by assessing the present implementation of RPL in the current version of Contiki, then designing and implementing modifications to several areas of this operating system, enabling the full hierarchical, multiple-destination capabilities of this routing protocol. This portion of the work is separable from the HOIST design and may be used for general applications of WSN deployments that require multiple sink networking environments.

1.1 Contribution to the Community

The issues addressed in this work are twofold. First, the modifications to the operating system make possible the ability to logically route to multiple, mobile sinks, as specified by the RPL protocol for low-powered and lossy communications[21]. This extension to the standard multipoint-to-point routing is not only essential to the hierarchical nature of this architecture, but also enables such secondary observers and other collectors to access data. This extension adds the capability to route under more than one set of metrics that are used by an objective function to select the best routing path for each device. This is a feature that is presently missing, which adds a great benefit for emergency modes of operation.

Secondly, this work presents the HOIST architecture, which enables deployments of devices to be scaled in practice without succumbing to network failure from flooding or the drastically uneven energy consumption that the bordering devices typically face[20]. This scaling, by using the hierarchical model, allows the logarithmic increase in transmissions with respect to the expansion in the number of collectors. By way of an example, the

increase of 15 new collector devices in a deployment on a Zolertia Z1 device, using the Contiki 2.7 operating system, could only add one additional transmission per unit time to the mobile sink under this designed architecture if they all associate with a single new bridge sink.

This level of deployment affects fields outside of small, home based uses. Precision agriculture, as the motivating example for this implementation, requires large deployment areas for field coverage and, as multiple fields may be geographically segregated from each other, a means for effectively bringing this data back for processing. While a line of devices would enable routing back to a base station, stringing fields together would only succumb to the same energy consumption problems as each device now needs to retransmit multiple fields worth of data per transmission cycle. Implementing HOIST enables mobile messenger devices to acquire the data from each field and then bring them back to the data collection point in an energy efficient manner. The role of the mobile messenger device is tested on an unmanned aerial system (UAS) for suitability.

In this manner, the overall objective for this implementation is to transport data from geographically segregated deployment areas, while accomplishing two principle objectives: demonstrate the ability to increase the number of devices arbitrarily while maintaining the ability to support downward routing and reduce of the energy drain issues related to the nodes closest to the sink.

1.2 Roadmap

This thesis is divided into a series of chapters. Each chapter seeks to introduce an aspect of this architecture, assess the current environment for factors to consider ahead of a design, motivate its need under the current environment, and analyze its implementation. The second chapter begins by discussing a background into the field of wireless sensor networks and the common routing protocol used therein. The third chapter discusses the myriad considerations for the design of an architecture, as well as the design of the HOIST architecture itself. The fourth chapter continues with the implementation and modification of

the Contiki operating system to enable the implementation of the designed architecture. Chapter five then explores the efficacy of the architecture through evaluations of both practical and simulated environments. Chapter six provides a basic set of evaluations of HOIST as a viable framework of this thesis. Finally, chapter seven ends with a conclusion and the continued work which may be undertaken.

Chapter 2: Background and Related Work

This work presents the Hierarchical network of Observable devices with Itinerant Sinks Transporting data (HOIST), a WSN architecture I designed to alleviate the effects of the disadvantages that lie within a WSN deployment. The objective of this work was to design a network architecture which would enable the effective collection of data from multiple, geographically segregated fields of WSN devices.

This chapter introduces the background necessary for the design of such an architecture by looking at the broad topic of modern, wireless ubiquitous computing and introducing the concepts of how wireless sensor networking devices are fulfilling a primary role therein. This chapter describes one of those devices to provide a background on the type of hardware currently in use, the protocols the hardware runs, and the networking protocol stack that enables the data to be successfully transmitted. This chapter then frames all of these topics by introducing Contiki, the popular open-source operating system that connects all of these concepts together.

Finally, this chapter also provides a brief overview of the related work in this field and how it relates to the architecture design of this thesis, by way of a motivating example for a practical implementation.

2.1 The Internet of Things

A modern trend in computing involves the inter-connection of ubiquitous devices to achieve a connection between the physical world and the internet. This trend is encapsulated within the growing concept of the Internet of Things (IoT), in which such devices gather and communicate information to put IoT capable objects "at the services of humanity"[13]. IoT initially grew out of the Radio-Frequency Identification (RFID) community, however, the

term has come to imply the interconnection of ordinary objects to provide users assistance with myriad applications, from household appliance power savings to agricultural crop management[23]. Recently several companies, such as [24], [25], and [26] have begun marketing IoT capable systems to the consumer market. Nest Labs, for instance, offers not only smart thermostats that are able to alert appliances to enter sleep modes when users are no longer home, but a partnership with Mercedes-Benz also allows your vehicle to notify your thermostat to begin cooling your home as you begin your evening commute[27]. Companies such as Nest Labs are able to leverage modern technology to bridge the divide between traditional computer systems and household appliances.

2.2 Wireless Sensor Networks

As a subset of IoT, Wireless Sensor Network (WSN) devices provide a direct bridge between real world events and the networked community at large. As was alluded to in the previous section, in order for the thermostat to alert the lights to enter a sleep mode, it must first sense that each of the rooms in the house are empty. WSN devices provide that key functionality by using myriad integrated sensors to detect the current state of their local environments and relaying that information to an interested party. As described by [5], wireless sensor networks are "multihop [networks] consisting of spatially distributed autonomous sensors with sensing, computation, and wireless communications capabilities." As with other IoT objects, WSN devices, hereinafter referred to as motes, are designed to use very little power and operate without maintenance for extended periods of time[6]. These motes can be configured for ad-hoc networking and are able to autonomously form their own networks during their own deployment. This makes motes ideal for many situations in which sensor data is required across a geographically large area, where traditional wired systems are not feasible for deployment.

2.3 WSN Hardware

There are several models of WSN devices on the market, to include the TelosB[6] and Tmote Sky[28]; however, this work will exclusively feature one of the newest motes on the market, the Zolertia Z1 (figure 2.1). Like many other devices in the WSN family, the Z1 is a low-power device that communicates over the 2.4GHz ISM band and is designed for long-term unattended operations[1]. Among the device features listed in [1] is an integrated temperature sensor as well as a three-axis accelerometer. The radio is the IEEE 802.15.4 compliant CC2420, which is controlled by a TI MSP430 ultra-low power processor. The MSP430 is a key feature of this device as it is the same processor used in other models, making the code easier to port between the devices. The Z1 is able to run its microprocessor as high as 16MHz at full power and features 96KB of flash memory for program storage and 10KB of RAM, representing an improvement over previous WSN devices[29]. In addition

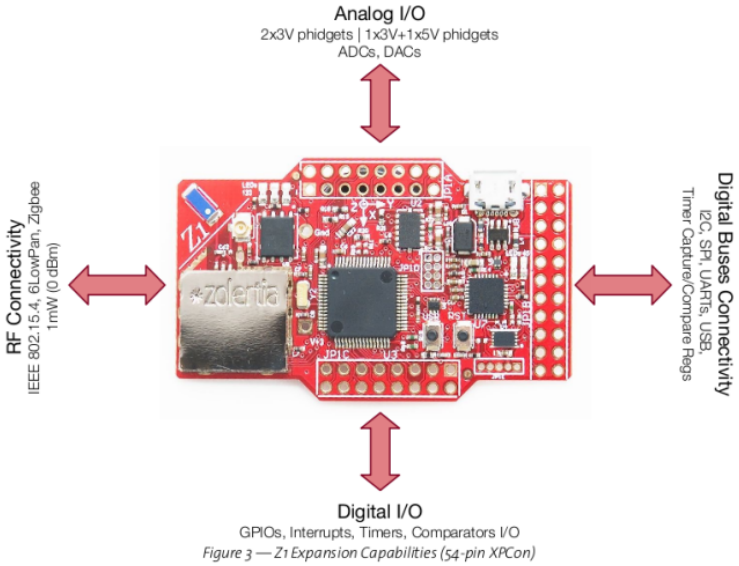


Figure 2.1: Zolertia Z1 WSN Mote[1]

to the Z1, TelosB WSN devices may be used to supplement a WSN implementation. These devices are older than the Z1 and feature half of the flash memory; however, they use

the same CC2420 radio and MSP430 processor, allowing them the ease of running nearly identical code for the same task[6].

2.3.1 Hardware Considerations for Deployment

While these devices have several strong features for use in an IoT situation, certain considerations must be taken into account with regards to their use within a network. The low-power nature of these devices means that the effective communications range is a factor. The radiated signal strength of radio frequency energy at any point in space decays on the order of the square of the distance from the transmitter, however, this ideal decay is only valid in open space. When placed very low, ground interference causes the radiated strength to decay as the fourth-power of the distance, dramatically reducing the range[30].

Additional considerations to the deployment of such devices also include other factors in the environment that may attenuate the signal, such as vegetation, buildings, and terrain occlusions. Placing these devices too close to each other also leads to the opposite problem, with receivers able to detect radiated signals from multiple transmitting devices at once, leading to local flooding, causing local interference[15].

Operarational life is also a factor in any implemented intended for long-term, unattended operations. Though counterintuitive, the CC4240 radio, like others in the class, uses approximately the same amount of power transmitting data as it does receiving data[1]. This goes against the intuition that a radio can stay in reception mode until ready to transmit, minimizing lost packets. To conserve battery life, these devices often use a radio duty cycling protocol, which only powers a radio on very briefly to check for any signals, before returning to a low-powered state[30]. While that is the primary means of conserving power, placing the motes in smaller clusters, geographically closer to each other and their effective root enables each node to reduce transmission power. The datasheet on the CC2420 radio, for instance, shows that cutting the power level by 5 dBm leads to a reduction in current consumed by 3.5 mA (21%)[31]. While such energy conserving techniques are not directly within the scope of this work, these protocols are used in this implementation.

The issue with energy on small and low-powered devices is critical for any implementation that seeks to deploy devices over long periods of time – up to and including years – with minimal maintenance required. The Zolertia Z1 device, for reference, in its low-power standby mode, uses 23.2 μA of power[1]. Using two standard 2,000 mAh rated AA batteries, this device would remain operational for approximately 10 years in standby mode. In full active mode, while either transmitting or receiving, the Z1 draws 18.8mA, which would run for approximately 70 hours on the same battery source.

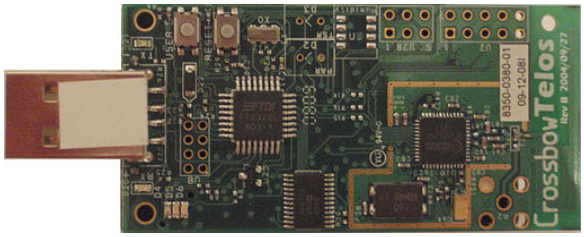
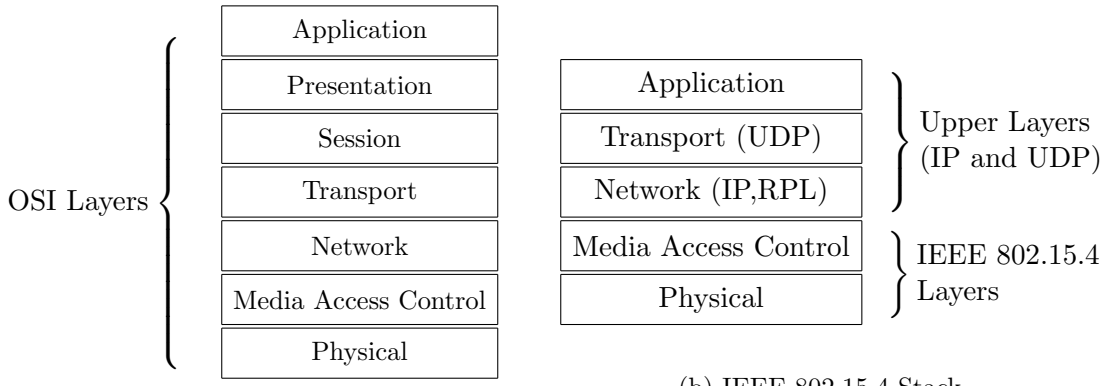


Figure 2.2: TelosB WSN Mote[2]

2.4 IEEE 802.15.4

WSN hardware, as described in the previous section, are very small and the computing power therein is limited and memory-constrained. A common standard for enabling such computationally constrained devices to communicate efficiently is IEEE 802.15.4, which is the IEEE Standard covering Low-Rate Wireless Personal Area Networks (PAN) [32]. This standard only defines the lowest two layers of the network stack, leaving the implementation of the upper layers to software stacks, such as ZigBee, XBee, or Contiki's uIP. Figure 2.6a shows the standard, 7-layer conceptual network stack model as compared with the implementation provided by IEEE 802.15.4. Completed with a the upper layers, IEEE 802.15.4 is currently being used as a communication standard for IoT applications [33].

The advantage this standard has over traditional wireless implementations is that the Physical (PHY) and Media Access Control (MAC) layers of the standard OSI-Model are



(a) OSI Stack

(b) IEEE 802.15.4 Stack

Figure 2.3: Comparison between the canonical Open Systems Interconnection model (OSI) and the IEEE 802.15.4 Implementation

defined by IEEE 802.15.4 to enable low-powered devices to conserve power by spending most of their time in sleep-mode. When a device wishes to transmit, it must first power-on its receiver to ascertain if there is any Electromagnetic Interference (EMI) detected on the assigned channel. In the presence of EMI, the device will wait in accordance with the PHY implementation of Carrier Sense Multiple Access (CSMA) with Collision Avoidance (CA) [32]. Once the channel is clear, it will broadcast the data and then resume a sleep-state. [32] further describes this as a "robust wireless technology that could run for years on standard primary batteries."

2.5 IPv6

The stack defined by IEEE 802.15.4 ends at the MAC layer, leaving the upper protocol layers up to the implementation to define. The first of these upper layers is the network layer. In this model, the network layer uses the Internet Protocol (IP), which enables the various machines on the Internet to inter-communicate. The first IP public addressing scheme to emerge from DARPA in 1984 was Internet Protocol version 4 (IPv4). For the time, this provided a sufficient address space for up to 2^{32} different systems to be uniquely

addressable. In a recent feature article, networking corporation Cisco Systems published the figure of 8.7 billion objects that are currently connected around the world [34]. The means of allowing more connected objects than available global addresses is covered through private or link-local addressing, in which small Local Area Networks (LANs) are able to locally address objects without direct global addressing capabilities. With [34] estimating that there may be 1.8 trillion connected things by 2020, the need for an enhanced addressing space is present.

This was one of the chief catalysts behind the creation of the Internet Protocol version 6 (IPv6) in 1998. IPv6 increases the addressing size of devices from 32 bits to 128 bits [35]. In essence, this increased the limit on directly addressable devices from approximately 4 billion, to 2^{128}, which is a value that approaches the approximate number of atoms in the Earth, as estimated by the Department of Energy, Jefferson Laboratory[36]. As this far exceeds the estimate by Cisco, this provides the ideal protocol for working with the growing IoT category of devices, allowing each to have their own globally accessible address. This protocol also supports extensions and options to the base packet, readily enabling custom protocol routing for WSN devices. One principle disadvantage of using this protocol is the increase in the size of the header for the packet frames, though this is addressed in section 2.7.

2.6 Operating System (Contiki)

The model that synthesizes together the lower and upper stack layers is implemented by an operating system. I have chosen the Contiki Operating System, which an open-source system that both completes the networking implementation in addition to providing application support for the motes[37]. Contiki, currently on version 2.7, provides this common implementation in addition to two other key networking features for WSN operations: a radio duty cycling protocol and 6LowPAN.

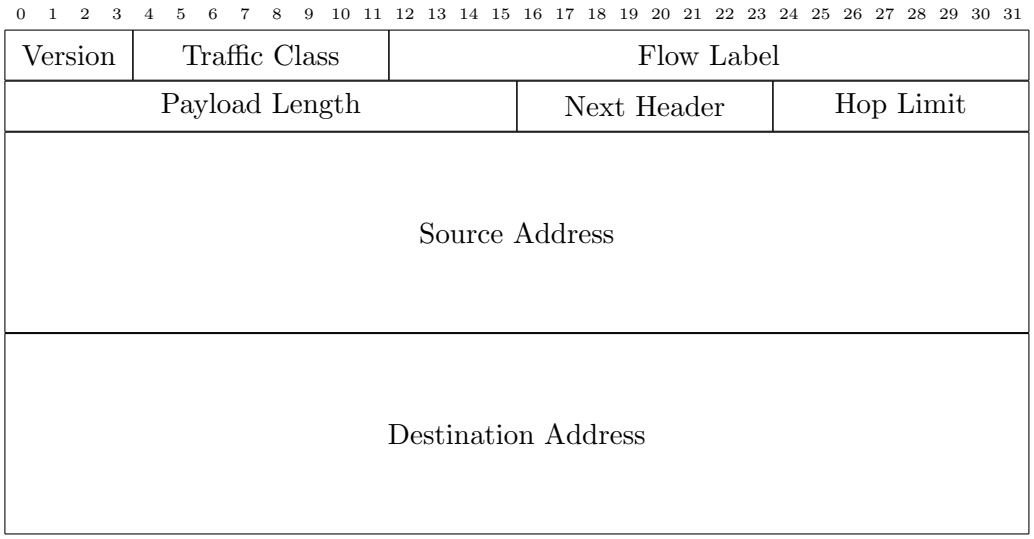


Figure 2.4: IPv6 Packet Header (40 octets)

Enhancing the IEEE 802.15.4 stack, Contiki implements a Radio Duty Cycling Protocol (RDC) in the form of ContikiMAC. ContikiMAC provides responsive communications service wherein roughly 99% of the time the radio is powered off [3]. ContikiMAC achieves this first by periodically conducting a pair of Clear Channel Assessment (CCA) probes of the current channel.

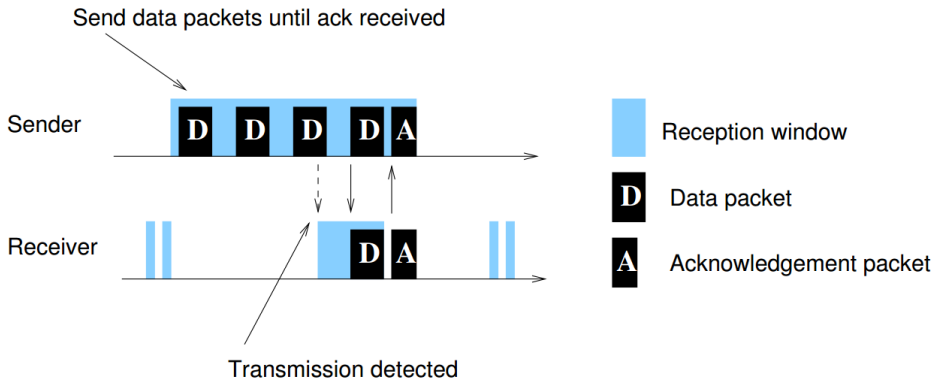


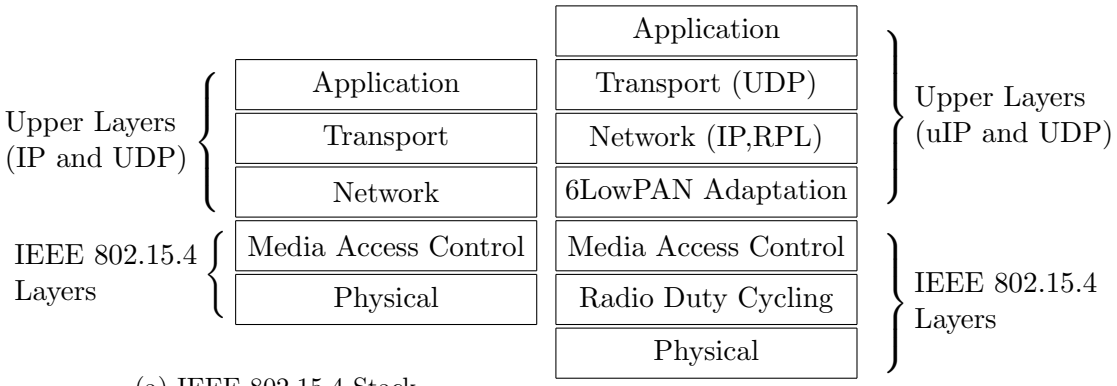
Figure 2.5: ContikiMAC Operation[3]

The two probes, shown in figure 2.5 as narrow, parallel vertical bars on the Receiver timeline, are timed such that they occur on a slightly longer delay than maximum time between a rebroadcast event of another mote to ensure transmission events are not missed. If neither of the CCA probes result in any detected signals, the mote will resume a powered off state. If non-relevant EMI is detected, the mote will receive just enough to discern that a proper message is not present and will likewise return to a powered down state. Only when a proper message is detected will the device begin receiving the first data packet and respond with an acknowledgment message. The sender will continually send only the first packet of data until the receiver responds back with an acknowledgment.

2.7 IPv6 Transmission over IEEE 802.15.4 (6LowPAN)

As IEEE 802.15.4 leaves the details of transmissions up to the application, Contiki still required a means for transmitting the IPv6 packets over the motes. For this reason, the Internet Protocol stack was chosen to fulfill this implementation. RFC 6282 (6LowPAN), which is the standard for the transmission of IPv6 packets over IEEE 802.15.4 networks, addresses this [38]. This RFC primarily addresses the reduction in size of a standard IPv6 packet (Fig. 2.4) to fit within the physical limitations presented by the motes. The standard IPv6 header is 40 octets in length, however, this only governs IP routing. UDP adds another 8 octets for port addressing. With the IEEE 802.15.4 Maximum Transmission Unit (MTU) of 127 octets and its own header size of 25 octets without security, this only leaves 54 octets for the payload [39]. Adding in AES-128 security, this drops by another 21 octets to 33 data octets. This also does not include any of the routing protocol extension headers, which, though relatively small, only serves to further demonstrate the need for a compression scheme.

6LowPAN, at its core, handles this form of header compression. In the best case, 6LowPAN can compress the IPv6 header down to only 2 octets, however, for IP routing, it still requires 7 octets of space [38]. 6LowPAN accomplishes this by assuming certain



(a) IEEE 802.15.4 Stack

(b) Contiki Stack

Figure 2.6: Comparison between the IEEE 802.15.4 stack and the Contiki Stack

field values that are essentially fixed within most IPv6 packets and removing them from the header. This protocol also removes the IP addressing and replaces it with a link-local addressing scheme using a link-local prefix. This enables direct addressing of the WSN devices while also allowing them the ability to transmit significantly larger data payloads without fragmenting the packets. This is precisely the implementation that will enable aggregated data to be transmitted efficiently to a mobile receiver.

2.7.1 Transmission Concerns

The transmission of the data across the medium between devices is a driving concern when planning deployments. The base problem is that if there are two devices communicating with each other on the same frequency, there exists the potential for interference.

Wireless communications on these devices are half-duplex in nature. This means that the wireless radio can not receive and transmit simultaneously. When a radio is transmitting, it does so at a base power level that is greater than the current strength of the signal it wishes to receive, thereby preventing reception. Due to this occurrence, if two devices communicate with each other and the transmission of the first device arrives immediately after the second device begins transmitting, then the first message will be lost in the noise.

The technique for dealing with the problems of a wireless medium involves listening

for any sign of incoming transmissions before attempting to send. On any sensed activity, the transmitter will delay its own transmission before trying again. This alleviates the problem by shifting the outgoing signals temporally. The Contiki operating system handles this through the Carrier Sense-Multiple Access (CSMA) MAC layer. Contiki implements CSMA as a default to complement the previously mentioned ContikiMAC[3] (figure 2.5.

For a large, dense mesh network in which multiple devices are continually within each other's range, these interferences can occur frequently. Increasing the deployment size of this scenario by adding more devices will only serve to exacerbate this problem. Even devices beyond the expected transmission range of each other will nevertheless still pose the danger of interfering with communications.

2.8 Network Layer Routing (RPL)

With Contiki and IEEE 802.15.4 and 6LowPAN, we have addressable WSN devices operating over IPv6; however, there remains a lack of a selected protocol for routing messages between the devices. I have selected to use RFC 6550 (RPL), which is the IPv6 Routing Protocol for Low-Powered and Lossy Networks [12]. With RPL, each device that exists within this logical structure, hereinafter referred to as nodes, participates in the creation of an ad-hoc network. The primary function of RPL, is to facilitate multipoint-to-point communication to transfer the collected data from the sender nodes to a single root node, also known as a sink [19]. RPL features instances, each of which may contain a single routed network and contains one set of routing metrics and constraints for routing [11]. The formed network within each instance creates a Destination Oriented Directed Acyclic Graph (DODAG), in which each node selects a preferred parent towards the root.

This process begins when a specially designated sink starts transmitting a DODAG Information Object (DIO) message over the IPv6 Link-Local Multicast address to all nodes within range for the current RPL instance. When a node receives this message, it first checks to see if it is already a member of the RPL instance being announced. If not, the

node will immediately join the instance, select a preferred parent towards the root, increase the advertised current rank, and rebroadcast the DIO to enable more distant nodes the ability to join the instance. If the receiving node was already in the instance, it checks to see if the rank of the received message is lower than its own and adjusts its preferred parent accordingly. Each instance has its own Objective Function (OF), which, among other things, specifies a metric to determine which of the node's neighbors to select as the preferred parent.

2.8.1 Routing within RPL (DODAG)

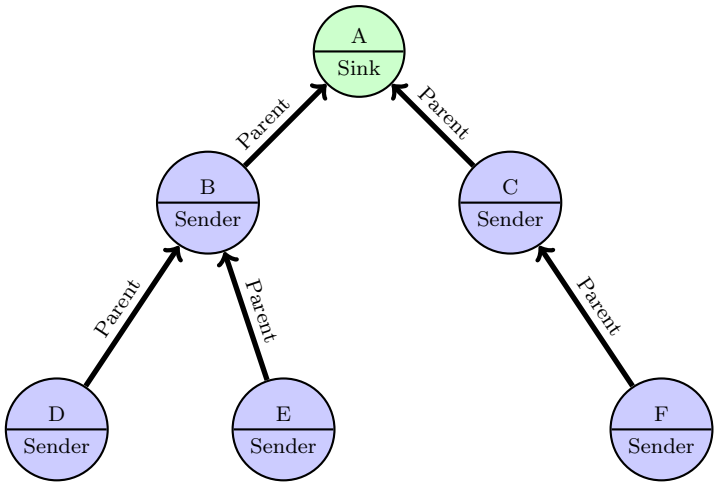


Figure 2.7: Sample RPL DODAG

This list of neighbors forms as the result of a second message type that is sent between the nodes. This message is a DODAG Information Solicitation (DIS) and serves as a means to receive information about the neighbors in communications range [11]. These, like the DIO, are sent out periodically to allow each node to detect any changes in the network topology or in the quality of the link between the nodes. Link quality is determined through an assessment of the conditions in the network. This is primarily achieved through one of two different means. The first of these means is the Estimated Number of Transmissions

(ETX) necessary to deliver a successful message to an adjacent node. A second metric that is also commonly used is the Received Signal Strength Indication (RSSI). This relies on the radio to measure the strength of the received signal, which is difficult to do without more sensitive equipment, making this moderately unreliable[40]. For this reason, the ETX link quality assessment metric has been assessed by [41] as the best of the readily available metrics.

With a formed DODAG, we have the ability to quickly send messages from all of the nodes to the sink, however, there are not yet any means for the sink to send any messages to the nodes, either individually or en masse. RPL handles this by giving each node the ability to create and maintain a table of all of its children. This is accomplished with a Destination Advertisement Object (DAO) message, which is sent by the nodes up to the root. The DAO contains the IPv6 address of the device originating the message as well as the address of the device currently transmitting it. In RPL, the feature which governs this is called storing-mode. When storing-mode is enabled, each device receiving a DAO will add two entries to its downward routing table: the originator's IPv6 address and the currently transmitting device's address[19]. In this manner, each node knows not only which devices are its children, but, more importantly, which devices to send messages through in order to reach them.

By way of an example using figure 2.7, if sender **D** originates a DAO, it will be sent up to sender **B**. **B**, if storing-mode is enabled, will add **D** to its table with a next-hop of **D**, before retransmitting the DAO to its own parent, **A**. When sink **A** receives this DAO, it too will add **D** to its downward routing table, but the next-hop will be **B**. In this manner, if any device now wishes to send a message to **E**, it is able to send its message up to the first common ancestor of **E**, which will have the information on how to route back downward to ensure delivery.

Storing-mode in RPL does come with a memory overhead and Contiki places limits on this functionality to attempt to curb the overhead. In Contiki, only 12 neighbors are ever stored by a node and it was recommended by [19] not to exceed clusters of 30 nodes for that

reason. Even with the proper precautions taken, support for point-to-multipoint is limited and point-to-point transmissions are described by [19] as "esoteric".

2.8.2 Assessment of RPL

The problem with the present implementation of RPL in Contiki is that only one DODAG is permitted per instance [11] and only one instance is routable per node. For this reason, I have designed and implemented a modification to the RPL implementation in Contiki to allow multiple instances to be routable by each node; these modifications are detailed in chapter 3. This enables each node to work as a part of multiple instances, each with its own DODAG and routing metrics. By employing these in a hierarchical manner [42], [43], a cluster of sensors can each maintain a static network to the nearest sink, which, in turn, can form a DODAG and send the blocks of collected data to a mobile messenger sink when present. Moreover, this also allows a direct observer to enter the cluster, form its own DODAG amongst the sensors, and receive real-time data. RPL, with just a few modifications, performs these tasks well.

One of the biggest failings with RPL is that is has no explicit design support for mobile nodes [19]. Mobility is handled through as a side-effect of the parent recalculation process. When a regularly scheduled timer recalculation timer goes off, all of the neighboring devices are assessed for the quality of their link and their rank to the destination. In an environment with a mobile sink, as it moves around, the nodes that are now closest will receive an updated DIO and change their ranks to reflect their new proximity to the sink. After updating their internal DODAG table, the devices will rebroadcast a DIO reflecting their new rank. When a node runs its rank recalculation routine, it will compare the ranks and link quality information of the neighbors it has received a DIO from and ultimately change its preferred parent to reflect the new position of the mobile device. The down side of this process is that each device may take a while to update its parent, due to a combination of hysteresis and stale DIO messages showing ranks from an outdated position.

Another serious problem is that the process RPL uses to construct the DODAG causes

the devices nearest to the sink to suffer from "high energy drainage"[20]. According to [19], [42], and [20], this occurs because the nodes nearest to the sink act as a router for all of their children's messages, in addition to their own, as shown in 2.8. By having to route a disproportionate number of packets, that node is spending more of its time with the radio powered on and in use than its children, causing it to drain its power faster.

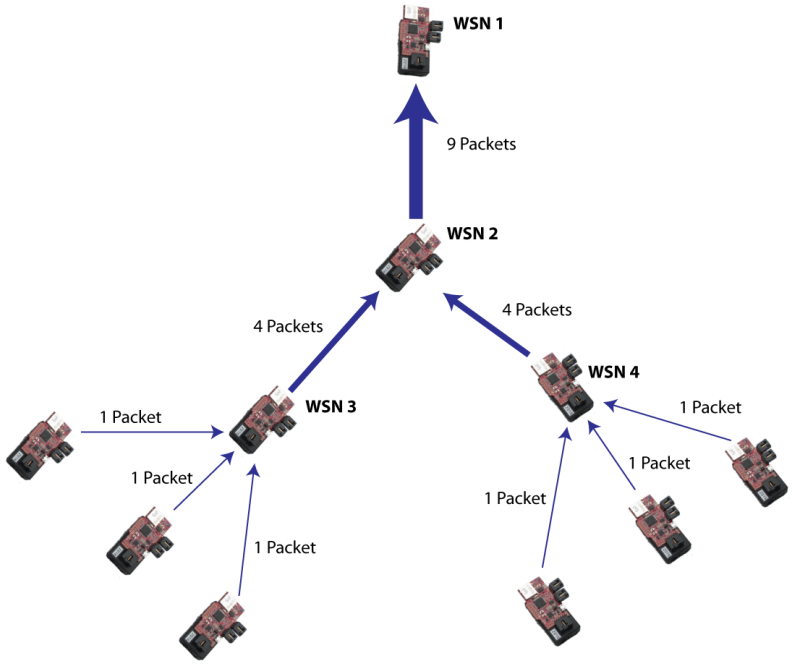


Figure 2.8: WSN Packet Routing Per Send Period

2.8.3 Overview of Design Goals

The HOIST architecture I designed and present in the coming chapters of this thesis began with the goal of creating an implementation which will allow for multiple, geographically segregated networks to send data back to a data processing center. Such a network must be able to function for long periods of time without maintenance and be able to cover multiple, large deployment areas without succumbing to problems of local flooding or uneven, high energy drainage.

One solution to this problem that has been proposed widely by [19], [44], [42], [15], and [18] is to have the sink move around the cluster to prevent any one set of nodes from succumbing to this energy drain situation. By distributing the load around the network, it will lead to a longer lasting network over time. The problem remains, however, that while RPL can adjust to the environment and change over time as a sink moves around, there is a considerable delay in the route changes, leading to disconnected networks, particularly when using ETX as the OF metric [41]. One proposed solution to this problem in RPL was also investigated by [41], who found that using the number of hops from the sink to the node, the Hop Count, as the metric yielded a 36% higher throughput than with ETX, owing to the faster reform time. This metric does poorly, however, with static networks as it does not take into account link quality information.

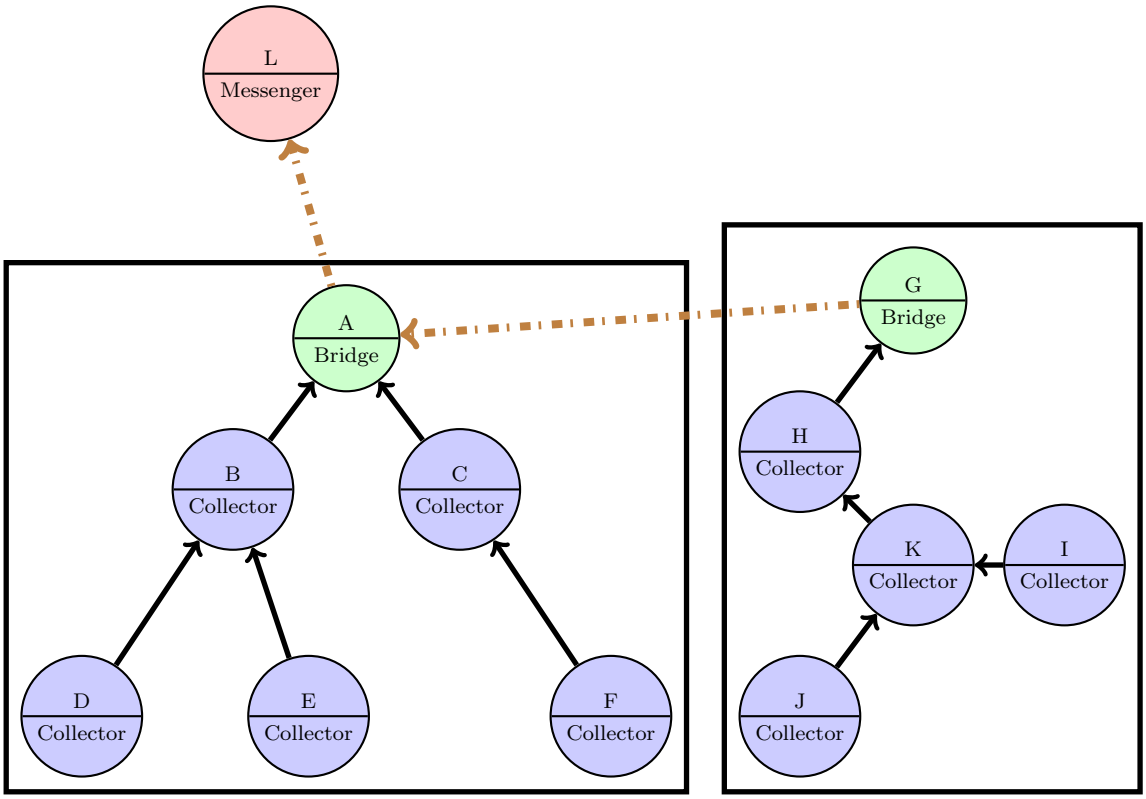


Figure 2.9: Hierarchical DODAG

For this implementation, I have elected to use the best of each approach by separating the static portion of the network from the mobile portion of the network, using a three-tier hierarchical model. In this design, I use ETX as the routing metric for the lowest tier of static collectors, and Hop Count for the bridge sinks dealing with the mobile data messenger, as shown in figure 2.9. In this manner, the static DODAG will persist over time and form the best routing based on link quality, without any disruptions by the mobile collector; the links to that collector will only affect the sinks, which will be able to track it faster by favoring direct connections.

2.9 Motivating Implementation

The above analysis of each of the components of WSN devices and its communications shows both strengths and failings, however, only by placing this within the context of a practical implementation can a real assessment be made on the feasibility of the technology. As a motivating example to illuminate the various aspects of this technology, and a means through which I can frame my own architectural design, I have chosen to utilize field of precision agriculture.

Precision Agriculture (PA) is a new, yet growing field within the farming community. As described by [45], PA involves five critical steps: "data collection, diagnostic, data analysis, precision field operation, and evaluation." The goal of PA is to allow farmers to adjust their land and crop management techniques to meet the immediate needs of particular areas of their land. In order for PA to work, there naturally needs to be some variability to the land and a means to allow such adjustments to affect the growing conditions. Until recently, PA has typically involved commissioned satellite flights to provide images of the farmland; such images are critical for presenting information about growth using the Normalized Difference Vegetative Index (NDVI) to measure crop health [46]. As such commissions can take up to two weeks to get the data, a lot of farmers are turning to Unmanned Aerial Systems (UAS) to provide a cheaper and near-realtime picture of their land [47].

With this rise in PA, farmers are also beginning to explore other options for monitoring

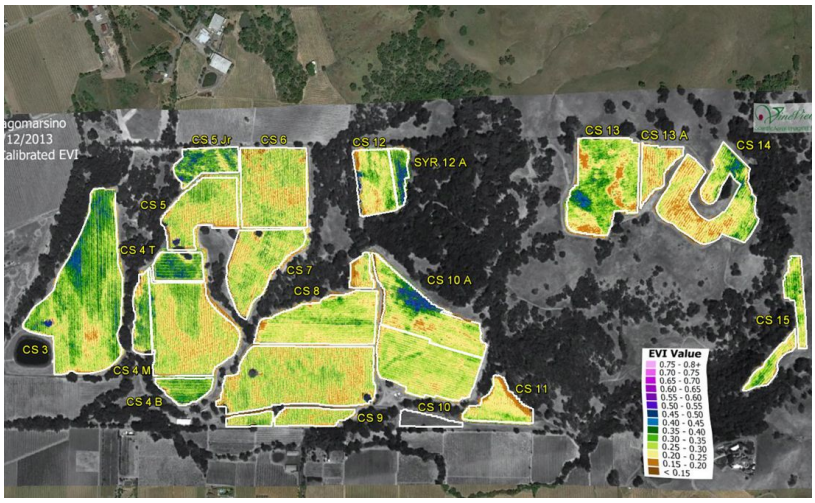


Figure 2.10: Vegetative Index Sample from St. Francis Winery and Vineyards[4]

their crops by using pH sensors, anemometers, thermometers, and humidity sensors, which they deploy throughout their fields [45]. These fields, as [45] describes and as shown in Fig. 2.10, are often hundreds of meters in size and each parcel of land may be separated from its neighbors by kilometers; wired technology is simply no an option.

By embracing modern networking and sensing technology, farmers are presently providing an ideal infrastructure for deployment and the practical use of WSN devices. Moreover, this practical implementation scenario involves each of the aforementioned concepts and stresses the advantages of employing a new architecture within these frameworks. This architecture is described in detail in the coming chapters of this work.

2.10 Related Work

There are several key aspects of this work, each of which has its own representative body of literature. The core of this work is the use of wireless sensor networks in a field environment in which direct communications with the network is not feasible.

The first aspect of my design employs multiple sinks to split the load of the network. In [14], the researchers employed a combination of infrared cameras and WSNs in order to more

rapidly alert first responders to fires in regions containing cultural heritage sites. In their work, they utilize multiple sinks for the purpose of redundancy as "data collection centres may also be affected by the fire [14]." Their technique differs from this work in that their sensor traffic is simultaneously routed to multiple sinks, with each sink being co-located at a data collection center. In my architecture, centrally located sinks split the network of sensors, leading to shorter routes and a tendency towards the star topology, further reducing power and hop counts of the sensors, as is described in the following section. Their model focuses on having multiple collection centers in which the sinks are located, however, in this work the assumption is that the deployment area is not such that is feasible to connect directly with.

With respect to the real-time observer, [43] describes a fire detection system scenario in which firefighting units carrying mobile sinks are able to directly connect with the network. This connection allows the first responders information on the location of the fire to better direct their efforts. This is a one-tier model in which an existing network of static nodes sends their data to one or more sinks. As the fire fighter enters the area, their sink joins the network and begins receiving a copy of all of the sensor data. In this work, I use a similar technique for each of the clustered fields, allowing direct, real-time observation of the local environment. My work implements this along side the mobile collection and ferrying of aggregated field data. Furthermore, the Contiki operating system does not support multiple sinks; a feature that my work proposes a design and implementation for.

Concerning the multi-tier approach and mobile ferrying, the research conducted by [44] proposes a three-tier model for data collection, aggregation, and distribution via a mobile sink. The Mobile Ubiquitous LAN Extension (MULE) approach is a design for sparse sensor networks that enables data to be locally collected and aggregated before being transmitted to wandering mobile sink devices [44]. Once collected, these MULEs transport their sinks to within range of an access point, where the data can be downloaded and connected to the larger Internet. The authors elected this approach for the same reason that [45] cites, namely the cost in motes necessary to bridge the gap would not only be prohibitive, but each

of those bridging motes leading to the data collection center would all suffer greatly from the energy drain effect as they are relaying entire deployments of traffic. My approach and implementation seeks the same benefit of the MULE implementation, however, my approach maintains a static network between two tiers of static motes, holding a link quality derived network to wait for the mobile sink to arrive. Once the mobile sink arrives, the lower tier of static motes is able to send through pre-existing routes to a second-tier static sink, which then is able to send data across its dynamic network to the mobile system. This achieves the data collection of the MULE approach, while reducing the need for dynamic routing on the first tier.

Other papers, such as [20] and [5] discuss the advantages of using mobile sinks for increased throughput and lower power requirements. These approaches show the benefit to a mobile sink, however, in their research, such mobility is to reduce the prevalence of the excessive energy drain problem and not for any form of data ferrying. My work addresses the reduction of the energy drain problem through centrally located static bridging sinks, which tend towards star topological formations, and using mobile sinks traveling in a predictable manner to perform periodic data collection from a smaller tier of static sinks brings with it the same advantages as discussed in the referenced works.

One recent publication, [42], presents a very similar approach using a three layer system in an approach called Load Balanced Clustering and Dual Data Uploading (LBC-DDU). Their first layer consists of the base collectors, deployed throughout an area. A clustering technique is then used to elect cluster heads from among the collectors, with these devices distributed throughout the deployment area. These cluster heads coordinate their local cluster members; each cluster member must be able to communicate with a cluster head using exactly one hop. Once the cluster heads are elected, they coordinate data transfers with the top layer, which consists of a mobile sink called a SenCar. This SenCar uses a planning technique to determine a route such that it is able to collect data from each cluster head using exactly one hop. This network features many of the same approaches I have

chosen to use, such as a bias towards a star topology for first and second layer communications to reduce hop counts. The notable difference between this work and HOIST is that the architecture presented in this thesis is immediately implementable using standard hardware and the common RPL networking protocol. This enables the deployment of a HOIST system to use low computationally capable devices, which do not have to run elections or coordinate clustering for one-hop transmissions. Furthermore, HOIST is designed to work using a simple mobile messenger sink that only has to be within communications range of one bridging sink. Implementations, such as those present with precision agriculture, may prevent mobile devices from maneuvering throughout a deployment area, as is needed by the LBC-DDU framework proposed by [42].

Chapter 3: HOIST Architecture

After examining the technology available for this implementation, I designed an architecture to transport data from geographically segregated deployment areas, while accomplishing two principle objectives: the ability to increase the number of devices arbitrarily while maintaining the ability to support downward routing and to reduce of the energy drain issues related to the nodes closest to the sink. I have designated this architecture as the Hierarchical network of Observable devices with Itinerant Sinks Transporting data (HOIST). The goal of this is to be able to send mobile sinks to remote deployment fields to retrieve up and transport back all of the collected data. Additionally, at any point, the devices in the fields may be directly observed, without interrupting either data collection or data retrieval by the mobile messenger.

The HOIST architecture creates a network able to cover multiple, large deployment areas and provides a means for transferring this data back to a data processing center. In addition to meeting these goals, I have also added the secondary task of enabling live data collection by an observer device, which may operate alongside the primary data couring, messenger device. I employ a three-tiered hierarchical design, featuring four distinct device roles, as shown in figure 3.1.

The collector device communicates over a standard RPL network to a bridging sink. These bridging sinks coordinate all communications between the collectors and mobile, destination sinks. These mobile sinks consist of an observer device, which is sent live data from each collector as they gather it, and a messenger device that is sent all of the data collected from each collector since the last data dump event. The bridges coordinate all of this traffic such that only one of their children are transferring data at a time, enabling the amount of concurrent transmissions to scale logarithmically with respect to the increase in the number of collectors.

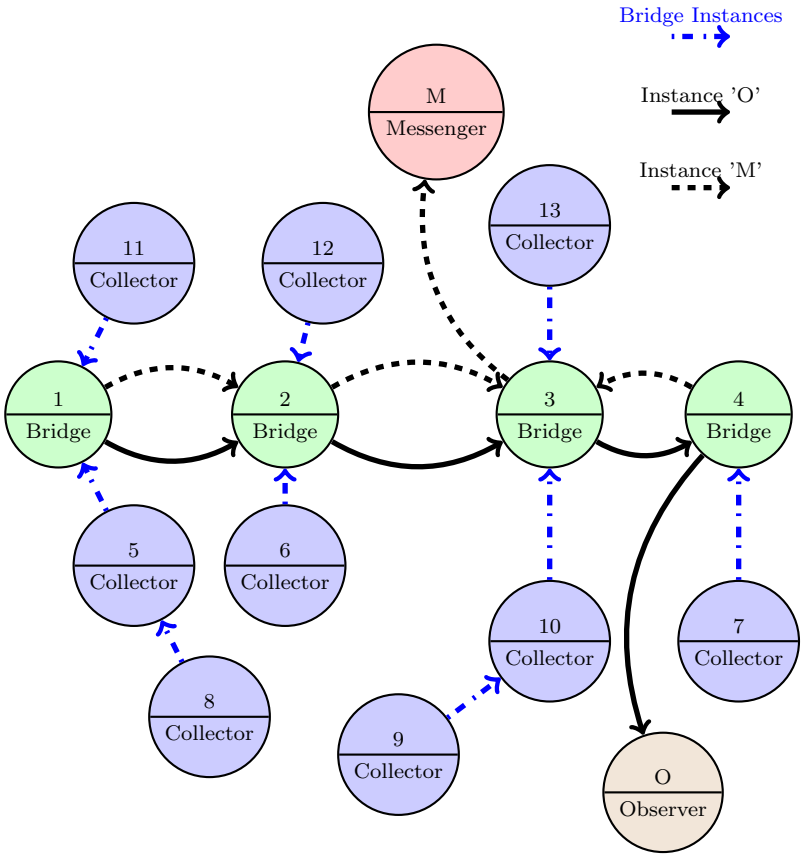


Figure 3.1: The HOIST Architecture

To properly motivate the design decisions behind this architecture, I will begin this chapter by detailing the considerations necessary to examine for any WSN architectural development of this type. I will continue this chapter by presenting a standard, single-layer example WSN network under which an examination of these considerations may be made. The conclusion of this analysis serves to motivate the design of the architecture, which is presented in the final section of this chapter.

3.1 Design Considerations

The principle concerns for this intended deployment of wireless systems is the ability to arbitrarily increase in the size of the deployment and the ability to utilize mobility to

reduce the prevalence of energy drains, while simultaneously enabling delivery of data to remote locations.

3.1.1 Increasing the Size of the Deployment

A paramount concern to WSN architectural development is the ability to arbitrarily increase the number of deployed devices in a manner such that the network is able to continue to support it[42]. With the Contiki operating system, the first practical limitation on the size of a deployment is the restriction on the size of the downward routing table. This table, as mentioned in section 2.8.1 is necessary to enable point-to-point or point-to-multipoint routing in the network. As HOIST is concerned with delivery assurance, the sinks will need to send acknowledgment messages back down to the senders.

With the limitation in Contiki of 15 entries in the downward routing table, a single-sink implementation will only be effective for up to 15 devices before downward routing is no longer possible. Moreover, as Contiki only uses a single downward routing table for all instances on the same device, multiple instances may enter a state wherein each sink is at the opposite end of the network form the other. In this example, all devices would be downward with respect to at least one of the instances. For this reason, I have elected to utilize two concepts in my architectural design.

First, I will be employing a hierarchical model to separate the collecting devices from their destination by using a bridging sink. In this manner, the collector devices are able to be coordinated in groups up to 15 by this bridge device, which will manage all downward routing to the collectors on behalf of the destination sinks. I employ multiple bridge sinks across each deployment area of the network. Each of these sinks is able to increase the number of collectors by up to 15. The practical constraint on the expansion of a deployment is now 15 bridge sinks, which allow for a total of 225 collector devices which may be directly addressed by the single destination sink. Following on with this model, if the bridge sinks coordinate data delivery to the destination by instructing each of their collectors to send data one at a time, then the local flooding issue is also mitigated. Instead of 225

devices attempting to send data over 15 bridges to 1 destination sink simultaneously, in the worst case, now the model can limit this to 15 devices communicating over 15 bridges to 1 destination sink. This implies that the increase in the transmissions is logarithmic with respect to the increase in the number of collectors.

Secondly, this three-tier model presents itself with the ability to reduce the hop-count of the devices by placing the bridge devices in the center of each of their clusters of collector devices. An analysis of this design consideration is presented in section 3.2.

Routing under Mobility

The second of the primary considerations is the reduction of the energy drain of the devices closest to the sink. As [20] mentions, this phenomena may be reduced by adding mobility into the design of the network. By moving the sink about the network formation, no one set of border devices will be burdened continually with a disproportionate amount of messages to reroute during each time period; encircling the deployment area will allow all border devices to share evenly in this burden, better distributing the energy expenditures.

The difficulty with mobility, as alluded to in section 2.8 is that it takes time for all of the DIO and DAO messages to propagate to allow each device to update its tables to reflect the new location of each sink. The parent recalculation process uses a series of metrics to assess the quality of each neighbor before selecting one of them at the new parent. This process is guided entirely under the direction of the Objective Function (OF) set forth by the sink via the DIO message. There are presently only two registered OF functions[48]: OF0 and MRHOF.

MRHOF [49], which is the default OF in Contiki 2.7, uses link metric information to determine which neighbor of the device might be the best parent. As link quality metrics, such as ETX, can fluctuate, this function uses hysteresis to maintain the current parents over short-term assessment periods to prevent frequent routing changes. Only when a link to a new parent is consistently better over time will the device change its routing information.

OF0 [50], on the other hand, is a simple function which primarily utilizes hop-count as

the metric for parent selection. During each recalculate event, all neighbors are assessed for their ranks within the current instance. The neighbor with the lowest rank is then selected as the new parent. While this does result in far more rapid changes to the network, a node's rank is only a rough gauge of its radial distance from the sink. A route utilizing two hops to the sink may offer better throughput than a route utilizing a single hop, albeit with very poor link quality. OF0 has no ability to assess this and will always select the greedy choice to minimize rank.

The primary concern with routing under mobility is the length of time it will take to rebuild the network after the sink begins moving. Once the initially adjacent nodes are no longer able to communicate with the sink, it would be ideal for the devices to immediately seek alternative parents. For this reason, as assessed by [15], ETX "cannot handle mobility in practice." Likewise, if a child node detects that they are now a single hop from the destination sink, it should immediately change routing to prefer the direct connection. The role of the OF in the design of the architecture was selected to address this consideration.

Now that the these primary considerations have been presented along with the options available, I will describe the common architecture used for WSN deployment and analyze which aspects need to be modified for an architecture to meet the design goals.

3.2 Analysis of the Common WSN Architecture

The commonly used model [14], [15], [16], [17], [18], [11] for WSN deployment involves the emplacement of devices in two principle roles. This first role is that of the sender device, whose sole function is to collect and then send that data to a sink device. The sink device, under RPL, may exist alone or in conjunction with multiple such sinks. Many papers describing large deployments, such as [14], allow the sender devices to select the most appropriate destination sink for their data. Furthermore, RPL permits multiple DODAGs within a given DODAG instance as well as multiple instances within a network, allowing for multiple different destination sinks to interoperate.

A sample deployment under such an architecture is displayed in figure 3.2. This example

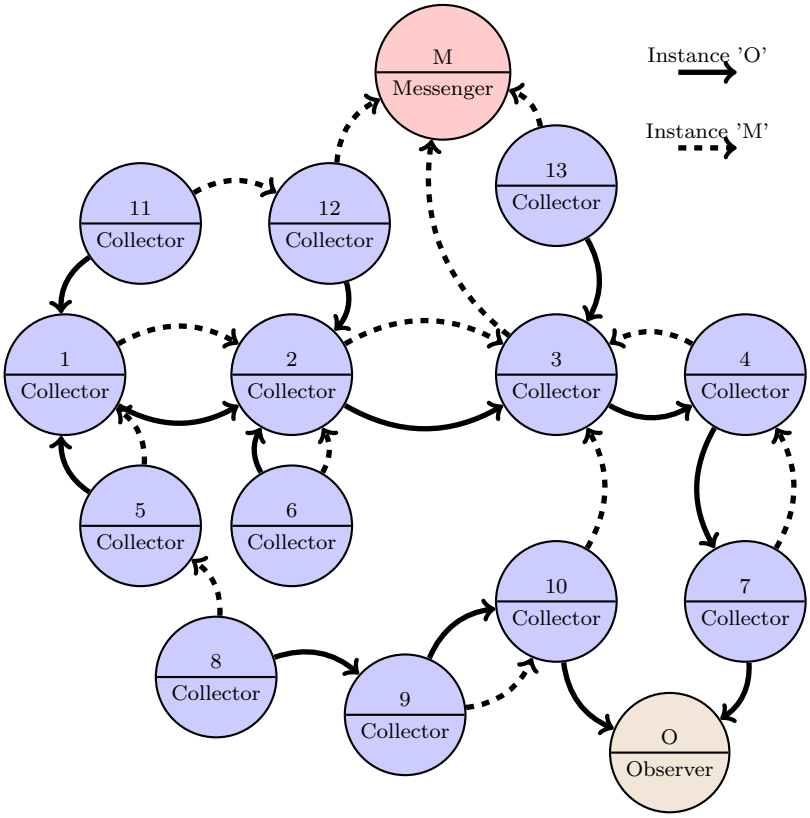


Figure 3.2: Basic RPL Architecture

will serve as the baseline for analysis of several key concepts that drive the creation of my architecture. As depicted, this multiple-sink architecture allows for the desired level of multipoint-to-point routing, with each collector node showing its preferred parent for routing to each of the two sinks. As depicted below, the network is partitioned into two different DODAG instances to correspond with different sinks. The messenger sink, designated **M** is able to receive select data under different routing metrics from the observer sink, designated as **O**. The sender devices here are all designated as collectors to highlight their role in this implementation.

In this network, by way of an initial example to highlight the key nodes, the route to **O** is such that all devices must route their packets through either collector **10** or collector **7**. Collector **7** must route 10 packets during each transmission period for the collectors, whereas collector **1** only has to route three packets each period. This disproportionate use of

the radio to transmit packets highlights the disproportionate energy use in a standard WSN deployment. In this model, collector **7** will run out of power significantly before collector **1**, fragmenting the network and causing a modification of the routing structure. Assuming collector **10** is the only remaining device within range, it will then bear the task of routing and making 12 transmissions each period until it too ceases to function, bringing the entire network down.

Mobility Analysis

Techniques such as described in [17] and [15] alleviate this problem by proposing strategies wherein the sinks move about the structure, allowing a more even distribution of the load. Note that these techniques still suffer drawbacks when using multiple sinks, as depicted in figure 3.2. While the ability of the sinks to move about does address this problem, scaling networks upwards only serves to exacerbate the energy drain effect. The only means of alleviating this effect is if each mobile sink is likewise able to penetrate the mote deployment field and move about the devices, providing equitable amounts of time for each mote. Figure 3.3 below shows a depiction of the initial network following movement about the topology.

In this depiction, the prior unbalanced routing overload formed at collector **7** has been alleviated insofar as it is now only responsible for transmitting two packets per send period. Collector **8**, however, now has the full load of the network under it, forcing it to relay and transmit all 13 packets per send period.

Although the mobility of the sinks has lessened the burden, the implementation of the mobility still leads to systematic inefficiencies. Let us assume that the messenger sink **M** is physically constrained from entering the deployment area. Circling about the motes fully will preclude any one of the perimeter devices from succumbing to the energy drain problem, however, the interior nodes remain continually under higher load than those on the perimeter. Collector **2**, for instance, had to transmit 5 packets per send period towards messenger sink **M**. After its movement, the next period will require collector **2** to transmit 3 packets per send period. While an improvement, it will never be directly adjacent to the

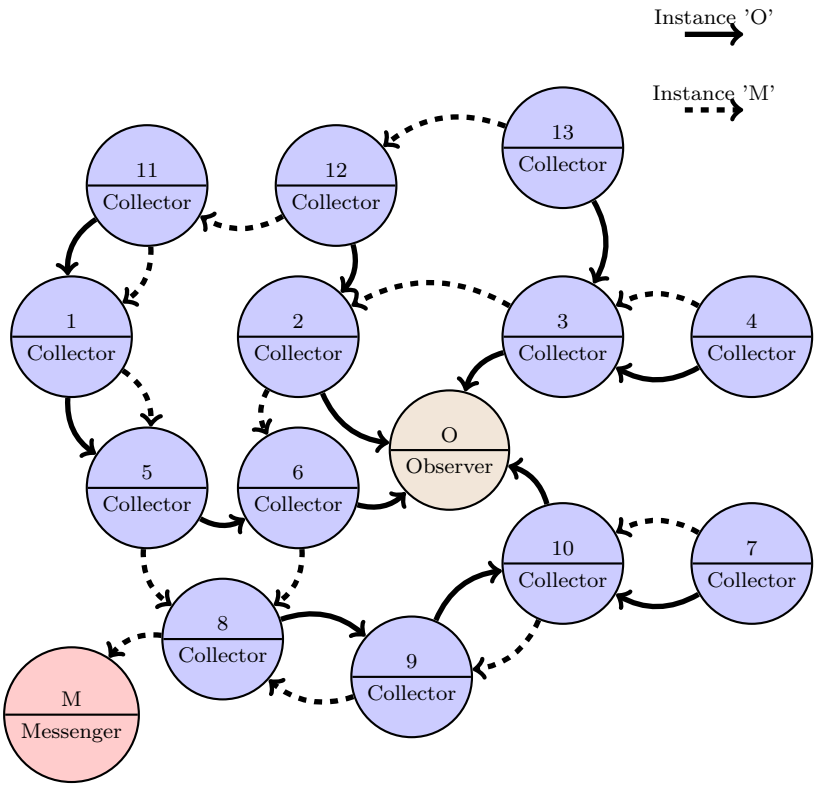


Figure 3.3: Basic RPL Architecture Following Movement

destination. As the number of devices in this deployment scales upwards, the inner devices will route a disproportionately high number of packets over the full duration of the network relative to those on the perimeter.

The second instance, headed by the observer sink **O**, has a more desirable effect. By moving to the center of the topology, more collectors have direct communications access with the observer sink **O**, reducing the overall load on the network. For this state, the most duress on any sender is from collector **6**, which must route four packets per period. The average number of transmitted packets across the four directly adjacent senders is now just over 3. This topology reduces the hop count of the network, however, for a larger deployment, this sink would need to travel to each of the senders to provide them all an equitable opportunity to be adjacent to the sink. This is not practical for large deployments and may not be physically possible depending on the deployment conditions.

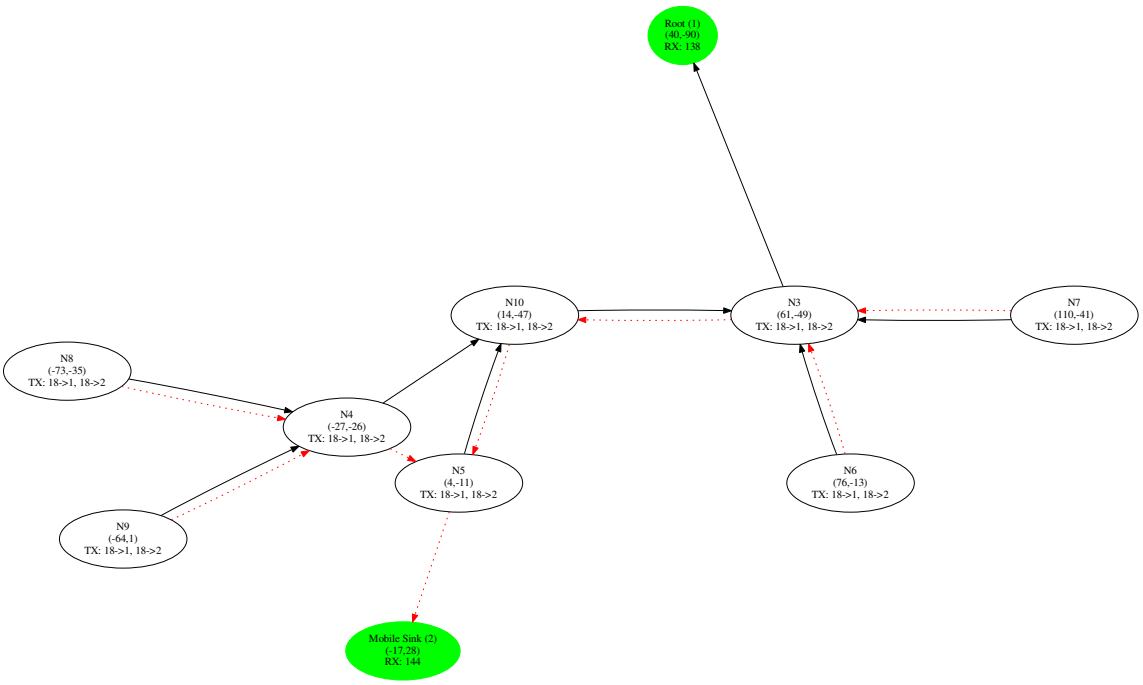


Figure 3.4: Multi-Sink Tree Topology in a Simulated RPL Network

Topological Analysis

One important item to consider is that of an ideal topology to favor for the designed architecture. Is there a simple topological formation which may provide a limitation on hop-count, while not degrading packet delivery rates to the sink?

The observations on the changes to routing in **M** and **O** serve to introduce two principle networking topological formations to examine: tree and star. The tree topology, as shown in figure 3.4[1], is representative of a closely clustered deployment of senders wherein the sink is on the perimeter of the formation. The star topology, depicted in figure 3.5[2], however, offers a much lower overall hop-count across the entire network. A perfect star, wherein each sender is an immediate neighbor of the sink, eliminates the hotspot problem by definition, however, may introduce a greater level of interference as each node must be within

[1]Multi-Instance RPL DODAG depicted after 300s of simulation in Cooja
[2]Multi-Instance RPL DODAG depicted after 300s of simulation in Cooja

transmission range of the sink.

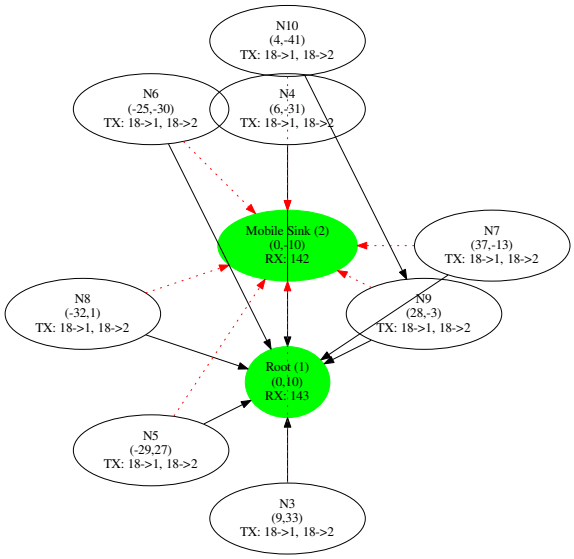
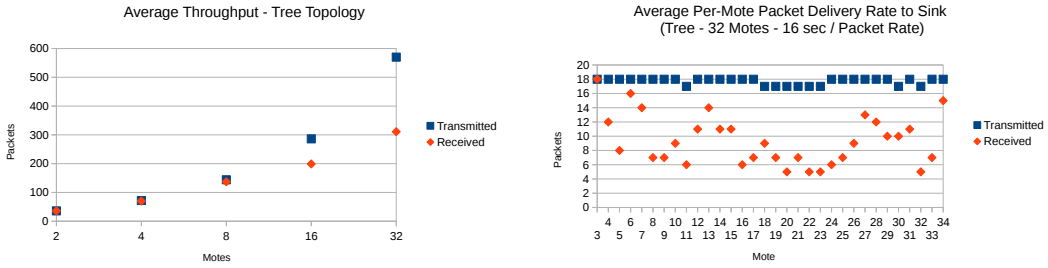


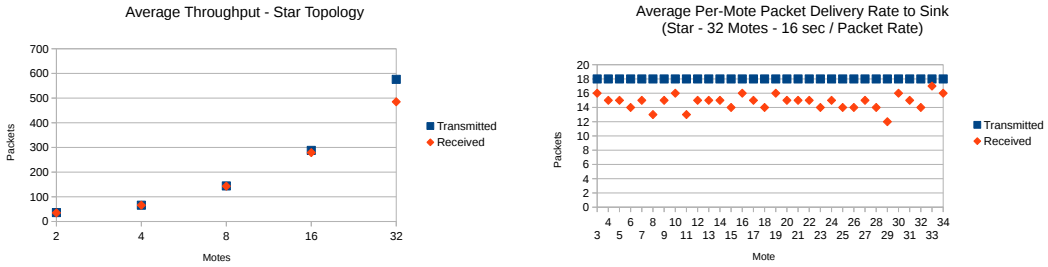
Figure 3.5: Multi-Sink Star Topology in a Simulated RPL Network

While reducing the hopcount of the sender nodes would greatly reduce the energy drain, it still needed to be demonstrated that reducing the problem to a star topology would not negatively affect the collection rate of the deployment field. The following charts are based on simulated runs on a two instance deployment. The simulation in Cooja ran for 300 seconds and data was collected using 2, 4, 8, 16, and 32 sender devices each. Each simulation of the star topology was run using positions selected randomly using the Box-Muller transform[51] using a mean of 3/4 of the transmission range and a variance of the difference between the mean and the transmission range, divided by 3. This enabled each newly added device to be placed within the transmission range of the sink for the star topology. The tree topology used manually selected positions to provide a deterministic tree-like structure. Each test was repeated three times and averaged for the following data. This collection was not aimed to be an exhaustive analysis, but rather serves to provide initial confidence to begin exploring a star based topology.



(a) Average Throughput (b) Average Per-Mote Packet Rate

Figure 3.6: Analysis of the Tree Topology



(a) Average Throughput (b) Average Per-Mote Packet Rate

Figure 3.7: Analysis of the Star Topology

This set of quick tests demonstrated that even in a cluttered environment where a single sink receives data from 32 devices, featuring a star topology in the architecture design would not present throughput degradation as compared to a more typical tree-based structure.

RPL Analysis under Resource Constraints

One further problem with such a deployment presents itself once the architecture is implemented on physical, memory-constrained devices. Every consideration towards this architecture has been solely focused on routing data packets upwards. Upward routing towards a sink remains easily implemented through the storage and use of a single key IPv6 address representing the logical parent of the node. Downward routing, however, become untenable even in small implementations due to the need to store information about all of a node's

children. In RPL, downward routing enables the ability to send messages from the sink down to a particular sender device. This also enables point-to-point routing by sending a message up to the common ancestor and then routing it downward to the destination.

RPL implements this form of routing by storing the IPv6 address of each child device in addition to the IPv6 address of each device the information was directly received from. This routing table of destination and next-hop information uses a lot of space on a memory-constrained device. The Contiki operating system, for instance, only allows 15 routes for the Zolertia Z1 device.

Table 3.1: Downward Routing Table for Collector **8** in figure 3.3

Destination	Next Hop
fe80::09	fe80::09
fe80::10	fe80::09
fe80::07	fe80::09
fe80::08	fe80::08
fe80::05	fe80::08
fe80::06	fe80::08
fe80::01	fe80::08
fe80::11	fe80::08
fe80::02	fe80::08
fe80::12	fe80::08
fe80::13	fe80::08
fe80::03	fe80::08
fe80::04	fe80::08

In a standard deployment, if one device served as the sole route to the sink and there were more than 15 other devices, the downward routing table on that device would not receive the complete list of devices, preventing proper routing. While the transmissions are typically upward [12], any form of transmission assurance will require the downward routing of an acknowledgment back to the originator. This simple architecture precludes proper responses, succumbing to the problems of scale.

The designed architecture must assess the capabilities of the standard devices for implementation and reduce the expected number of downward nodes in any given device's routing table. By preferring an architecture that biases the design to use smaller hop counts and distributes the collector devices among multiple intermediary sinks, this consideration will be met.

3.3 HOIST Architecture

The fundamental tenet of this design is to enable data collection to occur in multiple, geographically segregated regions and to transport that data back to a central processing center. Two primary aspects of this design were posed to address the issues of the common model: the ability to increase the size of the deployment and the mitigation of the energy drain issues of a static deployment.

The approach used in this architecture combines hierarchical, multi-tier routing with the reduction of the majority of the devices to a star topology to address these issues. A similarly constructed observer sink is also supported to allow live-collection to occur on site, while data stored on each collector may be retrieved periodically by a mobile messenger.

3.3.1 Design

The design meets the constraints by rebuilding the canonical network using a hierarchical model. There are three tiers of this hierarchical design, featuring four distinct device roles. In tier one, the WSN devices are actually collecting data on the environment. Tier two consists of bridging sinks that are responsible for routing to the mobile sinks and coordinating data transfers between collector devices and their destinations. Tier three consists of the two mobile sink types: the messenger and the observer. The messenger collects blocks of data from each deployment area for the physical transfer to a processing center, while the observer is sent real-time information from the field when in range.

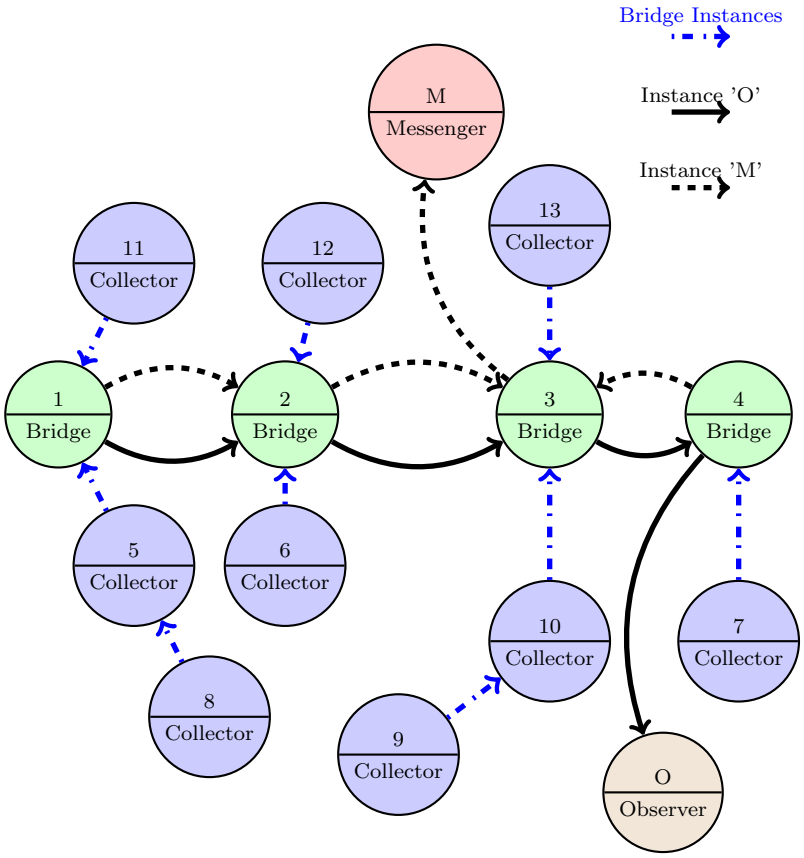


Figure 3.8: The Designed Architecture

Tier One - Collectors

The first phase of this design is to designate a geographically contained area for sensor deployment. The primary WSN devices may be placed throughout the environment as desired by the implementation. This first tier of devices represent the mainstay of the network and are hereinafter referred to as collectors. These collectors are programmed to activate their sensors on a routine schedule and store these data into an array for later access. As there is no routine data transmission on collection, the energy requirements of the network drop as compared to the traditional network deployment.

Upon reception of instructions to begin sending data for the messenger sink, the collector will stop collection and begin transmitting all of the stored values, beginning with the first

element of the array and proceeding until the array is exhausted and all data has been send and their proper transmission confirmed.

If, on the other hand, the collector receives instructions to begin transmitting data to the observer, then it will continue normal collections and populate the array; however, it will additionally send the live data out for ultimate reception by the observer. There are no delivery guarantees for the observer under this design. In an environment in which the observer and messenger are active simultaneously, those collectors sending to the messenger will cease communications with the observer until they have finished. At this point, the collectors will resume data collection and their live transmissions to the observer.

Tier Two - Bridges

The second tier of this design is the new addition to the standard model. These bridge sinks are the crux of this architecture. The bridges firstly enable the collectors to maintain static routing without any perturbation by the mobile sinks in the environment. By removing the necessity to have responsive updates to the routing on the part of the collectors, they can use the more powerful MRHOF function, which allows them to select the routing using link-quality aware metrics, such as ETX. This provides more stability and better transmission quality to the majority of the devices in the deployment, the collectors.

These bridge devices shall be placed in a linear topology along the central line of the collector deployment area. As bridges are the root of their own DODAGs, the nearest motes to each bridges will select them as their preferred root, fragmenting the collectors up into much smaller units that surround the bridges. This biases the constructed network towards a star topology, which both reduces hop-count and the effects of the energy drain phenomena.

Furthermore, as the bridges themselves are the only ones needing to route to the mobile sinks, they can use the more responsive OF0 to adapt to the movements of the sinks. The smaller number of these bridges compared to the collectors also reduces the number of routing changes that are necessary as the mobile sinks traverse the environment.

The bridges also actively coordinate all transmissions from their subordinate collectors. Instead of succumbing to a flooded environment when the messenger sink arrives, each bridge uses their downward routing tables to signal each collector, one by one, to transmit their data as fast as they are able to receive acknowledgments.

These features of the bridging tier principally allow for scaling of the environment on physical memory-constrained devices.

Tier Three - Mobile Sinks

The third tier represent the ultimate destinations of the data. The first device that may be a member of this tier is the messenger. This sink enters each deployment area using a nomadic pattern. The device will arrive, establish a network, receive the data while orbiting in the same geographic area, and then leave to either travel to the next deployment area or to return to the collection station for a data dump.

The messenger arrives and creates its network as the root of its own DODAG. As the only devices which will join the messenger are the bridges, this network forms much faster than the traditional model where the DIO sent out must first propagate to every sender before routing is completed. As the bridges have already established their static networks with the collectors, this initial network creation phase by the messenger is rapid. Furthermore as a second message, the DAO, must be sent out following DIO reception, this model also enables downward routing in a significantly more rapid manner, as the bridges would have already built their routing tables.

Once in position, the bridges signal their collectors to send and then serve to route this data to the messenger. Once each message arrives, the messenger sends an acknowledgment, which is repeated and propagated down to each collector. The data received by the messenger contains the collected value of the sensor, the index of the collected value for general collection time information, and the originator of the data. It then places all of these data into its own array. Once the bridges all signal that their data collection has ended, the messenger will signal the mobile carrier to proceed to the next destination.

The second device on this tier is the observer. Like the messenger, when it arrives, it sends a DIO to begin the process of receiving data. The bridges send a start signal to the collectors, which in turn send their data up to the observer. This data is live and presented as collected, with the first block of data from each device being the last recorded values. In this manner, the entity using the observer device can immediately see the conditions of the area. As new values are collected, they are directly sent to the observer.

As each mobile sink leaves the area, the instances they generated ultimately reach the end of their lifespan and the bridges drop their entries, awaiting the next event.

3.3.2 Demonstration of the Architecture

Figure 3.9 demonstrates a sample deployment of the same devices under this architecture. This model introduces a second tier of sink devices to serve as a bridge between the collectors and the messenger sink. These bridges are deployed in the center of the WSN deployment field to fragment the network into multiple, smaller star-based networks, aimed at reducing hop-counts to satisfy both the energy and resource constraint considerations. This architecture differs from the prior, standard architecture in two fundamental ways: hierarchical routing and star topological formations.

Hierarchical Routing

This design incorporates a new hierarchical layer to the existing sender and sink nodes. This layer of bridging sinks seeks to mitigate the most severe problems of the prior model by fragmenting the collecting nodes into smaller, geographically separated units and then completely removing the mobility aspect from their routing.

Examining the above example of different topological formations and the hotspot phenomena, this architecture places these bridging nodes in the center of the deployed collection fields. Segregating the collecting nodes into smaller units across a network by this technique not only reduces the hopcount between any given collector and the sink, but also reduces the size of the downward routing table. In a canonical example, one network may

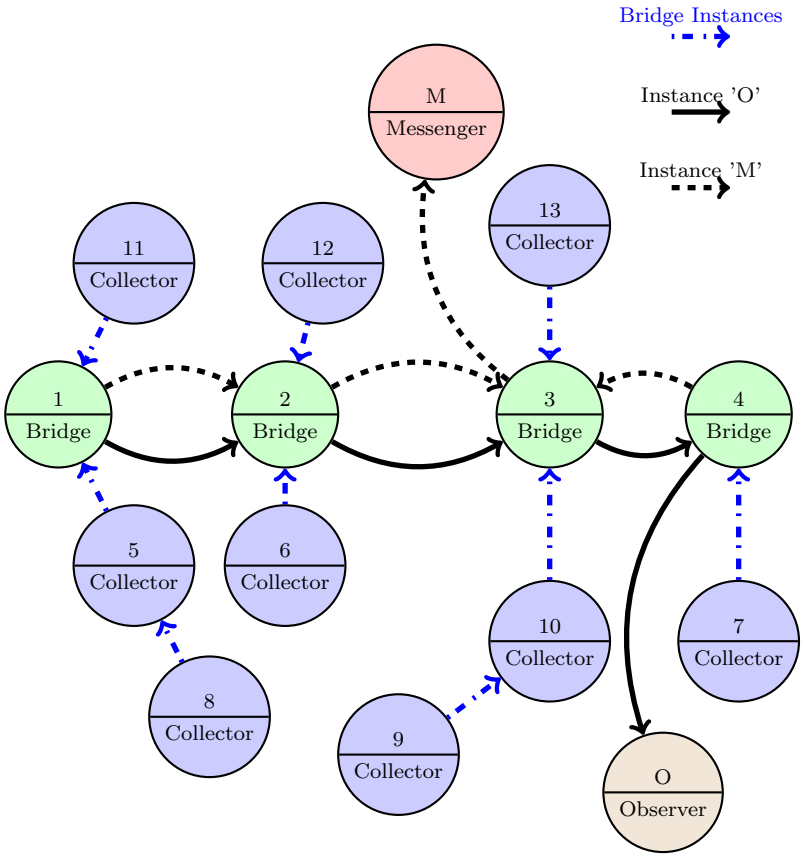


Figure 3.9: The HOIST Architecture

only be able to field 15 collectors due to this routing table limitation in Contiki. Each of these bridging sinks may now support up to 15 collectors, enabling the ability to scale the deployment through the addition of more bridging sinks.

On the other end of the bridging sinks are the messenger and observer devices. These nodes remain mobile and may traverse each environment as physically able. The difference under this architecture, however, is that the collectors no longer need to adjust routing as they wander the environment, as shown in figure 3.10. The bridging nodes are the only devices that need to adjust to the new locations. By selecting OF0 as the OF of these instances, the lack of hysteresis enables bridge sinks to more quickly adjust to the new locations of the two mobile sinks. As the link quality to these sinks are always in flux,

having a link-quality aware OF will not be as helpful as having one that is most responsive.

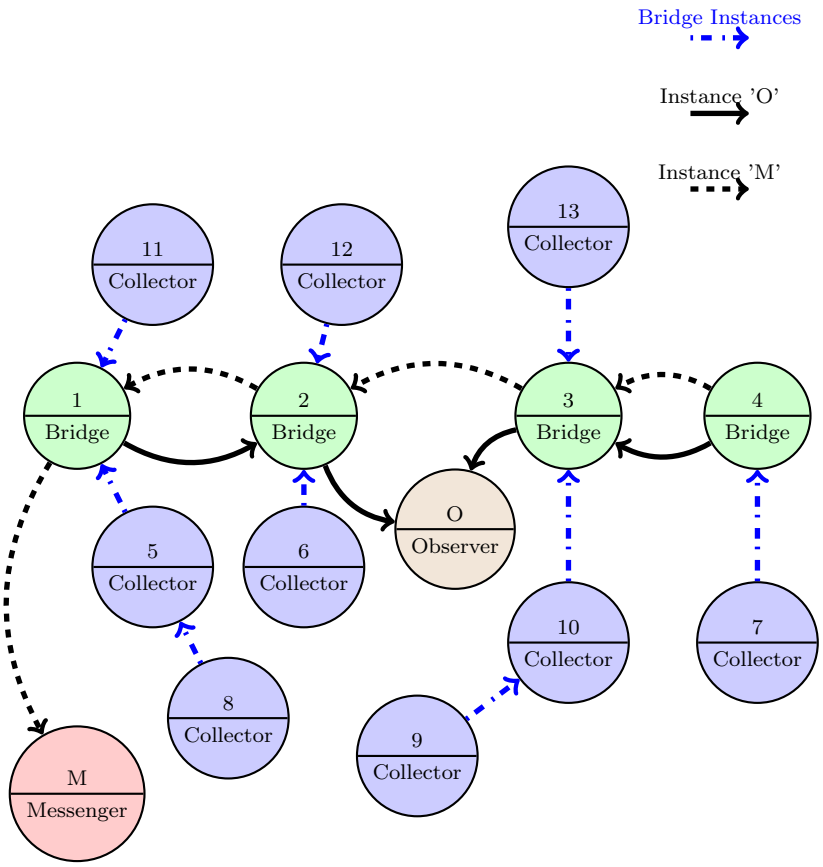


Figure 3.10: HOIST Network after Mobility

By developing this architecture hierarchically, this enables a deployment area to use link-quality based, static routing without any need for adjustments over time for most of the routes and rapidly changing routing based on minimizing the hop-count for routing to the transient devices. Once these mobile devices have left, the instances they manage decay and are purged, removing all communications and destroying the DODAGs until their return, further minimizing the energy expenditures.

3.3.3 Network Communications

The network communications model for facilitation of HOIST is described in this section. The goal of this communications model is to provide a means for the bridge sinks to detect and coordinate all data transfers from their collectors to the arriving mobile sinks.

The messenger sink requires the sending of all stored data by each collector with data delivery assurance. The bridge sinks need to coordinate each of their collectors, one at a time, to send their data until they have finished, at which point the next collector may begin their transmission.

The observer sink will not interfere with messenger operations or any of the collected data. When this sink arrives, the bridge will signal each of its collectors to send their latest collected data immediately, following which, they will send each new piece of data as collected. If the messenger is active in the area, the observer will not receive any data until the messenger has left, at which point, it will resume receiving live updates.

Messenger Arrival

The messenger, as an RPL sink, sends out regular DIO messages. If the DODAG formed around the RPL instance is stable, then the time between each message doubles, out to the maximum send time. The first DIO received by a bridge from the messenger sink is the catalyst that begins the networking process, as shown in figure 3.11.

The timeouts depicted are present to govern data assurance. The bridge begins the sequence after receiving the initial DIO by sending a **`M_BEGIN_TX`** packet to each collector in its downward routing table, one by one. If the collector does not reply within the given timeout window, a duplicate transmission is made. Once a collector replies with the same begin message, it will then begin sending data from its collection array. This array initially begins at index 0 and is reset to index 0 following each successful data dump to a messenger sink. Any interruptions in the link will resume at the index following the last successfully transmitted index.

Each data transmission from a collector begins a local retransmission timer. If it receives

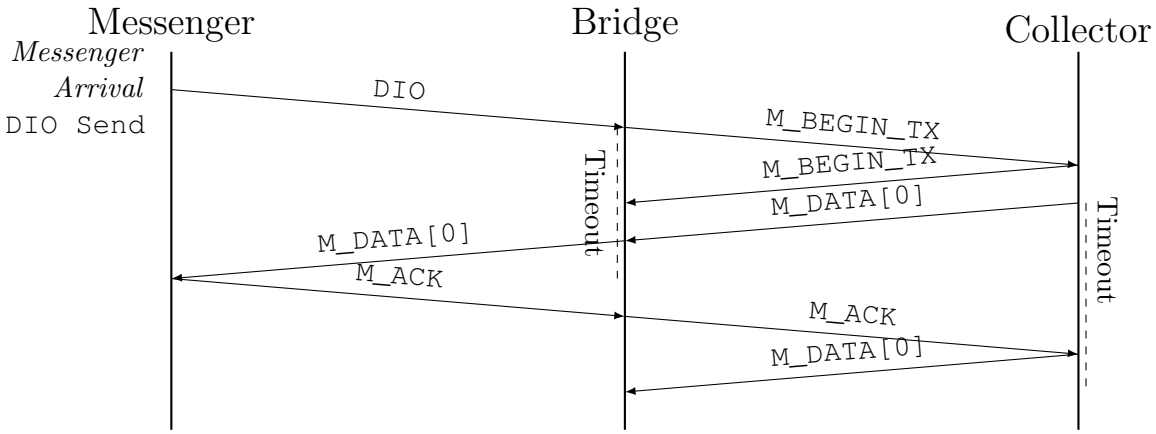


Figure 3.11: Start of Messenger Data Transmission

a **M_ACK** message in response before the timeout, it will reset the timeout and send the next block of data. Otherwise, it will resend the data and reset the timeout, in a cycle, until the data is either acknowledged or the link to the messenger is dropped by the bridge.

At the conclusion of the collector's data transmission, it will generate a **M_DATA_END** message for the bridge, as shown in figure 3.12.

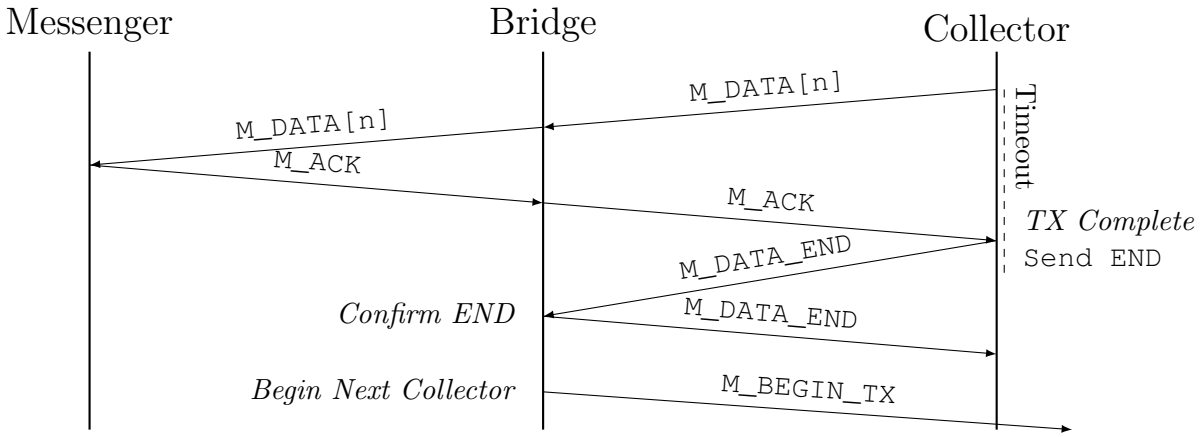


Figure 3.12: End of Messenger Data Transmission

Once the bridge receives **M_DATA_END**, it replies immediately with the same to the

sending collector and then sends **M_BEGIN_TX** to the next collector in the downward routing table. The collector receiving the reply will stop its timeout and resume normal collection operations. If the reply is lost, the collector will simply retransmit the end message again until acknowledged.

Once the bridge receives the end data message from its last collector, it will reset the current collector reference to the first entry in its downward routing table and then signal the messenger sink that it has finished. The messenger sink, having received a number of completions equal to the size of its downward routing table will have the option to either signal its carrier to continue, or will simply wait until the autonomous vehicle naturally leaves the area.

Observer sink operations behave similarly to the messenger data transfers. The beginning of this process is depicted in figure 3.13.

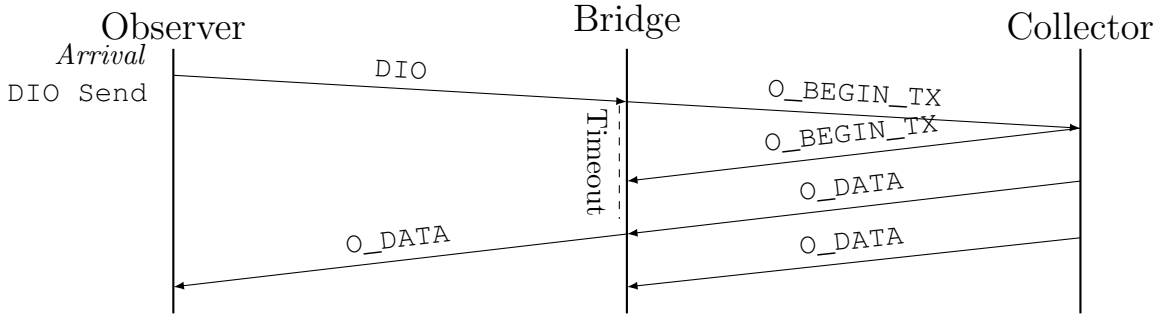


Figure 3.13: Start of Observer Data Transmission

The difference between this and figure 3.11 is that the collector begins by sending the most recently collected block of data and then only sends a new packet when data is normally collected. The only verification for reception is between the bridge and the collector, to ensure each collector receives the instructions to either begin or conclude transmissions.

One implementation item of note with this diagram is that the collector only sends data during its collection phase. When the messenger sink arrives and the bridge coordinates a transmission start message, the first action the collector takes is to disable its collection

timer. This effectively terminates observer transmissions. This is an intended effect to prevent undue interference during the messenger data transfers. Once the messenger has left the scene and its lifetime has decayed, the bridge will send the terminate signal and the collector will resume collection operations. At this point, the observer will resume receiving data.

As there is no natural conclusion to this form of observation, the process will only cease when the observer either leaves range or self-terminates. At this point the lifetime of the instance will decay until it is purged by the bridge. Once the bridge terminates this instance, it sends messages with delivery assurance to each collector that they can stop their transmissions, as shown in figure 3.14.

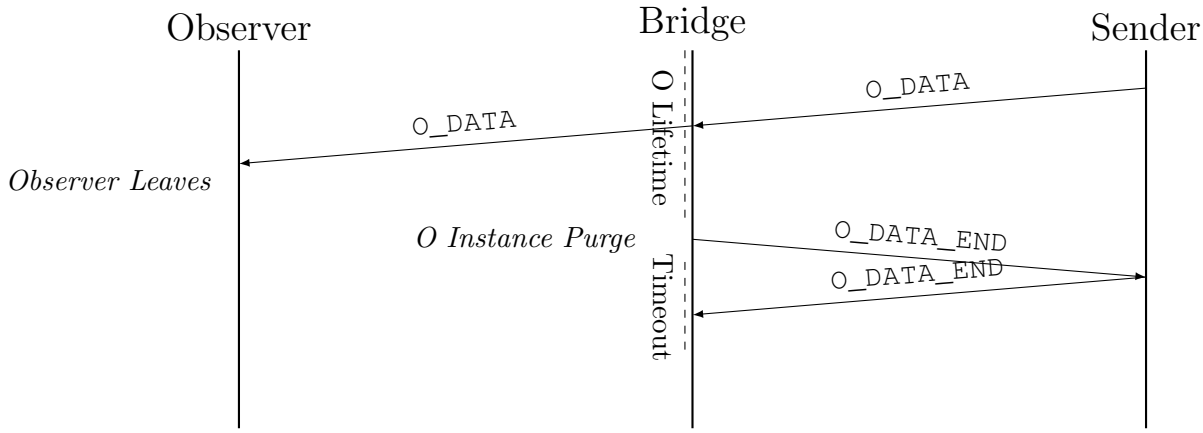


Figure 3.14: End of Observer

Chapter 4: Implementation

The HOIST architectural design from the previous chapter is based on the IEEE 802.15.4 and RFC 6550 (RPL) specifications. While the Contiki operating system implements both of these protocols, there are two key factors to note. First, these protocols do not fully define how they are to be implemented. RPL, for instance, specifies that while it permits multiple instance networks, it is beyond the scope of RFC6550 to provide an implementation thereof[12]. Second, the nature of WSN devices is such that they are memory-constrained. Developers of the Contiki operating system have left several aspects of RPL as presently unimplemented, likely for space restrictions for compatibility with older devices. This poses a problem for the implementation of the designed architecture, necessitating the modification of the operating system to complete or define the protocols prior to implementation.

This chapter will cover the assessment of the present RPL code in Contiki 2.7 and describes my design and implementation of modifications necessary to implement the necessary aspects of RPL for this architecture implementation.

4.1 Assessment of the Routing Protocol

The IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) is specified by IETF RFC 6550[12]. This standard was partially implemented by the Contiki operating system to facilitate the routing of messages over the IEEE 802.15.4. Each sink device is a designated destination for multipoint-to-point routing. In this common routing scheme, all non-sink senders route using only a single preferred-parent per instance, which serves as the next hop to transmit the message for ultimate destination to the sink. This simple mechanism allows such low-powered and memory constrained devices to successfully route messages to their intended destinations.

When a sink first comes online, it creates a logical collection of Destination-Oriented Directed Acyclic Graph (DODAG)s under a common DODAG instance. While an instance may contain multiple DODAGs, RFC6550 specifies they must all operate under the same Objective Function (OF). That is, they all use the same metrics for determining routing. This information is propagated initially by the sink, using a DODAG Information Object (DIO) message. When this DIO is received by the senders, if they have not previously associated with this instance they will join it and add the device they received this message from as its preferred parent for routing. After this action, the senders will rebroadcast the DIO to motes more distant from the sink than themselves. Any future routing this node performs on received messages will use these preferred parents as the next intermediate hop to send the messages to.

One of the core requirements of the designed architecture of this thesis is that multiple, discrete, routable tiers of devices must be able to inter-operate in order to facilitate scaling of device deployments. Mechanically, there are two aspects of RPL which may, from the outset, be employed to enable this multiple routing requirement: DODAGs and Instances.

RFC 6550, the definition standard for RPL, permits multiple DODAGs to be created under a single instance, allowing routing to separate destinations as befits the design of the network. Section 8.2.2.3 (DODAG Selection) of that document, however, specifies that "a node must only join one DODAG per RPL Instance," which rules single-instance implementations out as a feasible possibility to extend for allowing multiple-sink routing. Contiki, as of version 2.7, also does not presently support multiple DODAGs per instance.

As such, there is only one fundamental facet of RPL which may be used to achieve this objective: Multiple Instances. RFC6550 specifies in section 1.1 that "[a] network my run multiple instances of RPL concurrently," however, the implementation of such a network is left up to the designer of the network and is out of scope of the RFC. Contiki, unfortunately, also does not presently support this, rendering the implementation of this architecture impossible to directly perform. The analysis of this problem is detailed by the next section.

4.1.1 Problems with the Contiki 2.7 RPL Implementation

Despite RFC 6550 specifying that multiple instances are valid, the Contiki implementation only partially supports multi-instance networks. This partial support appears in the form of limited data structure support for multiple instances. The majority of functions in this implementation of RPL either use the **`default_instance`** of the intermediate routing node for all decision making on the next hop for a message, or are otherwise hardcoded assuming only a single instance exists.

The existence and use of a default instance itself is also specified in RFC6550 and would achieve a partial success towards the implementation of this architecture. As either the messenger or observer came online, the bridge could signal down to its children to begin transmissions. The collectors at this point would be free to send messages directly to either the messenger or observer. As the addresses used for this style of routing would not exist within the current collector-bridge DODAG, the confused collectors would simply route the messages up to their local sink. Once arrived, the bridge would assess routing and recognize the destination as its own registered sink and route it properly from this point forwards. Unfortunately, the downward routing of acknowledgments would be impossible so no message delivery assurance would be possible. Moreover, this approach still requires multi-instance networking to handle both an observer and a messenger. As this partial solution is not tenable, further assessments were required.

Continuing the assessments, it was determined that under the Contiki implementation, routing over RPL is never done using the addressing information on the messages themselves. The operating system uses the preferred parent of the default instance on the intermediate routing node solely for routing decisions. As there can be only one default instance, the last modification to this reference is the only one observed. For a single instance implementation, this will still result in the proper routing as there is only ever one parent to select from; however, once a second instance is added, the routing will always favor the last set default instance. As if it were bad enough that messages are being routed without checking its headers, but the headers themselves are actually overwritten by each

intermediate routing node's default instance values. This corrupts the addressing on each message while simultaneously ignoring it, as demonstrated in the log on table 4.1.

Further exacerbating this problem, each routing node must, by definition, maintain information about the list of potential parents that each DODAG may use to select a preferred parent to route to. Each DODAG by necessity must track its own parents and their rank, link metric, and Destination Advertisement Trigger Sequence Number (dtsn), among other metric-based values. These metrics must be tracked separately by DODAG and, ultimately, by instance.

In Contiki 2.7, there is only one, global, parents list for RPL. All updates to each parent overwrite the metrics in that parent's list entry, regardless of which DODAG or instance to which those metrics refer. This issue will ensure, regardless of the other improperly-implemented aspects of multiple instance routing, that each preferred parent recalculation will force the preferred parent of each instance to align to a single destination; this will prevent proper routing to one or more of the instances. This problem is demonstrated in Figures 4.2, 4.3, and 4.4 in the following section.

4.2 Analysis of Contiki 2.7 Multiple Instance Routing

To assess the problems with the routing, it was first necessary to understand the mechanisms in the operating system which govern instance creation, neighbor tracking, parent assigning, and message routing. The first stage in building the network is for DIO messages to be sent. Once a sink comes online, it logically creates its own DODAG instance and begins sending out DIO messages to announce to the network that an instance is available for joining.

The key phase of processing occurs when the receiving node calls the **`rpl_process_dio`** function. In this function, the instance information is extracted from the DIO and checked against the **`instance_table`** for a known match. If there is no entry in this table, then the new instance is joined through the **`rpl_join_instance`** function. The DODAG is likewise assessed and added, if not already present.

Once this processing is complete, the desired result is that a new instance is added to the

existing instances, the DODAG specified in the DIO is likewise added to the new instance, and a **preferred_parent** for that DODAG is added and refers to the node that sent this DIO message. The function allocates a new DODAG, adds the source as the new preferred parent, sets the OF as specified by the DIO, copies over the DODAG address, and sets this as the **default_instance** if one does not already exist. Finally, the function starts a DIO timer to handle local rebroadcasts of this DIO for more distant nodes.

4.2.1 Base Contiki 2.7 Multiple Instance Routing Analysis

For this initial test, a scenerio, depicted in the following figure, was implemented in the Cooja simulator. This linear topology features one sink at each end. The large circle that is centered around sender **4** represents the transmission range of the device. All of the devices have similar transmission ranges, ensuring they can only communicate directly with their immediate neighbors.

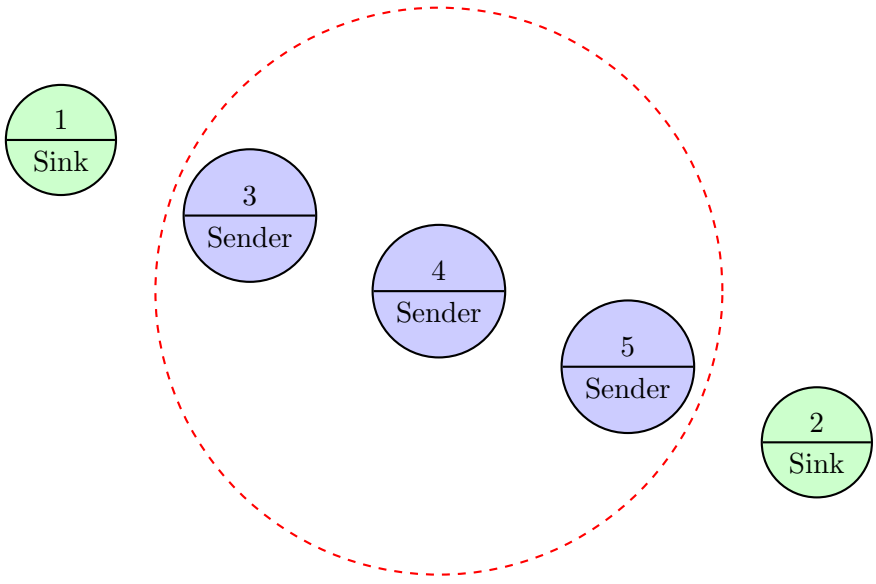


Figure 4.1: Multitest.csc Topology

In this scenario, each of the nodes are configured to send one message every four seconds.

The destination of each message alternates between sink **1** and sink **2** after each send. Correspondingly, if sender **3** sends one message to sink **1** at the 13 second mark of the simulation, it will send its next message at 17 seconds to sink **2**.

When this network comes online and all devices power up, sink **1** creates an instance with **instance_id == 1** and **dag_id == aaaa::1** and sink **2** creates its instance with **instance_id == 2** and **dag_id == aaaa::2**.

Sink **1** starts by sending a DIO out, which is initially received by sender **3**. Sender **3** processes this DIO and adds instance 1 to its table, with sink **1** as its preferred parent. Sender **3** then transmits its own copy of the same DIO, which is received by sink **1** and sender **4**. As it already has an entry for this instance at a lower rank, sink **1** discards the DIO immediately. Sender **4**, on the other hand, adds instance 1 to its instance table and sets sender **3** as it's preferred parent.

After the full cycle of DIOs from both sink **1** and sink **2**, the network looks like the situation depicted in figure 4.2.

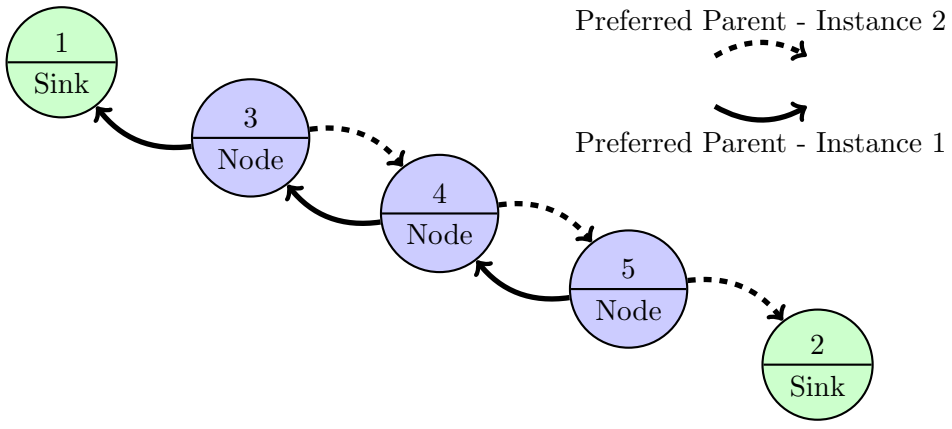


Figure 4.2: Full DIO Reception Showing Preferred Parents

At this point, the network is well formed for two instances and the first messages are ready to be transmitted. This is where the implementation of RPL begins to fall apart rather abruptly. The following table describes a sequence of six messages transmitted, as

was observed through a Packet Capture (PCAP) analysis with the Wireshark program. The entries that are italicized and in red show incorrect states.

Table 4.1: Routing of Messages using the Multiple Instance Scenario

Message	Source	Destination	Route	RPL Header	Delivery Status
1	4	2	4 ⇒ 5	***None***	Delivered to 5
			5 ⇒ 2	Instance 2	Delivered to 2 and Accepted
2	3	2	***3 ⇒ 1***	***None***	***Delivered to 1 and Rejected***
3	5	2	5 ⇒ 2	***None***	Delivered to 2 and Accepted
4	5	1	***5 ⇒ 2***	***None***	***Delivered to 2 and Rejected***
5	3	1	3 ⇒ 1	***None***	Delivered to 1 and Accepted
6	4	1	***4 ⇒ 5***	***None***	***Delivered to 5***
			5 ⇒ 2	***Instance 2***	***Delivered to 2 and Rejected***

This is the typical result of attempting to send messages to multiple instances in Contiki at present. There are several critical failures which led to this failed routing.

First, only two of the six messages contained a valid RPL header, and these messages only received the header after the first hop. This lack of headers on initial send is a bug in the uIP implementation within Contiki that affects all of Contiki's RPL traffic. While a single-instance environment has no technical need of headers as routing can be handled through the default router, once multiple instances exist in a network, headers become critical for proper routing. None of the messages initially transmitted by a sender attach a header. Even when a message is received for retransmission, the existence of an RPL header is never ascertained; a new header is generated using the current node's default instance information and blindly overwrites the header on the message.

This model presents a crippling flaw in Contiki's RPL routing; however, it only presents itself under the case of multiple instance routing. Under a single instance only, the addressing of the header would always point to the same location that each intermediate router's default instance would, mitigating the need to even check.

Second, the routes of half of the messages are incorrect. In figure 4.3, the initial preferred parents follow the sources of the DIO messages and properly reference the appropriate

parents; however, the parents of instances aren't directly used. All routing is done through the **preferred_parent** of the **default_instance** if the destination is neither a directly adjacent neighbor nor in the downward routing table.

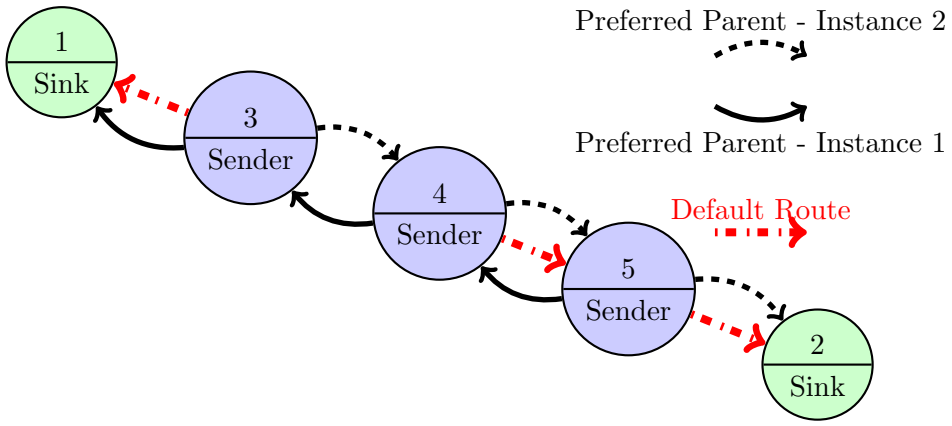


Figure 4.3: Showing the Default Instances in the Scenario

Figure 4.3 shows the path of the default instances as existed during this run. The default instances are represented as red, dashed arrows in the center of the routing. The preferred parents for the instance for sink **1** are solid black lines arced below and the preferred parents for the instance for sink **2** are dashed black lines arced above.

Any message sent from either sender **4** or sender **5**, regardless of which sink and instance it is addressing, will be routed directly to sink **2**. Likewise any message received by sender **3** will route directly to sink **1**. Table 4.1 shows the incorrect routing in italicized red. Beyond this, each new DIO received that affects the **default_instance** may arbitrarily switch the direction of the routing until the next DIO comes in from the other instance.

Furthermore, this depiction only survives until the first rank recalculation event for the instance. At this point, all parents are investigated for their ranks to see if a change is warranted for the **preferred_parent**. In the case of sender **4**, both sinks are equally distant, so hysteresis should allow its parents to remain stable. The other nodes, unfortunately, will not be so fortunate. Sender **3**, for instance, will perform the recalculation and

see that the rank on sink **1** is significantly less than the rank on sender **4** and adjust the **preferred_parent** for each instance to sink **1**.

This occurs because there is only one, global, RPL parent's table. This table holds entries for all neighbors seen and contains metric information for the communications link, however, there is no reference associating this to a specific DODAG or instance. Due to this, if sender **3** receives a DIO from sink **1** for instance 1, then the parent's table entry for sink **1** will contain a rank of 256, which is the minimum rank for Contiki, and will be the initial preferred parent for instance 1. When the same node receives a DIO from sender **4** for instance 2, then the parent's table entry for sender **4** will contain a rank of 712, and sender **4** will be the initial preferred parent for instance 2.

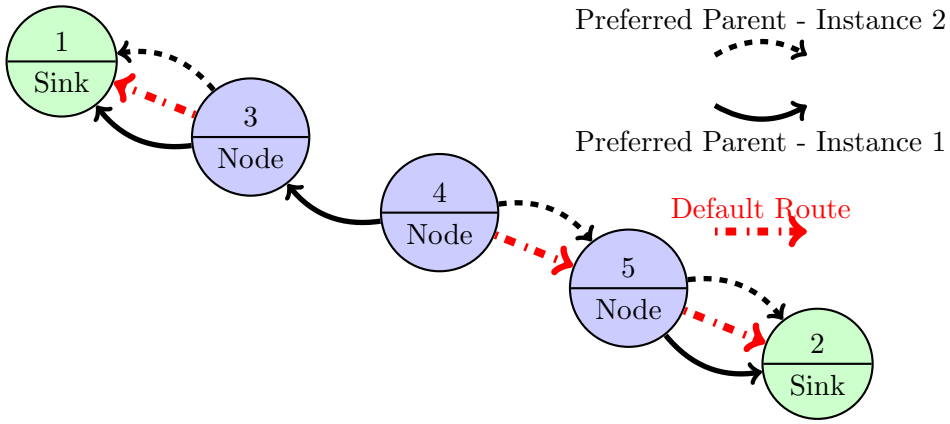


Figure 4.4: The Fatal Flaw of a Global Parents Table

When the recalculation event begins for instance 2, all parents are checked and it is noted that the rank for sink **1** is significantly lower than the rank for sender **4**. At this point, the preferred parent for instance 2 is erroneously changed to sink **1**. Figure 4.4 is a representation of the preferred parents over time, highlighting this critical flaw.

At this point in the scenario, even if routing could be based on the instance intended, the erroneously modified **preferred_parent** entries would prevent the appropriate destinations from receiving the messages. This highlights the initial state of Contiki 2.7 for

multiple sink operations.

4.3 Completed Modifications to Contiki 2.7 for Multi-Instance Routing

This section documents the updates to Contiki 2.7, in phases, that were made to support multiple instance routing, as needed for the implementation of this thesis.

Table 4.2: Summary of Changes for Multiple Instance Routing

Count	Actions Performed
8	Source File Changes
1	Function Obsoleted
5	Functions Added
5	Function Prototypes Added
25	Functions Modified
2	Structures Added
2	Structures Modified
1	Header Added
5	Function Debug Outputs Amended
1	Multiple Instance Debug Block Added

4.3.1 Optimizing the Environment

As this thesis implementation requires aggregation and storage of collected sensor data, the first stage of the application is to optimize the code environment. The Contiki operating system consists of a single process running on a microprocessor in an infinite loop. Application code for Contiki is linked into this kernel process and is executed as a module of the overall system. As such, Contiki supports a wide swath of options to support many different applications.

The first stage of any project is to optimize the environment to reduce the memory footprint on these severely constrained devices. The Zolertia Z1, for instance, has only 8KB of RAM and 92KB of flash memory for all data storage. Of this, only 64KB of memory had

been supported using the **msp430f2617** compiler toolchain, greatly reducing the available space. The most recent update to this compiler extends this, however, other devices, such as the Tmote Sky with its 48KB of total flash memory, must also be able to run this code.

The following snippets from the Makefile show the optimizations applied for all applications created for this thesis implementation.

Listing 4.1: Makefile

```
CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
CFLAGS += -ffunction-sections
LDFLAGS += -Wl,--gc-sections,--undefined=_reset_vector__,--undefined=InterruptVectors,
  --undefined=_copy_data_init__,--undefined=_clear_bss_init__,--undefined=
      _end_of_init__
```

Listing 4.2: project-conf.h

```
#define PROCESS_CONF_NO_PROCESS_NAMES 1
#define UIP_CONF_TCP 0
```

The following table shows the data collected on the efficacy of these optimizations at reducing the firmware size. The values are in bytes and are listed by their sections.

4.3.2 RPL Parents Table Modifications

One of the core problems with implementing multiple instance networking within Contiki 2.7 is that there is no means for storing metric information in each candidate parent that is associated with a particular instance. The solution I chose for this problem is to remove the existing global **rpl_parents** table and replace it with a similar table that is associated

Table 4.3: Firmware Size Comparison of Optimizations

Filename	Version	.text	.data	.bss	.dec
static-sink.c	Original	44289	210	7386	51885
static-sink.c	Optimized	40927	196	7144	48267
sender.c	Original	44262	210	7652	52124
sender.c	Optimized	40861	196	7410	48467

with each DODAG.

Logically, the parents are strongly associated with the DODAG, so this was the best place for the data structure to be anchored. Mechanically, since each instance in Contiki's RPL implementation may have only one DODAG associated, it functions strongly as the list of parents for the instance.

Listing 4.3: Modified DODAG Structure (rpl.h)

```
/* Directed Acyclic Graph */
struct rpl_dag {
  uip_ipaddr_t dag_id;
  rpl_rank_t min_rank; /* should be reset per DAG iteration! */
  uint8_t version;
  uint8_t grounded;
  uint8_t preference;
  uint8_t used;
  /* live data for the DAG */
  uint8_t joined;
  rpl_parent_t *preferred_parent;
  rpl_rank_t rank;
  struct rpl_instance *instance;
  LIST_STRUCT(parents);
  rpl_prefix_t prefix_info;
/* BEGIN - MOROMI_IN */
  /* Moved the global rpl_parents to a DAG based structure
   *   so it can handle multi-Instance routing */
  nbr_table_t *rpl_parents;
```

```
/* END - MOROMI_IN */
};
typedef struct rpl_dag rpl_dag_t;
```

This change does not substantially increase the size of the DODAG structure as I only added in a pointer to the new **rpl_parents** table. The set of individual DODAG tables of known candidate parents was added to the top of the main RPL source, directly replacing the existing table. This necessitated adding a sufficient number of such tables for the maximum number of instances as Contiki has only limited dynamic memory allocation functionality.

The potential parent tables are managed by the neighbor table system, which provides support for up to a maximum of eight tables. Contiki uses 3 tables outside of RPL, which provides support for up to 5 instances to operate simultaneously.

Listing 4.4: Modified rpl_parents Tables (rpl-dag.c)

```
/* BEGIN - MOROMI_IN */
/* Adds two arrays to store the RPL parents information.
 *   References to this table are stored in each DAG (dag->rpl_parents)
 * This completely replaces the original global rpl_parents table because it did
 *   not support multi-instance networks
 */
static nbr_table_t rpl_parents_table[RPL_MAX_INSTANCES];
static rpl_parent_t rpl_parents_memory[RPL_MAX_INSTANCES][NBR_TABLE_MAX_NEIGHBORS];
/* END - MOROMI_IN */
```

The above code defines the maximum number of parents tables as the maximum number of RPL instances as set in the Makefile of the project. The **rpl_parents_table** definition is what contains the necessary data for the neighbor table functions to operate with this

data set. The definition for the **rpl_parents_memory** table creates the space to hold each of the potential parents and their metrics based on the instance.

One very useful aspect of the parents information is that it also provides a pointer to the DODAG in which it is stored. This was included in the original Contiki 2.7 specification for the structure and becomes very useful now that a specific parent entry is stored within its DODAG.

A particularly egregious problem that arose when modifying the code to use the new DODAG based tables was Contiki's innate assumption that there would only ever be one instance and only one DODAG in existence. Several functions to perform operations on parents carried no information on the instance the operation was performed on.

Listing 4.5: Example of a Function Lacking DODAG/RPL Context

```
rpl_rank_t
rpl_get_parent_rank(uip_lladdr_t *addr)
{
  rpl_parent_t *p = nbr_table_get_from_lladdr(rpl_parents, (rimeaddr_t *)addr);
  if(p != NULL) {
    return p->rank;
  } else {
    return 0;
  }
}
```

This above function is seldom used, however, its sole purpose is to provide the rank of the parent, specified by its IPv6 address, in an RPL network. Each node may belong to only one DODAG within an instance, however, it may belong to multiple instances; each instance will have a different rank for each parent as fits its own DODAG. This lack of context leads to continuations of the earlier problem noted with the design. Such functions highlight the

problem with simply expanding the **rpl_parents** table.

4.3.3 RPL Header Modifications

In addition to the changes necessitated by the expansion of the **rpl_tables** structure into each DODAG, there exists the problem that RPL header information is either lacking or completely incorrect.

No RPL information is ever added to a message on its initial send operation. Compounding the problem of addressing, when such header information is added at each intermediate routing node, the instance information is that of the routing node's default instance. These headers are added without using any context from the received message and are facially incorrect.

RPL is a routing protocol that provides a means for delivering messages under the following three means: multipoint-to-point, point-to-multipoint, and point-to-point. Multipoint-to-point messages comprise most of the RPL traffic and refers to some number of nodes sending information to a single sink device. This delivery is facilitated by each router examining the RPL headers of the incoming messages and sending the message to the router's local preferred parent of that instance. The messages are not guaranteed the best route at all times to the sink, but they continue to make progress until final delivery is made.

In the current code, as specified in Section 4.1.1 and shown in Figure 4.3, arbitrarily routes messages to the default instance of the intermediate router. To address these problems, two fundamental changes needed to be made. First, all upward-destined RPL messages need to be sent with accurate RPL instance headers. Second, all intermediate routers, when recognizing the destination is neither a direct neighbor nor along a downward route, must use the RPL header information to select the proper instance's preferred parent to send the message to.

The first change involved two fundamental steps. First, the outgoing messages need to include RPL headers if the message is being sent upwards. This is logically a straightforward determination insofar as all upward routed messages will be destined for a known RPL

instance sink.

Listing 4.6: Code to Ascertain if Destination is a Known Instance Sink (rpl-dag.c)

```
/* Added to check if the destination address is a known instance sink
 * Only called on initial send and on re-routing when no RPL information
 *   is in the message.
 * Returns:  Instance pointer if dest_addr is a known sink and NULL otherwise.
 */
rpl_instance_t *
rpl_is_addr_sink(uip_ipaddr_t *dest_addr) {
  uint8_t instance_index;
  rpl_instance_t *instance;
  for(instance_index = 0; instance_index < RPL_MAX_INSTANCES; ++instance_index) {
    instance = &instance_table[instance_index];
    // If the destination IPv6 address matches the DAG ID of the selected instance,
        return 1
    if(instance->used == 1 && uip_ipaddr_cmp(dest_addr, &instance->current_dag->dag_id
        ) != 0) {
      return instance;  // The dest_addr is a known sink, this is RPL routable.
    }
  }
  return NULL; // The dest_addr is not a known sink, this is not RPL routable.
}
```

This above function iterates through the table of instances looking for a DODAG ID that matches the destination address. Using this scheme for instance creation – selecting the DODAG ID to match the local sink IPv6 address – allows RPL addressing information to be sent with the DIO and stored as a member of each node.

Listing 4.7: Setting the RPL Header from the Destination Address (rpl-ext-header.c)

```
void
rpl_set_instance_from_dest(uip_ipaddr_t *dest)
{
  uint8_t i;
  int uip_ext_opt_offset = 2;
  int last_uip_ext_len = uip_ext_len;
  rpl_instance_t *instance = NULL;
  uip_ext_len = 0;

  /* Searches all known instances for a matching destination--DAG ID
   * DAG ID is normally set using the sink's ipaddress using rpl_set_root()*/
  for(i = 0; i < RPL_MAX_INSTANCES; i++) {
    if(uip_ip6addr_cmp(dest, &(instance_table[i].current_dag->dag_id)) != 0)
      instance = &instance_table[i];
  }

  /* We haven't seen such an instance yet, add the header, but don't route
   * With Instance -1 as nonexistent, all recepients will discard */
  if(instance == NULL) {
    UIP_EXT_HDR_OPT_RPL_BUF->instance = -1;
    UIP_EXT_HDR_OPT_RPL_BUF->senderrank = 0;
  }
  /* Destination DAG is associated with an instance, set it on message */
  else {
    UIP_EXT_HDR_OPT_RPL_BUF->instance = instance->instance_id;
    UIP_EXT_HDR_OPT_RPL_BUF->senderrank = instance->current_dag->rank;
  }
  uip_ext_len = last_uip_ext_len;
}
```

Once it is determined that the IPv6 destination for the message is a sink, then the RPL information may be added to facilitate proper routing to the destination. The below code was added to the uIP **`uip_process`** function to insert a proper RPL header only if the destination is a sink. If the destination is not a known sink or is a multicast address, then no headers are added and the message is routed normally.

Listing 4.8: Adding RPL Header to Outgoing Messages (uip6.c)

```
/* Add the header to the packet after the checksum is calculated
 * rpl_update_header_empty generates the proper header to add.
 * rpl_insert_header, which was used previously, only adds the header if the
 *  message already contains an OPT RPL extension, so it was not useful.
 * Only adds RPL information if the destination address is a known sink.
 */
#if UIP_CONF_IPV6_RPL
  if(!uip_is_addr_mcast(&UIP_IP_BUF->destipaddr) && (rpl_is_addr_sink(&UIP_IP_BUF->
      destipaddr) != NULL)) {
    rpl_update_header_empty();
    /* The empty header now needs the instance to add to the OPT, set it here */
    rpl_set_instance_from_dest(&UIP_IP_BUF->destipaddr);
  }
#endif
```

This only addresses half of the original problem. The other facet of this issue is that intermediate routing nodes, when receiving this message, would replace the RPL header on the message with their own default routing information. This had to be changed along the same lines as the above listing. If the message contains RPL headers, then those headers should be left alone. On the other hand, if there are no RPL headers and the message destination is an instance sink known by the intermediate router, then it should add RPL headers to facilitate proper routing.

Listing 4.9: Reading the RPL Header from a Received Message (rpl-ext-opt.c)

```
rpl_instance_t *
rpl_get_current_instance()
{
  int i;
  rpl_instance_t *instance = NULL;
```

```
  int uip_ext_opt_offset = 2;
  int last_uip_ext_len = uip_ext_len;
  uip_ext_len = 0;

  /* Searches all instances for the one matching the received packet. */
  for(i = 0; i < RPL_MAX_INSTANCES; i++) {
    if(instance_table[i].instance_id == UIP_EXT_HDR_OPT_RPL_BUF->instance &&
        instance_table[i].used != 0)
      instance = &instance_table[i];
  }

  uip_ext_len = last_uip_ext_len;
  return instance;
}
```

Listing 4.10: Adding RPL Headers In Transit (uip6.c)

```
    /* If the packet does not contain a RPL OPT header with a known instance and
     *   is not a multicast addressed message, then see if it is RPL capable,
          otherwise
     *   continue to use current packet options.
     *
     *   This previously always added an empty header using default instance
          information.
     */
    if(rpl_get_current_instance() == NULL && !uip_is_addr_mcast(&UIP_IP_BUF->
        destipaddr)) {
      /* If the destination address is a known RPL sink, then add a new header */
      if(rpl_is_addr_sink(&UIP_IP_BUF->destipaddr) != NULL) {
        rpl_update_header_empty();
      }
    }
```

At this point Contiki is now sending and receiving RPL information in the messages properly, however, routing is still arbitrary and based on non-relevant default instance information on each intermediate routing node.

4.3.4 RPL Routing Decision Modifications

The final set of major modifications to the RPL implementation in Contiki 2.7 was to the primary routing function, **`tcpip_ipv6_output`**. Despite having TCPIP in the name of the function and its source file, this is the primary function for all message routing. The primary purpose of this function is to ascertain the next hop to add to the message on transmission.

The original function performed four steps in attempting to ascertain the proper next hop for the message routing.

1. If the message is destined for the multicast address (**`ff02::1`**), then send it immediately without further route determination.

2. Else, If the message is destined for a known immediate neighbor, then set **`nexthop`** to that destination and send the message.

3. Else, If the message is destined for an address in the routing table (downward routing), then set the **`nexthop`** to the nexthop field of the destination's entry in the downward routing table and send the message.

4. Else, Set the **`nexthop`** for the default instance's preferred parent and send the message.

At no point in this process is any RPL information used for upward routing towards a known sink. Routing to a sink has only used the fallthrough default rule of using the default instance of the intermediate routing node. The following listing and enumerated list show the changes made by adding a new step immediately prior to the fallthrough to route based on the RPL information contained within the message itself.

1. If the message is destined for the multicast address (**ff02::1**), then send it immediately without further route determination.

2. Else, If the message is destined for a known immediate neighbor, then set **nexthop** to that destination and send the message.

3. Else, If the message is destined for an address in the routing table (downward routing), then set the **nexthop** to the nexthop field of the destination's entry in the downward routing table and send the message.

4. Else, If the message contains RPL information for a known instance, then set the **nexthop** to that specified instance's preferred parent and send the message.

5. Else, Set the **nexthop** for the default instance's preferred parent and send the message.

Listing 4.11: Added RPL Destination Check (tcpip.c)

```
      /* No route was found, check message for RPL addressing
        *  If RPL found, route to that instance's parent
        */
#if UIP_CONF_IPV6_RPL
      if(route == NULL) {
        rpl_instance_t *instance = rpl_get_current_instance();
        if(instance != NULL) {
          //nexthop = &instance->current_dag->preferred_parent;
          nexthop = rpl_get_parent_ipaddr(instance->current_dag->preferred_parent);
        }
      }
```

Following these changes the multisink test was rerun and all apparent routing issues were resolved. Each node sends one message every four seconds alternating the destination

between the two sinks. In eight second intervals, each sink reports successful message receipt from all nodes.

4.3.5 RPL Instance Whitelist Modification

As with any hierarchical design, the devices need to determinstically join only the instance of the next superior tier in the hierarchy, without any incidence of joining sibling or subordinate instances. RFC 6550 itself only defines base operations under a single instance implementation, leaving such decisions to those implementing the protocol. As Contiki only operated under a single instance prior to these modifications, it too offered no means for selectively joining instances.

The modification for this design involved only a minimal addition to the ICMP6 implentation for RPL.

Listing 4.12: rpl-icmp6.c

```
/* BEGIN - MOROMI_DIO */
#ifdef RPL_ALLOWED_INSTANCES
  if(strchr(RPL_ALLOWED_INSTANCES, (char)dio.instance_id) == NULL) {
    PRINTF("The instance '%c' (0x%x) is not on the allowed list. Rejecting.\n", dio.
        instance_id,dio.instance_id);
    return;
  }
  else
    PRINTF("The instance '%c' (0x%x) is on the allowed list.  Joining.\n", dio.
        instance_id,dio.instance_id);
#endif
/* END - MOROMI_DIO */
```

This small modification makes use of a user defined macro, **RPL_ALLOWED_INSTANCES**, which is set to an array of values indicating valid instances to join. In this implementation,

for ease of use, the instance IDs chosen correspond with the first letter of the sink roles: **M**, **O**, and **B**. The makefile for each of the devices is then modified to specify which instances the device may join. For example, the following listing shows the relevant line for the bridge device.

Listing 4.13: Bridge Makefile

```
CFLAGS+= -DRPL_ALLOWED_INSTANCES="\"OM\""
```

This listing specifies that the bridge is able to join instances with an ID of either 0x4F ('O') or 0x4D ('M'). Any other DIOs this device receives will be ignored at the beginning of the processing phase. If **RPL_ALLOWED_INSTANCES** is not set at compile time, then the default action of accepting all instances will be followed.

4.3.6 RPL Instance Lifetime Modification

The final major modification to the Contiki 2.7 operating system is implied by the details of the architecture to implement, while not directly presenting itself as a problem affecting multiple instance routing environments. For the implementation, when either the messenger or observer sinks arrive, the send their next DIO to the network. Once each bridge sink receives a DIO from the mobile sink, they add the information to their instance table, set up routing, and begin rebroadcasting the DIO information for bridges more distant from the mobile sink than they are.

The complication arises when the mobile sink either ceases transmissions or leaves operational range while headed to a new destination. Though the routing information will eventually decay, the instance itself will remain ever-present in the system. The bridges will continue to output a DIO, flooding the network with out of date information and confusing the routing. Hours after the mobile devices have all left, the bridges will continue to

send each other copies of the DIO announcing their ranks to the mobile sinks. The final modification to this RPL implementation is not described in RFC6550, but rather uses its description of routing lifetimes to elect to clear instances from devices once their lifetime has expired.

DIO messages in Contiki carry with them an additional, optional DIO Configuration header, which describes, among other fields, the default route lifetime and the unit that lifetime is measured in. The default lifetime for Contiki's implementation is 65535 lifetime units, each of which is 255 seconds long. This provides a default routing lifetime of approximately 193 days. Even with this extreme lifetime, Contiki only assesses lifetime for the neighbor routing table entries.

The first modification follows this by adding a new **`lifetime`** field to the instance structure to allow instance-level lifetime tracking to occur.

Listing 4.14: rpl.h

```
/* Instance */
struct rpl_instance {
  /* DAG configuration */
  rpl_metric_container_t mc;
  rpl_of_t *of;
  rpl_dag_t *current_dag;
  rpl_dag_t dag_table[RPL_MAX_DAG_PER_INSTANCE];
  /* The current default router - used for routing "upwards" */
  uip_ds6_defrt_t *def_route;
  uint8_t instance_id;
  uint8_t used;
  uint8_t dtsn_out;
  uint8_t mop;
  uint8_t dio_intdoubl;
  uint8_t dio_intmin;
  uint8_t dio_redundancy;
  uint8_t default_lifetime;
/* BEGIN - MOROMI_DIO */
  uint16_t lifetime;
```

```
/* END - MOROMI_DIO */
  uint8_t dio_intcurrent;
  uint8_t dio_send; /* for keeping track of which mode the timer is in */
  uint8_t dio_counter;
  rpl_rank_t max_rankinc;
  rpl_rank_t min_hoprankinc;
  uint16_t lifetime_unit; /* lifetime in seconds = l_u * d_l */
#if RPL_CONF_STATS
  uint16_t dio_totint;
  uint16_t dio_totsend;
  uint16_t dio_totrecv;
#endif /* RPL_CONF_STATS */
  clock_time_t dio_next_delay; /* delay for completion of dio interval */
  struct ctimer dio_timer;
  struct ctimer dao_timer;
/* BEGIN - MOROMI_DIO */
  struct ctimer instance_lifetime_timer;
/* END - MOROMI_DIO */
};
```

The second modification to this system was to add a new timer when a DIO is received. This timer is set for the proper number of lifetime in seconds, as specified by the received DIO and is specific to each instance. Once the timer expires, the specified instance decrements its current lifetime by one unit. If the lifetime reaches 0, the instance is freed and all timers pertaining to it, to include the DIO resend timer, are halted.

The final modification is to assess incoming DIOs for the same instance for a larger lifetime than the current instance features. A larger lifetime indicates the instance was refreshed by an origin DIO from the mobile sink – which always sends the full default lifetime – and the current node should increase its lifetime to match.

Listing 4.15: rpl-dag.c

```
/* BEGIN - MOROMI_DIO */
  if((dio->default_lifetime * dio->lifetime_unit) > instance->lifetime) {
    instance->lifetime = dio->default_lifetime * dio->lifetime_unit;
    rpl_reset_dio_timer(instance);
  }
/* END - MOROMI_DIO */
```

The tested result of this is when a mobile sink arrives within range of a bridge sink, the bridge accepts the first DIO it receives and propagates the information to the remaining bridge sinks. Network communications occur as detailed in the implementation from this point forward, with each bridge sink decrementing the lifetime of their joined instance.

As the mobile sink sends out new DIO messages, the bridge sinks quickly increase the lifetime of those instances to match and the instance is refreshed. When the sink is removed from the environment, each of the bridge sinks slowly degrades, uniformly across the network, until the instance lifetimes all reach 0. At this point, each bridge sink purges the instance information and frees the slot in the instance table for a new instance to arrive.

Initially, the function **`rpl_free_instance()`** was assessed for suitability, however, this function had the side-effect of nullifying the node IPv6 address. Following the instance purge, the node was no longer routable whatsoever. This was bypassed by directly clearing the instance information in the table by manually stopping the DIO, DIO, and Lifetime timers before setting the **`used`** flag to 0 and clearing the instance.

Following a purge, when the mobile sink returns and sends its next DIO out, the instance is rejoined and operations resume.

Chapter 5: Validation of Modifications and Architecture

The implementation of the HOIST architecture has involved the heavy modification of the operating system as a necessary prerequisite to support this architecture. This chapter is dedicated to the proper validation of the design and of the system modifications.

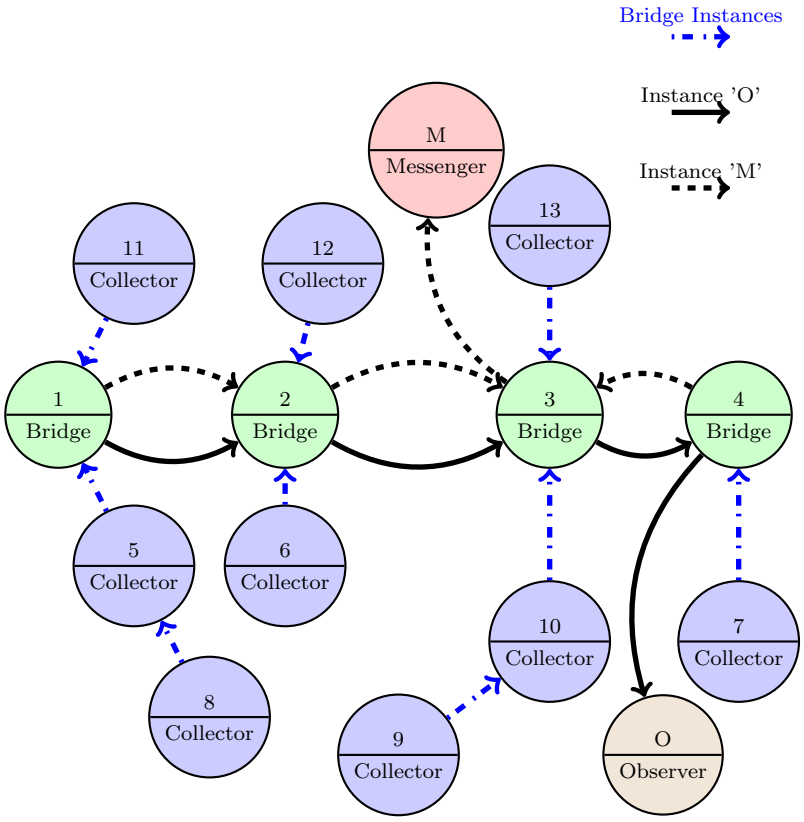


Figure 5.1: The Full Architecture

Figure 5.1 shows the intended implementation of the full designed architecture, which represents the final phase of verification.

5.1 Initial Validation

There are two primary aspects of the validation portion of this section. The first is flight verification testing. This aspect of the validation is primarily focused on the ability to successfully communicate with an unmanned aerial system that is orbiting about the WSN deployment zone.

The second aspect of validation is a brief breakdown of the architecture into five discrete phases, to highlight the various critical components necessary for proper validation. Each of these phases focuses on separate facets of the architecture for validation prior to the full evaluation of the design. This section concludes with a thorough run of the completed implementation.

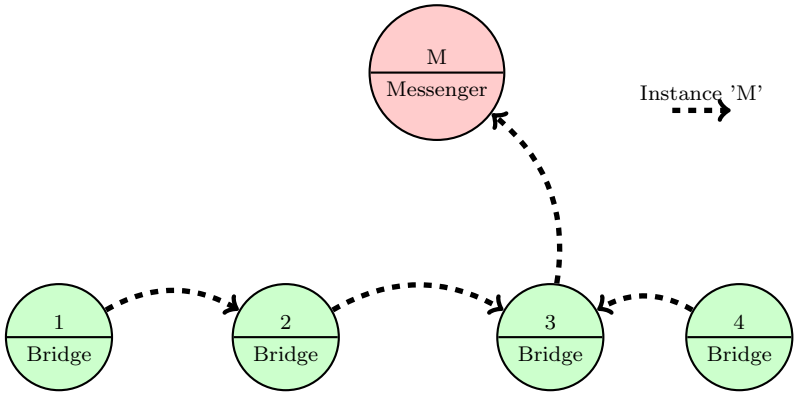


Figure 5.2: Implementation for the Unmanned Aerial System

5.1.1 Unmanned Aerial System Evaluation

I coordinated with a local Unmanned Aerial Systems (UAS) flight group, Aurora Flight Sciences, to assess the ultimate implementation of this work on a UAS platform. For agricultural implementations, UAS aircraft have a natural suitability to maneuver about the vegetation and their supporting structures to being used as effective collectors from embedded WSN devices.

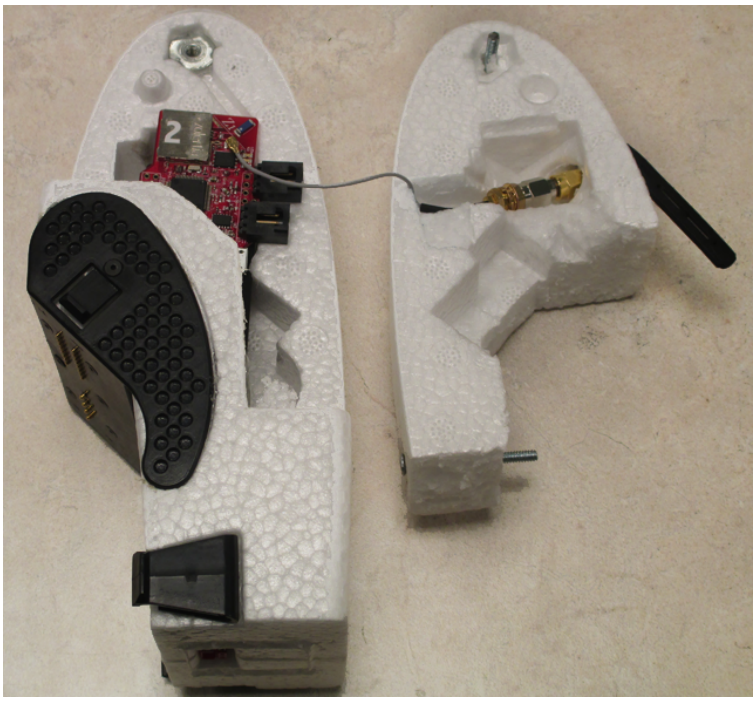


Figure 5.3: Payload Bay with WSN Device

On October 2, 2014, the Systems Laboratory met with Aurora Flight Sciences representatives to specifically assess the suitability of their Skate UAS model for use in the implementation of this architecture. At this point in the development cycle, phase one had been verified and evaluated in simulation and was selected as for the UAS test. I was provided in advance of the flight with a customizable payload pod, designed to carry up to 200g of equipment.

I modified the payload bay, as shown in figure 5.3, to carry 100g of payload, consisting of a Zolertia Z1 WSN device with battery pack, two AA batteries, and a Titanis 2.4GHz swivel antenna. I arranged the interior space such that while the bay is opened, both the reset and user buttons of the WSN device are easily accessible and the three LED emitters are clearly visible. While assembled and ready for flight, only the external antenna is accessible. As the device is powered from battery packs, testing procedures had to be devised under the assumption that the mobile sink would be fully powered and operational from the point of assembly, prior to flight.

A slot for the external antenna was cut into the nose of the airframe, providing it a stable location for flights. The swivel antenna was attached such that any position between vertical and rear-facing would result in a minimization of damage in the event of a hard landing. The WSN device began transmitting immediately upon power-on, prior to the closing of the payload bay. These transmissions occur very frequently in the beginning, then back off over time in accordance with the Trickle algorithm[52]. All transmissions took place on IEEE 802.15.4 channel 26, which is centered around 2480.0 MHz, with a 2MHz total bandwidth. External UAS downlink communications were configured to operate on IEEE 802.15.4 channel 1, which is centered at 2405.0 MHz, mitigating any possibility of RF interference.

Flight Test Preparations

I constructed the basic test to employ five motes, arranged in a linear pattern in a large, open field that would be free of obstructions or other sources of interference. Initially, for the first round of range testing, I configured two WSN devices, labeled as WSN **1** and WSN **2**, with the Titanis swivel antenna and loaded both with the default **radio-test** example program.

Figure 5.4: Control Table - WSN Devices along with the Payload Bay and Skate

The first test conducted was a basic range check to position the bridge device in the field beyond the range of the UAS mounted observer sink at the launch site. This test was added to ensure the bridge device would not receive messages from the control table prior to flight as the UAS device is always powered on. The expected range of the Z1 was approximately 50m of lateral distance using the Titanis antenna. Immediately, we noticed that the achieved range for bi-directional communications was greatly reduced, with communications failing at approximately 10m.

I elected to replace these devices with WSN devices **7** and **8** and repeat the experiment. At this point, we began receiving better signals, out to approximately 40m. The immediate analysis of the situation was the U.Fl connector on one of the first devices may have not been seated properly and the orientation of the antennas may not have been properly aligned. The Titanis swivel antennas are very sensitive to orientation, as ground testing would show.

Figure 5.5: Placement of WSN Device 7 in the Field

At this point, I attached bridge sink **7** to a post at an elevation of approximately 2.5m above the ground and planted it approximately 5m beyond the point of last signal reception, which was approximately 45m from the control table. The orientation of the

antenna was horizontal with respect to the ground, swiveled to face the control table. This was elected as the common antenna orientation for all ground devices. Figure 5.5 shows the final placement and inspection to ensure no communications were received. We used a cell phone based GPS unit to mark the geographic location of the device placement. This central device was positioned at **38.8300780N/77.807525W**.

Initial Range-Check Flight Test

Table 5.1: Initial Range-Check Flight Configuration Data

Mission	Altitude	Orbit	Payload	WSN Channel	Power	Temp	Wind
1	100ft AGL	20m	100g	26 (2480MHz)	0dBm	70 F	3-4 kts (10kts gust)
2	75ft AGL	20m	100g	26 (2480MHz)	0dBm	70 F	3-4 kts (10kts gust)

The first flight test employed a single device in the field and single mobile device mounted within the Skate UAS. The purpose of this first flight was to ascertain suitability for communications with the WSN devices and to determine the best altitude for communications.

The temperature was 70 degrees Fahrenheit on an overcast day with inclement weather in the distance. Wind speeds sustained approximately 3-4 kts, with gusts up to approximately 10 kts. I placed WSN observer sink **8** in the payload bay and attached the Titanis external antenna. We conducted a final check by inserting the batteries and verifying that neither device reported any activity (no blue or green LED lights emitted). Upon this verification, I sealed the payload bay and handed it off to the flight team. For this first test, the antenna was oriented horizontally and stowed partially within the nose section, facing the rear of the aircraft. The tip of the antenna was fully exposed, seated approximately 1/4" below the payload bay.

The flight team gave the permission for the operation and attached the payload bay to the Skate UAS. The UAS was brought around to the side of the control table and proceeded to start its flight checks. After a minor adjustment, the controller announced UAS startup.

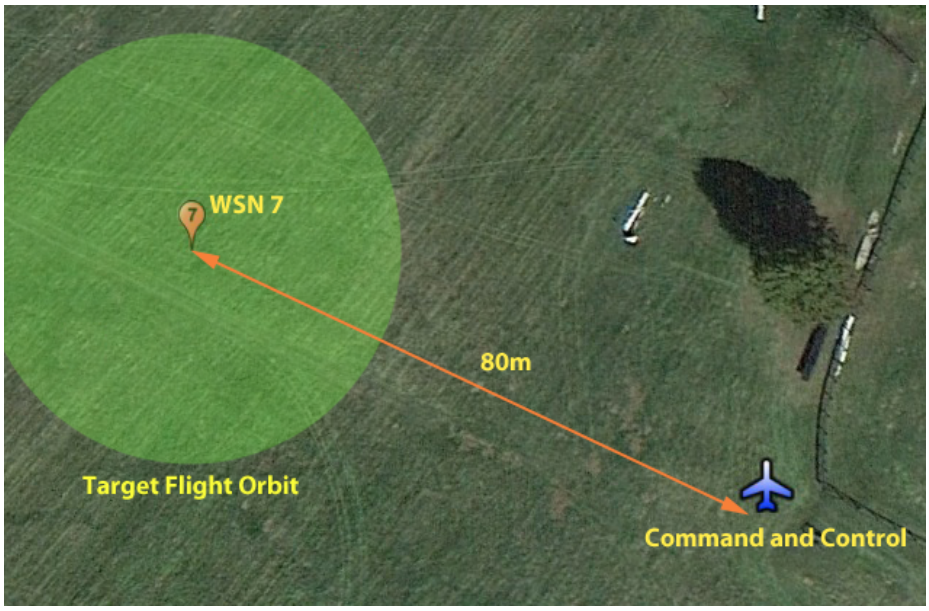


Figure 5.6: Planned Placement of WSN Device in the Field

After launch, the operator allowed the UAS to enter into a stable flight pattern before programming the orbit. The plan chosen used an altitude of 100ft AGL with a 20m orbit radius. The UAS entered into this orbit after several seconds, but had trouble maintaining a stable orbit due to the wind gusts. The UAS adjusted its flight to ensure it continued to center itself around the specified location, but the flight profile itself presented unpredictable antenna orientations. The initial orbit was also several meters north of the target device, which was expected from the precision of the cell phone positioning. The UAS operator adjusted the track to place the center over the device.

The bridge sink on the ground began receiving sporadic signals shortly after launch, but only began receiving relatively stable communications once the orbit was adjusted. In total, only approximately 60% of the transmissions at 100ft AGL were received, and these blocks of ideal communications came in bursts as the UAS presented a favorable angle before correcting to return for the next pass.

After several minutes of collections, I directed the UAS to reduce altitude to 75ft AGL to assess communications. Approximately 80% of the messages were delivered at this altitude,

(a) Flight Checks

(b) Launch of the UAS

Figure 5.7: Launch of the Skate UAS

though the orbit was slightly more stable and the UAS had already been adjusted over the location better.

The first flight provided the initial indication of suitability of the airframe for our research needs. Despite gusty conditions, the UAS persisted in its orbit and maintained approximately that 20m lateral radius from the device, at approximately the altitude selected. The 75ft AGL orbit provided significantly more reliable communications than did the 100ft AGL orbit, however, both featured the same pattern of bursts of reliable communications followed by moments of disruption.

The periodic communications pattern is not a large concern for our experiments as each device has a timeout system whereby it will attempt to retransmit the last packet until a successful reply is received. One observation made by the flight team was that the recessed antenna may be causing some trouble in our reception on the ground and it was recommended to repeat the test with the antenna extended.

At this stage, a second flight was requested to assess two more altitude profiles and a new antenna configuration.

Second Range-Check Test Flight

The temperature remained at 70 degrees Fahrenheit on an overcast day with inclement weather continuing to drift past in the distance. Wind speeds sustained approximately 4-5

Table 5.2: Second Range-Check Flight Configuration Data

Mission	Altitude	Orbit	Payload	WSN Channel	Power	Temp	Wind
1	120ft AGL	15m	100g	26 (2480MHz)	0dBm	70 F	4-5 kts (10kts gust)
2	80ft AGL	15m	100g	26 (2480MHz)	0dBm	70 F	4-5 kts (10kts gust)
3	60ft AGL	15m	100g	26 (2480MHz)	0dBm	70 F	4-5 kts (10kts gust)

kts, with gusts up to approximately 10 kts. I kept WSN device **8** in the payload bay with its Titanis external antenna. We conducted a final check by inserting the batteries and verifying that neither device reported any activity (no blue or green LED lights emitted). Upon this verification, I sealed the payload bay and handed it off to the flight team. For this second test, the antenna was oriented vertically at approximately 80 degrees incident to the payload bay, slightly swept towards the aft of the aircraft. We assessed that any hard landing would drive the antenna back into its original semi-housed orientation and reduce the chances for damage.

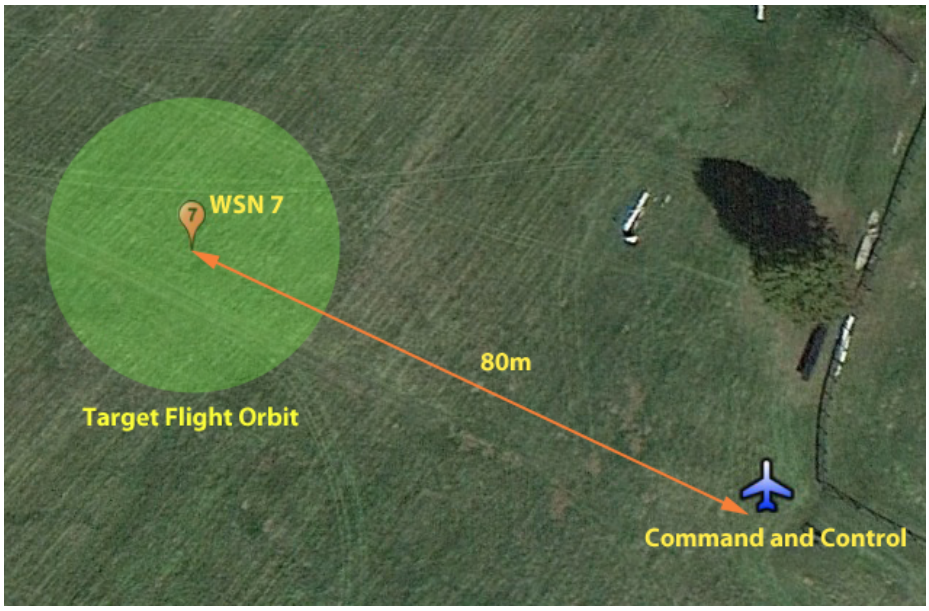


Figure 5.8: Placement of WSN Device in the Field

There was no servicing of either of the WSN devices prior to the second flight. The flight team gave the final permission for the operation and attached the payload bay to the Skate UAS. The UAS was brought around to the side of the control table and proceeded to start its flight checks. All tests passed and the controller announced UAS startup.

After launch the operator waited until the UAS achieved a stable flight pattern, then directed the aircraft over the target. The initial altitude selected was 120ft AGL with a tightened 15m radius orbit around the target. Once the UAS achieved this orbit, the ground communications device began to receive data, however, the rate of reception was very poor. After several minutes, we elected to drop the UAS to an altitude of 80ft AGL and we began receiving packets on a similar rate to the first test flight at 75ft AGL, which was approximately 80%.

I directed the the altitude to be lowered to 60ft AGL for several minutes and noted a much stronger communications pattern, as anticipated. With the antenna extended downwards and orbiting at 60ft AGL, there were only intermittent drops in communications. The UAS had some difficulty maintaining a stable flight pattern in the tighter orbit, but it spent more of its time around the target, providing a very strong communications pattern.

The second flight demonstrated that the UAS was viable at a lower altitude and at a tighter orbit. I elected to use 60ft AGL and a 15m orbit for the second test. The extended antenna seemed to provide longer bursts of communications between the UAS and ground station. We elected to use this antenna position for the first flight for the second test. The orbit seemed more erratic at this tighter radius, however, our communications improved with the longer time over target, so it was suitable for continued operations.

First Test Flight of Phase One Implementation

Table 5.3: Initial Phase One Implementation Test Flight Configuration Data

Flight Segment	Altitude	Orbit	Payload	WSN Channel	Power	Temp	Wind
1	60ft AGL	15m	100g	26 (2480MHz)	0dBm	70 F	1-2 kts

For this third flight, the temperature remained at 70 degrees Fahrenheit on a clear day, with some overcast conditions in the distance. Wind speeds dropped to approximately 1-2 kts, reducing throughout the duration of the flight. I kept WSN device **8** in the payload bay and attached the Titanis external antenna, though I reloaded this device with the **static_sink** program, which was the first iteration of the observer sink.

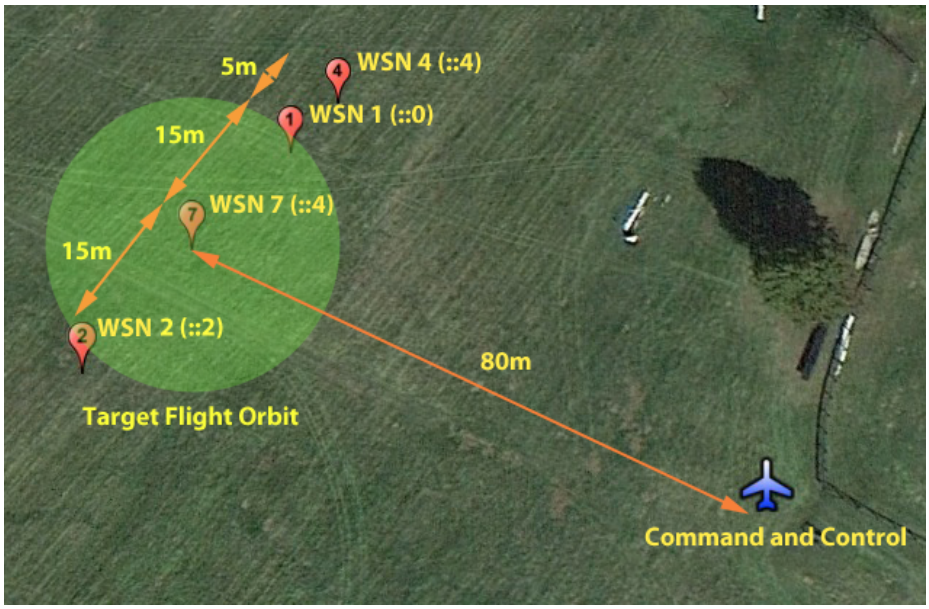


Figure 5.9: Placement of WSN Devices in the Field

I planned to deploy four additional WSN devices to create a linear topology network of five ground communications platforms. The deployed ground device location from the previous test was marked and that device was brought in for reprogramming. I loaded all five ground devices with the **sender** program, which was the first iteration of the bridge sink, and performed a function check at the table.

After loading the devices, I conducted a test of the network and discovered that WSN device **4** was unable to receive any replies. I checked the configuration of the device and saw it had been erroneously given an improper IPv6 address which did not match its Node ID value. I reprogrammed the device to return to its desired configuration, with the final

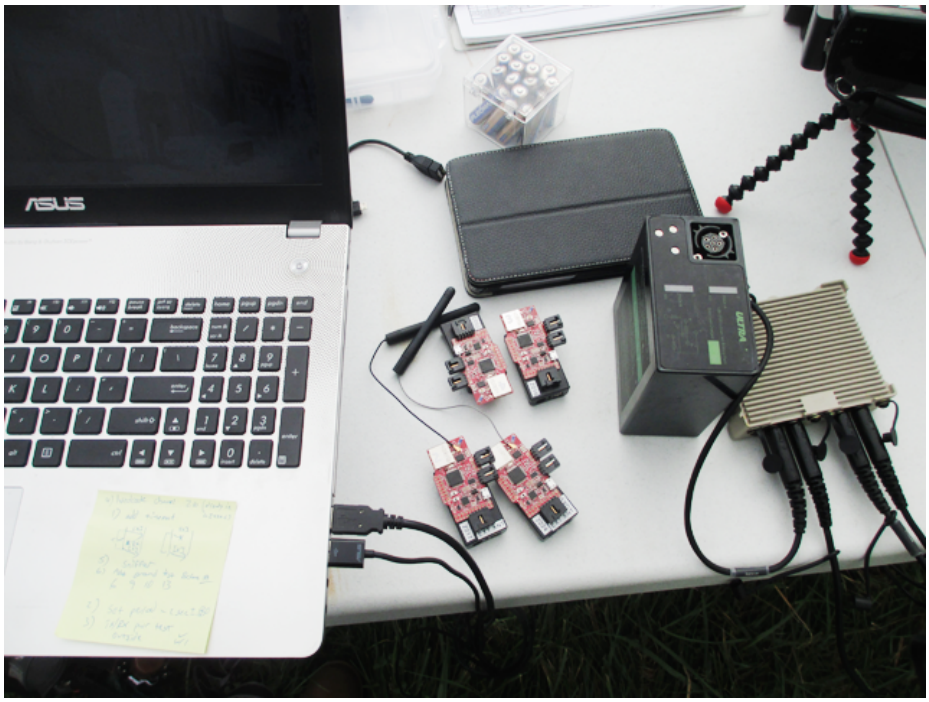


Figure 5.10: Loading the Devices for Test 2

octet of the Node ID and its MAC (and therefor IPv6) addresses to 4. Unfortunately, I did not account for the earlier substitution made in the beginning of the day, which had used WSN devices **7** and **8** in lieu of the planned devices. At this point, I now had two devices on the network with an IPv6 address ending in ::4. The repeat of the test passed because all five devices were able to directly communicate with each other on the same table, so when the aerial communications device replied to either of the ::4 devices, both acknowledged them. This problem with the networking would only affect multi-hop routing, which occurred during the actual test.

At this point, I moved the devices to the field in accordance with the above mapping of the placement. All four of the deployed ground devices were powered on and were reset at the deployment site.

I conducted a final check by inserting the batteries to the airborne WSN device and verifying that none of the deployed ground devices reported any activity (no blue or green

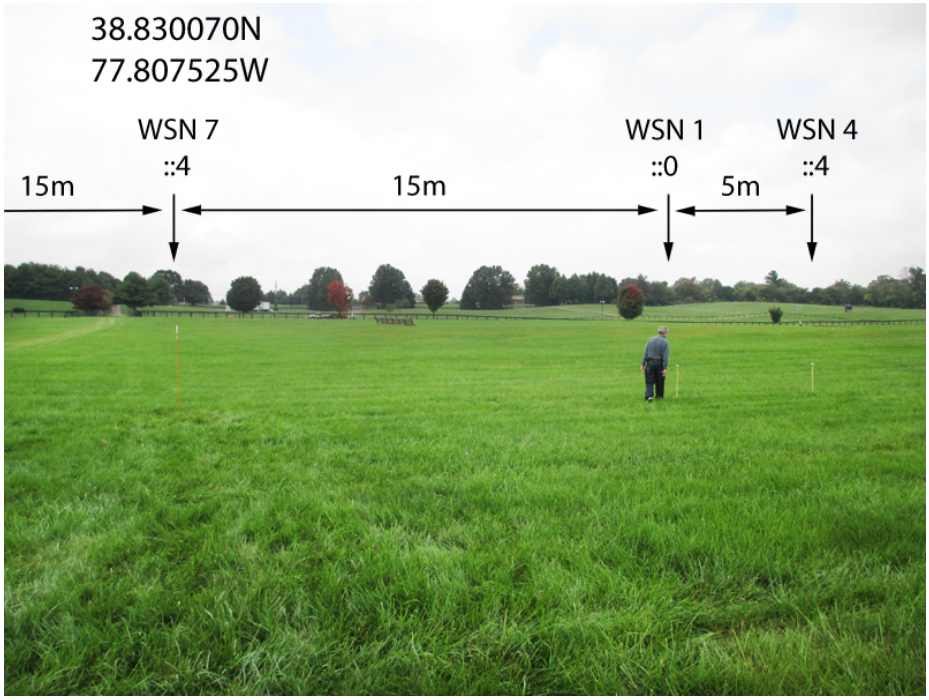


Figure 5.11: Placement of WSN Devices in the Field for Test 2

LED lights emitted). Upon this verification, I sealed the payload bay and handed it off to the flight team. For this test, the antenna was planned to be oriented vertically, however, the deployment was done with the antenna in the seated position, aft-facing and seated in the payload bay. The flight crew finished the preflight checks and launched the UAS from the control station. After achieving stable flight, the controller directed the UAS to orbit over WSN bridge sink **7**, which was located in the same spot as it was in the prior tests.

The flight proceeded normally for approximately the first four minutes of the test, however, nearing the end of the duration, the airframe lost altitude and began orbiting only a few feet off of the ground. The meaningful data collection had already ended at this point in the flight, so I directed the UAS to land for recovery. Following a rather abrupt landing, the flight team did a rapid assessment of the airframe and ascertained, and corrected, the problem immediately, preparing for further flights. As I had enough data from the flight, I elected to conclude operations for the day.

Upon opening the payload bay following this flight, I noted that the U.Fl connector that

attached the Titanis swivel antenna with the WSN device was disconnected. This likely occurred during the hard landing and was not likely separated during the flight. I noted that this connector should be secured with an adhesive to the device prior to any future flights.

WSN Network Analysis

In analyzing the network performance, all times were adjusted to start from the moment the UAS was directed to begin orbiting the target site. The airframe needed approximately 8 seconds from the input of this command until it was over the target location, due to the location of its initial orbit and its position in that orbit.

In addition to the visual logging through the debug LEDs, I also used a TMote Sky device loaded with an 802.15.4 sniffer program and connected to an Asus Nexus 7 tablet to perform packet analysis and live collection. I started collection approximately 8 minutes prior to the beginning of the flight, however, the device stopped recording after 10 and a half minutes, which left most of the flight unrecorded. This is likely due to a minor shifting of the device in the grass, disconnecting it from the tablet, which stopped the recording.

Table 5.4: Phase One Implementation UAS Test Data

WSN Device	IPv6	Position	DIO Time	RX Time	Completion
8	::3	UAV over WSN 7	N/A	N/A	N/A
7	::4	38.830070N/77.807525W	+19 sec	+32 sec	+172 sec
2	::2	15m SW of WSN 7	+22 sec	+33 sec	+ 179 sec
1	::0	15m NE of WSN 7	+11 sec	-	-
4	::4	5m NE of WSN 1	-	-	-

Initial DIS Configuration

The sniffer recorded 9 minutes and 4 seconds of communications between the four ground devices. These were all Destination Oriented Directed Acyclic Graph Information Solicitation (DIS) messages. DIS communicates occur very frequently on initial network startup.

These messages serve to announce the identity of the sender to all other WSN devices in range; this is the level of communications which build the adjacency neighbor lists and are used to establish signal quality information about each neighbor.

The DIO Received time in this log represents the average time between DIO reception and SYN transmission. These are recorded times made by observation and a simple wrist-watch. All times listed in italics are values not taken from the packet analyzer and indicate a general lack of precision. Also of note is that these mark the first transmissions detected by the RF sniffing device and may not represent the first packets transmitted.

Initial DIO Reception and Retransmission

Table 5.5: DIO Reception

Time (sec)	WSN	IPv6 Address	Event	Selected Parent
12	1	::0	DIO Received	8
13.416	1	::0	SYN Transmitted	8
14.092	1	::0	DIO Retransmitted	8
18	7	::4	DIO Received	8
19.104	7	::4	SYN Transmitted	8
21.549	7	::4	DIO Retransmitted	8
23	2	::2	DIO Received	7
24	2	::2	SYN Transmitted	7
25	-	-	END OF LOG	-

The network formed beginning with WSN **1** receiving the first DIO packets from WSN **8** in the UAS. WSN **7** then received a DIO directly from WSN **8** and began retransmitting the DIO to its neighbors. WSN **2** received the DIO from WSN 7 and added it as the parent, forming a perfect multi-hop routing tree up to WSN **8**. Unfortunately, at this point, the logging of the 802.15.4 sniffer failed and all packet logging ceased.

The best we were able to ascertain at this point was that WSN **1** changed its parent to WSN **7** after the UAS entered its stable orbit. WSN **4** received a DIO, however, once it attempted to associate with WSN **7**, it would have been immediately rejected and the

entire branch of the DAG would have been nullified as both devices had the same IPv6 address; WSN 7 would have seen WSN **4** as a loop. Messages destined for WSN **4** or WSN **7** would have been received by both, however, further routing would have been problematic due to the IPv6 address collision. While it is not possible to ascertain the precise nature of the results, the IP collision prevented proper responses to return to WSN **4** and WSN **1**, preventing their timely completion of the cycle.

Test Completion

Table 5.6: Test Completion Times

Completion Time (sec)	WSN	IPv6 Address
79	Ideal	Ideal
172	7	::4
179	2	::2

As downward routing was likely compromised by the IPv6 collision, only two of the devices maintained proper communications during the entire run, WSN **7** and WSN **2**. All of the initial communications from WSN **2** routed through WSN **7** and both finished their sequences with similar offsets from each other, implying they were both sharing the same communication link with the UAS; WSN **7**. Both finished within the expected range of 2-3 minutes of time that was seen in ground testing with occasional communications disruptions.

The packet send rate for each of the devices is one packet per every 2 seconds. With perfect communications, WSN **7** would have finished in 79 seconds from the initial dispatching of the UAS to the target location. Each missed packet due to communications disruption results in a 4 second timeout prior to a resend attempt. Several such disruptions occurred over the datalink between WSN **7** and WSN **8** in the UAS, which is precisely the behavior observed during the two earlier flights. The upper-bound of acceptability for timing on this run was 5 minutes, which was easily met; the two logically severed devices notwithstanding.

Assessment of the Flight Test

There were three deviations from the expected test profile, two of which affected communications. The first deviation was the precipitous drop in altitude at the end of the test, which did not affect the results of the experiment. The second deviation was the orientation of the Titanis swivel antenna on the UAS communications device, which ostensibly contributed to the minor attenuation of the emitted signal. This may have slightly extended the run time of each test, however, the test concluded within the expected timeframe. The third deviation was a duplication of an IPv6 address in the network, which led to the destruction of one branch of the linear formation.

The test concluded in the timeframe and manner as expected for the two devices which were unaffected by the routing issues, making the test a successful implementation of phase one of the designed architecture.

5.1.2 Initial Assessment

The objectives for phase one were successfully met in both simulation runs and during a physical implementation involving a UAS. In the presence of multiple sink devices, the bridge sinks all formed themselves into a DODAG with the mobile sink as its root. None of the bridge sinks attached themselves to any other bridge sink instance, validating the modification to enable instance whitelisting. Furthermore, each of the bridge sinks received replies back as demonstrated through the UAS test, validating the downward routing table of the mobile sink.

5.2 Validation Phases

This section presents several intermediate phases for the validation of the HOIST implementation. Each phase examines a different aspect of HOIST, in order to present a set of metrics necessary for the proper validation of that portion of the architectural implementation.

5.2.1 Phase One: Bridge and Messenger

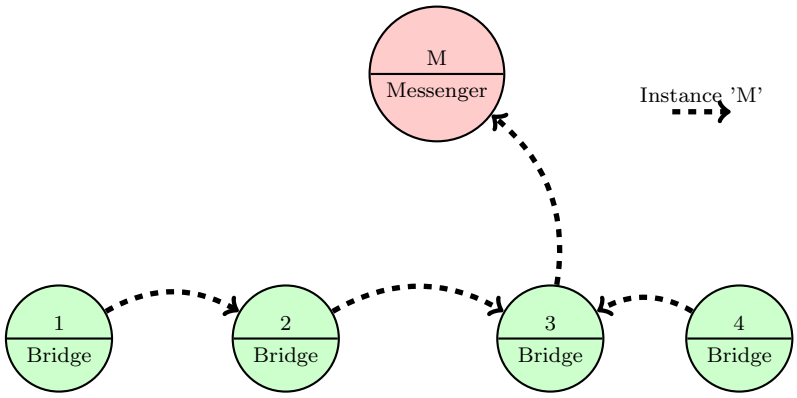


Figure 5.12: Phase One Validation

The first set of metrics of the implementation to consider involve the coordination between the bridge and the mobile devices. The following subsections describe the categories of metrics under this phase which must be met before HOIST may be considered as valid.

Selective Instance Joining

Neither the messenger, nor the observer, may join a bridge instance. This is a fundamental tenet of enabling a working hierarchical design. Following with this logic, the bridge sinks must also not join an instance run by another bridge. The method for controlling this is the whitelist modification, which was described in chapter 4.3.5.

Of note, the bridges will appear in the downward routing tables of other bridges. This is an unfortunate side-effect of the RPL routing process. As each bridge receives a DIO from a legal instance, it will increase the rank to reflect its own position and then rebroadcast that DIO out. If this rebroadcast DIO is received by a bridge farther from the mobile sink, it will do the same, increase the rank, and rebroadcast the DIO. The first bridge will receive this DIO and see the bridge as a downward neighbor, as it has a higher rank. This causes the situation in which one bridge will have a routing table entry for another. For this reason, a

timeout and retry attempt limiting mechanism is included in HOIST, enabling a bridge to attempt to send controls to another bridge, time out, then move on to a valid device.

Downward Routing

The data transmitted by the bridge sinks must be acknowledged through a reply by the messenger sink. This means all of the bridge sinks must be present in the downward routing table of each of the mobile sinks. As each packet of data is sent to the messenger, a reply is generated and sent back to the bridge, for ultimate retransmission to the originating collector. These downward routing tables use the Contiki default for the Z1 device of 15 entries.

5.2.2 Phase Two: Collector and Bridge

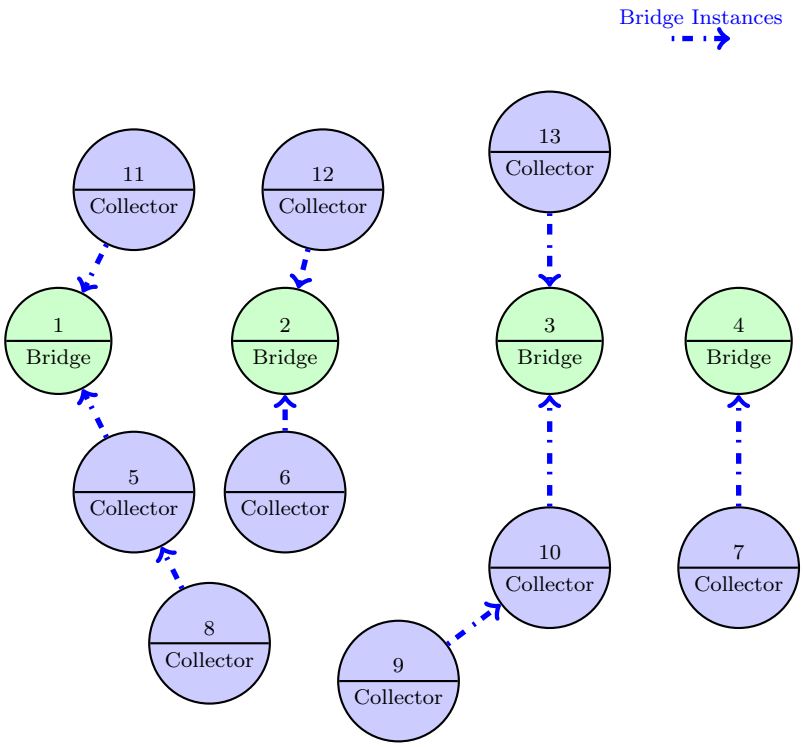


Figure 5.13: Phase Two

The second set of metrics of the implementation validate the proper star topology-based clustering of the collectors to associate with nearby bridge sinks.

Single Instance Joining

Each collector must join only a single bridge sink's instance. The collector must only route packets to this local bridge. HOIST implements this by having each of the bridges operating on a different instance ID, which is based on the last octet of the bridge's IPv6 address. The reason RPL will not allow each of these bridges to broadcast the same instance ID is that multiple bridges will receive and add that collector to their downward routing tables. While the collector will only have one preferred parent, ensuring a clean upward route to the messenger or observer sinks, any replies would be duplicated by each bridge containing an entry for that collector, increasing the traffic on the network. Furthermore, as the table entries are severely constrained, each occurrence of this event further reduces the amount of collectors each bridge can support.

5.2.3 Phase Three: Collector, Bridge, and Observer

The third set of metrics of the implementation validate the proper hierarchical communication between the tier one collectors, the tier two bridge sinks, and a tier three observer sink. When the observer enters the area and sends its first DIO message, the bridges need to rapidly form a DODAG to the observer and send messages to each of their children collector nodes to begin observer data transmissions.

Hierarchical Instances

At least one bridge must receive a DIO directly from the observer and join the **O** instance. The directly connected bridges must then propagate DIO messages out until all bridges are members of the **O** instance and a DODAG is formed with the observer as the root. No collector device may join the **O** instance. Bridge devices must then signal each of their collectors to begin transmitting data. This data is to begin immediately by invoking a new

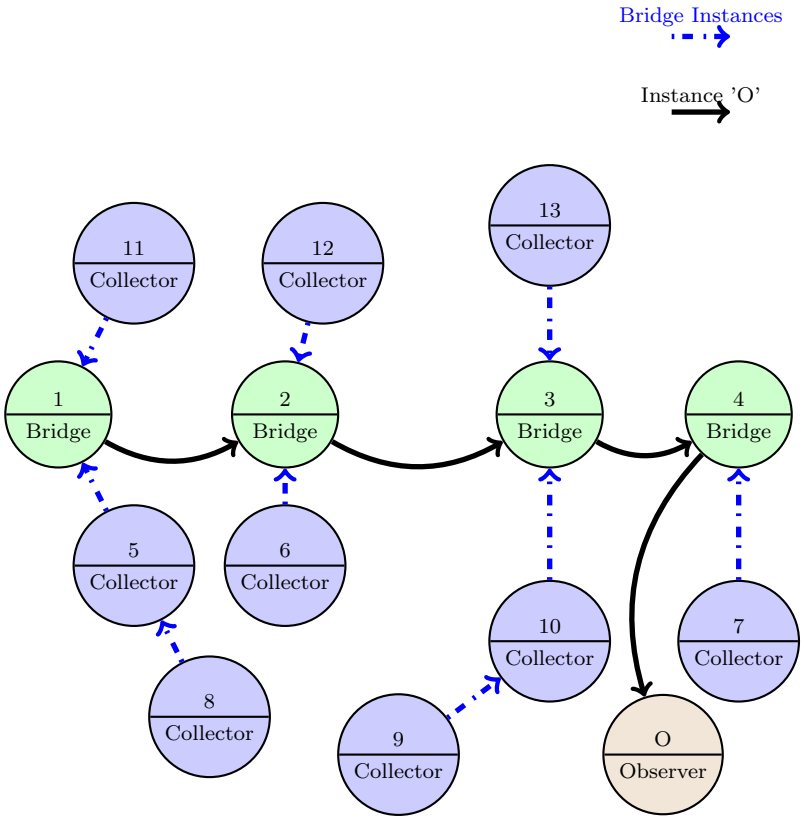


Figure 5.14: Phase Three

collection start event. Once the collector has acknowledged the event, the bridge sink then continues by transmitting the start message to the next collector in its downward routing table, until no more remain. Any sibling devices (other bridge sinks) in the table should bring about a similar transmission, but no reply should be received. After the maximum number of retry attempts, the bridge shall move to the next device in the downward routing table.

Each collector must immediately schedule a new collection event and send the collected data to their local bridge. Upon receipt, the bridge will forward that data to the observer. At this point, as long as the collector is collecting data, an transmission must be generated to send data upwards.

Instance Lifetime

When the observer device leaves range, the ⭘ instance must uniformly degrade in lifetime until all bridges reach a lifetime of 0, at which point they must all purge the instance. Any bridges with an outlying lifetime that purges its instance early must rejoin the instance on the next live DIO and resume operations. This implementation is described in chapter 4.3.6. The values governing the DIO send rate and the instance lifetimes will determine the promptness of the lifetime degradation, the efficacy of the distributed lifetime coordination, and whether or not an instance will prematurely decay. As a rule, the lifetime of a network must be at least as great as the maximum DIO send rate of the corresponding instance.

5.2.4 Phase Four: Collector, Bridge, and Messenger

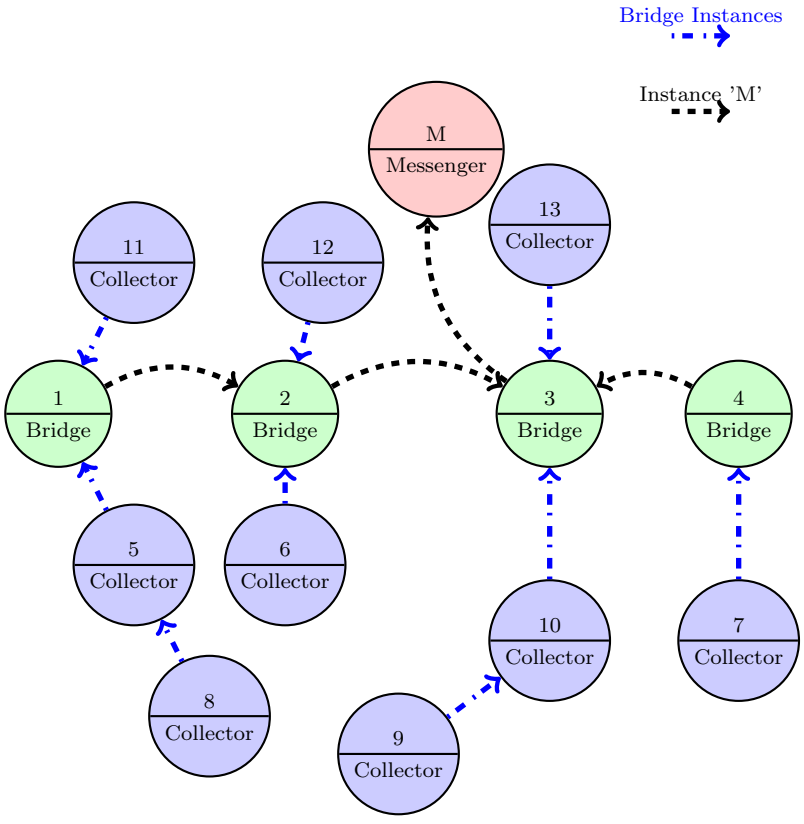


Figure 5.15: Phase Four

The fourth set of metrics of the implementation validate the proper hierarchical communication between the tier one collectors, the tier two bridge sinks, and a tier three messenger sink. When the messenger enters the area and sends its first DIO message, the bridges need to rapidly form a DODAG to the messenger and send messages to each of their first children collector nodes to begin messenger data transmissions.

Hierarchical Downward Routing

The bridges must first form their local networks of collectors and have their downward routing tables populated by the full listing of the collectors under their coordination. At least one bridge must receive a DIO directly from the observer and join the **M** instance. The directly connected bridges must then propagate DIO messages out until all bridges are members of the **M** instance and a DODAG is formed with the observer as the root. No collector device may join the **M** instance. Following from the previous validation phase, once **M** receives a packet containing a dump of the data, a reply must be sent to the appropriate bridge and then relayed to the originating collector of the data. This is implemented throughout chapter 4.3, representing the main portion of this work.

Multiple Events per Packet

When data transmission is started, each collector will suspend new data collection and observer transmissions and send a number of data samples not to exceed the defined maximum number per packet to the bridge. On successful reception, the collector will send the next batch of data. On a timeout event, the collector must transmit the same block of data. This re-transmission will continue until either a proper acknowledgment is received, or the bridge sends a termination event. After a termination and re-establishment, the collector must begin transmitting the same data until acknowledged. Once all data has been transmitted, the collectors must signal the bridge, which will then alert the next collector to begin data transmissions. This process shall continue until all collectors have transferred their data. Once each collector has finished its transmission, they will resume collection

operations.

5.2.5 Phase Five: Collector, Bridge, Observer, and Messenger

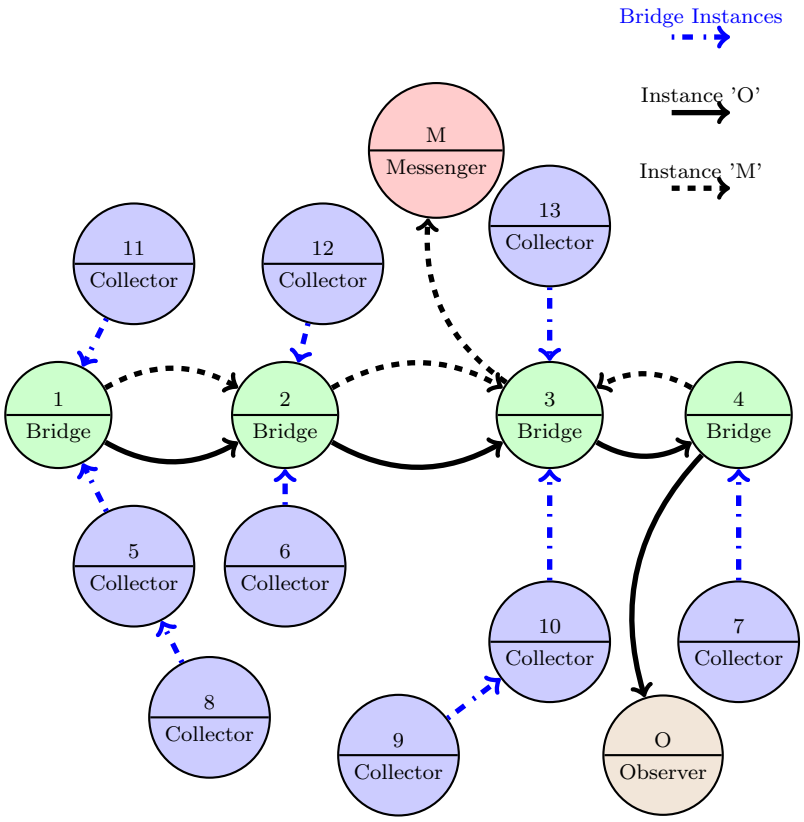


Figure 5.16: Phase Five

The final set of metrics considers the full implementation as a whole. The key to this phase is the operation of the network in the presence of both the messenger and observer sinks. The following components are critical for this full implementation:

Observer Suspension

When the observer arrives, the network should form and collectors must transfer the data, as it is collected, to the observer sink. Upon arrival of the messenger, the bridges must signal

the collectors to begin data transmission to the messenger, one at a time. The selected collector must suspend its collection operations during the data dump and, with it, suspend all communications with the observer. Once either the full data transfer has completed, or the messenger instance has been terminated as the result of lifetime expiration, the collector will resume collection operations and resume transmissions with the observer.

Observer coordination

When the observer is active and in the transmission range, all collectors not currently transmitting data to the messenger will continue collecting and transmitting to the observer. When the observer is removed through lifetime termination, sending to the observer must cease. This action must occur with or without the presence of the messenger.

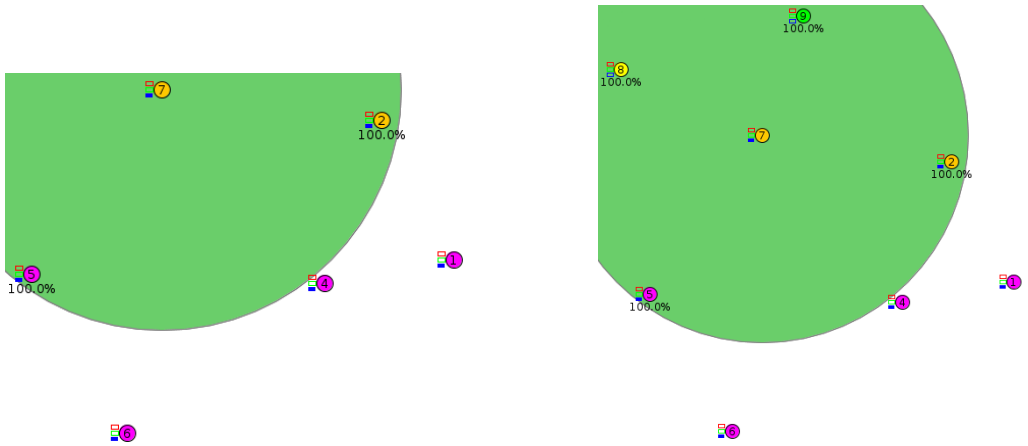
5.3 Validation through Simulation

The previous section described the myriad aspects of this implementation necessary to consider the architecture to be implemented as designed. This section presents the analysis of a sample implementation designed to assess the aforementioned aspects of proper execution. This validation was run using the Cooja simulator and assessed through internal logging over the serial ports of the devices and through Wireshark.

5.3.1 Validation Setup

The initial configuration of the network is displayed in figure 5.17a. In this figure the green circle represents the range on each of the devices, with the range for device **7** depicted. The purple devices labeled **1**, **4**, **5**, and **6** are collector nodes. The orange devices **7** and **2** are bridge sinks. The yellow device **8**, shown in figure 5.17b is the messenger sink and the green device **9** is the observer sink.

The initial configuration features four collectors and two bridges, without any of the mobile sinks present. Collectors **4** and **1** are each in direct communications range of bridge



(a) Initial Deployment (b) Messenger and Observer in Range

Figure 5.17: Cooja Simulation

2. Collector **5** is in direct communications range of bridge **7** and collector **6** is in direct communications range of collector **5**. This node was placed as such to demonstrate hierarchical, multi-hop networking during this validation.

At the indicated time periods in the next section, the observer and messenger devices will enter and leave. The positions they take when they enter the network are displayed in figure 5.17b.

5.3.2 Mobility Definitions

I conducted this validation in Cooja using the Mobility extension to place and move the nodes at precise time intervals. The only mobile nodes are the observer and the messenger. The position file used is as follows.

Listing 5.1: positions.dat

```
# Node 1
0 0.0 62 55
# Node 2
1 0.0 47 26
```

```
# Node 4
2 0.0 35 60
# Node 5
3 0.0 -26 58
# Node 6
4 0.0 -6 91
# Node 7
5 0.0 -1 21
# Node 8
6 0.0 -231 86
# Node 9
7 0.0 -225 -33
##### MOBILE #####
# Node 9
7 30.0 5 -19
# Node 8
6 90.0 -40 5
6 300.0 -231 86
7 330.0 -225 -33
6 540.0 -40 5
```

The positions.dat file uses a space separated, four column format. The first column specifies the logical node number in the simulation, beginning with the first created node as 0. Appropriate comments are included in the file to provide the translation between logical node number and the device number in cooja.

The second column is the time at which the device will enter the specified position, given in seconds. The third and fourth columns represent the cartesian coordinates of the devices in the simulation plane, with x increasing to the right and y increasing to the bottom.

This file begins the simulation with the two bridges and four collectors separated from the mobile sinks. At the 30 second mark, observer **9** is moved to a position within transmission range of bridge **7**. This forces bridge **2** to form a multi-hop DODAG to **9** through a peer bridge. At the 90 second mark, the messenger **8** moves to a similar position, again with

transmission range to only **7**. At 5 minutes into the simulation, messenger **8** is removed from the environment. 30 seconds later, observer **7** is removed. Finally, at the 9 minute mark of the simulation, the Messenger is returned to the deployment area.

5.3.3 Validation of Implementation

At the beginning of the simulation, both bridges began sending out DIO messages.

Table 5.7: Validation Initial DIO Messages

Time Sent	Bridge	Transmitter	Time Received	Receiver	Preferred Parent
00:04.385	2	2	00:04.496	1	2
00:04.385	2	2	00:04.512	4	2
00:04.399	7	7	00:04.710	5	7
00:08.619	7	5	00:08.702	6	5
00:12.611	7	6	N/A	N/A	N/A

Table 5.7 shows that bridge sinks **2** and **7** began transmitting a DIO to announce their instances within 5 seconds of boot. The three collectors that are in direct transmission range received these messages and joined their instances within 350ms. Collector **6** is two hops away from the originator and had to wait for collector **5** to retransmit a copy of the DIO before it could join. By the 9 second mark, all four collectors had joined the instance.

At this point, the DODAGs have been properly formed and multipoint-to-point routing is possible from collectors to their respective bridges. Downward routing, on the other hand, is not yet possible. To enable downward routing, each of the collectors must send a DAO message up to their respective bridges, to inform all devices above them of their IPv6 address and the next hop through which they may be reached.

By the 20 second mark after boot of the devices, all devices have full tables of their subordinate nodes and the network is capable of handling point-to-point and multipoint-to-point routing. This is critical to occur before the arrival of the mobile devices as this is the mechanism through which replies are routed. At this point, the deployment area is

Table 5.8: Initial DAO Messages

Time	Receiving Device	Destination	Next Hop
00:11.959	5	6	6
00:14.384	2	4	4
00:15.508	7	5	5
00:17.276	2	1	1
00:19.248	7	6	5

fully functioning and is capable of supporting either of the mobile sinks.

At this point in the simulation, each collector began collecting data in 15 second intervals. Table 5.9 shows the initial collection times for each collector, leading up to the arrival of the messenger and the cessation of collection. One item of note is that the fifth collection for each device occurs only a couple of seconds after the fourth collection. This occurrence coincided with the arrival of the observer **O**, which triggered an immediate collection and send by all devices.

Table 5.9: Initial Data Collection Prior to Messenger

Collector	1	2	3	4	5	6
5	00:16.200	00:31.208	00:46.216	01:01.224	01:03.232	01:18.296
1	00:16.234	00:31.242	00:46.250	01:01.291	01:07.886	01:22.890
6	00:16.567	00:31.637	00:46.646	01:01.653	01:03.552	01:18.871
4	00:16.751	00:31.759	00:46.767	01:01.775	01:08.393	01:23.400
Collector	7	8	9	10		
5	01:33.248	01:48.255	02:03.263	02:18.271		
1	01:37.898	01:52.906				
6	01:33.567	01:48.575	02:03.583	02:18.591		
4	01:38.408	01:53.415	02:08.423			

Table 5.10 shows the first several transmissions following the arrival of the observer **9** at the 30 second mark of the simulation. The first DIO was sent by **9** at 00:57.248, which was its next scheduled send time. This DIO was received by bridge **7** at 00:57.312 and by

Table 5.10: Data Transmissions Following Observer Arrival

From	Type [Data]	From	To	TX Time	RX Time	From	To	TX Time	RX Time
7	**OBS_START**	7	5	01:01.319	01:01.455	5	7	01:01.710	01:01.817
7	**OBS_START**	7	6	01:01.827	01:02.195	6	7	01:02.444	01:02.672
5	**OBS_DATA [5]**	5	7	01:03.300	01:03.419	7	9	01:03.442	01:03.550
6	**OBS_DATA [5]**	6	7	01:03.611	01:03.902	7	9	01:03.924	01:03.425
2	**OBS_START**	2	1	01:06.384	01:06.487	1	2	01:06.736	01:06.875
2	**OBS_START**	2	4	01:16.885	01:07.005	4	2	01:07.254	01:07.371
1	**OBS_DATA [5]**	1	2	01:07.945	01:08.365	2	9	01:08.388	01:08.371
4	**OBS_DATA [5]**	4	2	01:08.451	01:08.598	2	9	01:08.621	01:08.765
5	**OBS_DATA [6]**	5	7	01:18.309	01:18.386	7	9	01:18.409	01:18.740

bridge **2** at 01:01.489 (after being rebroadcast by bridge **7**.

The packet transmission design for this architecture after reception of the first DIO from an observer is shown below, in figure 5.18.

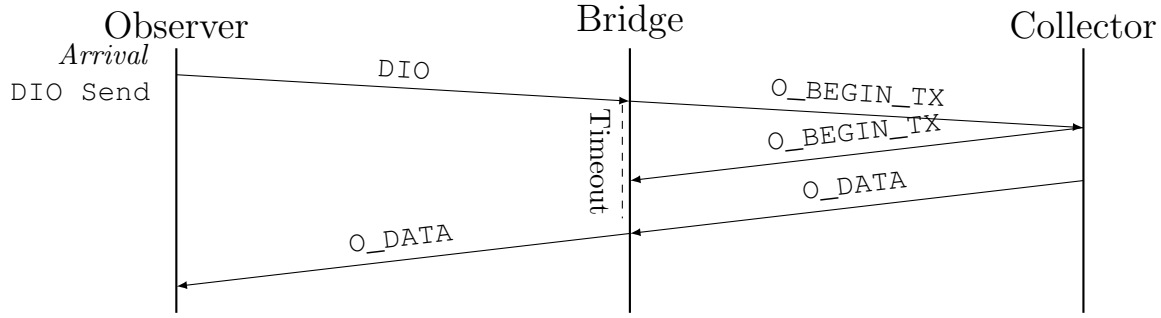


Figure 5.18: Start of Observer Data Transmission

This design is represented in the data collected during the simulation. Looking at the first row of table 5.10, when bridge **7** received the first DIO, it immediately transmitted the **OBS_START** message to the first collector in its downward routing table, collector **5**. Upon receipt, collector **5** immediately sent a reply of the same code back to bridge **7**. Shortly thereafter, it began transmitting data towards the observer. In this case, as noted earlier, it immediately scheduled a new collection and transmitted data block 5 to its bridge, **7**. When bridge **7** received this data, it immediately routed it to observer **9**. Table 5.10 shows

all four collectors receiving, and confirming, the **OBS_START** messages and beginning an immediate collection and data transfer. The final displayed row in the table shows the next scheduled collection of collector **5**.

This collection cycle continued unabated until the arrival of messenger **8**, at 01:30.000. Messenger **8** sent its first DIO following this movement at 01:59.742. Bridge **7** received this DIO immediately thereafter, at 01:59.805 and relayed this to bridge **2**, which received the DIO at 02:03.057.

Table 5.11: Data Transmissions Following Messenger Arrival

From	Type [Data]	From	To	TX Time	RX Time	From	To	TX Time	RX Time
4	**OBS_DATA [8]**	4	2	01:53.474	01:53.808	2	9	01:53.831	01:54.097
7	**MES_START**	7	2	02:01.601					
5	**OBS_DATA [9]**	5	7	02:03.332	02:03.791	7	9	02:03.814	02:04.306
6	**OBS_DATA [9]**	6	7	02:03.641	02:03.916	7	9	02:03.939	02:06.377
7	**MES_START**	7	2	02:04.670					
2	**MES_START**	2	1	02:06.721	02:06.864	1	2	02:06.864	02:07.315
7	**MES_START**	7	2	02:07.741					
1	**MES_DATA [1..5]**	1	2	02:08.430	02:08.806	2	8	02:08.814	
4	**OBS_DATA [9]**	4	2	02:08.483	02:08.547	2	9	02:08.571	02:09.174
7	**MES_START**	7	2	02:10.811					
1	**MES_DATA [1..5]**	1	2	02:11.711	02:11.920	2	8	02:11.928	02:12.572
8	**MES_ACK**	8	2	02:12.611	02:12.906	2	1	02:12.909	02:13.115
1	**MES_DATA [6..9]**	1	2	02:13.125	02:13.285	2	8	02:13.292	02:13.685
8	**MES_ACK**	8	2	02:13.721	02:14.144	2	1	02:14.147	02:14.488
7	**MES_START**	7	2	02:13.961					
1	**MES_FIN**	1	2	02:16.510	02:16.650	2	1	02:16.653	02:16.864
2	**MES_START**	2	4	02:16.719	02:17.003	4	2	02:17.254	02:17.413
7	**MES_START**	7	5	02:20.100	02:20.329	5	7	02:20.578	02:20.742

This second selection of data in table 5.11 is a subset representing the packet transmission activity following the DIO receipt from the newly arrived messenger **8**. This selection has several interesting aspects to it, relating to this validation. The first row shows the continued, normal transmission of data to observer **9**, which is still in reception range. The second row shows the first attempt by bridge **7** to start the messenger data dump. Unfortunately, bridge **2** is listed as one of its neighbors and has been entered into the downward routing table. This represents both validation for the design and an interesting aspect of

the RPL routing table implementation. As bridge **2** broadcasts its DIO for messenger **8**, bridge **7** receives it. As the DIO is a valid instance for its whitelist and it has a larger rank, it is added to the table as a downward route. This is an unfortunate, yet unavoidable effect of using RPL for routing.

As a validation for the design, this data selection shows multiple attempts for bridge **7** to make communications with what it believes is one of its children. The present setting for retry attempts is 5. The final row of this table represents this successful start, having skipped over a few prior events as indicated by the horizontal rule. This validates the delivery assurance mechanism that allows for a certain number of delivery attempts before pressing on with the next device. In this manner, if a device dies, it will miss the events occurring during its down time, but will still be polled and communicated with upon its return.

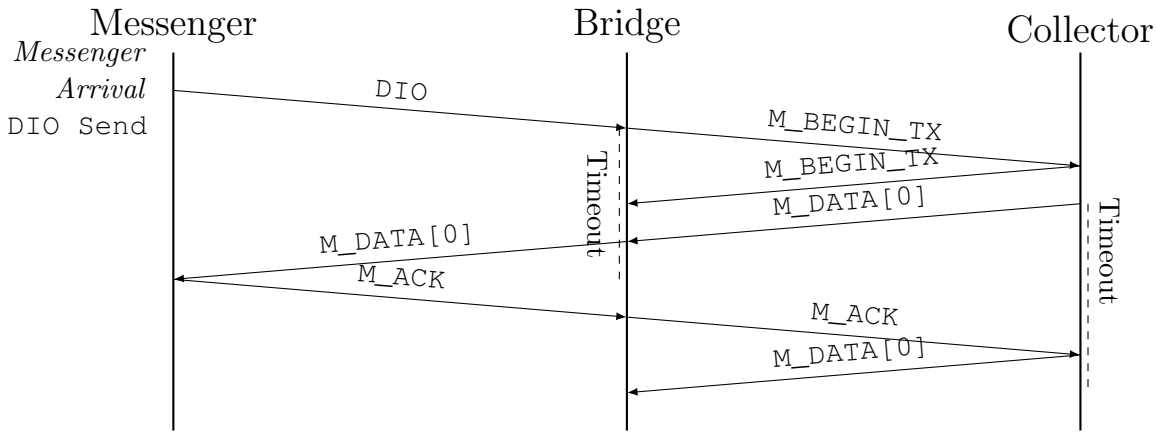


Figure 5.19: Start of Messenger Data Transmission

The second key validation, as seen in the table entry showing the first **MES_START** message from bridge **2**, is that the messenger data start procedure does follow the design, as shown in figure 5.19. After receiving the start message, collector **1** sends the same message back to bridge **2** as an acknowledgment and then begins broadcasting its first five blocks of data to its bridge. Once bridge **2** receives this data, it retransmits it to messenger **8**.

At this same moment, another transmission occurred, which prevented messenger **8** from receiving the message. After the 3 second timeout period, collector **1** retransmitted the same first five blocks of data. This message did get through and was received by messenger **8**. Following this reception, **8** then sent an acknowledgment message back down to bridge **2**, which ultimately sent it to **1**. Upon receipt of the acknowledgment, **1** sent its next block of data, which was similarly acknowledged. Lacking any new data, collector **1** sends the finish message to its bridge. Bridge **2** at this point cycles to the next collector in its routing table and sends the start message to **4**.

This sequence both validates the network design, but also validates the timeout and resend of the same data following a timeout. This subset of message transmissions serve to validate both messenger and observer interactions and show the cessation in observer transfers by a device during messenger communications. After a collector has their finish message acknowledged by their local bridge, it resumes collection and resumes sending data to the mobile observer, if it is still in range.

The data collection continued and all other collectors had successful data transmissions to the messenger, then resumed collections and transmissions to the observer. At 300 seconds, the messenger was removed from the environment. This was followed 30 seconds later by the removal of the observer. At this point, both instances are still active in the system. The lifetime of each of these instances was set for 255 seconds. At the point of the removal of the messenger, it had approximately 3 minutes seconds remaining. This implies that it had sent a new DIO out one minute prior, refreshing its lifetime.

Finally, the table also shows proper finish and transfer of control from one collector to the next by the bridge, as shown in the design figure 5.20.

The final point of validation that is shown in this simulation is the validation of the instance lifetime decay and rejoin mechanics. As the neither the mobile devices, nor the bridge sinks know when they will join together, instance decay is absolutely essential to the termination following the mobile device leaving and the rejoining once the mobile device has returned. The following listing shows the actions of bridge **7** as pertaining to the lifetime

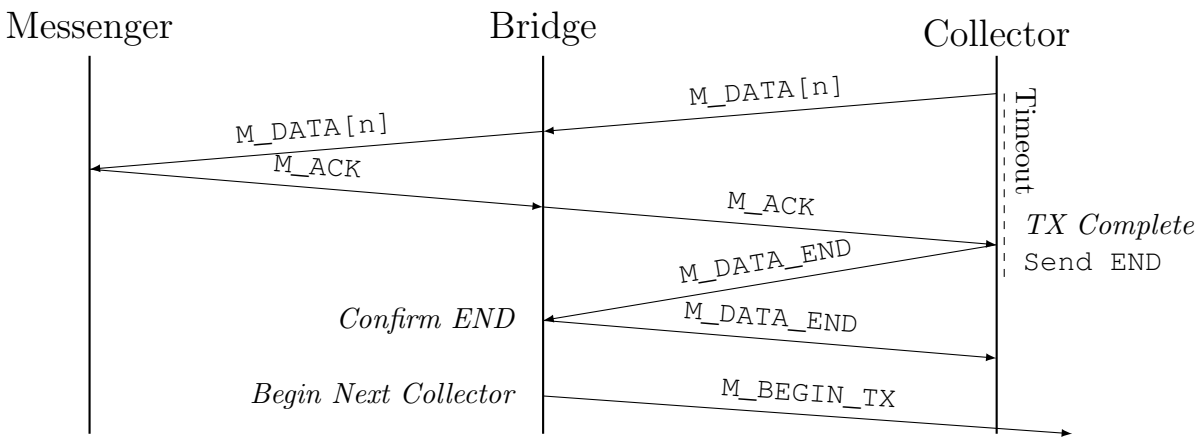


Figure 5.20: End of Messenger Data Transmission

of stale messenger instance, **M**.

Listing 5.2: Lifetime Analysis

```
08:24.380       ID:7    RPL: Instance Lifetime Timer triggered
08:24.385       ID:7    The instance lifetime for 0x4f is now 1
08:24.388       ID:7    Killing Instance 0x4f
08:24.988       ID:7    My downward routing table is:
08:24.995       ID:7    aaaa::c30c:0:0:2 via fe80::c30c:0:0:2
08:25.002       ID:7    aaaa::c30c:0:0:5 via fe80::c30c:0:0:5
08:25.008       ID:7    aaaa::c30c:0:0:6 via fe80::c30c:0:0:5
08:25.096       ID:7    uip_ds6_route_rm: removing route: aaaa::c30c:0:0:2
08:25.109       ID:7    No more routes to aaaa::c30c:0:0:2
08:26.012       ID:7    My downward routing table is:
08:26.018       ID:7    aaaa::c30c:0:0:5 via fe80::c30c:0:0:5
08:26.025       ID:7    aaaa::c30c:0:0:6 via fe80::c30c:0:0:5
08:28.644       ID:7    Instance Dead, starting timeout for 10
                        Sending (attempt #0) 10 to aaaa::c30c:0:0:5
...
10:49.018       ID:7    Received an RPL control message
10:49.024       ID:7    RPL: Received a DIO from fe80::c30c:0:0:8
10:49.029       ID:7    RPL: Neighbor already in neighbor cache
10:49.035       ID:7    RPL: Incoming DIO (id, ver, rank) = (77,240,256)
```

```
10:49.042       ID:7    RPL: Incoming DIO (dag_id, pref) = (aaaa::4d:1, 0)
10:49.048       ID:7    The instance 'M' (0x4d) is on the allowed list.  Joining.
...
10:49.122       ID:7    Instance[0].used == 1, ID == 'B'
10:49.126       ID:7    Instance[1].used == 1, ID == 'M'
10:49.136       ID:7    Initial DIO, starting timeout for 3
                        Sending (attempt #0) 3 to aaaa::c30c:0:0:5
```

This above listing from this simulation log shows the tracking of the instance lifetime for instance **O** (0x4f). Once this lifetime hit 1 at the beginning of the assessment phase, bridge **7** purged instance **O** from its instance table, removed all routes on that instance, displayed the full instance table to verify, and began sending the timeout messages to its children. In this case, it is sending the timeout to the first entry in its routing table, which is collector **5**.

The reason bridge **2** was purged from the downward routing table was that bridge **2** sent it a DIO for instances **O** and **M** with a higher rank, causing bridge **7** to consider it a downward child. At this point in the code, **M** had already timed out and **O** is being purged, leaving no references tying bridge **2** to bridge **7**, flagging it for a purge. Now that **2** is no longer in the table, when **7** begins sending data type 10 (**OBS_TERMINATE**) to its first child, collector **5**.

Finally, this shows receiving the first DIO from messenger **8** following its return. It checks the instance against the whitelist and joins it. Now that the bridge has joined instance **M** again, it begins sending code 3 (**MES_START**) to each of its children, starting with collector **5** in this instance.

5.4 Analysis of Validation

This validation has demonstrated the fundamental operational nature of the designed architecture in Contiki. The mainstay of chapter 4 pertains to the extensive modification

of the operating system to enable each of the necessary components for multiple instance routing, hierarchical routing, and instance lifetime decay. The architecture as designed in chapter 3 has validated through a careful analysis of the message traffic and internal logs of the devices, as demonstrated through comparisons of the packet transmissions and the network transmission design diagrams.

5.4.1 Necessary Adjustments for Implementation

There are several compile-time definitions that need to be adjusted prior to an implementation of this architecture. The first of which is the lifetime of each instance. This constant, defined by **RPL_CONF_DEFAULT_LIFETIME_UNIT** and **RPL_CONF_DEFAULT_LIFETIME**, now controls the viability of the instance over time. During the above simulation for validation, this was set to provide a lifetime of 255 seconds for all three instances, which was the value I selected for testing to ensure the lifetime decay mechanic was operating. The failure in this was made evident when conjoined with a second adjustment constant: **RPL_CONF_DIO_INTERVAL_DOUBLINGS**. I set this constant to 5 for the Messenger and Observer codebases, giving them a maximum time period between DIO messages of 131 seconds to ensure a DIO would always arrive before the lifetime of an instance decayed. Unfortunately, I neglected to change the default value for the Bridge codebase, leading to the following selection below.

Listing 5.3: Incorrect Lifetime Results

```
11:12.504       ID:4    Received a packet, appdata[0] == 2
11:12.511       ID:4    .. Start Index = 20, Current Index = 20, Last Index = 43
11:12.515       ID:4    Beginning Messenger Data Send
11:12.522       ID:4    .. Start Index = 20, Current Index = 20, Last Index = 43
11:12.524       ID:4    Generated 1
11:12.525       ID:4    ..Type: 1
11:12.527       ID:4    ..Source: 4
11:12.530       ID:4    ..Number of Entries: 5
```

```
11:12.534       ID:4    ..Seqno for this Instance: 26
11:12.537       ID:4    ..From 4, Index 21, Temp 20
11:12.541       ID:4    ..From 4, Index 22, Temp 21
11:12.545       ID:4    ..From 4, Index 23, Temp 22
11:12.549       ID:4    ..From 4, Index 24, Temp 23
11:12.553       ID:4    ..From 4, Index 25, Temp 24
11:12.556       ID:4    Sending MES Data to
11:12.558       ID:4    aaaa::c30c:0:0:2
...
11:12.654       ID:4    RPL: Instance Lifetime Timer triggered
11:12.659       ID:4    The instance lifetime for 0x42 is now 2
...
11:13.585       ID:4    The instance lifetime for 0x42 is now 1
11:13.588       ID:4    Killing Instance 0x42
```

As the above listing shows, very shortly after sending blocks of data 21 to 25, the primary instance **B** (0x42) linking collector **4** with its bridge died. The reason for this is that the default maximum value for the DIO send time on a stable network is 1048 seconds, which is longer than the 255 second lifespan of the instance. This is a critical problem to understand when implementing a lifetime. If the means to synchronize the distributed devices with each other has a longer duration between sends than the lifetime of the network it was meant to refresh, then that network has a high chance of death. The adjustment necessary to transition from this validation to an experimental environment is to increase the lifespan on instance **B**, as it is designed to be static and stable for the lifetime of the deployment.

Aside from the lifetime issue, I could not find any other issues with this validation run. Having demonstrated each of the components of this architecture in operation, the next step is to begin evaluating the efficacy of the design.

Chapter 6: Evaluation

This chapter covers experiments to demonstrate the basic functionality of the HOIST architecture. While these experiments are not exhaustive in nature, these evaluations demonstrate that this work provides a solid foundation of this architecture as a viable platform for further research.

6.1 Evaluation of Architecture

This section covers a couple of experiments designed to provide a basic level of evaluation over this designed architecture. This section is divided into subsections, each of which will provide an overview of the evaluation, the scenario setup for Cooja, and the results of the experiments.

6.1.1 Evaluation 1: Twenty Collectors

This evaluation is designed to examine the architecture under an implementation which places more collectors than a single traditional Contiki RPL network is able to support downward routing for. The objective of this evaluation is to demonstrate that the number of devices may be increased provided enough bridging sinks to divide the load. Moreover, this evaluation demonstrates the capability of this architecture to handle a large degree of overlap between collectors wherein multiple bridges are within direct communications range. This demonstration further shows data assurance over such a dense grouping of devices.

Figure 6.1 shows the Cooja simulation setup for this evaluation. The central line of devices in purple, labeled **3** through **8** are bridge sinks. The green node labeled as **1** is the deployment location for the messenger sink. The remaining twenty devices in yellow are all collectors. The bridge sinks were separated such that each is at the maximum transmission

range of its two neighbors. This places most of the collectors within range of multiple collectors, as well as multiple bridges to choose from.

Figure 6.2 represents the same simulation, albeit using a logical representation of the network formed. The solid black lines denote the preferred parent observed during the simulation for each of the collectors. The dashed red lines represent the DODAG formed when the messenger arrived. The network formed using a star topological formation around the central bridge sinks, as intended. This reduces the number of hops that data has to travel before it reaches its ultimate destination.

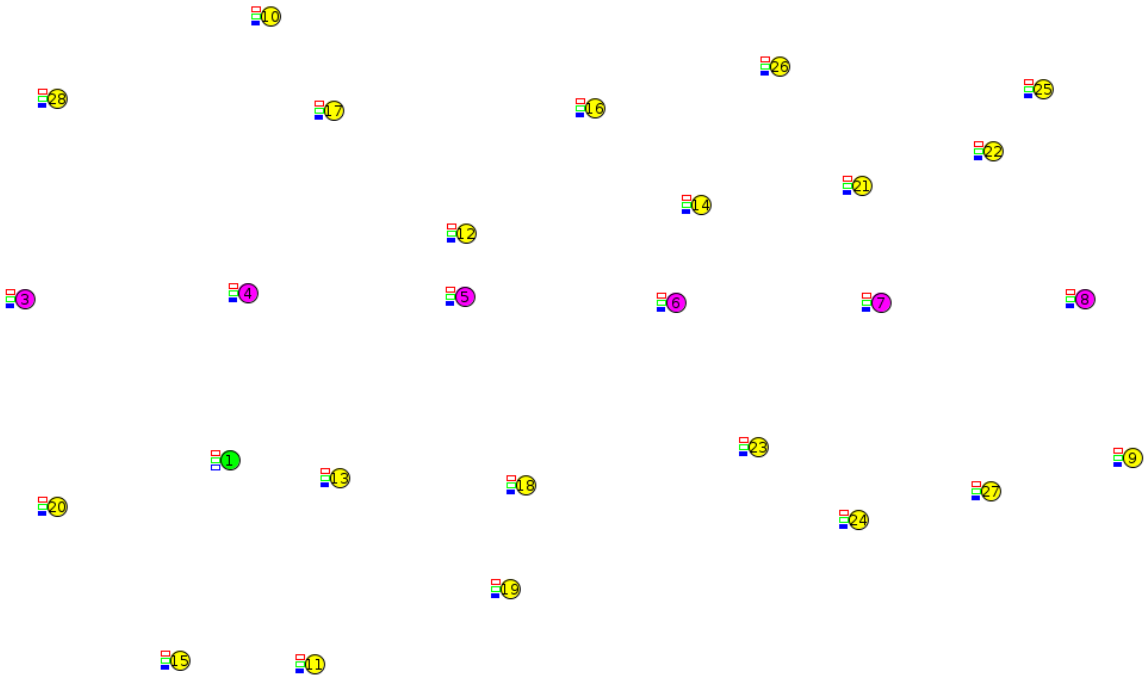


Figure 6.1: Twenty Device Implementation in Cooja

Despite the growth in the size of this deployment over the four collector validation, the full network formed with all downward routing tables completed in 19.845 seconds. The messenger sink **1** was moved to the deployment area at the 30 second mark of the simulation. It sent its first post-move DIO at 57.558 seconds.

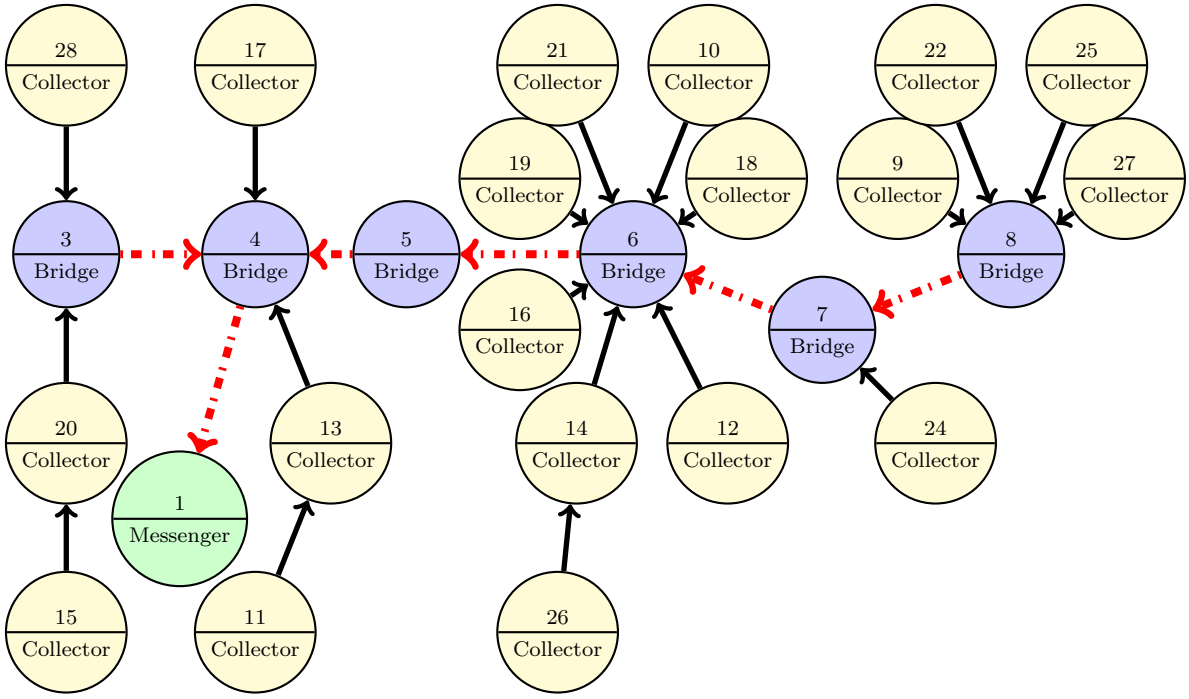


Figure 6.2: Twenty Collector RPL Logical DODAGs

The downward routing necessary to ensure full delivery of responses back to the collectors is solely dependent upon the time needed to receive a DAO from each of the bridge sinks. This final DAO was received at the 1:23.866 time mark, however, the bridge closest to the messenger was ready for full communications at 01:01.248. During this 22 second window as bridges were routing their DAO messages towards the messenger, the closest collectors were already sending their data. The following table shows the arrival times for the data to the messenger.

Over an 80 second period, 36 packets were transmitted and 31 were successfully delivered with confirmation, as detailed in table 6.1. Of the five unsuccessful packets, four were sent to the messenger before the collector's bridge had been added to the messenger's downward routing table, preventing the acknowledgment from arriving. Only one message was lost in transit. After failing to receive the acknowledgment, each of the five resent the same data successfully following the 10 second timeout.

Table 6.1: Twenty Collector Data Arrival Times

Time	Collector	Data	Time	Collector	Data
01:03.336	17	1..4	01:38.650	28	1..5
01:04.575	20	1..5	01:40.263	27	6..7
01:08.925	10	1..5	01:40.987	28	6..7
01:09.414	14	1..5	01:46.347	16	1..5
01:14.616	24	1..5	01:46.347	16	1..5
01:15.604	20	1..5	01:46.710	22	1..5
01:15.753	13	1..5	01:49.322	16	6..7
01:19.692	9	1..5	01:50.060	22	6..7
01:20.056	14	1..5	01:55.251	18	1..5
01:21.668	11	1..5	01:57.619	25	1..5
01:22.032	15	1..5	01:57.788	18	6..8
01:24.895	24	1..5	02:00.977	25	6..8
01:28.121	26	1..5	02:04.840	19	1..5
01:30.222	9	1..5	02:07.941	19	6..9
01:32.605	15	1..5	02:14.420	21	1..5
01:34.061	12	1..5	02:16.018	21	6..9
01:36.048	12	6..7	02:22.343	23	1..5
01:36.912	27	1..5	02:23.845	23	6..9

6.1.2 Evaluation 2: Fifteen Collector Comparison

This evaluation compares a traditional RPL architecture of 15 collecting devices against the HOIST architecture. Both environments will use the same operating system, however, the traditional architecture will use a single observer to collect live data from 15 collectors. Under both models, the observer will enter at the 30 second mark of the simulation.

In this comparison, both implementations use the same position file on mobility. The simulation runs for 90 seconds before terminating. At the 30 second mark the mobile collector enters the deployment area at the position occupied by the blue node **16** in figure 6.3. The traditional architecture comparison uses a collector with a single data packet to deliver. Once a DIO is received, the collector will attempt to deliver this data packet to the mobile sink after two seconds. If the data is not acknowledged, a four second timeout is triggered, after which the collector will attempt another send within one second. This process will repeat until a reply is received from the mobile sink, at which point the next message received by the mobile sink will count as a delivery success with confirmation.

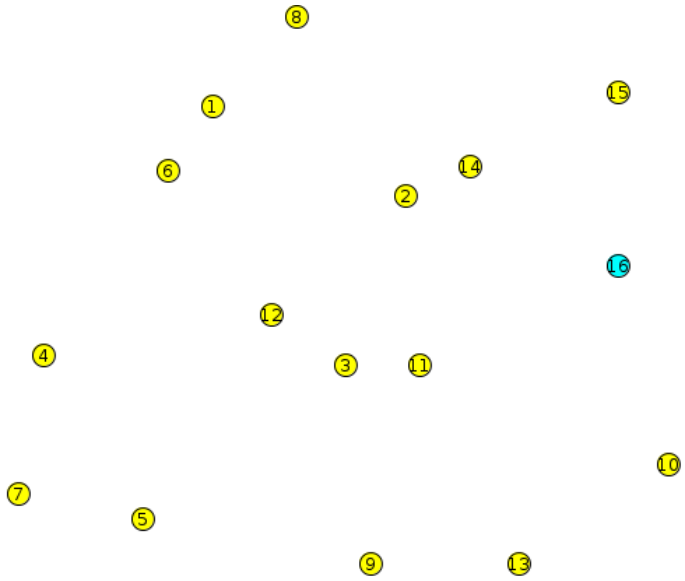


Figure 6.3: Fifteen Collector Comparison - Traditional Architecture

After a 90 second run, this traditional deployment architecture resulted in a successful packet transmission with delivery acknowledgment for nine out of fifteen collectors. The initial successes were the devices closest to the mobile sink location, with later successes arriving from collectors farther away. The most distant success was collector **8**, with a hop count of 2. As of the two minute mark, the final data packets received their initial acknowledgment at 01:55.989. Only one node, 14, was unable to receive a response within a two minute cap.

After a 90 second run, the HOIST architectural deployment, shown in figure 6.4, resulted in eleven out of fifteen collectors delivering data. In this scenario, four bridge sinks (annotated using white circles) were added. All other positions were maintained as they were in the prior test. Allowing the simulation to continue, the final of the fifteen data packets arrived at 1:43.473. One stark difference between this and the traditional model is that initial data came in from across the deployment area; it was not confined to the immediate surroundings of the mobile sink, expanding outwards over time. The first nodes to report back were collectors **11**, **1**, **10**, and **13**, which represent collection nodes across

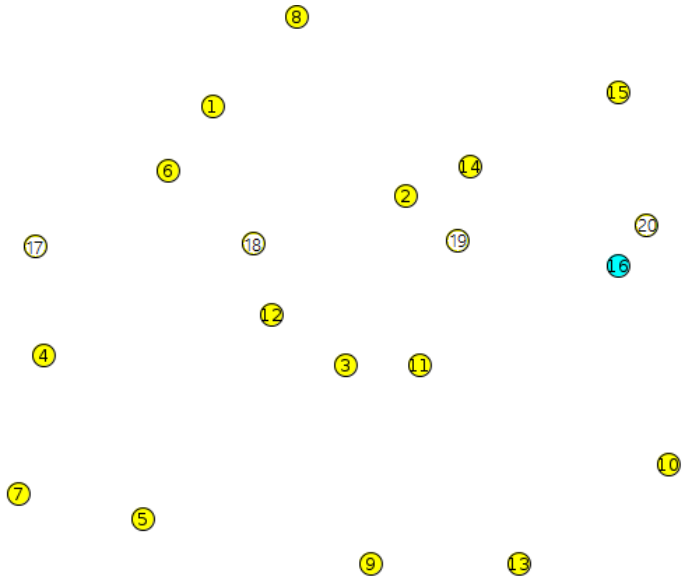


Figure 6.4: Fifteen Collector Comparison - HOIST Architecture

the deployment field.

Chapter 7: Conclusion

My desire for this work was to develop a viable WSN architecture that would support multiple sink destinations, mobility of the sinks, and the ability to increase the size of the deployment field without a commensurate increase in the amount of network traffic. I have pursued this design under the motivating scenario to provide a low-maintenance system, using commonly accessible hardware and a popular open source operating system, that would benefit large, geographically segregated deployment areas, such as is present in the field of precision agriculture.

7.1 Accomplishments

I have provided extensive modifications for the Contiki 2.7 operating system to enable a core feature of RPL: multiple sink routing. In pursuing these modifications, I have analyzed and modified core networking structures, reimplemented portions of the IP and RPL network layers, and added new features not covered by RFC6550, such as the instance join whitelist and the coordinated instance lifetime decay. These modifications and new features enable general purpose, multi-instance, hierarchical routing for uses beyond the original aim of this thesis.

I have also designed, implemented, and validated the HOIST RPL networking architecture. This architecture has already demonstrated the ability to increase a practical deployment size for WSN implementations. Pairing this architecture with the unmanned aerial system testing and evaluation conducted for this thesis, will enable the ability to manage multiple, independent WSN deployment sites without succumbing to the inherent problem of energy drain that connecting the sites with repeaters would entail.

7.2 Limitations of this Technology

As with any wireless deployment, local flooding will still occur with any dense cluster of devices. The Contiki RPL system has further issues affecting this implementation at present. RPL has no means for message-identification of other sinks in a deployment. In a hierarchical deployment, it is not presently possible to discriminate neighbors against known sibling sinks and reject their routes. As as result of this, the limitation on the size of the downward routing table is smaller in actuality, as sibling sinks may be added. In addition to wasting the very limited amounts of memory for this table, it also slows down the network by triggering the timeout mechanism on transmissions that are sent to, and ignored by, such other sinks.

One additional problem was discovered with the implementation of HOIST on the modified system. This may be present in Contiki 2.7 without the modifications; further testing is required to continue assessing this newly identified problem. OF0 failed to properly form a multi-hop DODAG. While this objective function is ideal for the mobile sink instances in the HOIST design, in this implementation, the OF was reverted to MRHOF.

7.3 Future Work

This architecture makes provisions for a messenger sink to enter a deployment area, collect the data from the sensor devices, and then leave to either deliver the data or to collect from the next geographic deployment area. The next stages of this work are to exploit the weakness in the routing delays inherent with the RPL protocol. One aspect of research in this would be developing a mobility prediction model for a messenger with a deterministic circuit of travel. By assessing the delays between the initial DIO reception and each change in routing, a bridging sink should be able to determine offset times whereupon it could preemptively change its routing to enhance connectivity. By using this simple technique for mobility prediction, downtime in the network due to restructuring may be minimized.

Further work should be done to attempt to build a better mobility model for the messenger sink. This architecture specifies only a nomadic model, wherein the device would enter range, remain in range during the data dump process, then leave once all messages have been received. To more effectively address the energy drain issues, further work should be undertaken to either manually, or autonomously, program the messenger devices to circle the deployment at a slow speed, enabling each of the bridge devices an opportunity for direct connection with the mobile sink.

Finally, while the UAS aircraft provided a valid platform, further testing is required under the framework of the fully implemented HOIST. This is a continued area of interest that will require time to pursue. Additionally, the unknown nature and scope of the operating system modifications precluded a final implementation using the Pioneer P3-AT robots in a large field implementation. This is a logical next step in the validation of such an architecture, to demonstrate the viability of HOIST on actual devices in a controlled environment, using such an autonomous robotic system.

Bibliography

Bibliography

[1] S. Advancare, "Z1 datasheet," 2013. [Online]. Available: http://zolertia.com/sites/default/files/Zolertia-Z1-Datasheet.pdf

[2] Jbasic, "English: A wireless sensor node of type TelosB by crossbow technology," Nov. 2009. [Online]. Available: http://commons.wikimedia.org/wiki/File:TelosB.jpg

[3] A. Dunkels, "The ContikiMAC radio duty cycling protocol," Swedish Institute of Computer Science, Technical Report T2011:13, Dec. 2011. [Online]. Available: http://soda.swedish-ict.se/5128/1/contikimac-report.pdf

[4] J. Terrell, "Sustainable wine making & tech: The perfect pairing," Jul. 2014. [Online]. Available: http://www.foodtechconnect.com/2014/07/17/sustainable-wine-making-technology-the-perfect-pairing/

[5] E. Hamida and G. Chelius, "Strategies for data dissemination to mobile sinks in wireless sensor networks," *IEEE Wireless Communications*, vol. 15, no. 6, pp. 31–37, Dec. 2008.

[6] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, April 2005, pp. 364–369.

[7] Thingsquare, "Thingsquare - internet of things technology," 2013. [Online]. Available: http://www.thingsquare.com/tech/

[8] K. Islam, W. Shen, and X. Wang, "Wireless sensor network reliability and security in factory automation: A survey," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, no. 6, pp. 1243–1256, Nov 2012.

[9] Y. Yang, A. May, L. Yang, and S.-H. Yang, "Opportunities for WSN for facilitating fire emergency response," in *Future Wireless Networks and Information Systems*, ser. Lecture Notes in Electrical Engineering, Y. Zhang, Ed. Springer Berlin Heidelberg, 2012, vol. 143, pp. 479–486. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27323-0_60

[10] A. De San Bernabe Clementa, J. Dios, and A. Baturone, "A wsn-based tool for urban and industrial fire-fighting," *Sensors*, vol. 12(11), pp. 15 009–15 035, Nov. 2012. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3522951/

[11] E. Ancillotti, R. Bruno, and M. Conti, "The role of the RPL routing protocol for smart grid communications," *IEEE Communications Magazine*, vol. 51, no. 1, pp. 75–83, Jan. 2013.

[12] T. Winter, P. Thubert, A. Brandt, J. Hui, and R. Kelsey, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," Internet Requests for Comments, IETF, RFC 6550, March 2012. [Online]. Available: http://tools.ietf.org/html/rfc6550

[13] IEEE, "IEEE internet of things," 2014. [Online]. Available: http://iot.ieee.org/about.html

[14] N. Grammalidis, F. Tsalakanidou, A. Benazza, E. Kuruoglu, D. Torri, E. etin, K. Dimitropoulos, K. Kose, C. Ersoy, O. Gunay, B. Kosucu, F. Chaabane, S. Tozzi, and B. Gouverneur, "A multi-sensor network for the protection of cultural heritage," in *ZENODO*, Aug. 2011. [Online]. Available: http://zenodo.org/record/1192

[15] K. Karenos and V. Kalogeraki, "Traffic management in sensor networks with a mobile sink," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 10, pp. 1515–1530, Oct. 2010.

[16] A. Somov, A. Baranov, D. Spirjakin, A. Spirjakin, V. Sleptsov, and R. Passerone, "Deployment and evaluation of a wireless sensor network for methane leak detection," *Sensors and Actuators A: Physical*, vol. 202, pp. 217–225, Nov. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0924424712007297

[17] S. Gandham, M. Dawande, R. Prakash, and S. Venkatesan, "Energy efficient schemes for wireless sensor networks with multiple mobile base stations," in *IEEE Global Telecommunications Conference, 2003. GLOBECOM '03*, vol. 1, Dec. 2003, pp. 377–381 Vol.1.

[18] B. Nazir and H. Hasbullah, "Mobile sink based routing protocol (MSRP) for prolonging network lifetime in clustered wireless sensor network," in *2010 International Conference on Computer Applications and Industrial Electronics (ICCAIE)*, Dec. 2010, pp. 624–629.

[19] T. Clausen, U. Herberg, and M. Philipp, "A critical evaluation of the IPv6 routing protocol for low power and lossy networks (RPL)," in *2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Oct. 2011, pp. 365–372.

[20] J. Rao and S. Biswas, "Network-assisted sink navigation for distributed data gathering: Stability and delay-energy trade-offs," *Computer Communications*, vol. 33, no. 2, pp. 160–175, Feb. 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366409002394

[21] T. Winter, "RPL: IPv6 routing protocol for low-power and lossy networks." [Online]. Available: http://tools.ietf.org/html/rfc6550

[22] A. Dunkels, B. Grnvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004. [Online]. Available: http://dunkels.com/adam/dunkels04contiki.pdf

[23] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568

[24] I. Nest Labs, "Works with nest," 2014. [Online]. Available: https://nest.com/works-with-nest/

[25] T. Inc., "Intelligent climate control for heating & air conditioning," 2014. [Online]. Available: http://www.tado.com/de-en

[26] P. G. Corporation, "SmartThings," 2014. [Online]. Available: www.smartthings.com

[27] R. Brown, "Developer program makes nest a focal point for the smart home," Jun. 2014. [Online]. Available: http://www.cnet.com/news/developer-program-makes-nest-a-focal-point-for-the-smart-home/

[28] M. Corporation, "Tmote sky datasheet," 2006. [Online]. Available: http://www.eecs.harvard.edu/ konrad/projects/shimmer/references/tmote-sky-datasheet.pdf

[29] Zolertia, "Z1 compared to telosb/tmote," 2009. [Online]. Available: http://zolertia.sourceforge.net/wiki/index.php/Z1_compared_to_Telosb/Tmote

[30] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Commun. ACM*, vol. 43, no. 5, p. 5158, May 2000. [Online]. Available: http://doi.acm.org/10.1145/332833.332838

[31] C. Products, "CC2420 datasheet," Texas Instruments, Datasheet, 2014. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2420.pdf

[32] J. Adams, "An introduction to IEEE STD 802.15.4," in *2006 IEEE Aerospace Conference*, 2006, pp. 8 pp.–.

[33] D. A. Warren, H. Fiennes, J. A. Dutra, D. Bell, A. M. Fadell, M. L. Rogers, I. C. Smith, J. Satterthwaite, J. E. Palmer, G. M. Erickson, and A. Mucignat, "United states patent: 8788103 - power management in energy buffered building control unit," Patent 8 788 103, Jul., 2014.

[34] I. Cisco Systems, "Connections counter: The internet of everything in motion - the network: Cisco's technology news site," Jul. 2013. [Online]. Available: http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342

[35] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6)," Internet Requests for Comments, IETF, RFC 2460, December 1998. [Online]. Available: http://tools.ietf.org/html/rfc2460

[36] D. Weisenberger, "Questions and answers," 2003. [Online]. Available: http://education.jlab.org/qa/mathatom_05.html

[37] A. Dunkels, "Contiki: Bringing IP to Sensor Networks," ERCIM News, Jan. 2009. [Online]. Available: http://ercim-news.ercim.org/content/view/496/705/

[38] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," Internet Requests for Comments, IETF, RFC 6282, September 2011. [Online]. Available: http://tools.ietf.org/html/rfc6282

[39] "Ieee standard for local and metropolitan area networks part 15.4: Low-rate wireless personal area networks (lr-wpans)," *IEEE Std. 802.15.4-2011*, pp. 1 –294, 2011.

[40] J. A. R. Azevedo and F. E. Santos, "Signal propagation measurements with wireless sensor nodes," University of Madeira, Portugal, Campus da Penteada, Tech. Rep., Jul. 2007. [Online]. Available: http://cee.uma.pt/people/faculty/amandio.azevedo/Indoor.pdf

[41] R. Draves, J. Padhye, and B. Zill, "Comparison of routing metrics for static multi-hop wireless networks," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, p. 133144. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015483

[42] M. Zhao and Y. Yang, "A framework for mobile data gathering with load balanced clustering and mimo uploading," in *INFOCOM, 2011 Proceedings IEEE*, April 2011, pp. 2759–2767.

[43] C. Tunca, S. Isik, M. Donmez, and C. Ersoy, "Distributed mobile sink routing for wireless sensor networks: A survey," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 877–897, 2014.

[44] R. Shah, S. Roy, S. Jain, and W. Brunette, "Data MULEs: modeling a three-tier architecture for sparse sensor networks," in *2003 IEEE International Workshop on Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE*, May 2003, pp. 30–41.

[45] J. Valente, D. Sanz, A. Barrientos, J. d. Cerro, . Ribeiro, and C. Rossi, "An air-ground wireless sensor network for crop monitoring," *Sensors*, vol. 11, no. 6, pp. 6088–6108, Jun. 2011. [Online]. Available: http://www.mdpi.com/1424-8220/11/6/6088

[46] C. Malveaux, "Investigating the potential for drone use in agriculture," *Louisiana Agriculture*, vol. 57, no. 1, pp. 8–11, 2014. [Online]. Available: http://text.lsuagcenter.com/NR/rdonlyres/CDDE1EE5-BB6B-4CD4-B100-B0BA0BCC721D/96270/PDFI.pdf

[47] J. Otto, "UAVs are next wave of agricultural technology - AgriNews," Apr. 2014. [Online]. Available: http://agrinews-pubs.com/Content/News/MoneyNews/Article/UAVs-are-next-wave-of-agricultural-technology-/8/27/10106

[48] Internet Assigned Numbers Authority, "Routing protocol for low power and lossy networks (rpl)." [Online]. Available: https://www.ietf.org/assignments/rpl/rpl.xml

[49] O. Gnawali and P. Levis, "The minimum rank with hysteresis objective function." [Online]. Available: http://tools.ietf.org/html/rfc6719

[50] P. Thubert, "Objective function zero for the routing protocol for low-power and lossy networks (rpl)." [Online]. Available: http://tools.ietf.org/html/rfc6552

[51] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 06 1958. [Online]. Available: http://dx.doi.org/10.1214/aoms/1177706645

[52] T. Clausen, O. Gnawali, J. Ko, and J. Hui, "The trickle algorithm." [Online]. Available: http://tools.ietf.org/html/rfc6206

Curriculum Vitae

Kevin Michael Andrea received his Bachelor of Science in Computer Science from George Mason University in 2012. He has previously served as a Sergeant in both active service in the United States Marine Corps and in reserve service for the Army National Guard of the State of California and the Army National Guard of the Commonwealth of Virginia. He is a member of the ACM Special Interest Group for Computer Science Education and his research interests include computer science education, wireless sensor networks, and robotics.