UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SWITCHING WITH ADAPTIVE INTERVAL LABELS FOR WIRELESS NETWORKS

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Kevin Moua

June 2018

The Thesis of Kevin Moua
is approved:

Professor J. J. Garcia-Luna-Aceves, Chair

Professor Chen Qian

Rick Graziani

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

Abstract

Switching with Adaptive Interval Labels for Wireless Networks

by

Kevin Moua

The initial design of the Internet and its protocols did not impose limitations to performance as its purpose intended to use and share expensive computing resources [5]. Modern networks consists of different communication models that expand the capabilities of how resources are exchanged with one another. Wireless sensor networks (WSN) and the Internet of Things (IoT) are examples that have become dominant network platforms for data acquisition, allowing service providers and consumers to collect and exchange data with sensors and devices embedded throughout the physical world. However, these types of networks suffer from the limitations of lossy communication media and low-powered devices. Additional drawbacks that these networks still face are that conventional design methods intended for traditional networks are still being used in protocol design and implementation. An area of such design concern includes routing. RPL [22] has been proposed as a routing-protocol solution for WSN's by catering to the specific needs of low-power and lossy networks. Although RPL is emerging as a standard for routing in WSN's, it still faces many challenges concerning scalability with increases to network size, point-to-point traffic, and underspecification of handling node failures [8]. In this thesis we propose Switching with Adaptive Interval Labels (SAIL), which takes a clean-slate approach from traditional routing. SAIL utilizes compact routing

rather than destination-based routing, and caters to the issues RPL faces by replacing destination-based identifiers with interval labels. We implement SAIL using the ns-3 simulator and compare its performance to RPL in a wireless-network deployment. Our results show that SAIL outperforms RPL in energy conservation and forms shorter forwarding paths in small to medium-size networks. For larger deployments we show that SAIL provides tremendously lower storage overhead where routing table sizes remain constant, while RPL routing tables grow linearly at $O(n)$, where n equals the number of nodes in the network.

Acknowledgments

I owe my deepest gratitude to my advisor Prof. J. J. Garcia-Luna-Aceves for his continuous support of my Masters study and related thesis research and for his immense knowledge. His guidance helped me in all my time of research and writing of this thesis.

Chapter 1

Introduction

Wireless sensor networks (WSN's) typically consist of devices that are resource constrained in regards to battery life, memory storage, and processing capabilities. Existing routing protocols such as OSPF [19] are not suited for WSN's as they require extensive resources and cannot scale to optimize routing in these networks [18]. RPL was specifically designed to overcome the limitations posed by WSN's by implementing different functionalities, such as the trickle timer [22] to dynamically control the rate of messages transmitted in a network in order to conserve energy. These design features supported by RPL helped preserve the constrained resources under routing operations. However, RPL still faces many shortcomings including its lack of design support for point-to-point traffic, scalability in operating in large deployments, underspecification of design implementation, and complexity to implement in a real life deployment [8]. Many of the drawbacks RPL faces are a consequence from the fact that it still supports traditional destination-based routing. In this thesis, we propose a clean-slate approach

to routing that uses compact routing with interval labels to address the shortcomings of RPL.

This thesis focuses on analyzing the disadvantages of the Routing Protocol for Low-power and Lossy Networks (LLN), RPL [22], and proposes a high-level specification for a simple routing protocol, SAIL (Switching with Adaptive Interval Labels), which aims to scale in ways that RPL cannot for WSN's. SAIL and RPL are then implemented, tested, and compared in performance using the ns-3 simulator.

The rest of this thesis is organized as the following: Chapter 2 provides background information about WSN's, LLN's, and describes the challenges faced in these types of networks. A general specification of RPL is described, and we further analyze the shortcomings of RPL in more detail. Chapter 3 summarizes and describes the past work on different routing approaches proposed for WSN's and LLN's that differ from traditional destination-based routing. Chapter 4 presents SAIL and its protocol specifications. Chapter 5 presents SAIL's design implementation in ns-3, and in Chapter 6 we evaluate and compare the performance of SAIL through simulation tests against RPL in various network sizes. Chapter 7 provides a summary about our proposed protocol and also insights of future work.

Chapter 2

Background

2.1 Wireless Sensor Networks

Wireless sensor networks (WSN's) consist of sensors, typically small in size, with limited processing and computing resources that are deployed in difficult-to-access geographical locations [23]. Because of this positioning, WSN nodes use radio for wireless communication in order to transmit data collected from its physical environment [23]. These network topologies typically consists of few tens to thousands of nodes that are constantly collecting and pushing data to a base station node. Applications using WSN's include military target tracking and surveillance, biomedical healthcare monitoring, hazardous environmental monitoring and exploration, and other areas of geographical monitoring [23]. Batteries are the main source of energy in these networks, and nodes using radios for data transmissions must also perform as minimal routing operations as possible in order to conserve battery life. WSN's are typically deployed in

an unstructured manner where the network must self-configure and heal itself in the presence of node or link failures. With additional transmission and signaling to ensure network connectivity, WSN's call for a new routing protocol that can scale in the presence of these issues.

2.2 Low-Power and Lossy Networks

Low power and lossy networks (LLN's) are a variation of WSN's [20]. Applications using LLN's include home automation, building automation, industrial automation, and monitoring urban environments [20]. Traffic patterns in LLN's typically consist of multipoint to point (MP2P), point to multipoint (P2MP), and in recent cases point to point (P2P). These networks are heavily resource-restrained in terms of energy, memory, and processing power. Similar to WSN's, LLN's face low-powered battery constraints along with lossy communication challenges, requiring a new routing approach that must be able to cope with resource limitations while scaling and providing optimal routing service.

2.3 Criteria for Routing in WSN's/LLN's

Ideally, a routing protocol proposed for LLN's must be able to operate in constrained conditions where:

- Routing is optimized for energy conservation.
- Data patterns do not consist of only unicast flows [18].

- Packet sizes must be as minimal as possible [18].
- Efficiency and generality are equally weighted since LLN nodes are limited in computational and memory resources [18].

It is further stated in [18] that the minimal criteria that a routing protocol for LLN's must meet includes scaling with routing state, localizing response to link failures without global network repairs, low energy cost for autonomous route configuration, and the ability to take into account node and link costs while constructing routing paths. Existing routing solutions including Ad Hoc On Demand Vector (AODV) [21] and Optimized Link State Routing (OLSR) [15] for mobile ad-hoc networks (MANETs) fail to meet all requirements previously stated in the criteria for an optimal routing solution in LLN's [18]. Other routing protocols such as OSPF take into consideration link cost while constructing forwarding paths, but requires global flooding of link-state advertisements in order to inform all nodes of a failed link or node. If no current protocols can meet the requirements of LLN's, further work must be done in order to in introduce a new standard for these types of networks.

2.4 RPL

The IETF Working Group, known as Routing Over Low power and Lossy networks (ROLL), proposed a routing protocol for LLN's to overcome the routing challenges stated in the previous section [4]. Routing Protocol for Low Power and Lossy Networks (RPL) [22] had emerged as the most prominent solution for routing in WSN

deployments during its initial proposal [2]. RPL primarily supports routing from a set of nodes to a sink node (multipoint to point traffic) with some support for point to multipoint and point to point traffic. Its main attraction includes its features of constructing a logical Destination Oriented Acyclic Graph (DODAG) rooted at a sink node, use of trickle timer to control transmission rates based off of network consistency, and its flexibility to route based off of different objective functions.

2.4.1 RPL Specification

We briefly summarize the basic operation of RPL as its specification is very complex. RPL constructs different DODAG topologies based off a routing optimization metric specified by the objective function. RPL uses two modes of operation including storing and non-storing mode, and forms local and global routing state about the entire network topology through immediate neighbors by using a set of packets including DODAG Information Objects (DIO's) and Destination Advertisement Objects (DAO's). To further optimize energy conservation, RPL utilizes a trickle timer to decrease the interval of packet transmissions when routing has stabilized. We explore these features in more detail in the subsequent sections.

RPL forms a DODAG over the underlying network graph rooted at a sink node. The topology of the DODAG can vary based off of the objective function set, which determines the next hop adjacent edge to forward a packet. The objective function includes but is not limited to hop count, energy consumption, and expected number of transmissions until a data packet reaches a node [22]. For example, in Figure 2.1 the

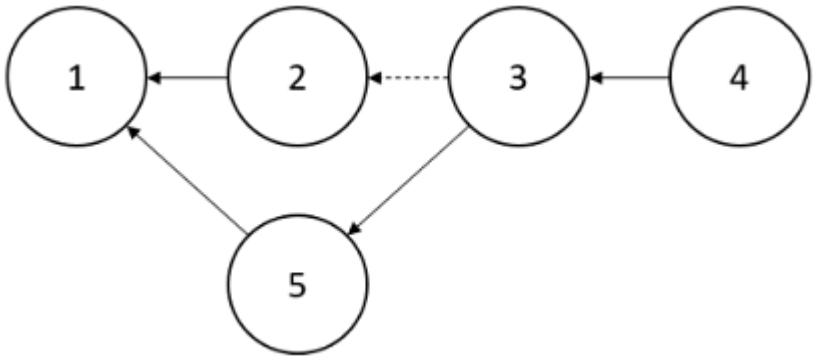

Figure 2.1: Simple RPL network in a chain topology

RPL network routes traffic based off of a node's energy level [22]. Observe that node 3 routes traffic through node 5 since node 5 has more energy than node 2. After some time, if node 5's energy link drops below node 2, node 3 will route traffic through node 2. RPL also supports construction of multiple DODAG's spanning over the physical topology, using identifiers known as RPL instance ID's to distinguish between the two logical topologies. This allows numerous DODAG topologies to be rooted at multiple sinks, increasing network productivity for different WSN and LNN applications.

The process of constructing a DODAG begins when a sink node transmits a DIO message to its immediate neighbors. A DIO message contains the RPL instance ID the node belongs to, network prefix, logical identifier, metric rank, and optional headers. Upon reception, a neighbor node adds route state information about the sender of the DIO and begins to participate in RPL routing. The node first assigns itself an IPv6 identifier using SLAAC [3], and chooses a preferred parent node from the DIO's received based off of the optimal objective function metric. It then begins transmitting its on DIO messages, and this process propagates until all nodes have joined the DODAG.

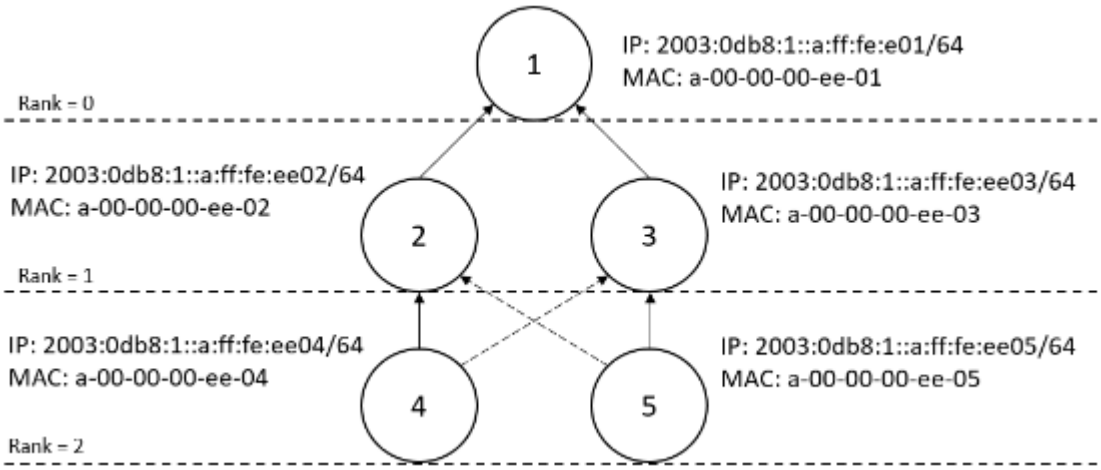

Figure 2.2: RPL tree topology depicting address plan and rank hierarchy

Downward routes from the root node to leaf nodes are formed and learned by intermediate nodes through this process. Concurrently, DAO's are periodically unicasted back to parent nodes from intermediate nodes. DAO's contain information about children nodes reachable in the sub-DODAG rooted at the sender of the DAO. This process propagates from leaf nodes towards the root. Through this process, downward routes are learned as nodes learn all reachable destination in its sub-DODAG. In Figure 2.2, we show a simple tree topology of a RPL network, addressing plan of how nodes use SLAAC, and a representation of the objective function set to hop count.

RPL further utilizes different modes of operations that affect the amount of routing state stored by nodes while routing traffic upwards. In non-storing mode, RPL nodes incur less storage overhead as DAO's are propagated all the way back towards the root. In this mode, source routing is utilized at the root in order to forward traffic to a children node. In storing mode, DAO's are unicasted upwards towards parent

nodes where routing stated is stored and not forwarded. If a parent node contains the destination in its routing table while in storing mode, it doesn't need to forward data packet upwards and transmits it to the next-hop children that contains the destination in its sub-DODAG.

2.4.2 Issues with RPL

Although RPL has been proposed as the current routing solution for WSN networks, it still faces many limitations. We briefly summarize a set of problems that RPL faces identified by Clausen [8].

RPL's design focuses on multipoint-to-point and point-to-multipoint traffic [8]. This is not the case in current WSN deployments such as the IoT where device to device (point-to-point traffic) is not uncommon as nodes exchange information with one another. In the current RPL design, device-to-device traffic pattern involves forwarding traffic all the way up to the root if a common parent node is not shared between source and destination. The root node will either perform source routing or destination-based routing in order to forward the traffic based off of the current mode of operation. This causes at risk redundant and longer pathways, since traffic is always routed up the tree. Under high point to point traffic load involving intermediate nodes with uncommon parent ancestry, congestion occurs between the communication medium near the root which increases the percentage of data loss. Hotspot nodes will also consume an excessive amount of energy in respect to the entire network since they are used majority as relays.

Although RPL aims to provide flexibility in conserving routing state information with different storage modes, both do not scale with increases to network size [8]. In storing mode, nodes will have to store route state for all nodes in its sub-DODAG. In non-storing mode, there are increased risks of fragmentation and loss of data packets because of source routing [8].

Another main issue that RPL faces includes network convergence after link failures and topology changes. If a node's preferred parent needs to change due to a link or node failure, it must change its own IP address if IP addresses are assigned in a hierarchical fashion. This requires all nodes in the sub-DODAG to do the same. Changing IP addresses can disrupt ongoing communication since they serve multiple purposes of identifying end-point communication and topological location [8]. This issue is also implementation-specific, and we do not explore this issue in this thesis as we implement RPL address configuration in a non-hierarchical manner.

RPL faces many more challenges and we briefly summarize a remaining few. RPL nodes do not test bidirectional operability when intermediate nodes select parent nodes [8]. Reception of DIO's enables a noid to join and participate in RPL routing without any form of bidirectionality verification [8]. If a child node receives a DIO on a unidirectional link and selects the sender as its preferred parent, traffic route upwards would be lost. Another big problem is RPL's underspecification in its implementations [8]. Many specific details are missing including timing of DAO transmissions, absence of jitters, and what to do when DAO's from intermediate routers along a path are not received [8]. These problems heavily influence design implementation, and without clear

specification of explicit details, protocol performance will vary tremendously.

RPL faces many scalability issues as we have discussed, which are all consequences of the traditional destination-based routing in which RPL has built off of. Destination-based routing requires a forwarding node to store information specifying where and how to reach all of the possible destinations in its network. This approach does not scale in IoT deployments for the reasons we have stated in RPL. In order to overcome this, we have to change our approach to routing and find alternative methods to destination-based routing. We explore different methods in the next section.

Chapter 3

Related Work

There has been a considerable amount of past research conducted in different areas of routing for WSN's including name-based routing, routing using distributed hash tables (DHT), routing based off of geographical location, and compact routing. These related approaches attempt to provide innovative techniques to routing in hopes of scaling and accounting for the limitations in WSN's.

3.1 Name-Based Routing

Name-based routing in information-centric networking (ICN) was one of the first alternative approaches proposed for routing in the WSN's [6]. This routing approach aimed to use content names in replacement of hostname as destination identifiers [6]. Mechanisms including name resolution and name-based routing are used to discover content based off a unique identifier. In name resolution, the content name is first translated to to a locator identifier, and then the request for that content is routed to that

destination [6]. In name based routing, traffic for the requested content is forwarded based on the content name through flooding techniques, and route state information is maintained along that path in order to establish paths back to the requester [6]. One of the earliest routing protocols proposed using name based routing in WSN's includes Directed Diffusion [14]. Interests specifying a content's attribute and value are disseminated and flooded throughout the network until they reach a node that contains the requested content. During an interest's transit to a content provider, paths known as gradients are formed allowing providers to unicast information back to requesters.

However, protocols such as Directed Diffusion do not provide optimal results in energy conservation. These protocols contain unwanted qualities attributed for WSN's due to the excessive overhead signaling from flooding interests throughout the network. Another common issue includes the fact that the gradients formed are not reliable due to the WSN's dynamic nature in topology changes.

3.2 Routing using Distributed Hash Tables

Routing with distributed hash tables has been proposed as another alternative solution to traditional destination-based routing. Routing information is distributed amongst the nodes in the network in order to solve issues such as load balancing, multiple replicas, consistency, and many others [24]. A hashing function is used to generate a key residing in the DHT. The DHT provides an abstracted global key space where nodes participating in routing or data are assigned an identifier in this space [24]. Pro-

tocols that use this approach include Chord, Kademlia, Tapestry, Viceroy, and many more [24]. A key benefit of routing in this approach in favor of WSN's is that route state information increases logarithmically with increased network sizes, and duplicate resources can be stored in the case of single-point failures.

However, state information involving the overlaying logical topology of the network must be maintained, so additional signaling must be incurred. Such overlaying topologies include rings and virtual links that can correspond to multi-path links in the physical network topology, and the information exchange to maintain the virtual topology can become excessive in WSN deployments.

3.3 Geographical Routing

Geographical routing protocols focus on the core issues of dynamic scalability in ad-hoc and sensor networks, where the rate of topology changes significantly impact routing due to the nature of the network's mobility [9]. Protocols in this category use a node's geographical position in the form of GPS coordinates to route traffic to an intended destination region. Routing protocols such as DREAM [7] flood the network with information about a node's position, where nodes compute the distance to that location through kinematics by using the speed of a destination's velocity, time of the last position information received, and the current time. Other protocols such as GPSR [16] use greedy algorithms to forward packets to neighboring nodes closest to the destination and also incorporate fallback mechanisms to ensure that greedy routing doesn't fail.

These protocols, however, suffer from the consequences of physical invariances such as line-of-sight reachability to satellites for GPS-based devices, and also incur additional overhead of discovering locational identifier of nodes. Alternative solutions using virtual coordinates in reference to beacon nodes are proposed through protocols such as Beacon Vector Routing [10] and LCR [13].

These solutions, however, are limited since virtual coordinates can correspond to multiple nodes, and there is no uniqueness to these virtual identifiers since positional coordinates are in reference to a relative node. This leads to incorrect routing or additional flooding in verifying if the node located at the virtual coordinate is the intended destination.

3.4 Compact Routing

Compact routing aims to minimize routing state and scales with the number of immediate neighbors rather than the number of nodes in the entire network [17]. This benefit, however, comes at the cost of increased path lengths. Information about distant nodes reachable by neighbors requires aggregation and grouping with respect to some kind of relation amongst nodes in a subtree [17]. Some relations include interval labeling [12] and prefix labeling [17]. In interval labeling, routing nodes are labeled so that all nodes reachable through the same edge belong to the same interval. An edge in this context is simply a link connecting a node to its immediate neighbor. Limitations to this approach are that all network labels have to be redone after a single link failure.

Prior interval-labeling approaches use interval-label space equivalent to the number of nodes in the network which allows for compact labeling. However, assigning single interval labels to a node leads to depth-first search algorithms as the labeling parent must know the number of interval labels to assign beforehand, and this incurs additional signal overhead. Our proposed routing protocol for WSN's differs from prior compact routing approaches as it utilizes continuous interval labels with a range of labels, and creates a hierarchical label assignment for the entire network rooted at the tree. Interval-label assignment in this way can utilize breadth-first search techniques, minimizing energy consumption while conserving state information.

Chapter 4

Switching with Adaptive Interval Labels

Switching with Adaptive Interval Labels (SAIL) is proposed as a clean-slate approach from destination-based routing for WSN's using compact routing. SAIL builds from prior research on automatic incremental routing (AIR) [12], where prefix labels are assigned to nodes relative to a root. In SAIL, a continuous range of interval labels from a common identifier space are assigned to each node, where each intermediate node's interval range is contained in its parent's interval. IPv6 addresses are used as identifiers, and the interval labels assigned are a continuous range of IPv6 addresses. SAIL constructs a logical spanning DAG containing a hierarchical interval-labeling assignment rooted at a controller node. Throughout the course of this thesis, we will refer to a continuous range of interval labels as an interval or interval range, and a single interval identifier as a label.

4.1 SAIL Ranking

SAIL utilizes a ranking system similar to RPL's objective function. Nodes are ranked according to their hop-count distance from the controller node. Those closer to the controller have a higher rank and form logical relations with nodes of lower ranks as a parent node. Children nodes of lower ranks can have a set of multiple parents in the presence of multiple neighbors with higher ranks. A parent node acts as a labeling node by partitioning its interval space and informing immediate one-hop children nodes of the interval subspace allocated for them. Nodes can only assign a neighbor a subinterval from its own interval as long as it is closer to the controller than the neighbor. With the controller node partitioning its interval space with its immediate neighbors, this process is repeated and propagates downwards from the root until all nodes are assigned their own interval contained within the root's interval.

4.2 Interval-Labeling Assignment

A key issue in designing SAIL includes the interval assignments. The core of the protocol design focuses on using of a range of continuous interval labels that can be excessively distributed throughout an entire network. Even with a plethora of interval labels, the protocol still must specify the average size of the intervals to be assigned for each node.

Routing nodes require a sufficient amount of interval range beforehand in order to sufficiently handle and distribute enough intervals to new nodes that join the net-

work. Other nodes, such as end devices, may only require a single label for themselves such as nodes not participating in routing. For these reasons, we propose two different approaches for label assignment using peer-to-peer signaling and controller-based signaling.

4.2.1 Peer-to-Peer Interval Assignment

Label assignment using peer-to-peer signaling is based on periodic HELLO messages such as those used in prior routing protocols. A HELLO message is sent periodically by each node to its immediate neighbors containing the sending node's interval and a list of labeling updates. The labeling update contains information about the entire interval range of the controller along with the sender's own interval. Labeling updates can provide additional routing information stating additional intervals that are reachable in the sending node's neighborhood. After receiving a HELLO message, nodes who have not joined the SAIL network will request to join in SAIL operations through the sender of that HELLO message. The sender then becomes a labeling parent node and partitions its interval so that enough sub-intervals can be provided to its children.

An example of how SAIL assigns intervals to nodes are shown in Figure 4.1. Simple numeric values are used instead of IPv6 addresses for simplicity, where the interval lengths are arbitrarily chosen. The root node is assigned the first non-zero value in its interval range, and intermediate nodes are assigned the first label in their assigned intervals. In this scenario, the root node is assigned label 1. Nodes can learn about intervals that are not in its one-hop neighborhood through a neighbor's route

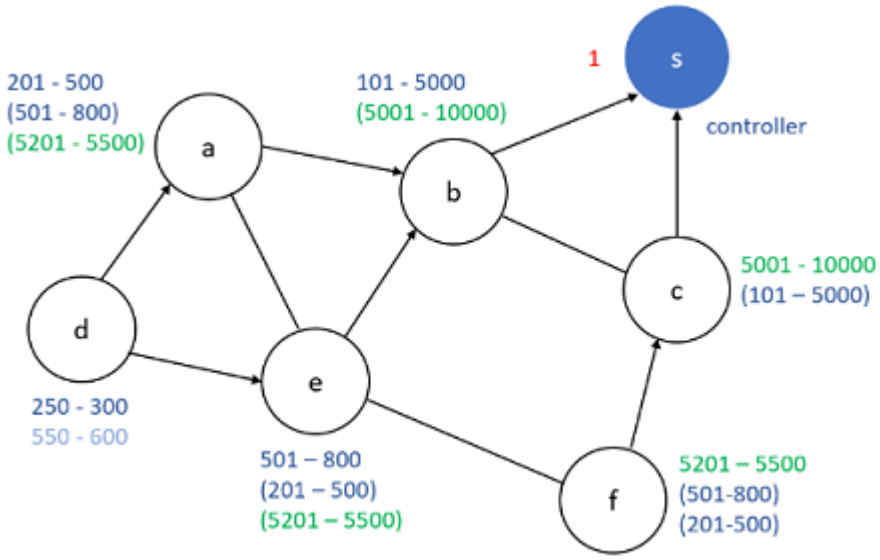

Figure 4.1: SAIL Interval Labeling using peer-to-peer signaling

state information from additional signaling. These intervals are shown as parenthesis in the figure. Node f knows that it can reach interval 201 to 500 through its sibling neighbor e. Nodes that are at the same distance with respect to hop count from the root are referred to as siblings, and they do not assign intervals to each other. Nodes can end up with more than one parent neighbor which results in multiple intervals being assigned to them. In this scenario, node d is assigned interval 250 to 300 from its parent node a and interval 550 to 600 from its parent node e.

4.2.2 Controller-based Interval Assignment

In a controller-based labeling assignment scenario, nodes communicate their network connectivity to a controller before interval assignments. The controller learns beforehand how many nodes reside in the network as intermediate nodes propagate

discover messages to the controller requesting to the join in SAIL routing operations. The controller learns the location of each node in the network in regards to which immediate child it can forward traffic to in order to reach any node. After some period of time the controller will learn about the entire network topology and partition its interval evenly since it initially knows about the network size.

In this approach, more compact labels can be assigned to nodes since the controller is aware of the number of nodes requesting to join. However, issues arise when new nodes request to join the network. This will require request and update packets to propagate and traverse path lengths equivalent to twice the tree's depth in order to notify the controller and requester respectively. This approach requires new signaling so that the controller node is aware of the entire network's link state. For these reasons, we explore peer-to-peer signaling and leave this approach for future research.

4.3 Forwarding with Intervals

Forwarding in SAIL takes advantage of route-state summarization, where information about the entire network topology is aggregated and self-contained within a node's edges. When point-to-point traffic needs to be forwarded, a forwarding node will search its routing table for the shortest range interval that contains the destination label and forward it to that neighbor. If there are no intervals that contain the destination label in a node's routing table, traffic is routed up the spanning DAG to any one of its parent labeling node. If a parent node does not not contain any route state about that

interval, the packet proceeds forwarding upwards until eventually it reaches the root. Since all interval assignments are contained within the root's interval, the root will know which of its children to forward the packet to, and the packet will be forwarded downwards until it reach its destination. In this manner, the number of forwarding entries a node stores grows linearly with the number of its immediate neighbors.

4.4 Loop-free Forwarding

Topology changes can affect routing performance as forwarding table information become inconsistent with the changes occurred. If there is a packet in-route to be forwarded during such changes, it can traverse a loop. SAIL eliminates the chances of loops occurring by adding a hop count to the header of a packet. Routing nodes can check this header against information stored in their routing tables specifying the minimum-hop distance to an interval or destination and drop packets that have a hop count exceeding the minimum distance reported [11].

4.5 Path Stretch

Compact routing protocols suffer from long forwarding paths since nodes only know information about their immediate neighborhood. By utilizing a limited distance-vector routing approach, SAIL can reduce and minimize path stretch from source to destination at the cost of additional signal overhead. This raises a question of how much additional signaling SAIL should incur in order to optimize path stretch. A node

can inform immediate neighbors of intervals it knows how to reach that are not contained in its own interval. In this manner, nodes become aware of unrelated intervals that are within another node's neighborhood. Additional route state including the depth of a node's neighborhood is limited since WSN deployments aim to minimize packet size, so we do not want to constantly flood packets that contain massive information such as an entire routing table. Nodes periodically multicast a range of neighborhood route state where we limit the number of hops contained within a neighborhood to minimize packet length.

4.6 Node and Link Failures

SAIL adapts with node and link failures by assigning nodes multiple intervals from different parents. In the case where a link to a parent node fails, SAIL does not need to perform global repair since nodes can simply use another one of its intervals assigned from other parent nodes. Timers can be used to maintain state about the last HELLO message received from an interval, and can detect link or node failure if the timers exceed a limit. This causes SAIL to expire routes that are no longer reachable, where this check can be executed only when there is data traffic is to be sent in order to minimize signal overhead.

In the case of root failure, SAIL will distributively elect a new root within the nodes that are one hop away from the root. We propose a greedy approach where the first neighbor to detect controller failure will broadcast a message stating its request

to be the new controller. This node will be termed as the request-to-control (RTC) node. When neighbors to the former controller receive this request, an acknowledgement (ACK) will be unicasted back to the RTC node. After some time after the RTC node receives the ACKs, it will configure itself to assume the position as the new controller. Unavoidably, global repair needs to occur in this scenario of a controller failure since one hop neighbors in reference to the old controller can rank at a higher degree with respect to the new controller.

Chapter 5

SAIL ns-3 Module Design

We use ns-3 to gather a deeper understanding of the protocol including its downfalls, to gain full control of its implementation, and also because the simulator provides full support for IPv6 at the routing layer. Descriptions of SAIL's operations are provided along with their respective classes in the code. We note that SAIL is not limited to these design decisions and improvements to these specifications should be explored in future work.

Figure 5.1 shows a simplified Universal Modeling Language (UML) visualization of the entire class diagram. The module is structured in this manner so that it is compliant with ns-3's IPv6 routing protocol class. The main SAIL class inherits from the ns-3 IPv6 routing protocol class which provides the basic IPv6 support for our routing protocol. We implement two additional classes referring to SAIL's routing table and a neighbor set in order to respectively record information about reachable intervals and to store additional metadata about node's neighbors. The last important set of classes

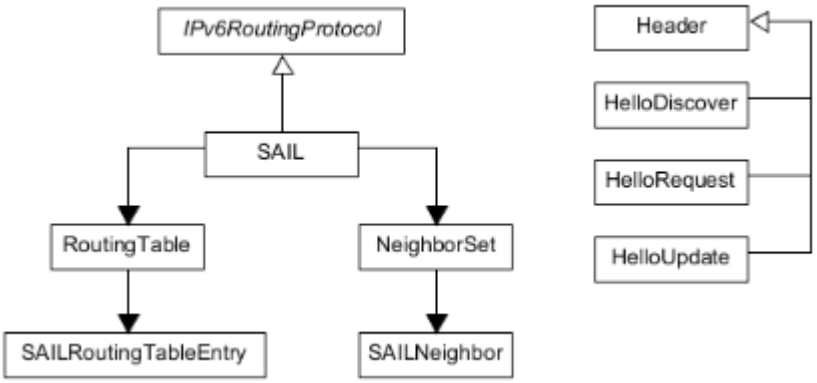

Figure 5.1: UML of classes used to implement SAIL in ns-3

that we mention includes SAIL's packet headers derived from the ns-3 header class.

5.1 SAIL Main Class

The main class represents the core of our protocol and contains the logic necessary to implement state recording, route forwarding, and message processing. This class makes use of the SAIL routing table, SAIL neighbor set, and various SAIL packet header classes. The entire main class is abstracted with the support of a helper class that provides encapsulation, making it simple for users to install the routing protocol onto a node's network stack with helper functions.

Although this class spans over a few thousand lines of code, we briefly describe its core operations and methods in the following subsections. This includes the process of joining the SAIL network, how nodes learn of distant reachable intervals, and forwarding a packet based off a destination label.

5.1.1 Root Configuration

Routing operations begin at the controller of the spanning DAG. The controller is configured with an IPv6 prefix and uses the interface portion of the IPv6 address space as its interval range. In our implementation, we provide a short network mask of four to eight bytes, allowing us to explore and allocate more interval space with the host portion. The first label in the interval is assigned to the controller. The controller performs other auto-configurations such as creating a socket associated with its new label, setting its SAIL rank to zero, and initializing timers. Once configuration is complete, the controller begins broadcasting HELLO messages.

5.1.2 Joining the SAIL Network

SAIL nodes broadcast a HELLO message periodically at a certain time interval. In our design implementation, we set this to one second with some jittering. A HELLO message will contain a node's interval, the controller's interval, and metadata describing reachable intervals in a node's distant neighborhood (the contents of a HELLO message are further described in section 5.4.1). We will refer to nodes broadcasting HELLO messages in this section as node X. When a neighboring node that is not participating in SAIL routing operations receives a HELLO message, it will store a route to node X and request to join the SAIL graph. Let us refer to this neighboring node as node Y. Node Y then creates a request message specifying which interval it wants to join out of the intervals broadcasted by node X, and node Y unicasts this request message to node X. Node X will then partition its requested interval into a sub-interval or label

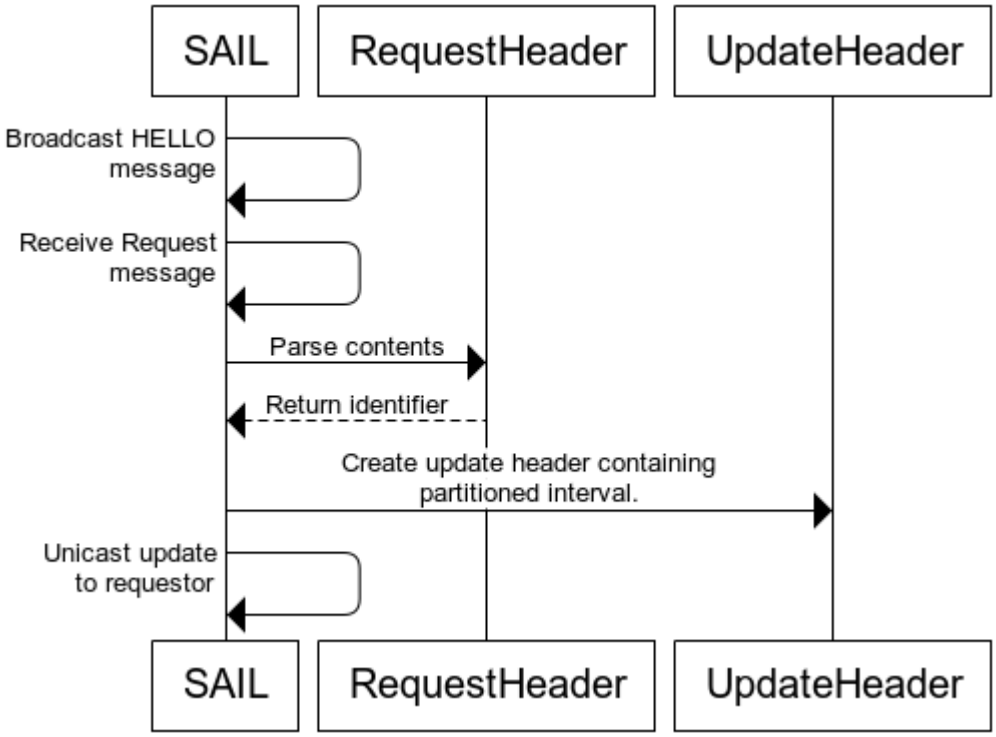

Figure 5.2: Sequence diagram of SAIL parent node

depending whether node Y is a router or leaf node, and X creates an update message with this new interval. Node X transmits the update message to node Y, and node Y stores information about its new interval and configures itself by opening a socket associated with the first label in the new interval assigned from node X. After node Y finishes configuring itself, it begins broadcasting its own HELLO messages and the process repeats until all nodes in the deployed network have joined in SAIL routing. We refer readers to view algorithms 1 to 5 in the Appendix A section for a more detailed description of the pseudo-code for this operation. For a more detailed description of the interval partitioning process, we refer reader to view algorithm 7 in Appendix A.

Figures 5.2 and 5.3 show sequence diagrams that generally describe how nodes

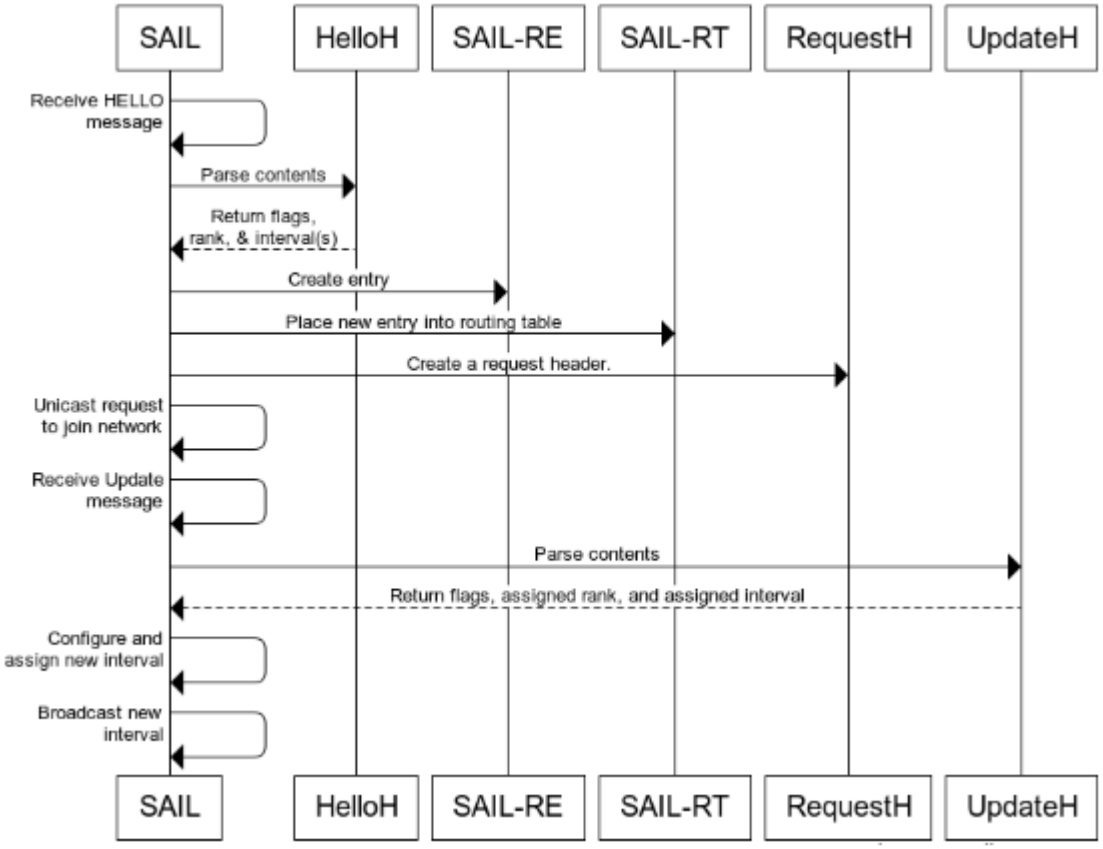

Figure 5.3: Sequence diagram of SAIL child node

join the SAIL network. In Figure 5.2, we describe the behavior of a parent node in how it broadcasts its network connectivity, and how it handles requests to join the network. In Figure 5.3, we show how a children node requests to join the SAIL network.

5.1.3 Path Reduction

In the process of broadcasting HELLO messages, SAIL attaches additional route state information in these messages containing a node's neighborhood of reachable intervals. Initially, information about a node's one hop neighborhood is shared amongst

its immediate neighbors. When a neighboring node receives this message, it stores routes to any intervals that currently do not reside in its routing table. Neighbors will also broadcast their own one hop neighborhood information when their HELLO timer expires. After a node receives information about its neighbor's one hop neighborhood, the node now attaches its two hop neighborhood and informs immediate neighbors. This process repeats until the distance of neighborhood depth information shared exceeds a threshold, and we limit this threshold to four hops.

5.1.4 Forwarding with Intervals

When a node contains a message to be forwarded, it checks if the destination label is contained within the entries in its routing table. A node contains a route to the destination label if the label is contained within an interval stored in the routing table. If there are multiple intervals that contain the destination label in their range of values, then the interval with the least difference is chosen as the forwarding path (where the least difference is in reference to the highest label in the interval subtracted by the destination label). The routing entry that contains the destination label in its recorded interval will specify which neighbor to forward the packet through. If there are no entries that contain the interval, the forwarding node will forward the interval through any parent node. We refer readers to view algorithm 6 for a more detailed description of the pseudo-code for this functionality.

5.2 SAIL Routing Table

The routing table class contains entries that specify routes to various intervals. A routing entry contains a destination interval range, the identifier of a neighbor that reported the destination interval, and status information specifying whether the destination interval is for a sibling, child, or parent node. This class allows a node to check and store route state information when packets are received from new interval neighbors.

5.3 SAIL Neighbor Set

A neighbor set is also used to store additional metadata about a node's immediate neighbors. This class tracks state about the number of intervals a node's parent neighbor has assigned it, and also reports information about a node's relation with this neighbor, where the relation states consists of the state whether a node is requesting to join through this neighbor and the reliability of the link.

5.4 SAIL Packet Headers

SAIL messages are all derived from the ns-3 header class. We note that these headers should initially have been derived from ns-3's ICMPv6 header class, and not their own custom headers. For faster development, we simply implement them as their own abstracted header types, where doing so does not impact the simulation performance. Implementation for route or interval invalidation has been implemented, but

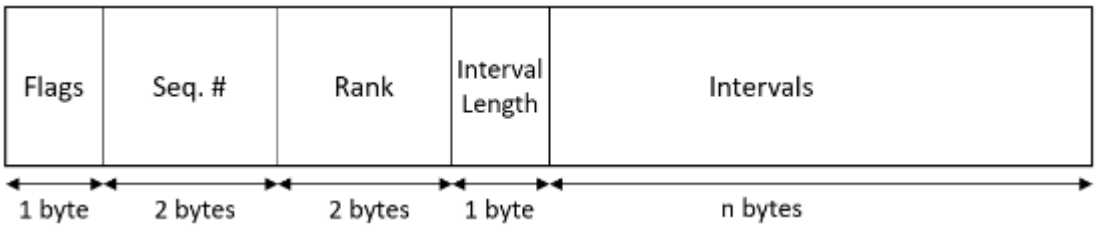

Figure 5.4: SAIL HELLO header fields

in the analysis comparison between SAIL and RPL we are not concerned with this operation.

5.4.1 HELLO Header

Figure 5.4 shows the contents of a HELLO message. Flags are used to discern between various type of HELLO message, distinguishing whether it is a labeling update with additional neighbor interval information or a regular HELLO packet. The sequence number is incremented every hop and is checked by a node with information in its FIB to ensure that forwarding loops do not occur. The rank states a node's degree in the spanning DAG specifying how many hop counts the sender is from the root. The intervals length specifies how many intervals are contained within this packet since node's can be assigned multiple intervals from its parent nodes, and the trailing header contains the sender's assigned intervals.

5.4.2 Request Header

Figure 5.5 shows the contents of a request packet. It contains a flag field to allow different options to be implemented in future works. For implementation purposes,

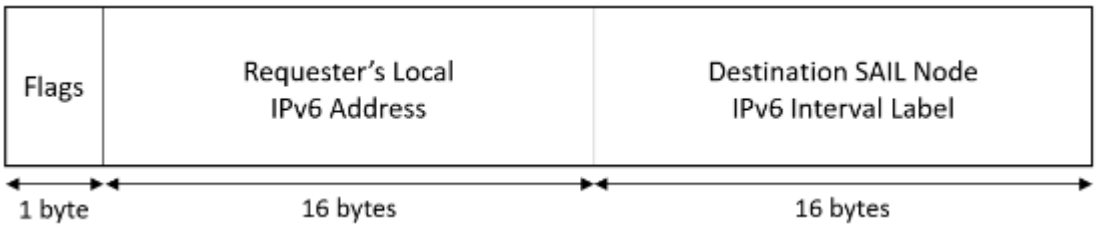

Figure 5.5: SAIL Request header fields

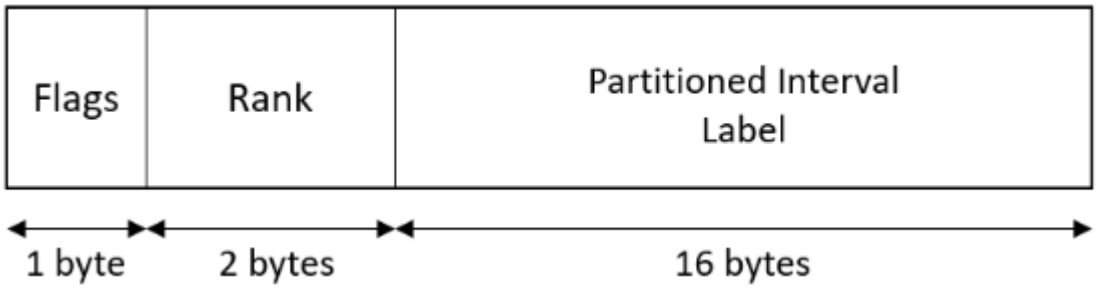

Figure 5.6: SAIL Update header fields

we include a node's local IPv6 link address in this field along with the destination node's interval label. This allows us to obtain unique identification at the routing layer rather than at the link layer.

5.4.3 Update Header

Figure 5.6 shows the contents of an update packet. A flag field is also presented to allow options such as ACKs or other options to be presented for future work to further optimize the protocol. The rank field specifies the rank to be assigned to the requesting child node, which is the sending parent's rank incremented by one. The last field contained in the packet is the new interval to be assigned to the requesting node.

5.5 Additional Design Choices

Some design choices have been made about many aspects of the protocol. Nodes cannot be assigned more than one interval from a parent node. If this were not the case, the amount of intervals assigned to nodes in the SAIL network would increase exponentially as the spanning DAG's rank increased. SAIL is designed in this thesis so that IoT deployments using SAIL can still operate with current access points running modern routing protocols. In this case, interval partitioning to create sub-intervals for children nodes are designed for this purpose and for simplicity. The controller node is assigned a prefix through its default gateway router to the Internet and partitions its IPv6 interface bits into intervals to be assigned to children nodes. As the degree of a SAIL network increases by an integer k, we create sub-intervals of the current interval by partitioning and reserving the next 8*k bits to the right of the IPv6 prefix bits. If for example the root node is assigned a 16 bit network, this gives us 112 bits, or 14 bytes, to be used for the interval-labeling space. We reserve the rightmost byte to be used for leaf node identifiers since they are to be assigned only one label. This gives us 13 bytes to use for interval partitioning, representing that our spanning DAG's in this implementation can have a maximum degree of 13. This gives us plenty of room to work with for our simulation purposes which we explore in the next chapter.

Chapter 6

Simulation Setup

SAIL's design and implementation have been tested on networks of various sizes. Its protocol performance is compared with a baseline version of RPL that we implement ourselves in ns-3.

6.1 RPL ns-3 Implementation

The ns-3 module and implementation of RPL has yet to be released, so we implement our own version of RPL based off of the RFC 6550 [3]. In our implementation, RPL uses storing mode and the objective function is set to measure hop count. DIO messages are periodically broadcasted at the same rate as HELLO messages in our SAIL implementation, and DAO's are unicasted back to parents every second. The trickle timer is not implemented simply due to the reason that such transmission rate control algorithm could also be implemented on SAIL. Our RPL implementation also assigns IPv6 address to children nodes through SLAAC as stated in RFC 6550 [3]. Nodes join

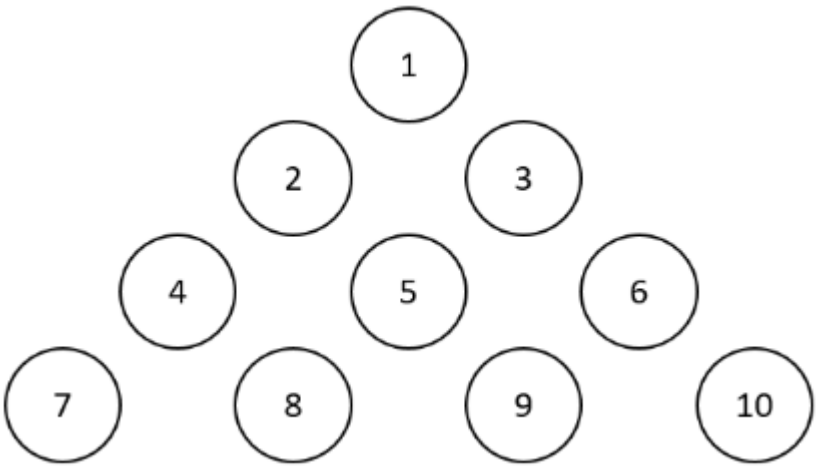

Figure 6.1: Perfect Binary Tree with 4 leaves

the RPL network by greedily accepting the first node that they receive a DIO message from as their parent node.

6.2 Test Networks

We first examine how the protocols compare in small to medium-size networks in a perfect binary tree topology. We run simulations on these topologies consisting of three leaves to twelve leaves. Figure 6.1 shows an example of our tested topology where a network consists of four leaves. For these small to medium-size networks, simulations are ran for an hour where data traffic is generated every minute for five seconds from one end leaf node to the other end-leaf node. We refer to the first and last leaf in the last row of a perfect binary tree as an end-leaf node. Point-to-point communication is emphasized in our simulations tests as we aim to analyze SAIL's design in supporting this feature whereas RPL provides minimal support

SAIL and RPL are also ran on a large-scale network to further test the performance of the protocol. The parameters of this large-scale topology consists of a square area (1 kilometer x 1 kilometer) with the root node at the center of the grid, and 1121 nodes disburse throughout this perimeter. The first 121 nodes act as routing nodes and are placed every 100 meters in both x and y axises. The remaining 1000 nodes act as leaf nodes and are randomly distributed inside the perimeter. Simulations are ran in large networks until all nodes have joined and are participating in the routing network. This large-scale simulation will give us insight on the amount of decreased storage overhead we predict SAIL has over RPL. Figure 6.2 shows our network topology visually displayed through the NetAnim process tool provided by ns-3.

6.3 Simulation Design Choices

6.3.1 Modes of Path Optimization

SAIL is implemented running in five different modes of path optimization. These modes correspond to the amount of neighborhood information shared with immediate one hop neighbors. We explore the behavior in simulation where SAIL shares one hop to four hop neighborhood connectivity information with its immediate neighbors.

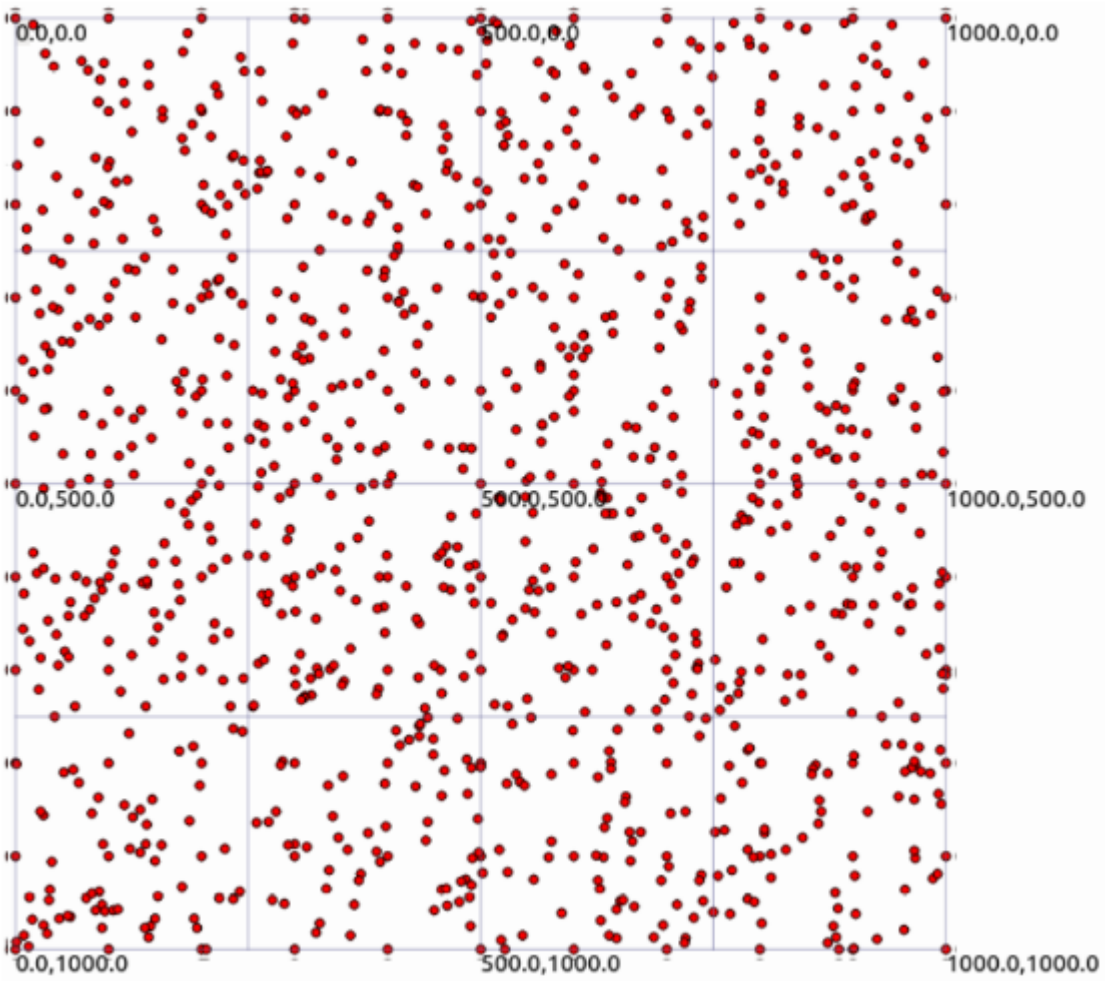

Figure 6.2: Large Network (1km x 1km) Simulation Topology

6.3.2 Simulation tests using 802.11

We note that the simulation network stack utilizes 802.11 as the link layer protocol. Although ns-3 currently has public distribution of their modules for 802.15.4 and SixLowPan, these modules do not support mesh routing at the current time. Both currently support static routing which is done at the abstracted simulation layer, however, we want our protocol to be installed as a simple plug-and-play protocol where the nodes will autonomously compute their own routing tables. For these reasons, we deploy our simulation tests on a wireless scenario using 802.11 and WiFi. Even though this solution is sub-optimal, it helps provide us with a general understanding of how SAIL operates and performs in comparison with RPL in a general wireless scenario deployment.

Chapter 7

Results

The metrics that we compare between SAIL and RPL consists of convergence set-up time, energy consumption, and the number of entries residing in a node's routing table. For small to medium-size networks, we measure performance in terms of average energy and delay of forwarding packets from an end-leaf node to the other end leaf node. For larger sized networks, we measure the average energy cost it takes for all nodes to join the network.

Our energy model uses the specifications from a Raspberry Pi Zero [1], as the Raspberry Pi is a common microcontroller used to embed network connectivity and sensors seamlessly together. The power specification sets energy consumption for radio transmissions, receiving, and idling to 320mA, 39mA, and 1.05mA respectively [1]. Test results are shown in the following subsequent sections.

7.1 Small to Medium-Size Networks

7.1.1 Convergence Time

Figure 7.1 depicts the time it takes for both protocols to converge when all nodes have joined the network and are participating in routing operations. RPL converges at a faster rate to be expected since SAIL requires nodes to test bi-directional link functionality through the message exchanges of HELLO, request, and update messages respectively in that order. SAIL's message exchange utilizes an entire 1.5 round-trip trip (RTT) in order for SAIL nodes be assigned an interval label, but the process allows nodes to consider which links to a neighbor are reliable before creating a logical link with a neighbor as a parent. RPL nodes simply accept a receiving DIO message without any forms of acknowledgements, producing a faster convergence but unreliable test of a bi-directionality link.

7.1.2 Routing Table Size

Figure 7.2 shows the average number of routing entries a routing node contains that is an immediate neighbor to the root node. These nodes are referred to as hotspot nodes since majority of resources such as network bandwidth and memory storage are consumed around these nodes as the network load increases. RPL node's routing table size increases linearly as the number of nodes in the network increases. However, SAIL node's routing tables maintain consistent and do not change as the network size increases. SAIL node's routing tables scale with the number of immediate neighbors as

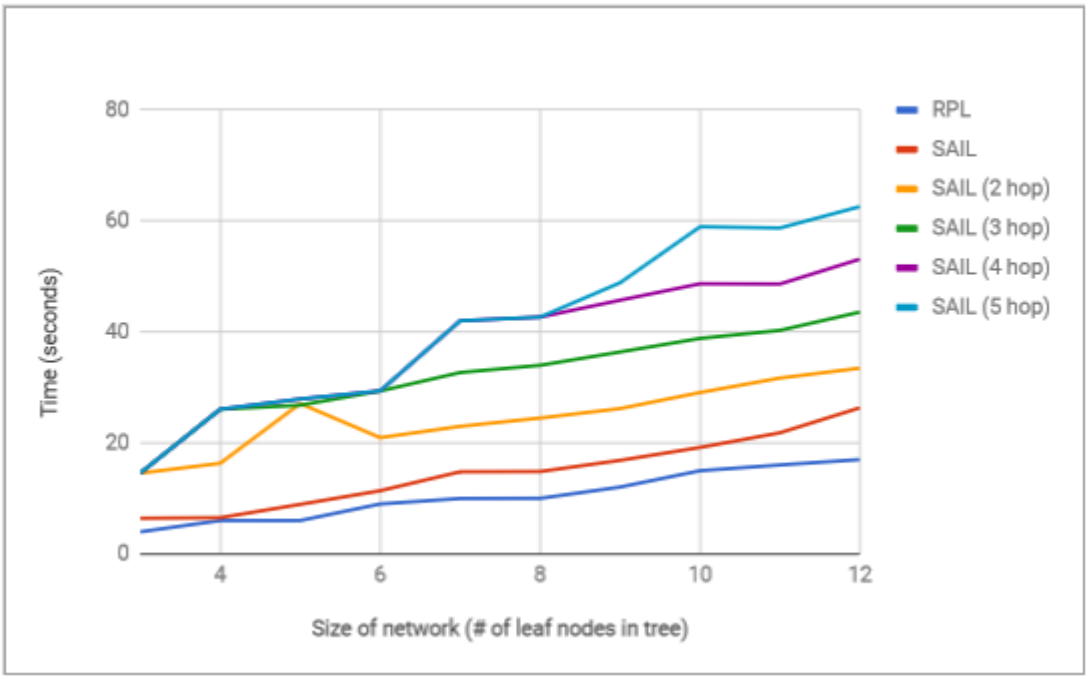

Figure 7.1: Time until all nodes have joined the network for small-medium size networks

been shown through compact routing protocols, which is favorable to WSN deployments with small storage capacity constraints.

We further analyze the average routing information overhead of all nodes in the tested networks shown in Figure 7.3. Our results show that RPL incurs less storage overhead averaged as a whole over SAIL. As the depth of the DODAG tree increases in RPL, nodes near the bottom of the tree will store less routing information. For example, leaf nodes will contain only one routing entry to their parent node. However, in SAIL, route state information remains consistent independent of the network's size.

The trade-off observed from our results show that RPL overall incurs less routing storage than that of SAIL in small to medium-size networks. However, hotspot nodes significantly incur more route state for all children in its sub-DAG in RPL. In

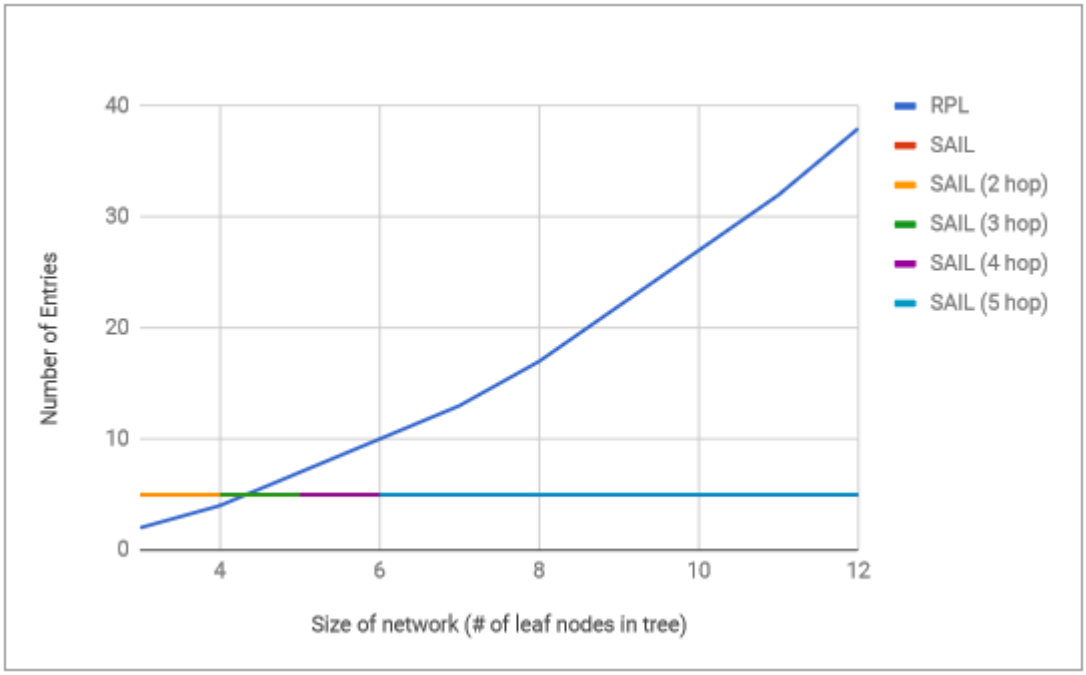

Figure 7.2: Average routing table size of hotspot nodes for small-medium size networks

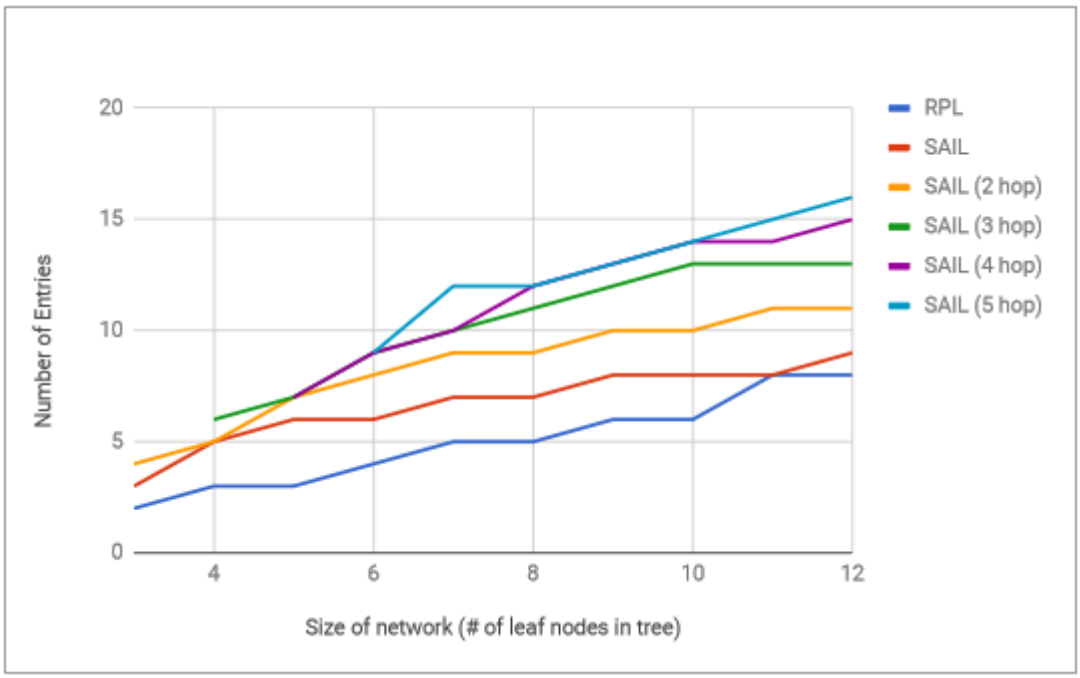

Figure 7.3: Average routing table size of all nodes for small-medium size networks

any WSN deployment, a node's resources is not dependent on the topology of the network, and memory storage can become depleted in such scenarios. SAIL scales in these scenarios and allows a fairly small and equivalent storage overhead to incur over all its nodes.

7.1.3 Energy and Delay of Data Traffic

Figure 7.4 shows the average energy consumption of a node operating in SAIL and RPL in an hour where traffic is generated for five seconds every minute, and each protocol's delay in forwarding that respective traffic pattern is shown in Figure 7.5. RPL consumes more energy since traffic from one end leaf to another require to be forwarded through the root node and back down to the respective destination. Another key factor we observe is that RPL consumes more energy since nodes periodically unicast DIO messages up the DODAG to parent nodes. Comparing the different path optimizations implemented in SAIL, more energy is consumed as additional transmissions occur to share more neighborhood interval information.

Our SAIL implementation that does not provide path optimization incurs the less amount of energy consumed, however, experiences higher end-to-end delay since traffic needs to be rooted up the tree since intervals through sibling nodes are not known. As we increase the amount of neighborhood information shared between immediate nodes in SAIL, end-to-end delays are decreased as shorter paths are formed. As we increase the amount of neighborhood information a node processes, it can find shorter paths to the destination interval label. Nodes that contain the destination interval

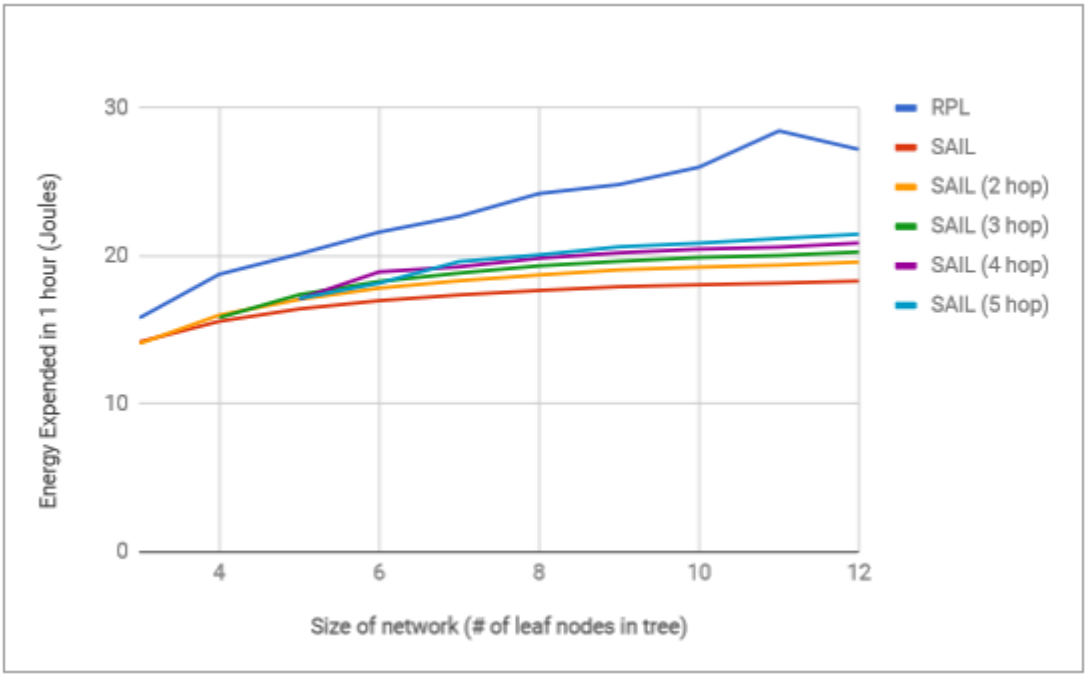

Figure 7.4: Average energy consumption of a node in 1 hour for small-medium size networks

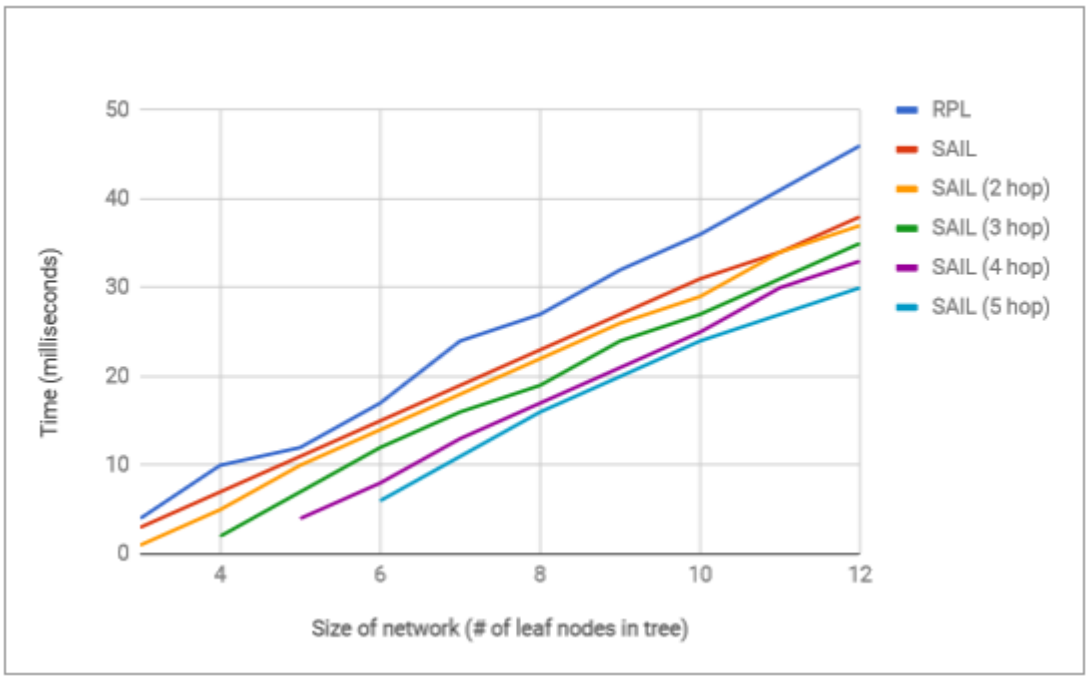

Figure 7.5: Delay traffic from end leaf node to another for small-medium size networks

in their routing tables can forward a packet across their sibling, and need not waste resources forwarding packets up towards the root.

7.2 Large Networks

7.2.1 Convergence Time

Comparing convergence times for larger networks, we observe the same results as those produced in small to medium network sizes. The additional overhead signaling SAIL incurs to ensure bi-directionality and interval-label assignments increases convergence time, but the difference compared to RPL is not at a significant scale to discredit SAIL.

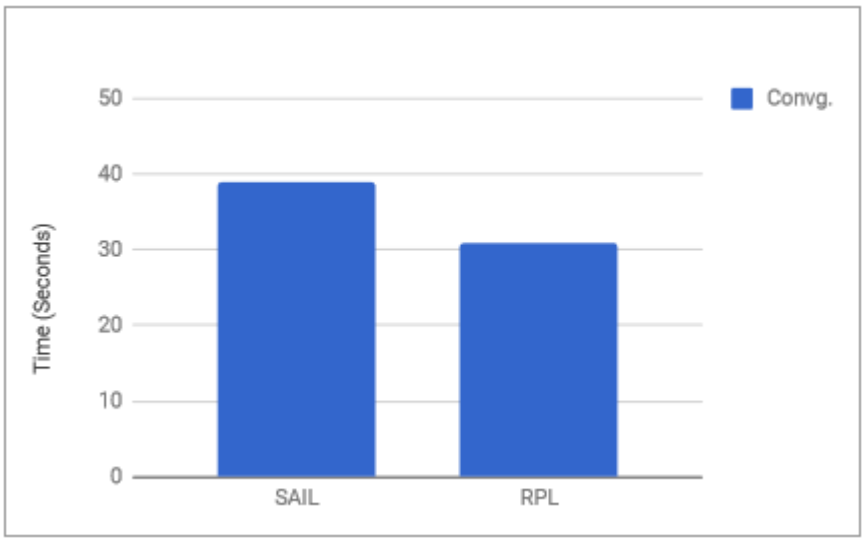

Figure 7.6: Time until all nodes have joined the network for large networks

7.2.2 Routing Table Size

Figure 7.7 shows the average number of entries in a node's routing table for nodes running both RPL and SAIL. The average storage overhead for route state remains unchanged from small to medium deployments, and we notice that RPL incurs over 900% more overhead than SAIL. For hotspot nodes, RPL incurs over 5200% more storage over SAIL shown in the results of Figure 7.8! RPL routing tables grow linearly as the number of nodes in the network increases as shown in the previous and current section. These results emphasize the core motivation in SAIL's compact routing approach on how the size of its routing table remains unchanged with different network sizes.

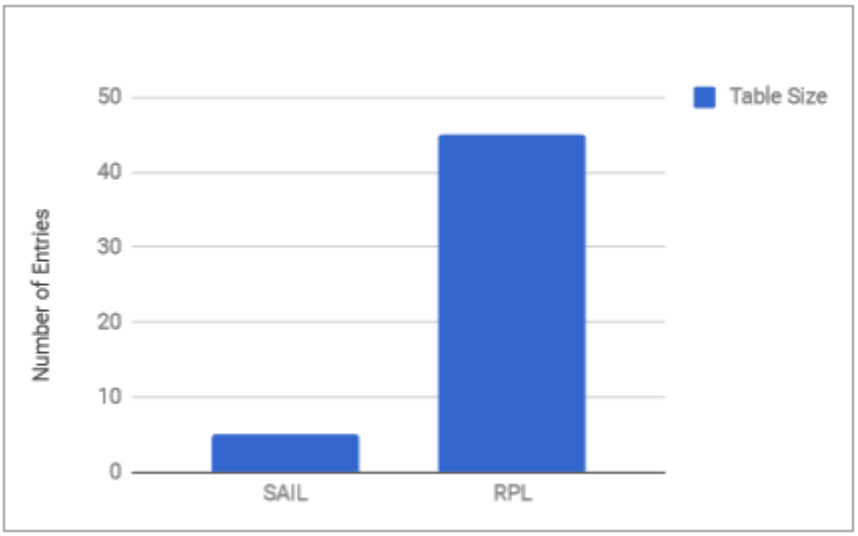

Figure 7.7: Average routing table size of all nodes for large networks

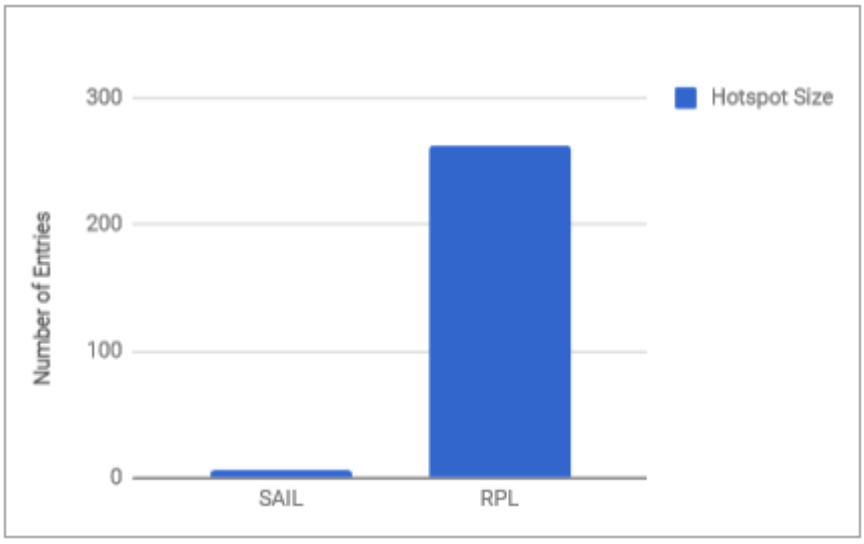

Figure 7.8: Average routing table size of hotspot nodes for large networks

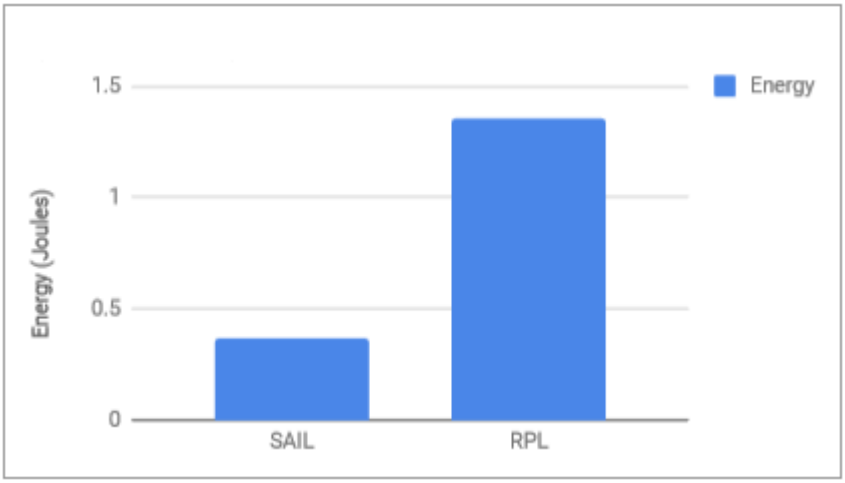

Figure 7.9: Average energy consumption to construct network connectivity for large deployments

7.2.3 Energy

We further examine the average energy cost that nodes in both routing protocols consume in order to construct the routing trees associated in a large deployment. We've shown that RPL converges at a faster time than SAIL, but the DIO messages used to propagate leaf location and route information to parent nodes consumes more energy depending on the rate of transmission. RPL nodes transmitting DIO messages at a one second interval as stated in RFC 6550 [3] consumes triple the amount of energy a SAIL node would require in order to build its entire routing table.

For various-sized deployments of wireless networks, SAIL outperforms RPL in conserving energy and minimizing the storage overhead of routing information. For point to point communication, we have shown that SAIL is able to compute shorter paths and provide shorter traffic delays by allowing nodes to forward packets through

sibling nodes when additional neighbor interval information is known. In a scenario where networks sizes can increase exponentially as nodes join the network, SAIL scales and does not need to inject any additional information about new nodes into the network since interval labels are assigned in a hierarchical manner, unlike RPL in scenarios where DAO's must be sent upwards. Although RPL routes converge at a shorter time, we show that SAIL benefits from the additional delay by allowing nodes to test bi-directionality before joining the network through a parent node.

Chapter 8

Conclusion

WSN deployments require alternative approaches to routing from the conventional routing methods proposed in computer networks. IoT deployments involve of dynamic behavior where a node's membership can constantly change in a deployed network. These nodes can be heavily energy resource constrained, requiring routing techniques that can utilize as minimal resources as possible in order to build the necessary routing information. Compact routing through continuous interval-labels is an alternative approach to routing in IoT deployments, having nodes identified with intervals rather than a single destination address. This routing approach allows routing tables to grow linearly with the number of neighbors a node contains rather than the number of nodes in the entire network. This paper focuses on presenting the design and evaluation of SAIL, Switching with Adaptive Interval Labels, that utilizes compact routing for IoT networks. We have shown through our results that this protocol provides optimal storage overhead and energy conservation in comparison with the standard IoT

routing protocol, RPL.

For our future work, we point out areas in our protocol design that require further development and investigation:

- Design and implement the operation of having of multiple root labeling nodes.
- Design and implement in simulation a controller-based scenario of SAIL instead of peer-to-peer signaling.
- In a peer-to-peer scenario, propose different methods on how to partition IPv6 identifiers into intervals that can further scale and become independent from the degree of the DAG tree. This will allow our simulations tests for networks that span over a larger area.
- Integrating name resolution protocol to map a device (through MAC address or content naming) with interval label.
- An evaluation of the protocol in a dynamic scenario where nodes and links are constantly failing. This will have to be compared against RPL when its public ns-3 distribution is released.

Bibliography

[1] Cyw43438 single-chip ieee 802.11ac b/g/n mac/baseband/radio with integrated bluetooth 4.1 and fm receiver. http://www.cypress.com/file/298076/download. Accessed: 2018-22-3.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[3] R. Alexander, A. Brandt, JP. Vasseur, J. Hui, K. Pister, P. Thubert, P. Levis, R. Struik, R. Kelsey, and T. Winter. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, March 2012.

[4] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[5] P. Baran. On distributed communications networks. *IEEE Transactions of the Professional Technical Group on Communications Systems*, January 1964.

[6] M. F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and B. Mathieu. A survey

of naming and routing in information-centric networks. *IEEE Communications Magazine*, 50(12):44–53, December 2012.

[7] S. Basagni, I. Chlamtac, V. Syrotiuk, and B. Woodward. A distance routing effect algorithm for mobility (dream). In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '98, pages 76–84, New York, NY, USA, 1998. ACM.

[8] 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)*, pages 365–372, Oct 2011.

[9] K. Dar, M. Bakhouya, J. Gaber, and M. Wack. Evaluating geographic routing protocols for vehicular ad-hoc networks. In *2009 3rd International Conference on New Technologies, Mobility and Security*, pages 1–5, Dec 2009.

[10] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 329–342, Berkeley, CA, USA, 2005. USENIX Association.

[11] J. J. Garcia-Luna-Aceves. Towards loop-free forwarding of anonymous internet

datagrams that enforce provenance. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2016.

[12] J. J. Garcia-Luna-Aceves and D. Sampath. Scalable integrated routing using prefix labels and distributed hash tables for manets. In *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 188–198, Oct 2009.

[13] Y. K. R. Govindan, B. Karp, and S. Shenker. Lazy cross-link removal for geographic routing. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 112–124, New York, NY, USA, 2006. ACM.

[14] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, February 2003.

[15] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Proceedings. IEEE International Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century.*, pages 62–68, 2001.

[16] B. Karp and H. T. Kung. Gpsr: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, MobiCom '00, pages 243–254, New York, NY, USA, 2000. ACM.

[17] J. Leeuwen and R.B. Tan. Compact routing methods: A survey. Technical Re-

port UU-CS-1995-05, Department of Information and Computing Sciences, Utrecht University, 1995.

[18] P. Levis, A. Tavakoli, and S. Dawson-Haggerty. Overview of existing routing protocols for low power and lossy networks. IETF ROLL Working Group Internet-Draft, 2009.

[19] J. Moy. Ospf version 2. RFC 2328, Internet Engineering Task Force (IETF), April 1998.

[20] A. Oliveira and T. Vazão. Low-power and lossy networks under mobility. *Comput. Netw.*, 107(P2):339–352, October 2016.

[21] C. Perkins and E. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 90–, Washington, DC, USA, 1999. IEEE Computer Society.

[22] T. Tsvetkov and A. Klein. Rpl: Ipv6 routing protocol for low power and lossy networks. 2013.

[23] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52(12):2292–2330, August 2008.

[24] H. Zhang, Y. Wen, H. Xie, and N. Yu. A survey on distributed hash table (dht): Theory, platforms, and applications. 2013.

Appendix A

SAIL ns-3 Pseudo-code for Main Operations

Algorithm 1 Receiving and handling various SAIL packets

```
1:  procedure RecvPacket(packet)
2:      typeHeader ← packet.getType()
3:      if typeHeader == HELLO then
4:          ProcessHello(packet)
5:      else if typeHeader == Request then
6:          ProcessRequest(packet)
7:      else if typeHeader == Update then
8:          ProcessUpdate(packet)
9:      else if typeHeader == Invalidate then
10:         ProcessInvalid(packet)
11:     else
12:         Drop packet
```

Algorithm 2 Processing a HELLO packet.

```
1: procedure ProcessHello(packet)
2:     if state == IDLE then
3:         Parse contents of packet
4:         if packet.rank != ROOT and packet.flags == regularHELLO then
5:             for every interval announced in packet do
6:                 CheckRouteExists(packet.helloInterval[i])
7:                 if packet.helloInterval[i] is not found then
8:                     AddGlobalInterval(packet.helloInterval[i])
9:                     if packet.rank <parentRank then
10:                        UpdateRoute(packet.helloInterval[i], PARENT)
11:                        if joinSAIL == true then
12:                            for every interval currently assigned do
13:                                Invalidate and inform children nodes
14:                        RequestToJoin( any of packet.helloInterval)
15:                    else if packet.rank == parentRank then
16:                        UpdateRoute(packet.helloInterval[i], PARENT)
17:                        RequestToJoin( any of packet.helloInterval)
18:                    else if packet.rank >parentRank then
19:                        UpdateRoute(packet.helloInterval[i], CHILD)
20:                    else if packet.rank == parentRank then
21:                        UpdateRoute(packet.helloInterval[i], SIBLING)
22:                else
23:                    Update expiration timer for route entry packet.helloInterval[i]
24:        else
25:            CheckRouteExists(packet.helloInterval[i])
26:            if packet.helloInterval[i] is not found then
27:                AddGlobalInterval(packet.helloInterval[i])
28:                RequestToJoin(packet.helloInterval[i])
29:                UpdateRoute(packet.helloInterval[i], PARENT)
30:            else
31:                Update expiration timer for route entry packet.helloInterval[i]
```

Algorithm 3 Sending a request packet

```
1: procedure RequestToJoin(requestedInterval)
2:     if number of intervals assigned from this parent <intervalThreshold then
3:         state = IN_REQUEST
4:         Create requestHeader
5:         if status == ROUTER then
6:             flags = ROUTER_REQUEST
7:         else if status == NON_ROUTER then
8:             flags = LEAF_REQUEST
9:         requestHeader ← local_identifier
10:        requestHeader ← requestedInterval
11:        requestHeader ← flags
12:        requestPacket ← requestHeader
13:        Send requestHeader to neighbor that sent initial HELLO message.
```

Algorithm 4 Processing a request packet.

```
1: procedure ProcessRequest(packet)
2:     Parse contents of packet
3:     if root == true or joinedSAIL == true then
4:         Create updateHeader
5:         if packet.flags == ROUTER_REQUEST then
6:             subInterval ← Partition an interval range contained within
   packet.requestedInterval
7:             updateFlags = ROUTER_UPDATE
8:             updateHeader ← subInterval
9:             updateHeader ← updateFlags
10:            updateHeader ← rank + 1
11:        else if packet.flags == LEAF_REQUEST then
12:            singleLabel ← Assign a label from packet.requestedInterval
13:            updateFlags = LEAF_UPDATE
14:            updateHeader ← singleInterval
15:            updateHeader ← updateFlags
16:            updateHeader ← rank + 1
17:        Create updatePacket
18:        updatePacket ← updateHeader
19:        Send updatePacket back to requester
```

Algorithm 5 Processing an update packet.

```
1: procedure ProcessUpdate(packet)
2:     Parse contents of packet
3:     if number of intervals assigned from this parent <intervalThreshold then
4:         myRank = packet.rank
5:         Assign packet.interval as an identifier and configure new socket
6:         Increment number of intervals assigned from this neighboring parent.
7:         if packet.flags == ROUTER_UPDATE then
8:             Configure next sub-interval available that can be assigned to new children
   nodes
9:         joinSAIL = true
10:        state = IDLE
11:        Begin broadcasting HELLO messages.
```

Algorithm 6 Check if a route exists to an interval label

```
1: procedure LookUpInterval(intervalDest)
2:     for each entry rte in the routing table where rte.routeState != PARENT do
3:         if (intervalDest <rte.lastLabel and rte.firstLabel <intervalDest) or inter-
   valDest == rte.label then
4:             return rte
5:     if no route consisting interval label found then
6:         for each entry rte2 in the routing table where rte2.state == PARENT do
7:             if (intervalDest <rte2.lastLabel and rte2.firstLabel <intervalDest) or in-
   tervalDest == rte2.label then
8:                 return rte2
9:     if still no entry consisting interval label found then
10:        if node is not root then
11:            Foward packet to any parent node
12:    else
13:        Drop packet
```

Algorithm 7 Partitioning a sub-interval or label to a child node

```
1: procedure IntervalPartition(interval, flags)
2:     if flags == ROUTER_REQUEST then
3:         index = rank + lastPrefixByteIndex
4:         if nextAvailableInterval[index] + 1 >0xFF then
5:             No more sub-intervals can be assigned from this interval, return
6:         else
7:             newInterval = nextAvailableInterval[index]+1
8:             return newInterval
9:     else if flags == LEAF_REQUEST then
10:        if nextAvailableInterval[15] + 1 >0xFF then
11:            No more labels can be assigned from this interval, return
12:        else
13:            newLabel = nextAvailableInterval[15]+1
14:            return newLabel
15:
```