2017-06

Achieving sink node anonymity in tactical wireless sensor networks using a reactive routing protocol

Haakensen, Thomas J.

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ACHIEVING SINK NODE ANONYMITY IN TACTICAL WIRELESS SENSOR NETWORKS USING A REACTIVE ROUTING PROTOCOL

by

Thomas J. Haakensen

June 2017

Thesis Advisor: Preetha Thulasiraman
Second Reader: Murali Tummala

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE ACHIEVING SINK NODE ANONYMITY IN TACTICAL WIRELESS SENSOR NETWORKS USING A REACTIVE ROUTING PROTOCOL	**5. FUNDING NUMBERS** W7B46
6. AUTHOR(S) Thomas J. Haakensen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Marine Corps Systems Command and Naval Research Program	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

A wireless sensor network (WSN) collects and routes information from the environment to an aggregation point, known as a sink node. The sink node processes the information or acts as a gateway to forward information to another network. Due to its essential role in the network, the sink node is a high priority target for an attacker who wishes to disable a WSN. In this thesis, we focus on the mitigation of sink-node vulnerability in a WSN. Specifically, in this thesis we study the issue of protecting the sink node through anonymity techniques. In particular, we use a technique known as *k*-anonymity. To achieve *k*-anonymity, we use a specific routing protocol designed to work within the constraints of WSN communication protocols, specifically IEEE 802.15.4. We use and modify the Lightweight Ad hoc On-Demand – Next Generation (LOADng) reactive-routing protocol to achieve anonymity. This modified protocol prevents an attacker from identifying the sink node without adding significant complexity to the regular sensor nodes. We simulate the modified LOADng protocol using a custom-designed simulator in MATLAB. We demonstrate the effectiveness of our protocol and also show some of the performance tradeoffs that come with this method.

14. SUBJECT TERMS Sink node anonymity, base station anonymity, Wireless Sensor Networks (WSN), Mobile Ad hoc Network (MANET), Lightweight Ad hoc On-Demand – Next Generation (LOADng)	**15. NUMBER OF PAGES** 95
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89) Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

ACHIEVING SINK NODE ANONYMITY IN TACTICAL WIRELESS SENSOR NETWORKS USING A REACTIVE ROUTING PROTOCOL

Thomas J. Haakensen
Major, United States Marine Corps
B.S., University of Minnesota Duluth, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2017

Approved by: Preetha Thulasiraman, Ph.D.
Thesis Advisor

Murali Tummala, Ph.D.
Second Reader

R. Clark Robertson, Ph.D.
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A wireless sensor network (WSN) collects and routes information from the environment to an aggregation point, known as a sink node. The sink node processes the information or acts as a gateway to forward information to another network. Due to its essential role in the network, the sink node is a high priority target for an attacker who wishes to disable a WSN. In this thesis, we focus on the mitigation of sink-node vulnerability in a WSN. Specifically, in this thesis we study the issue of protecting the sink node through anonymity techniques. In particular, we use a technique known as *k*-anonymity. To achieve *k*-anonymity, we use a specific routing protocol designed to work within the constraints of WSN communication protocols, specifically IEEE 802.15.4. We use and modify the Lightweight Ad hoc On-Demand – Next Generation (LOADng) reactive-routing protocol to achieve anonymity. This modified protocol prevents an attacker from identifying the sink node without adding significant complexity to the regular sensor nodes. We simulate the modified LOADng protocol using a custom-designed simulator in MATLAB. We demonstrate the effectiveness of our protocol and also show some of the performance tradeoffs that come with this method.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

6LoWPAN	IPV6 over low-power wireless personal area network
ADC	Analog-to-digital converter
AES	Advanced Encryption Standard
ALERT	Anonymous Location-Based Efficient Routing Protocol
AODV	Ad hoc On-Demand Distance Vector
AOI	Area of interest
BEB	Binary Exponential Back-off
BLAST	Base-station Location Anonymity and Security Technique
CH	Cluster Head
COTS	Commercial off the shelf
CSMA-CA	Carrier-Sense Multiple Access with Collision Avoidance
DOD	Department of Defense
GPS	Global Positioning System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv6	Internet Protocol version 6
ITU	International Telecommunication Union
LLN	Low-Power Lossy Network
LOAD	6LoWPAN Ad Hoc On-demand Distance Vector Routing
LOADng	Lightweight Ad hoc On-Demand – Next Generation
LoWPAN	low-power wireless personal area network
LR-WPAN	Low-Rate Wireless Personal Area Network
MANET	Mobile Ad hoc Networks
MCRP	Marine Corps Reference Publication
MTU	Maximum Transmit Unit
OSPF	Open Shortest Path First
PDR	packet delivery ratio
RF	radio frequency

RFC	Request for Comments
RIP	Routing Information Protocol
RPL	Routing Protocol for Low-Power and Lossy Networks
RREP	Route Reply
RREP_ACK	Route Reply Acknowledge
RREQ	Route Request
RERR	Route Error
RX-TX	receive-transmit
SMMS	Sensor Mobile Monitoring System
TLV	Type-Length-Value
TRSS	Tactical Remote Sensor System
USMC	United States Marine Corps
WSN	wireless sensor network

ACKNOWLEDGMENTS

First, I would like to thank my wife, Miki, for her love, support and patience throughout the long hours of studying, research, and writing over these past two years. I would also like to thank my son, Ian, for being as patient and understanding as a 6-to 8-year-old boy could be when his dad cannot always play with him.

I would like to thank my thesis advisor, Professor Preetha Thulasiraman, for providing the guidance, knowledge, and direction needed to complete this process.

Finally, I would like to thank all of the electrical engineering professors at Naval Postgraduate School who have given me the knowledge to be successful.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Advances in the miniaturization of integrated circuits, transmitters, and sensing devices have facilitated the creation of small-yet-capable remote wireless sensors that can be deployed over large areas inexpensively. This has a wide range of applications in both military and civilian functions, including environmental monitoring, presence/intrusion detection, ranging, imaging, and noise detection [1]. The versatility of remote sensors has made them especially appealing for use in military applications.

The *Marine Corps Reference Publication* (MCRP) 2–10A.5 on remote sensor operations [2] details the ways and means in which sensor operations are conducted in the United States Marine Corps (USMC). It states that remote sensors provide an economical means to expand the commander's situational awareness on the battlefield by deploying a persistent presence to monitor an area-of-interest (AOI) without having to employ troops in dangerous and hostile areas. They act as a force multiplier by reducing the requirement for personnel and the associated risks when conducting reconnaissance and surveillance operations. It is because of these advantages that the use of sensors on the battlefield continues to increase.

In [1], the authors performed a survey of 13 papers covering military applications of wireless sensor networks and identified a variety of current and future uses including soldier detection and tracking; perimeter protection; chemical, biological, and explosive vapor detection; acoustic sensing; and gunshot detection and localization. This wide range of applications makes wireless sensor networks a versatile tool that will grow in importance on future battlefields.

A. WIRELESS SENSOR NETWORKS

A wireless sensor network (WSN) is a system of specialized devices, or nodes, which communicate data from sensor inputs through a wireless medium to a base station, which we refer to as the sink node. These devices are generally resource constrained, meaning they have enough computational and transmit power to accomplish their task

while using as little power as possible. This is especially important for sensors that rely on a battery for power and are expected to function for long periods without replacement.

Sensor nodes are always equipped with a radio frequency (RF) transceiver, a transducer for sensing, a microcontroller, and a power supply that is usually a battery [3]. A high-level diagram of a typical wireless sensor node architecture is shown in Figure 1.

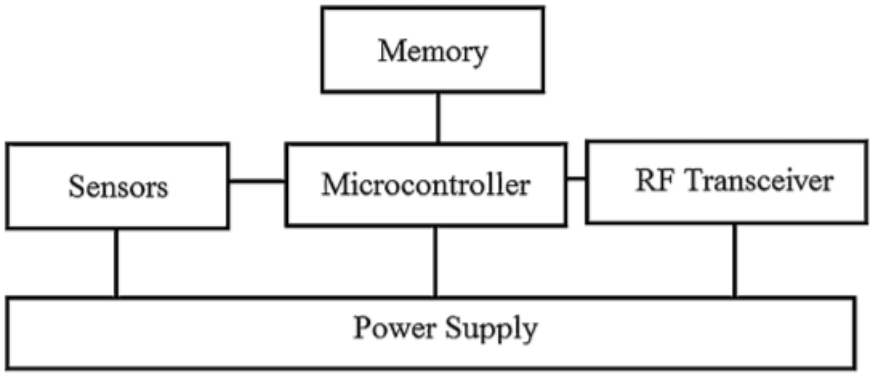

Figure 1. Typical Sensor Node Architecture. Source: [3].

The transducer acts as the sensor, which collects input from the environment and converts that input to analog electrical signals. Those signals are converted to digital signals through an analog-to-digital converter (ADC) and are sent to the microcontroller for processing. The microcontroller processes those inputs, makes decisions based on the inputs, and acts on them accordingly. The RF transceiver sends and receives information to and from other nodes in the network as instructed by the microcontroller. Sensor nodes may also have other capabilities, such as Global Positioning Systems (GPS), based on the information requirements.

A WSN contains multiple nodes, which are each connected to at least one other node utilizing a wireless protocol. The sensor nodes transmit their information through the network based on the specific protocols that are implemented. Nodes may be designed to perform any combination of sensing, data relaying, or external network data communication functions [3]. A node designed for sensing is only able to act as a sensor and has to transmit its information to a relaying node to be forwarded through the network. The relaying nodes act as routers and forward traffic through the network based

on the routing protocol implemented. The sink node is the central node to which data is sent by other sensor nodes for processing. It acts as the gateway node between the WSN and an outside network. An example of a typical WSN depicting a sensor field and a sink node acting as a gateway is shown in Figure 2.

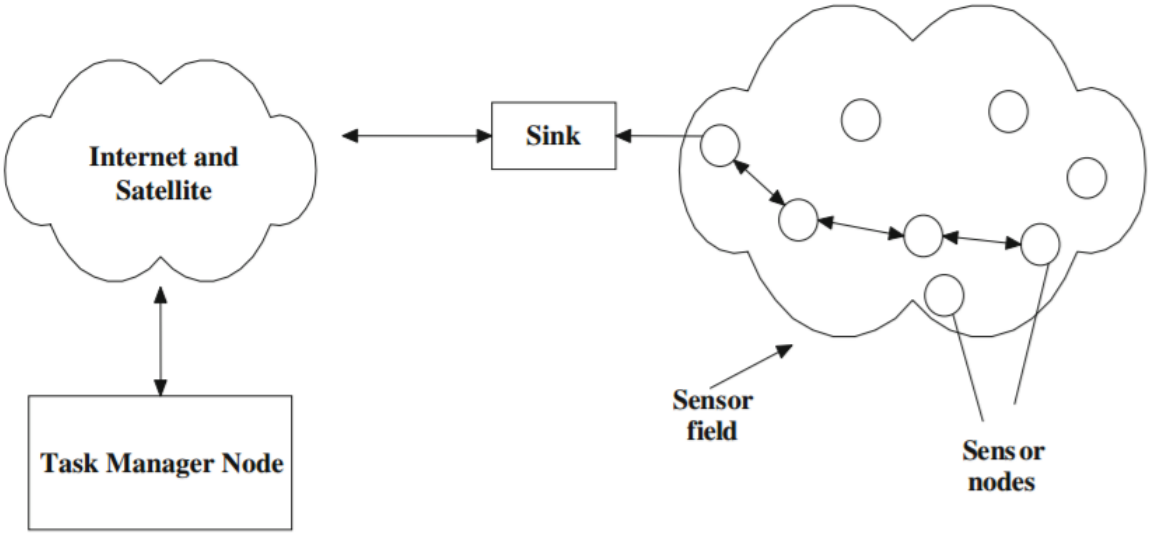

Figure 2. Structure of a Typical Wireless Sensor Network. Source: [3].

The open nature of the WSN environment allows nodes to be easily compromised, leading to several security problems. In particular, the sink node, which is the aggregation point of all network data, is considered a single point of failure. An attack that compromises the sink node results in the network becoming isolated and non-functioning. This makes the sink node the priority node for an adversary to locate and disable.

B. STANDARDS FOR LOW-POWER COMMUNICATIONS

The restricted energy and computational requirements of battery-powered remote sensor nodes and the unreliable nature of Low-Power Lossy Networks (LLNs) prompted the development of new communications protocols. Existing standards, such as Institute of Electrical and Electronics Engineers (IEEE) 802.11, were not designed for the requirements of such restricted devices. Additionally, existing routing protocols were too

resource demanding for devices with such limited memory and computing power. LLNs use the IEEE 802.15.4 standard, which is a data link and physical layer protocol that provides communications between low-power devices [4]. Internet Protocol version 6 (IPv6) over IEEE 802.15.4 low-power wireless personal area networks (6LoWPAN) was later developed to address interoperability between LLNs and IPv6-enabled networks. 6LoWPAN is an open networking standard that provides compatibility between existing Internet-connected devices and low-power WSNs. It allows for IP packets to be carried within IEEE 802.15.4 link layer frames by reducing the overhead associated with the IPv6 protocol.

C. RESEARCH MOTIVATIONS AND OBJECTIVES

The objective of this research is to develop a method to provide anonymity to the sink node in a WSN while incurring minimal computational overhead to the sensor nodes in the network. This allows the use of inexpensive, resource-constrained, low-power wireless sensor motes, which reduces the cost of procurement and deployment. Deployment of large numbers of these devices over large areas greatly enhances the situational awareness of units in the field without being cost-prohibitive. The devices can easily be viewed as expendable and can be air-dropped into hostile or contaminated areas without fear of losing them. This makes the need to hide the sink node even more critical, as the network is more exposed to a hostile attacker in a situation where the network was deployed far forward of friendly lines.

The USMC has employed remote sensors on the battlefield since 1967 during the Vietnam War [2]. It currently employs the AN/GSQ-261 Tactical Remote Sensor System (TRSS) for remote sensor operations within an AOI. The sensors in this system are capable of detecting the presence and movement of vehicles and personnel and can operate continuously for 30 days on internal batteries [2]. The data from the sensors is relayed back to the AN/MSC-77 Sensor Mobile Monitoring System (SMMS) for processing. The SMMS acts as the sink node for the WSN and is critical to the function of the entire network.

While the AN/GSQ-261 is a capable system that enhances the capabilities of the USMC, it is large and requires sensor nodes to be manually placed, potentially putting personnel at risk if placed forward of friendly lines. There is a potential for smaller and more resource-constrained devices, which can easily be air dropped into an area or placed in large numbers by units on patrol. These devices can provide a similar remote sensing capability as the AN/GSQ-261's sensor nodes while being much more cost effective and expendable. There are a wide range of commercial-off-the-shelf (COTS) solutions that can economically provide this capability with minor modifications [5]. These sensors require the use of more power-friendly communications protocols while still maintaining security and their ability to transmit data back to the monitoring station.

Power-friendly communication protocols such as 6LoWPAN and IEEE 802.15.4 are prime candidates for maintaining connectivity in WSNs. In this thesis research, we focus on the mitigation of sink node vulnerability in a WSN used at the tactical edge by the U.S. Marine Corps. Specifically, in this thesis research, we study the issue of protecting the sink node through anonymity techniques. In particular, we use a technique known as *k*-anonymity to obfuscate the actual sink node. To achieve *k*-anonymity, we use a specific routing protocol designed to work within the constraints of IEEE 802.15.4 and 6LoWPAN. For this thesis research, we modify the Lightweight Ad hoc On-Demand – Next Generation (LOADng) reactive, or on-demand, routing protocol.

D. THESIS CONTRIBUTIONS

To achieve the above stated objectives, we develop a modification to the LOADng routing protocol that accomplishes sink-node anonymity while adding minimal computational overhead to the resource constrained sensor nodes.

The contributions of this thesis are as follows:

- Development of a modified LOADng routing protocol that provides *k*-anonymity to the sink node while limiting the computational overhead for the sensor nodes.
- Simulation of the modified routing algorithm to measure and quantify anonymity and performance versus the standard LOADng protocol.

- Measurement of the performance of the modified routing algorithm compared to the standard LOADng protocol for average route length, latency, power consumption, and packet delivery ratio (PDR).

During our literature search, we found no other research that merges *k*-anonymity for the sink node with reactive routing protocols.

E. THESIS ORGANIZATION

The remainder of this thesis is organized as follows. In Chapter II, we cover relevant background information and discuss some of the previous research done on this topic. The method used to determine anonymity as well as how sink node anonymity is achieved are discussed in Chapter III. In Chapter IV, we cover how the experiment was designed, implemented, and run to compute specific performance metrics. In Chapter V, the results of the simulation are presented and discussed in relation to the metrics measured. In Chapter VI, we conclude this thesis and discuss future work topics. All code for the implemented simulation is included in the appendix.

F. CHAPTER SUMMARY

In this chapter, we introduced to the concept of WSNs and their application in the Department of Defense (DOD). Research motivations and the objectives of this thesis were discussed, followed by a brief outline of the contributions of this thesis.

II. BACKGROUND AND RELATED WORK

In this chapter, we discuss some of the protocols that are developed for different layers of the WSN protocol stack as well as some existing research into sink node anonymity. This lays the informational foundation for the later discussions of our implemented protocol and its application to sink node anonymity in WSNs.

A. IEEE 802.15.4 STANDARD OVERVIEW

The IEEE chartered the IEEE 802.15 Task Group 4 to address the need for a wireless standard for low complexity devices that require low data rate and low power consumption. This led to the publication of the IEEE 802.15.4-2003 standard in 2003, which was superseded by the IEEE 802.15.4-2006 standard. The standard contains the following features [4]:

- Data rates of 250 kbps, 40 kbps, and 20 kbps.
- Two addressing modes; 16-bit short and 64-bit IEEE addressing.
- Support for critical latency devices, such as joysticks.
- Carrier-Sense Multiple-Access with Collision Avoidance (CSMA/CA) channel access.
- Automatic network establishment by the coordinator.
- Handshaking protocol for transfer reliability.
- Power management to ensure low power consumption.
- Sixteen channels in the 2.4-GHz ISM band, ten channels in the 915-MHz band, and one channel in the 868-MHz band.

The IEEE 802.15.4 standard is designed to provide Low-Rate Wireless Personal Area Network (LR-WPAN) capabilities for devices with constrained power and computation resources that only require low data throughput [3]. The IEEE 802.15.4 standard outlines the specification for the physical and MAC layers of the WSN protocol stack. Other protocols must be used to implement the higher layer functions of the WSN protocol stack.

We are specifically interested in the security implementation at the MAC layer, which is a critical component of protecting the sink node. The IEEE 820.15.4 standard designates MAC layer encryption using Advanced Encryption Standard (AES)-128 with 128-bit symmetric keys as specified in FIPS Pub 197 [4]. When implemented in accordance with FIPS 140–2, this encryption meets the DOD requirements for communications transmission security [6].

B. 6LOWPAN

6LoWPAN was developed by the Internet Engineering Task Force (IETF) as a means to enable IPv6 to be used on low-power IEEE 802.15.4 WSNs [3]. This enables sensors to communicate directly on the Internet without having to utilize a gateway to translate between protocols. The challenge for designers of this protocol was fitting the IPv6 header, which is 40 bytes, within an IEEE 802.15.4 frame, which is limited to 127 bytes total. The 25-byte MAC frame header and optional 21-byte encryption header leave just 81 bytes for upper layer headers and payload data [7].

6LoWPAN adds an adaptation layer between the MAC and network layers to provide header compression in IEEE 802.15.4 networks and fragmentation and reassembly when transitioning between networks which use the standard IPv6 1280-byte maximum transmit unit (MTU) [8]. The adaptation layer compresses the IPv6 header to two, 12, or 20 bytes depending on the node's knowledge of its destination. Fragmentation and mesh headers of four to five bytes and five to 17 bytes, respectively, are added as needed to support fragmentation of larger IPv6 packets and multi-hop routing [8]. This dramatically reduces the header size of IPv6, allowing its use in the restricted IEEE 802.15.4 frame.

C. REACTIVE VERSUS PROACTIVE ROUTING PROTOCOLS

Reactive, or on-demand, routing protocols differ from proactive routing protocols, such as Open Shortest Path First (OSPF) and Routing Information Protocol (RIP), in that nodes only determine a route to a destination when they need to send data. There are no periodic control packets sent throughout the network for route maintenance, and the nodes do not maintain large routing tables with a full picture of the network as is required

in a proactive protocol. Each node only maintains routes to nodes for which it needs to send data. Due to these characteristics, they are well suited to run on devices that are restricted in computational power and memory. According to Clausen et al. [9], we see that reactive protocols are preferable to proactive protocols under the following conditions:

- Few concurrent traffic flows in the network (i.e., traffic flows only between few sources and destinations);
- Low data traffic overall, and, therefore, the traffic load from periodic signaling (for proactive protocols) is greater than the traffic load from flooding route requests (for reactive protocols);
- State requirements on the router are very stringent; i.e., it is beneficial to store only few routes on a router.

Reactive routing protocols have an advantage over proactive protocols in low-traffic Mobile Ad Hoc Networks (MANETs) because they tend to use less power in many scenarios [10], [11]. This is important when using battery-powered sensors that need to function for long periods on their own internal power. This is especially true in sensor networks that may sit for long periods before a sensor is triggered and needs to send data. In a proactive protocol, route updates are sent out periodically even if there is no event to trigger a data transmission, using more power each time. A reactive protocol only attempts to determine a route when it needs to send data to another node, conserving power.

D. LIGHTWEIGHT AD HOC ON-DEMAND—NEXT GENERATION (LOADng) ROUTING PROTOCOL

The LOADng routing protocol is a reactive routing protocol developed for use in MANETs and is currently a draft at the IETF Network Working Group [9]. It was derived from the Ad hoc On-Demand Distance Vector (AODV) routing protocol, which was originally published in 2003 in Request for Comments (RFC) 3561 by the IETF.

6LoWPAN Ad Hoc On-demand Distance Vector Routing (LOAD) was the first derivative of AODV developed by the 6LoWPAN working group, but development was suspended while the group worked out adapting IPv6 for IEEE 802.15.4 [12]. LOAD was

designed as a layer-2 mesh under protocol and was designated as the routing protocol for utility metering networks by the International Telecommunication Union (ITU) in recommendation ITU-T G.9903. Despite the suspension, development of AODV derivatives continued, and LOADng was created as an improvement to LOAD that also offered the ability to work as a layer-3 route over protocol [12]. The ITU superseded LOAD with LOADng in the recommendation ITU-T G.9903 in May 2013.

1. Overview of LOADng

As the name implies, LOADng is a lightweight protocol designed for use in devices that are resource constrained. It eliminates some of the functions of AODV while maintaining the core ability to provide end-to-end routing efficiently. In AODV, each node maintains a precursor list, which has the IP addresses of all other nodes that it thinks will use it as a next hop to all destinations. LOADng does not have precursor lists and only cares about its next hop to a destination, reducing the memory requirement in the sensor nodes.

AODV allows an intermediate node to respond to a route request (RREQ) if it has a route to the destination. LOADng only allows the destination to respond to RREQs, which serves to lower the amount of network traffic and simplifies the protocol. This tends to further highlight the sink node since all route replies (RREPs) come from the sink node, assuming the traffic from the sensors is all destined for the sink node.

LOADng allows for protocol extensions through the use of Type-Length-Value (TLV) elements, making it possible to provide additional functionality to the protocol easily. The ability to modify LOADng is a key characteristic and one of the main reasons it was chosen for the application discussed in this thesis.

2. Operation of the LOADng Protocol

As a reactive routing protocol, LOADng accomplishes route discovery through the use of RREQ, RREP, and RREP acknowledge (RREP_ACK) packets. In the following sections, we detail the operation of the route-discovery process.

a. Route Requests

When a node has data it needs to send to another node, the source node first determines whether it has a route-tuple to the destination node in its routing set. Each tuple contains the next-hop node address and the routing metric used to obtain the route towards the destination [9]. If there is a route, the source node simply unicasts a data packet to the destination node. If there is no route in the routing set, the source node generates a RREQ packet and floods the RREQ packet to its neighbors. The RREQ packet is flooded through the entire network until all nodes have received the packet or the packet hop limit is exceeded. When each node receives the RREQ, the node updates its routing set by updating an existing route-tuple or adding a new tuple with a route to the source address in the RREQ. This generates the reverse route, as each node receiving the RREQ generates a route-tuple to the source node. The node then checks the destination address to determine if it is the destination. If it is not the destination, the node continues flooding the RREQ in accordance with the flooding scheme implemented. LOADng supports optimized flooding, which reduces overhead when compared to classical flooding.

b. Route Replies

If the node is the destination, it generates a RREP message and unicasts it back to the source via the reverse route. As each node receives the RREP, it similarly updates or creates a tuple in its routing set with a route to the RREP originator. This creates the forward route toward the RREP source node. The node then determines if it is the destination for the RREP. If it is not the destination, the node forwards the RREP packet based on the reverse route generated in the RREQ flooding that was executed previously. An example of the RREQ and RREP process for the LOADng protocol with the route numbers showing the hop count from the source is shown in Figure 3.

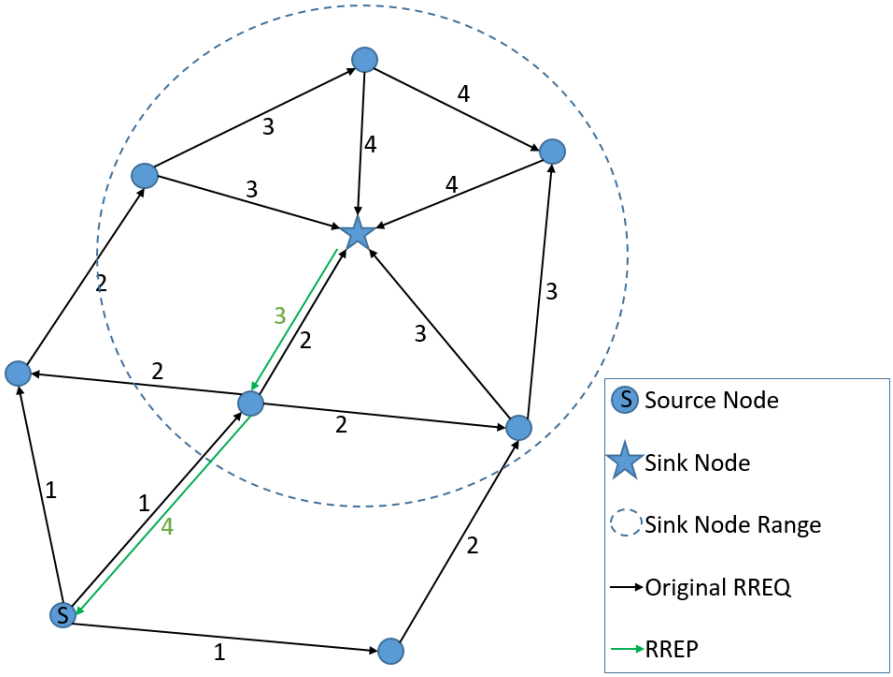

Figure 3. RREQ Flooding and RREP Unicast for LOADng Routing Protocol

c. Route Reply Acknowledgement

If it is the destination of the RREP, the node generates a RREP_ACK packet and unicasts it to the source of the RREP via the forward route. If the node has data to send, it then sends the data to the destination node via the same forward route.

d. Route Errors

If any node in the route determines that the route is broken at any time, it generates a Route Error (RERR) message and unicasts it back to the source node of the packet it was attempting to forward. Upon receipt of the RERR, the source node sends a new RREQ to establish a new forward route to the destination.

3. LOADng Performance Comparisons

LOADng was chosen for this research due to its low resource overhead for sensor nodes and its good performance compared to other WSN routing protocols. When compared to the Routing Protocol for Low-power and Lossy Networks (RPL), a proactive routing-distance vector protocol for MANETS, it was found that LOADng

showed significantly lower network overhead while maintaining a better PDR and average path length [12]. It did show a longer end-to-end delay due to the route discovery process required before it is able to send data. When compared to AODV, LOADng showed better PDR and significantly less routing overhead in multi-point to point scenarios, which is what we are interested in for this thesis research. It did show higher average end-to-end delay as node density increased, which is likely due to intermediate nodes not sending RREPs in LOADng [7].

E. APPROACHES TO PRIVACY IN WIRELESS SENSOR NETWORKS

There are many approaches to achieving sink node anonymity, each of which makes certain assumptions about the capabilities of the sensor nodes. Sink node anonymity is often a tradeoff between the level of anonymity and latency, power consumption, node complexity, and PDR. Our design does not assume that the nodes have prior knowledge of the network layout or the ability to determine their physical location. The only assumption is that the nodes know the address of the sink node and have been configured with the same symmetric key prior to deployment. Knowing this, it is useful to examine some other methods for achieving sink node anonymity

1. False Packet Injection

One of the simplest anonymity schemes involves injecting false packets into the network to deceive an adversary by making it difficult to recognize traffic patterns. Since there is generally more traffic from nodes closer to the sink node, the sink node's location can quickly be determined by a global observer. This method seeks to make this more difficult by increasing the traffic in areas that normally have lower traffic. This method can range from a packet being sent to every node in the network to more advanced methods that create false packets only in the areas that normally have lower traffic volume.

Deng et al. [13] proposed a method called fractal propagation to create false packets in a WSN. When a node hears that one of its neighbors is sending a packet to the sink node, that node generates a fake packet with a probability p_c. This packet is sent to a random neighbor node, which in turn forwards to another random neighbor node, continuing until k nodes have forwarded the packet. Additionally, another neighbor of the

original fake packet generating node creates a fake packet with a probability p_c and hop limit $k - 1$. This creates traffic away from the sink node and obscures the location of the sink node. This method creates a lot of unneeded traffic and may still not provide anonymity to the sink node because of the uncontrolled routing of the fake packets. The authors add more routing control to the scheme to create pockets of increased traffic, but this increases the complexity of the routing algorithm.

2. Deceptive Sink Nodes

One approach to hiding the location of the sink node is to route traffic to nodes that act as decoy sink nodes. This hides the sink node from a global adversary because the traffic never actually traverses the sink node. Base-station Location Anonymity and Security Technique (BLAST) [14] utilizes a ring of nodes around the sink node that act as endpoints for the traffic to the sink node. When a node sends traffic, it chooses one of the blast nodes to act as the endpoint, which then transmits the packet with a range that covers all of the other blast nodes and the sink node. This creates a ring of protective nodes around the sink node, masking the identity of the sink node. Each time a node transmits, it chooses a different blast node as the destination. This provides anonymity for the sink node from a global attacker but adds computational burden to the nodes and does not account for how the routes are learned. In addition, it assumes that all nodes know the address of all of the blast nodes and assumes routes have already been established. By not addressing the route discovery process, we see that this method leaves a potential vulnerability that can highlight the sink node.

3. Location-Aided Routing

Location-aided routing anonymity schemes require that the nodes know their physical location and that of the other nodes. This requires additional hardware, such as GPS or a dedicated location server, so each node knows its physical location. The Anonymous Location-Based Efficient Routing Protocol (ALERT) [15] protocol utilizes location-aided routing to provide anonymity to a destination node. When a node wants to transmit to a destination, the node partitions the physical space and chooses a random forwarder node in the neighboring partition and forwards the packet to it. This random

forwarder executes the same partitioning and forwarding. This continues until the partition with a random forwarder and the destination node contains a number of nodes *n* where $n \leq k$. At this point, the random forwarder broadcasts the packet, so all of the nodes, including the destination, can see the packet. This provides anonymity to the destination node because a global observer cannot distinguish the destination node from the other nodes in the partition. This scheme also has the advantage of being able to obscure the identity of any chosen destination; however, ALERT requires the use of a dedicated location server to tell the nodes where the other nodes are. This can be very impractical in an ad hoc network, especially one that is deployed in a combat zone. There is also additional cryptographic and message overhead for the dissemination of information on the nodes' location.

4. Cluster Head Routing

The authors of [16] and [17] segment their networks into clusters and utilize cluster heads (CHs) to act as the gateway for traffic leaving each cluster. CH routing is a common technique used in WSNs and can have the advantage of saving energy in a flat topology [16]. If all nodes have the capability, CHs can be rotated to conserve power and extend the life of the network. The traffic is routed to adjacent clusters through the CHs until it reaches the sink node's CH. The sink node's CH then broadcasts the traffic to allow the sink node to receive the data. This approach provides anonymity for the sink node among the members of its cluster but assumes that the nodes have the ability to elect their CHs and the CHs have the ability to build routes through adjacent CHs. This requires additional computational capabilities for all nodes that may become CHs, which increases cost and energy consumption.

F. CHAPTER SUMMARY

In this chapter, we discussed common WSN protocols, specifically IEEE 802.15.4 and 6LoWPAN. We also introduced the LOADng routing protocol that is used in this thesis research to achieve sink node anonymity. The discussion of these protocols builds a foundation of knowledge for the remainder of this thesis. Other related methods for sink node anonymity were presented, detailing their strengths and drawbacks relating to WSNs.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ACHIEVING SINK-NODE ANONYMITY IN REACTIVE ROUTING PROTOCOLS

Reactive routing protocols present a unique challenge to the problem of masking the identity of the sink node. As discussed previously, the act of route discovery through the flooding of RREQ packets and the RREP response quickly highlights the sink node; however, reactive protocols offer distinct advantages in their lower computational overhead and power consumption when compared to proactive protocols [10], [11]. In order to obfuscate the identity of the sink node, the traffic to the sink node must be indistinguishable from a set of other nodes in the network. In the following sections, we discuss the scheme used to anonymize the sink node as well as the modification to the LOADng routing protocol to achieve that anonymity.

A. *K*-ANONYMITY

K-anonymity was first proposed in 2002 [18]. The original premise of the theory related to protecting the identity of patients in a database by ensuring that accessible data does not link a specific record to an individual person. If multiple pieces of identifying information are put together in a patient record, and any of those pieces of information are unique, then a patient can be identified and linked to the medical condition associated with that record. By taking steps to ensure none of the personal information uniquely identifies an individual within the set of patients, a level of anonymity is provided to the patients.

The authors of [18] demonstrate the *k*-anonymity principle using a table $RT(A_1,...,A_n)$ with QI_{RT} as the quasi-identifying information associated with the table. If there are at least *k* occurrences of each sequence of values that appear in $RT[QI_{RT}]$, the table is *k*-anonymous, where *k* is the smallest number of identical sequences. The example table containing patient quasi-identifying information is shown in Table 1. In this example, $QI = \{Race, Birth, Gender, ZIP\}$ since the data in the *problem* column of Table 1 are not quasi-identifiers.

Table 1. Example of *k*-Anonymity, where $k = 2$ and $QI = \{Race, Birth, Gender, ZIP\}$. Source: [18].

	Race	**Birth**	**Gender**	**ZIP**	**Problem**
t_1	Black	1965	m	0214*	short breath
t_2	Black	1965	m	0214*	chest pain
t_3	Black	1965	f	0213*	hypertension
t_4	Black	1965	f	0213*	hypertension
t_5	Black	1964	f	0213*	obesity
t_6	Black	1964	f	0213*	chest pain
t_7	White	1964	m	0213*	chest pain
t_8	White	1964	m	0213*	obesity
t_9	White	1964	m	0213*	short breath
t_{10}	White	1967	m	0213*	chest pain
t_{11}	White	1967	m	0213*	chest pain

Sets (t_1, t_2) and (t_{10}, t_{11}) each contain identical quasi-identifying information among the records in their set. The data in Table 1 has a *k* value of two because there are at least two occurrences of each sequence of quasi-identifiers in the table. It is not possible to identify the person associated to a problem with certainty because of the anonymity provided by the information in the table. As long as *k* remains greater than one, this is true. If record t_{11} were removed, record t_{10} would be unique since the birth date of 1967 is different from records t_7, t_8, and t_9 despite the rest of the information being identical. In this case, $k = 1$, and there is not anonymity for record t_{10}.

B. APPLICATION OF *K*-ANONYMITY IN WSNS

For *k*-anonymity in a WSN, we are making the assumption that the attacker knows limited information about the network traffic because the communications are encrypted at the MAC layer; however, the attacker could begin to build a basic understanding of the types of traffic in the network from passively observing the traffic. Passive observation enables one to determine the types of traffic based on the length of the transmission time, number of bits in a packet, and the order of transmissions between nodes.

1. Identification of the Sink Node through Passive Observation

The transmission times for LOADng packets encapsulated in IEEE 802.15.4 frames using the maximum transmission speed of 250 kbps is shown in Table 2. We assume that the frames are not padded and the network is encrypting the traffic at Layer 2 using AES-128, so there is no change to the frame size. A full frame for IEEE 802.15.4 is limited to 127 bytes [4]. The frame header is 25 bytes and the encryption overhead is 21 bytes, leaving 81 bytes for the frame payload [4].

Table 2. Size of LOADng Packets in Bytes and Transmission Time Based on 250-kbps Data Rate

Packet Type	Packet Size (bytes)	Frame Size (bytes)	Transmission Time (ms)
RREQ	30	76	2.432
RREP	34	80	2.56
RREP_ACK	18	64	2.048
Data	81	127	4.064
RERR	30	76	2.432

Transmission rates may vary within a wireless network due to link quality and congestion, which can make determining frame types by time difficult but not impossible. Unless there is padding of the packets, the attacker may be able to see the number of bits in each transmission even if the traffic is encrypted. This makes the bit-rate irrelevant and makes the system more vulnerable.

In a WSN, it is assumed that the majority of the traffic is destined for the sink node; therefore, a majority of the RREQs that are sent have the sink node as the destination. Once the RREQs are received, the sink node does not forward RREQs any further. If this is the case, the majority of the RREP packets originate from the sink node, and the majority of the RREP_ACK and data packets are destined for the sink node. By observing the traffic and determining the ratio of the RREPs sent to the RREQs forwarded, the attacker can determine where the majority of packets are destined and conclude that this is the sink node. Additionally, once the network has converged and the

majority of traffic is data to the sink node, it is easy for an attacker to determine the sink node.

2. Obscuring the Sink Node through *k*-Anonymity

The *k*-anonymity principle in a WSN seeks to prevent sink node identification by having at least one node act similarly to the sink node. By altering the traffic patterns, we find it is possible to change the behavior of the sink node and its neighbor nodes so that they are similar enough to be indistinguishable to an attacker. This is equivalent to patient records having the same identifiable information among multiple patient records [18]. As *k* increases, that identification becomes more difficult.

Let N be the set of all nodes and N_{SN} be the set of nodes that include the sink node and its one-hop neighbors where $N_{SN} \subseteq N$. If the sink node is indistinguishable from the other nodes in N_{SN}, then $k = N_{SN}$. We determine the distinguishability of the sink node by looking at two parameters: the total number of transmissions and the ratio of RREP to RREQ packets sent for each node in N_{SN}. With these numbers, we can measure the standard deviation among the nodes to determine how easy it is to identify the sink node among its neighbors. Using T_{SN_i} as the number of transmissions or the ratio of RREP to RREQ packets for node N_{SN_i} and μ_{SN} as the mean, we can calculate the standard deviation as

$$\sigma_{SN} = \sqrt{\frac{1}{N_{SN} - 1} \sum_{i=1}^{N_{SN}} \left| T_{SN_i} - \mu_{SN} \right|^2} \tag{1}$$

where

$$\mu_{SN} = \frac{1}{N_{SN}} \sum_{i=1}^{N_{SN}} T_{SN_i} \ . \tag{2}$$

If the number of sink node transmissions and the ratio of RREP to RREQ packets are both within one standard deviation of the mean μ_{SN}, then the sink node cannot be distinguished from its neighbor nodes and is considered *k*-anonymous where $k = N_{SN}$. With T_S as the number of transmissions or the ratio of RREP to RREQ packets for the sink node, the sink is *k*-anonymous if it satisfies the inequality

$$\left|T_S - \mu_{SN}\right| \leq \sigma_{SN}. \quad (3)$$

The advantage of this system is that even if the sink node's behavior falls outside of one standard deviation of the mean, it may still be anonymous within a smaller subset of nodes. For example, assume that a sink node has seven neighbors ($N_{SN} = 8$) and its number of transmissions is greater than μ_{SN} by more than one standard deviation. It is possible that there is one node in the set that has a very low transmission number that is causing the mean of the set of nodes to go down. Removing this node from the set raises the mean and may bring the sink node within one standard deviation of this new set, yielding $k = 7$.

C. MODIFIED LOADNG FOR SINK-NODE ANONYMITY

Our method obscures the identity of the sink node by employing the *k*-anonymity metric previously described. The sink node's identity is hidden to an adversary by using additional nodes that act as the sink node, making it difficult for anyone observing from the outside to determine which node is actually the sink node. This is a common technique employed using different methods in numerous anonymity schemes in wireless sensor networks. To the best of our knowledge, this approach to sink-node anonymity has not been accomplished utilizing a reactive, or on-demand, routing protocol. Due to the nature of reactive routing protocols, maintaining anonymity of the sink node can be difficult to accomplish.

1. Protocol Overview

As noted previously, the sink node is particularly vulnerable to identification, even if the traffic is encrypted because a majority of the traffic in the network is destined for this node. By observing the frequency of the transmissions from the node, an adversary can quickly determine the importance of the node and determine with a high level of certainty that this is, at the very least, a high priority node.

Our scheme works by developing a zone of regular nodes around the sink node. The nodes within this zone act as false endpoints for the traffic. The nodes that are one-hop neighbors to the sink node are chosen to be part of the zone. This is because the one-

hop neighbors are within transmission range of the sink node, thereby allowing the sink node to have knowledge about the various routes each of its one-hop neighbors has in its routing set. In other words, the sink node knows all of the other nodes to which its one-hop neighbors have routes. This information is necessary in order for the modified LOADng protocol to operate efficiently. The sink node does not know the routes that non-neighbor nodes have, so they are not used as part of the zone.

2. Choosing a Neighbor Node

Since all nodes in the network build their knowledge of the network through the use of RREQ packets, it is critical for efficient routing that some RREQ packets from the source node(s) reach the sink node. The sink node must have an understanding of which of its neighbor nodes have active routes to the node requesting a route (source node). In order to ensure this, the sink node must not respond to the first RREQ it receives from each node in the network. The sink node sees which of its neighbors have forwarded the RREQ and knows that this neighbor node has a route to the source node. When the timeout for the RREQ expires, the source node floods another RREQ packet through the network.

When the sink node receives the second RREQ from this source node, it looks in its routing set and finds all nodes, including itself, which have a route to the source. Of these, it excludes those nodes that the sink node knows have already forwarded the new RREQ. This is critical because if a node is seen forwarding a RREQ and then sending a RREP, an adversary might assume that this node is not the sink node. From the nodes that are left, including itself, the sink node chooses one node to act as the sink node.

3. Forwarding the Altered RREQ and Sending the RREP

If the sink node does not choose itself, it alters the RREQ packet by changing the destination address to the chosen neighbor node and setting the *sink flag* in the packet. This flag tells the chosen neighbor node that it is acting as the sink node and relays to the source node that any future packets destined for the sink node need to go to this address. The sink node then continues flooding the altered RREQ, acting as a normal node. When the chosen neighbor node receives the RREQ, it creates a RREP packet with the *sink flag*

set and unicasts it back to the source node. If the sink node chooses itself, it creates a RREP packet with the *sink flag* set and unicasts it back to the source node. An example of the RREQ and RREP packets is shown in Figure 4 with link numbers indicating the hop count from the source node.

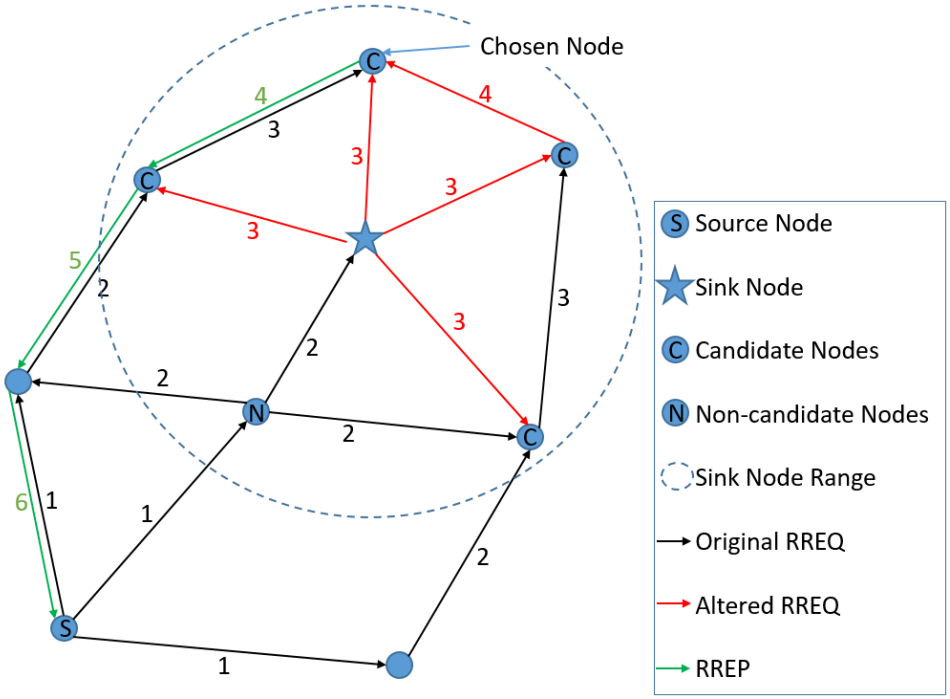

Figure 4. Example of RREQ Flooding and RREP Unicast for the Modified LOADng Routing Protocol

The addition of the *sink flag* to the RREP, RREP_ACK, and data packets is accomplished through a protocol extension enabled by TLV elements. The extension adds an additional TLV element of type FLAGS as defined in [9] to the RREP_ACK and data headers since the RREP already contains the TLV. Bits 1–7 are reserved for future use in this TLV, allowing us to use bit 1 for the *sink flag.* RREQ packets do not require the addition of the *sink flag* since it is not used in the route discovery process. A RREP header with the FLAGS TLV highlighted is shown in Figure 5. The *Values* field contains the eight bits used in this TLV.

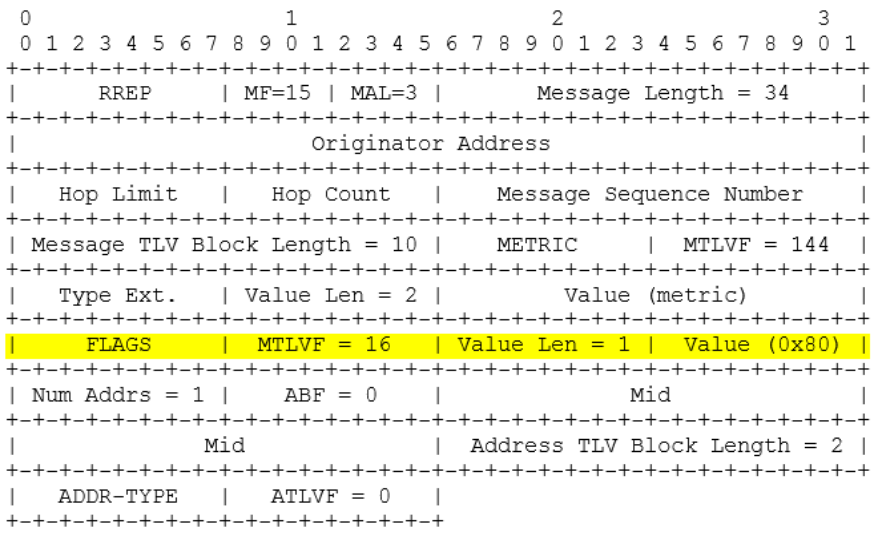

Figure 5. RREP Packet Header with FLAGS TLV Highlighted. Source: [9].

4. RREP_ACK and Data Packets

Upon receiving the RREP, the source node first sends a RREP_ACK to the source address in the RREP. It then sends its data packet(s) to the same address. Upon receipt of a data packet, the destination node, whether the actual sink node or a chosen neighbor, sends a broadcast of the data packet with the *sink flag* set. All neighbor nodes see the broadcast, but only the actual sink node accepts the packet since the *sink flag* is set. It is important that the actual sink node also broadcast if it is the destination to ensure that it is behaving the same as the fake sink nodes.

D. DESIGN CONSIDERATIONS

Since the traffic is assumed to be encrypted at Layer 2, the adversary is not able to see any of the Layer 3 header information. This is important because if the Layer 3 header was visible, the address of the sink node can easily be determined using a simple packet sniffer, and finding the address of the node to which the majority of the traffic is destined would be trivial. A potential issue is when the sink node changes the RREQ destination address, the adversary can see a change in the encrypted packet and determine that this node is the sink node. If there is no change to the underlying frame information as it is routed through the network, there is no change to the cipher text visible to the

observer. When the sink node alters the RREQ by changing the destination address and sink flag, it causes a change in the cipher text. This signifies a change in the underlying frame information and alerts the observer that this is the sink node. The RREQ header contains a *hop count* field, which is incremented as each node forwards the packet. In other words, the hop count field also changes as the frame moves through the network. Under the principle of cryptographic diffusion, this one-bit change creates a large change in the encrypted packet as it moves through the network; therefore, changing the destination node and the sink flag are not distinguishable to the adversary.

This protocol puts the majority of the requirements on the sink node, which generally has more resources than the sensor nodes [3]. The only change to the routing protocol for the sensor nodes is the addition of the *sink flag*. The nodes must be aware of the *sink flag* for the purpose of routing traffic to the sink node and to signal to the destination node that it must broadcast.

Other than these modifications, there are no changes made to the LOADng protocol. This was done to ensure as little computational overhead as possible for the sensor nodes. The modified LOADng is designed to function on nodes with very basic capabilities and assumes that the nodes only have the ability to wirelessly communicate with neighbors in accordance with IEEE 802.15.4 and store route tuples as defined in the LOADng protocol. The nodes do not need to have any pre-defined knowledge of their physical location or network topology for this scheme to work as some other anonymity schemes require.

E. CHAPTER SUMMARY

In Chapter III, we discussed the anonymity scheme used to hide the identity of the sink node in a WSN. A background of the *k*-anonymity principle was outlined as well as its use to quantify the anonymity of a node relative to a set of other nodes. As long as the sink node's number of transmissions and its RREP-to-RREQ forwarded ratio values fall within one standard deviation of the mean of those values for a set N_{SN}, which includes the sink node and its neighbors, the sink node is said to the *k*-anonymous with $k = N_{SN}$. The modified LOADng routing protocol achieves this by choosing a node from a subset

of N_{SN} to reply to a RREQ, which then continues to act as the endpoint for future transmissions from that source node. The modified LOADng protocol was discussed. The design considerations that were taken into account during the development of the modified LOADng protocol to achieve *k*-anonymity were addressed.

IV. EXPERIMENTAL DESIGN

All simulations were designed and run in a custom simulator built in MATLAB. The simulator is designed to simulate any number of sensor nodes in either a uniform spacing or a random distribution. This flexibility allows the testing of the effectiveness of the modified LOADng protocol in the scenarios of deliberate or random placement of nodes. In each simulation, five metrics were measured: sink-node anonymity, average route length, latency, power use, and PDR.

A. EXPERIMENTAL PARAMETERS

The experimental parameters for the simulation are discussed in the following sections. In particular, the transmission range of the nodes, the number of nodes, and the spacing between them are discussed. We also describe how power consumption is measured within the simulations.

1. Selecting Sensor Parameters

The type of sensor node and the transmission range requirements of the nodes is not the focus of this thesis research and are not taken into account in the determination of range. Many available COTS IEEE 802.15.4 transmitters that operate at 1-dBm transmit power have a maximum unimpeded line-of-sight range up to 100.0 m. Accordingly, we chose the transmission range of the nodes to be 50.0 m. We assume that the maximum transmission range and any signal degradation due to range do not greatly impact the outcomes of the experiments. We make this assumption because we are comparing the performance between LOADng and our modified LOADng protocol on identically distributed networks under identical circumstances; therefore, any change in data rate due to transmission range is assumed be the same between the two networks and will not change the outcome of the experiment.

2. Determining Node Quantities

The number of nodes for a uniform distribution was determined based on the size of the field and the transmission range of the sensor motes. Using a transmission range of

50.0 m, we modeled the transmission area as a circle with a radius of 50.0 m with the sensor mote situated at the center. Using a right triangle with the hypotenuse as the diameter of the circle (100.0 m), the length of the other two sides is 70.71 m each. Spacing the sensor motes 35.35 m apart is the minimum requirement to provide transmission coverage for the entire area and requires 205 sensor motes to cover a 500×500 m^2 area if the sensors are deployed to the edges. The geometric representation of the calculation of the uniform node spacing is shown in Figure 6.

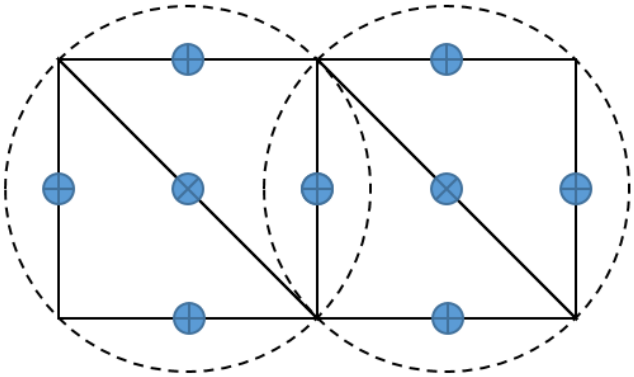

Figure 6. Uniform Spacing of Sensor Nodes

This calculation works for uniformly spaced sensors but often caused problems when the sensor motes were placed using a random distribution. With the random distribution, there were often pockets of nodes isolated from the sink node, meaning their traffic always failed to reach the sink node. An example of a simulation in which there is a pocket of isolated nodes in a random distribution is shown in Figure 7.

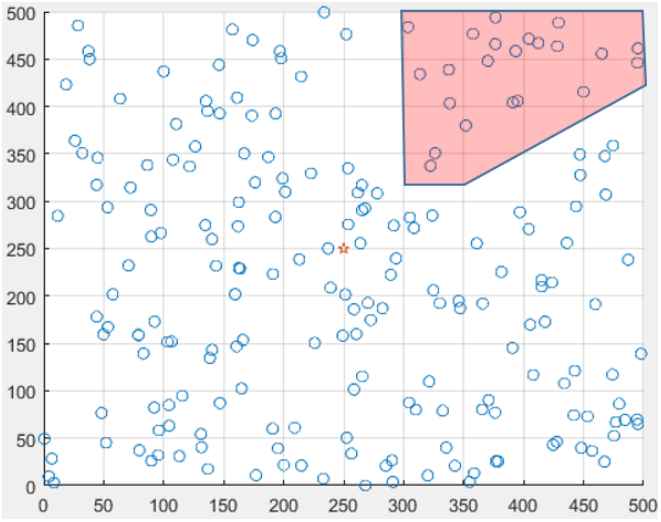

Figure 7. Isolated Pocket of Nodes in a Random Distribution within a 500×500 m^2 Field Containing 205 Nodes

Through experimentation, we found that increasing the number of sensor nodes by approximately 25% dramatically reduced the occurrences of isolation for a random distribution; therefore, for the same field size of 500×500 m^2, we chose to deploy 250 nodes for a random distribution. For the uniform spacing, 196 nodes were chosen because it provides an even 14×14 sensor node field. This maintains the same node neighbors as 205 nodes because the small distance increase between sensors does not extend the distances beyond the 50.0-m range. The size of the field was chosen to be 500×500 m^2 to ensure there was a large sample set of nodes outside the transmission range of the sink node. This allows for numerous hops between nodes on the edge of the field and the sink node. We assume that there is only one sink node in the network.

3. Measuring Power Consumption

When measuring power consumption, we assumed that nodes are in three general states: idle, receiving, and transmitting. Receiving and transmitting encompass all functions to go from the idle state, complete their task, and return to the idle state. In Table 3, the power consumption values used to determine power usage during the transmit and receive phases are shown.

Table 3. Phases of Node Operations and Power Usage. Adapted from [19].

Transmit	Receive	Power Draw	Duration	Power Used
Wakeup & Preprocessing	Wakeup & Preprocessing	44 mW	1.5 ms	1.833×10^{-5} mWh
	Receive	66 mW	32 µs/Byte	5.867×10^{-7} mWh
CSMA/CA		72 mW	1 ms	2×10^{-5} mWh
RX-TX switch		54 mW	0.4 ms	6×10^{-6} mWh
Transmit		90 mW	32 µs/Byte	8×10^{-7} mWh
Post-processing	Post-processing	24 mW	1.4 ms	9.33×10^{-6} mWh

The energy consumption for transmitting a full 127-byte IEEE 802.15.4 frame is 1.55×10^{-4} mWh, while the energy to receive the same frame is 1.02×10^{-4} mWh. The case when the node enters the CSMA/CA state and finds that it cannot transmit must also be taken into account. In this case, the node skips the *RX-TX switch* and *transmit* states and goes straight to the *post-processing* state before going to idle.

In addition to the above power consumption, we need to account for the power consumed during the cryptographic key setup, encryption, and decryption for AES-128. This must be taken into account at each node since the encryption is executed at the MAC layer. Every node needs to decrypt each packet it receives and encrypts each packet it transmits. The power used by a node during the encryption and decryption phases is shown in Table 4.

Table 4. Power Consumption for AES Encryption and Decryption. Adapted from [20].

	Key size (bits)	Duration	Power Used
Encryption	128	1.53 ms	1.09×10^{-5} mWh
Decryption	128	3.52 ms	2.47×10^{-5} mWh

B. SIMULATOR DESIGN

All simulations were performed with a custom simulator designed in MATLAB. The network for the modified LOADng protocol is simulated first followed by the simulation for the standard LOADng protocol using identical node distributions, transmit

order, and transmit times. The simulator works by placing nodes into different matrices when they enter certain states and associates a timer with that node. For each iteration of the simulator, all timers are decremented by 100 μs. When a timer reaches zero, the simulator acts on that node based on the function it is currently performing. In the following sections, we elaborate on the details of the simulator.

1. Building the Network

The user is initially prompted for the number of nodes, the size of the field, the transmission range, the number of transmissions, and whether it is a uniform spacing or random distribution. The simulator builds the physical distribution of the sensor field based on these inputs and places the sink node at the center of the field. Examples from the MATLAB simulator of the uniform and random distributions of sensor nodes are shown in Figure 8. A matrix of neighbor nodes and physical distances, named *neigh*, is created based on the transmission range and the physical locations of the nodes relative to each other. This matrix is used to determine which nodes are within the transmission range of other nodes in the network.

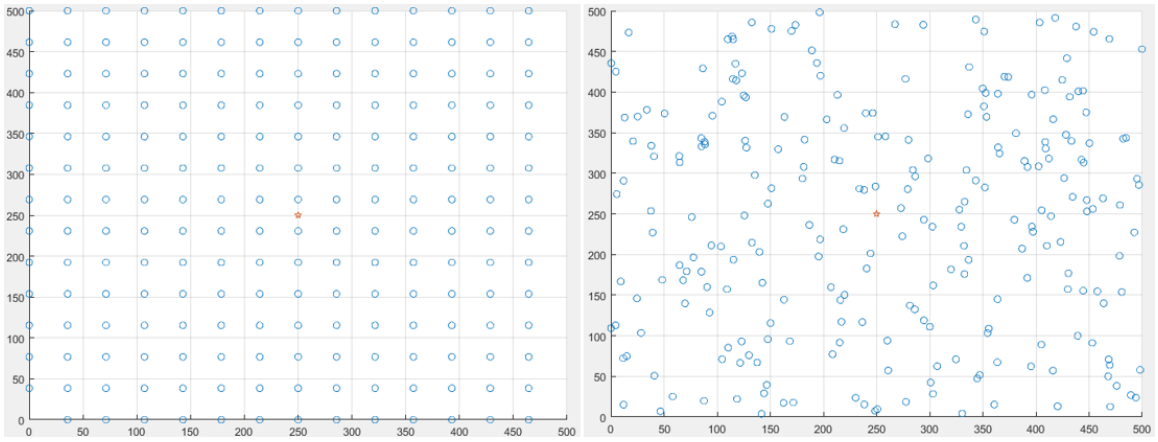

Figure 8. Uniform Spacing (Left) with 196 Nodes and Random Distribution (Right) with 250 Nodes

2. Determining Transmission Order

Once the nodes are placed and the neighbors are calculated, the simulator determines the transmission order and the number of times the nodes attempt to transmit. The simulator builds a $2 \times N$ matrix called *trans_order*, where N is the number of transmissions and randomly assigns node addresses to each cell in the first row. The node in the first cell is given a transmit time of zero, and each following node is given a transmit time randomly calculated between zero and 10.0 s following the previous node. These values are stored in the second row. This creates the transmission queue for both the modified LOADng and LOADng simulations. When the timer for a node reaches zero, that node is put into a matrix named *pending_txmit* with all of the information that is contained in the packet.

3. Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)

A basic CSMA/CA algorithm was designed into the simulator and implemented in the function *CSMA_CA*. When a node in *pending_txmit* has a back-off timer of zero, it attempts to transmit. The node first listens to see if a neighbor node is transmitting. The simulator determines this by comparing the nodes in the *txmitting* matrix (containing nodes that are currently transmitting) with the neighbor nodes in *neigh* of the node attempting to transmit. If there are no matching nodes, there are no neighbor nodes transmitting, and the node is able to transmit. The node is put into the *txmitting* matrix with a transmitting timer set based on the values in Table 2 and is now in a transmitting state. If there is at least one neighbor node transmitting, the node goes into a back-off equal to the longest remaining transmit time of the transmitting neighbor. When its back-off timer reaches zero, it attempts to transmit again.

4. Collision Detection

The *collision* function determines if there is a collision due to the hidden node problem. If two transmitting nodes that are not neighbors have a common neighbor, there is a collision. The simulator determines if there are common neighbors by comparing all of the nodes in *txmitting* and finding common neighbors in the *neigh* matrix. If two nodes

have a common neighbor, both nodes are put back into *pending_txmit* and are given a back-off time based on a Binary Exponential Back-off (BEB) mechanism.

The BEB value is determined based on the number of times the node has gone into a back-off state. The back-off intervals are based on the transmission time for a full 127-byte frame, which is 4.1 ms. The node randomly chooses an integer value from zero to the number of times it has gone into back-off and multiplies that by 4.1 ms. This is the node's back-off timer. After failing to finish a transmission five times, a node quits trying to transmit and simply drops the packet. In this case, the simulator removes the node from the *txmitting* matrix.

5. Packet Transmission

When the transmit timer for a node in the *txmitting* matrix reaches zero, the *tx_complete* or *tx_complete_mod* functions are called for standard LOADng or modified LOADng, respectively. These functions first determine the next hop for the packet based on the destination and the transmitting node's routing set. The function *flood*, *flood_mod*, *sink_flood*, or *unicast* is then called based on the type of packet the node is transmitting. If it is a unicast packet, the next hop is passed to the unicast function. The values used to calculate the metrics for the simulation are initiated and updated in the *tx_complete* and *tx_complete_mod functions*.

6. Flooding RREQ Packets

When a node needs to send a packet to the sink node, it first determines if it has a route to the sink node in its routing set. If it does not have a route, it floods a RREQ packet to its neighbor nodes. The flooding is accomplished using a classical flooding routine. There are three functions that handle packet flooding in the simulator: *flood*, *flood_mod*, and *sink_flood.*

a. Flood

The *flood* function handles flooding for the normal LOADng protocol. An intermediate node that is not the destination node floods the RREQ to all of its neighbors except the neighbor node from which the RREQ was received. This is accomplished by

placing these neighbor nodes into the *pending_txmit* matrix with the same packet information and a decremented hop limit. If a node receives a RREQ packet that it has already forwarded, it ignores the packet. If a neighbor node is the destination for the RREQ, it is placed into the *pending_txmit* matrix with a packet type of RREP and a destination address equal to the source of the RREQ.

Jitter is introduced into the system by assigning a back-off time between zero and three times the transmission time of a data packet. This is known as window jitter and is shown by Cordero et al. [21] as a simple, effective way to reduce the number of collisions during the flooding of RREQ packets during route discovery. The introduction of jitter in the flood routines dramatically reduces the number of collisions due to hidden nodes, which increases the success rate of the RREQ process.

After the neighbor nodes are placed into the *pending_txmit* matrix, the *update_route_table* function is called. Each neighbor node updates its routing sets if the route to the source and neighbor nodes have new routes or have a lower hop count than an existing tuple. This creates the reverse route that the RREP takes back to the source.

b. Flood_mod

The *flood_mod* function performs the flooding functions for the modified LOADng protocol. It works identically to the *flood* function discussed previously with a few additions. This routine must take into account the *sink flag* when looking at the destination because of the modified RREQ from the sink node. This is to distinguish the modified RREQ from any normal RREQ packet that may be destined for a non-sink node.

For the sink node, *flood_mod* is called the first time the sink node receives a RREQ packet from a source node. This simulates the sink node ignoring the packet and continuing to flood the packet as if it were not the destination. The *flood_mod* function is not called to handle flooding of modified RREQ packets from the sink node.

c. Sink_flood

The *sink_flood* function handles the flooding of RREQ packets that were modified by the sink node during the route discovery process. It is called when the sink node receives a second RREQ from a source node. The sink node first determines all of its neighbors that have not forwarded the RREQ and adds itself to the set with a probability of 0.5. It then randomly chooses a node from this set to act as the sink node. If it chooses itself, the sink node is placed into the *pending_txmit* matrix for a RREP back to the source. If it chooses another node from the set, it alters the RREQ packet with the new node as the destination and sets the *sink flag* to 1. The sink node then floods the altered RREQ packet to all of its neighbors that have not forwarded the original RREQ in accordance to the rules of classical flooding. This process is critical to achieving *k*-anonymity.

7. Sending Unicast Packets

When a node wants to transmit a RREP, RREP_ACK, or data packet, this traffic is sent via a unicast transmission from the source node to the destination node. The sending node sends this packet to one neighbor based on the next hop address in its routing set for the destination node. This is handled by the *unicast* function. The *unicast* function determines whether the receiving node is the destination. If it is not, the receiving node is added to *pending_txmit* and continues to unicast toward the destination. If it is the destination, the node reacts based on the type of packet received. If it is a RREP packet, the node is first put into *pending_txmit* for a RREP_ACK packet with a back-off time of zero. It is then put into *pending_txmit* for a data packet with a back-off equal to the transmit time for a RREP_ACK to ensure that it sends this packet second. If the packet received is a RREP_ACK or a data packet, the receiving node does nothing.

8. Broadcasting

During the simulation of the modified LOADng routing protocol, a data packet is broadcast when the destination node receives the packet with the *sink flag* set. In the *unicast* function, if the packet is a data packet with the *sink flag* set and the next hop node is the destination, the next hop is put into *pending_txmit* with the destination address set

to the broadcast address. The node is put into the *txmitting* matrix through the *CSMA_CA* function. When the transmission timer reaches zero, the *tx_complete_mod* function calls the *broadcast* function, which signals that the data packet was received by the sink node.

9. Measuring Metrics

Five metrics are measured in the simulation: sink-node anonymity, average route length, latency, power use, and PDR.

a. Sink-node Anonymity

The simulator records the total number of transmissions completed by each node as well as the number of RREP packets originated and RREP packets forwarded by each node. It separates these between nodes that are neighbors to the sink node and those that are not. From these numbers, the mean and standard deviation are calculated for the node set including the sink node and its neighbors and then separately for the set of all other nodes. This is performed for the standard and modified LOADng protocols separately.

b. Route Length

The average route length is calculated at the end of the simulation by finding the average of the non-zero route metrics within the total routing set. This is the average route length for all routes that were determined during the simulation.

c. Latency

Latency is measured from the time a unicast packet is sent to the time it is received at its destination. The simulator tracks the total latency for each node in the network and divides this by the total number of unicast packets originated by that node. The average is then taken across all nodes in the network to obtain the average latency for all nodes.

d. Power Use

During the simulation, the simulator maintains a running count of the number of times each node attempts to transmit, completes a transmission, and the total time each

node is transmitting. These values are used with the values in Table 3 and in accordance with the description in Section A.3 to determine the total power used by each node in the network.

e. PDR

Packet Delivery Ratio (PDR) is determined based on the number of unicast packets originated and the number of unicast packets received. The simulator maintains a total count of the number of unicast transmissions originated by nodes in the network as well as the total number of unicast packets received by nodes in the network. Dividing the total number received by the total number originated gives the PDR.

C. CHAPTER SUMMARY

In this chapter, we provided an overview of the simulation environment and the design of the simulator in MATLAB. The simulator runs the scenario for the modified LOADng protocol followed by the standard LOADng protocol for identical node placements, transmit order, and transmit times. Classical flooding was used to transmit RREQ packets through the network. CSMA/CA and collision detection were implemented to simulate a realistic network environment. Information on network traffic was gathered to calculate five metrics: sink-node anonymity, average route length, latency, power use, and PDR.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SIMULATION RESULTS AND ANALYSIS

The simulations were run for a uniform spacing of 196 nodes and a random distribution of 250 nodes. Under each scenario, simulations were run using 500, 1000, and 2000 transmissions from nodes to the sink node. The transmission order of the nodes and the time spacing were randomly generated, as previously noted.

A. SINK-NODE ANONYMITY

Achieving sink-node anonymity utilizing a reactive routing protocol is the primary goal of this research. We define the sink node as anonymous if the number of transmissions sent and the ratio of RREPs sent to RREQs forwarded for the sink node are within one standard deviation of the mean of the same metrics for its neighbors. In this section, we present the results of the anonymity test for the modified LOADng protocol and compare the results to the standard LOADng protocol. The results presented for each number of transmissions in both sections represent the results from the same simulations.

1. Number of Transmissions

The number of individual transmissions completed for each node was tracked separately during the modified LOADng and LOADng simulations. A completed transmission means a single node completes the transmission of one packet to a neighbor node. Transmissions that were incomplete due to a collision are not counted in the numbers. The transmission number values for 500, 1000, and 2000 transmissions are shown in Tables 5, 6, and 7, respectively. These tables contain the mean and standard deviation of the number of transmissions for the sink node's neighbor nodes and the nodes that are not neighbors to the sink node. The numbers for both our modified LOADng and the standard LOADng protocols are included for comparison.

Table 5. Number of Node Transmissions for 500 Total Transmissions

		Uniform Spacing			Random Distribution		
			Neighbors			Neighbors	
		Sink Node	Sink	Non-sink	Sink Node	Sink	Non-sink
Modified LOADng	Mean	976	969	628.66	1079	1130.67	856.06
	Std-dev		39.29	41.79		121.1409	111.46
LOADng	Mean	592	315.43	115.89	682	357.33	192.25
	Std-dev		146.35	37.83		189.38	53.69

Table 6. Number of Node Transmissions for 1000 Total Transmissions

		Uniform Spacing			Random Distribution		
			Neighbors			Neighbors	
		Sink Node	Sink	Non-sink	Sink Node	Sink	Non-sink
Modified LOADng	Mean	1267	1261	678.18	1519	1694.60	1261.65
	Std-dev		74.03	69.56		160.24	83.43
LOADng	Mean	1084	515	130.81	1230	518.30	234.88
	Std-dev		292.78	76.19		371.71	70.72

Table 7. Number of Node Transmissions for 2000 Total Transmissions

		Uniform Spacing			Random Distribution		
			Neighbors			Neighbors	
		Sink Node	Sink	Non-sink	Sink Node	Sink	Non-sink
Modified LOADng	Mean	2100	1924	839.15	1861	2016.50	977.88
	Std-dev		297.52	127.46		298.4598	661.39
LOADng	Mean	2186	938.00	185.45	2225	1103.11	279.40
	Std-dev		584.55	136.47		179.33	162.12

For the modified LOADng protocol, the sink node was within one standard deviation of the mean number of transmissions for its neighbor nodes for all cases tested. The uniform node spacing consistently achieved anonymity for the number of transmissions based on our criteria. The random node distribution met the criteria in most cases but failed under certain circumstances. These circumstances are explained later in

this chapter. The number of sink-node transmissions in the standard LOADng protocol was never within one standard deviation of the mean number of transmissions for its neighbor nodes.

There was a significant increase in traffic among all nodes in the modified LOADng simulation compared to the standard LOADng protocol. This is partially due to the extra RREQ packets sent during the initial route discovery process to give the sink node awareness of its neighbor's routes to the source. It was also found during simulation that fewer RREQs were sent during the standard LOADng simulation than initially expected. This is due to the forward route generation during the RREP process, which generates routes to the sink node for the intermediate nodes. When these intermediate nodes send data to the sink node for the first time, they already have a route and do not need to send a RREQ. In the modified LOADng protocol, these routes to the sink node are built only when the sink node has chosen itself, which is a much smaller number of times. The comparison of the total transmissions between our modified LOADng protocol and the standard LOADng protocol is shown in Figure 9.

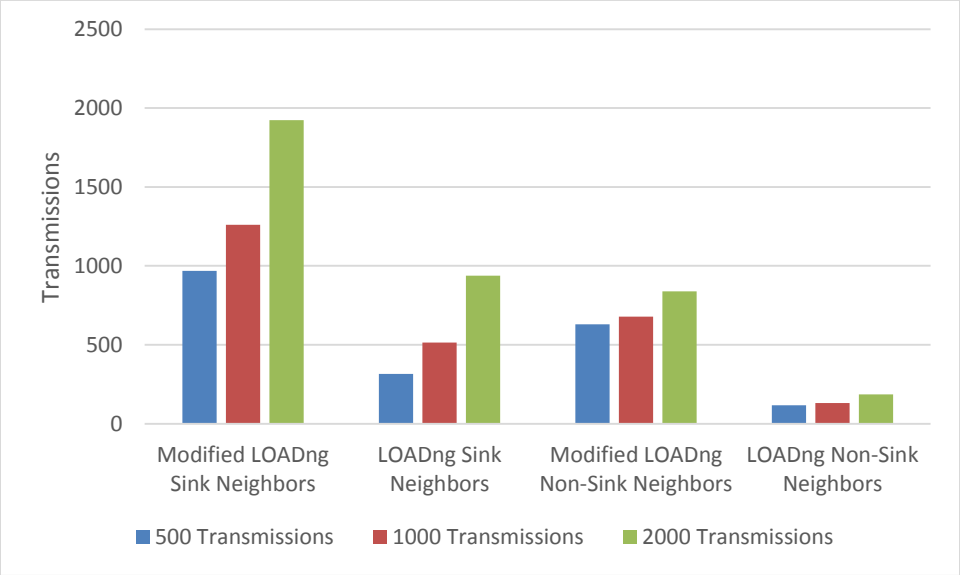

Figure 9. Modified LOADng and Standard LOADng Total Transmissions Average per Node

2. RREP to RREQ Ratio

Sink-node anonymity was achieved for the ratio of RREPs sent to RREQs forwarded as well. The results shown in Table 8 are from the same simulations as those in Tables 5, 6, and 7 for the respective number of transmissions. This demonstrates that the modified LOADng protocol meets our criteria for sink-node anonymity for both the number of transmissions and the ratio of RREPs sent to RREQs forwarded simultaneously. The standard LOADng protocol is not shown because only the sink node sent RREPs in that scenario, meaning the ratio for all other nodes is zero.

Table 8. RREPs Sent to RREQs Forwarded Ratio

	Uniform Spacing			Random Distribution		
Transmissions	**Sink node**	**Avg**	**Std-dev**	**Sink node**	**Avg**	**Std-dev**
500	0.0647482	0.070986	0.010574	0.072289	0.059148	0.013748
1000	0.0831974	0.072779	0.013318	0.044426	0.05115	0.011477
2000	0.0861582	0.072693	0.016489	0.087585	0.076675	0.013895

B. NON-ANONYMITY PERFORMANCE METRICS

The results presented in this section represent the metrics not directly related to sink-node anonymity. All measurements were performed during the same simulations as presented in the previous section.

1. Power Usage

The higher network traffic directly correlates to higher power usage, as shown in Table 9. The mean ratio of power usage between the two protocols is 4.48:1, demonstrating a dramatic increase in power usage. The main cause of the increase in power is the additional flooding of RREQ packets for route discovery. There is also a significant increase in collisions as depicted in Figure 10 for a simulation with 2000 transmissions. These collisions cause nodes to retransmit their packets, resulting in an increase in power usage.

To determine the effect of collisions on total power use, additional simulations were performed with collisions disabled. The result was a mean ratio of power usage of 3.36:1, down from 4.48:1. This shows that there is significant impact caused by the increase in collisions. Nevertheless, even this lower ratio represents a large increase in power usage over the standard LOADng protocol. From the *Ratio* row in Table 9, we see that the ratio for power usage decreases as the simulations get longer (i.e., more transmissions). In longer simulations, the RREQs are a lower proportion of the total transmissions, resulting in fewer collisions proportional to time. As the simulation progresses, there are fewer RREQs and, correspondingly, fewer collisions, thereby decreasing average power usage. Both the modified LOADng and standard LOADng had zero collisions during times without RREQs.

Table 9. Average Power Use per Node (mWh)

	Uniform Spacing			Random Distribution		
Transmissions	**500**	**1000**	**2000**	**500**	**1000**	**2000**
Mod. LOADng	0.5238041	0.602645	0.691146	0.523804	1.198195	1.038033
LOADng	0.1004773	0.140199	0.186704	0.100477	0.225695	0.267513
Ratio	5.21316	4.2985	3.70184	5.21316	5.308916	3.880316

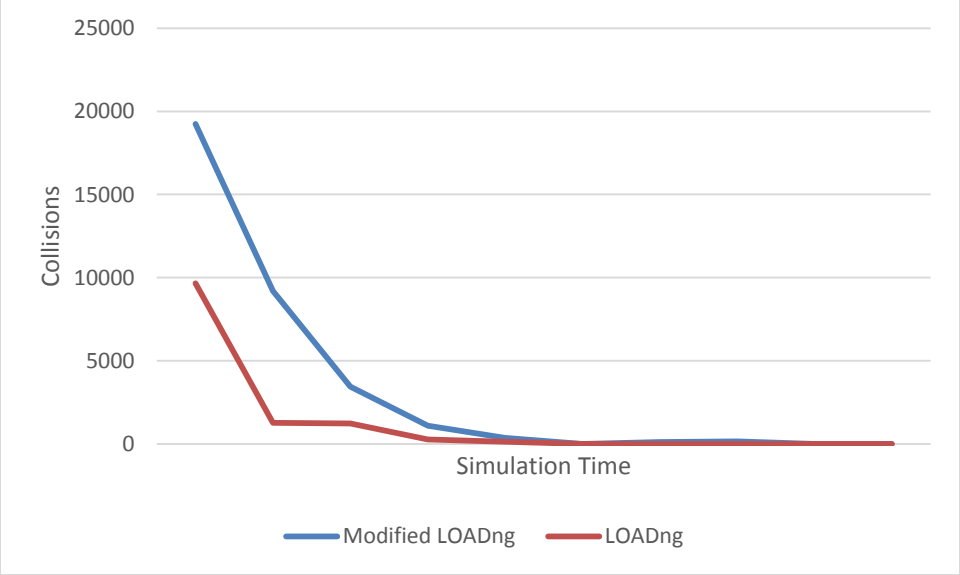

Figure 10. Transmission Collisions over Time for 2000 Transmission Simulation

2. Latency

The average latency of unicast packets between the source node and the sink node increased in the modified LOADng protocol by 5.0 ms, as is shown in Table 10. This is mainly due to the broadcast of packets, which add an extra 4.1 ms to the total latency. The number of hops to the chosen neighbor node also averages out to be slightly longer than to the sink node itself. This is the likely cause for the additional 1.0 ms in latency.

The average path length of all established routes is 0.83 hops fewer for the modified LOADng protocol, as shown in Table 11. This is due to the increased number of RREQ packets allowing the nodes more opportunity to form shorter routes.

Table 10. Latency of Unicast Packets between Source Node and Sink Node (sec)

	Uniform Spacing			Random Distribution		
Transmissions	**500**	**1000**	**2000**	**500**	**1000**	**2000**
Mod. LOADng	0.028694	0.024974	0.023346	0.028694	0.025767	0.022405
LOADng	0.0229192	0.020159	0.018787	0.022919	0.019655	0.018596
Delta	0.0057748	0.004815	0.004559	0.005775	0.006112	0.003809

Table 11. Average Path Length Measure in Hops between All Nodes

	Uniform Spacing			Random Distribution		
Transmissions	**500**	**1000**	**2000**	**500**	**1000**	**2000**
Mod. LOADng	10.344919	10.48034	10.36545	10.32775	9.137914	9.414233
LOADng	10.804386	11.41985	11.04272	11.73658	9.779416	10.30082

3. PDR

The PDR for modified LOADng is moderately lower than standard LOADng, resulting in an average of −0.0225 and −0.0362 compared to standard LOADng in the uniform spacing and random distributions, respectively. These results are shown in Table 12. This is caused by the increase in collisions as our tests without collisions showed a

PDR of 1.0 for both, as expected. In addition, the increased node density of the random distribution caused more collisions and a corresponding lower PDR for both protocols.

Table 12. Packet Delivery Ratio

	Uniform Distribution			Random Distribution		
Transmissions	**500**	**1000**	**2000**	**500**	**1000**	**2000**
Mod. LOADng	0.94291	0.96189	0.967798	0.928807	0.880139	0.957223
LOADng	0.962	0.984715	0.993474	0.969099	0.935319	0.970252
Delta	-0.01908	-0.02283	-0.02568	-0.04029	-0.05518	-0.01303

C. FAILURE CASES

There were cases in the random distribution in which the modified LOADng protocol failed to achieve sink-node anonymity. An example of the node placement in a normal failure scenario is shown in Figure 11. When a RREQ comes through the lone bottom node (shown as the bottom red arrow in Figure 11) to the sink node (shown as a star in Figure 11), that lone node is excluded from the candidate node set. This makes the three nodes above the sink node (three red arrows above sink node in Figure 11) and the sink node the only candidate nodes. Also, in this scenario, there are no routes between the three neighbor nodes (above the sink node) to nodes below the sink node that are shorter than the route that goes through the sink node. This forces all traffic to and from nodes below the sink node to go through the sink node, causing the total transmissions at the sink node to be higher than its neighbor nodes.

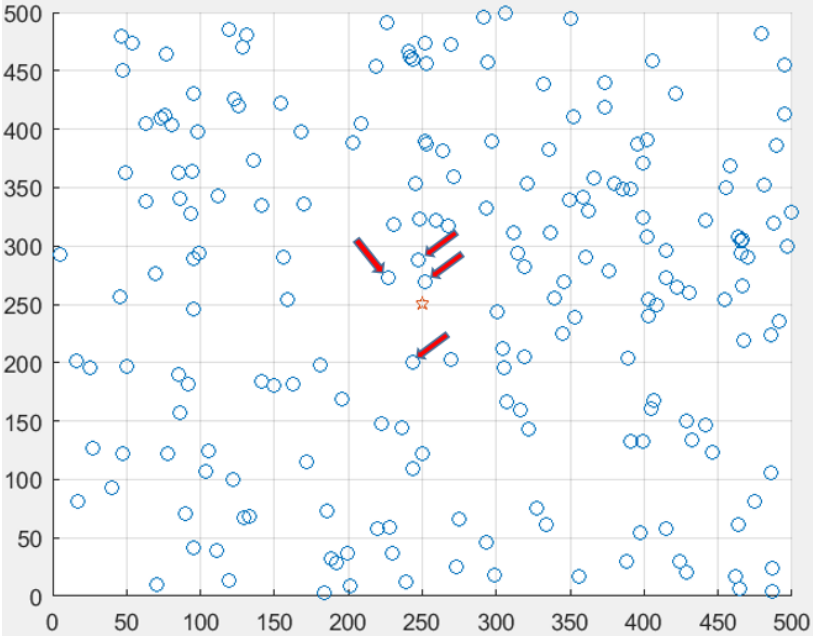

Figure 11. Random Distribution that Fails to Achieve Sink Node Anonymity with Arrows Highlighting Sink Neighbor Nodes

D. CHAPTER SUMMARY

We achieved sink-node anonymity based on the total transmissions and the RREPs sent to RREQs forwarded ratio using a modified LOADng protocol in both uniform spacing and random node distribution. There are sacrifices to power usage, latency, and PDR to achieve anonymity using this approach. The increase in total power consumption is the most dramatic drawback to this approach to anonymity, with an average power ratio of 4.48:1 between the two protocols. There were a small number of cases in the random distribution in which sink-node anonymity was not achieved due to the location of the sink node's neighbors.

VI. CONCLUSIONS AND FUTURE WORK

A. SUMMARY AND CONCLUSIONS

WSNs have been a vital asset for the military for many decades and continue to increase in capabilities and applications. Current sensors employed by the USMC are bulky and complicated to deploy. There are small and inexpensive COTS devices that can fulfill the need for easily deployable wireless sensors. This thesis research was motivated by a desire to enable sink-node anonymity on these computationally restricted devices with minimal change to the standard communication protocol for the sensor nodes. The existing research in sink-node anonymity, such as location-aided routing, extra layers of encryption, or complex routing schemes, add increased computational complexity to the network and require all nodes to have a broader understanding of the network.

LOADng is designed to be lightweight and requires low processing resources at the nodes. Our modified LOADng method achieves sink-node anonymity while preserving the lightweight nature of LOADng for the standard sensor nodes in the network. The majority of the additional computational overhead is assigned to the sink node, which is assumed to have more resources to accomplish the computations. The only additional requirement of the normal sensor nodes is being aware of the *sink flag* for the purpose of acting as the sink node.

Our modified LOADng routing protocol was simulated in MATLAB and compared to the standard LOADng routing protocol. From our simulations, we were able to show that sink-node anonymity was achieved in most network topologies except for a specific case in which the sink-node transmission number exceeded one standard deviation above the mean of that of its neighbors. This only happened when there was a single sink-node neighbor on one side of the sink node and multiple neighbors on the other, forcing traffic through the sink node.

The main tradeoff with our method to achieve sink-node anonymity is the increased power usage due to the extra transmissions per node and extra collisions. There are always tradeoffs to achieve anonymity. We showed in our results that as the number

of transmissions increased (i.e., longer simulations), the average power consumption decreased. This is because the RREQs are a lower proportion of the total transmissions, resulting in fewer collisions proportional to time.

Overall, despite the short-term tradeoffs in power, the modified LOADng routing scheme achieved sink node anonymity for the majority of cases, adding a level of cybersecurity that is not found in the standard LOADng protocol.

B. CONTRIBUTIONS OF THIS THESIS

Our goal was to develop a sink-node anonymity protocol using a reactive routing protocol that would function on IEEE 802.15.4 wireless sensor nodes with highly constrained computational resources. In this thesis research, we have contributed the following to the study of sink-node anonymity:

- Development of a modified LOADng routing protocol, which provides *k*-anonymity to the sink node while limiting the computational overhead for the sensor nodes.
- Simulation of the modified routing algorithm to measure and quantify anonymity and performance versus the standard LOADng protocol.
- Measurement of the performance of the modified routing algorithm compared to the standard LOADng protocol for average route length, latency, power consumption, and PDR.

C. FUTURE WORK

While we successfully achieved sink-node anonymity, there are potential refinements to the protocol to improve the anonymity results and reduce the negative impact on some of the performance metrics.

1. Intelligent Neighbor Selection

Despite distributing the sink node role among the sink node's neighbors, there is still sometimes a large standard deviation within the set of *k* nodes. This shows that the distribution between the nodes in the set is not consistent due to their being chosen randomly. If the sink node tracked the choice of nodes, it could more effectively

distribute the choice evenly among the nodes. This would narrow the standard deviation and make the set of *k* nodes more indistinguishable from one another.

2. Optimized Flooding

Collisions due to flooding are a major cause of increased energy usage and packet drop in the modified LOADng protocol. We utilized classical flooding in our simulation, which is not as efficient as many optimized flooding schemes. A more optimized flooding routine can be developed to lower the overall traffic in the network, thereby reducing the power consumption and collisions. This routine must be carefully designed to ensure that it does not sacrifice anonymity for efficiency. Additionally, more research can be done on the optimal jitter and BEB schemes to further reduce collisions during the flooding of RREQs.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX. SIMULATOR CODE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Main.m
% Sink node anonymity of a wireless sensor network using LOADng
reactive
% routing protocol.
%
% Major Haakensen, Thomas J.
% Student, M.S. Electrical and Computer Engineering
% Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;
clc;

%Declare global variables
global timer;
global time_inc;
global route_table_sink;    %Sink node routing table
global route_table;     %Non-sink nodes routing tables
global pending_txmit;   %[node, source, dest, type, hop_limit, seq_no,
backoff, backoff_num, sink_flag]
global txmitting;       %[node, source, dest, type, hop_limit, seq_no,
txmit_time, sink_flag]
global pending_response; %Stores nodes waiting for a response to a
packet
global txmit_times;     %Stores transmit times for different packets
global neigh;           %Stores neighbor nodes and distances
global time_out;
global RREQ_fwd;   %Tracks nodes that sent/forwarded RREQ. Used in
flooding
global seq_nos;
global sink_RREQ;   %Tracks the times the sink received RREQ from each
node
global success;
global trans_num;   %Tracks the number of transmissions for each node
global trans_num_comp;  %Tracks the number of completed transmissions
for each node
global run;
global node_latency;    %Track latency of transmission for nodes
global latency;     %Tracks total latency
global trans_tot;   %Total non-RREQ transmissions
global retries;
global sim_time;
global collisions;
global out_RREP_ratio;
global trans_tries;

Data_rate = 250000;     %250kbps. Max rate for IEEE 802.15.4
run = 0;
route_len = [0 0];      %Stores average route lengths
N = input('How many nodes?: ');
```

```
field_size = input('What is the field size?: ');
Range = input('What is the transmission range?: ');  % Transmission
range in m
Num_tx = input('How many transmissions?: ');
distro = input('Enter (1) for uniform, (2) for random distribution: ');

disp('Deploying nodes');
if distro == 1
    x = floor(sqrt(N));
    spacing = field_size/x;
    for i = 1:x
        nodes(i*x - x+1:i*x, 1) = i * spacing - spacing;
        nodes(i*x - x+1:i*x, 2) = linspace(0, x*spacing, x);
    end
elseif distro == 2
    nodes = field_size*rand(N, 2);    %Randomly generate node locations
end
nodes(1, 1:2) = field_size/2;       %Puts sink node in center of field
scatter(nodes(2:N, 1), nodes(2:N, 2));  %Display node locations
hold;
scatter(nodes(1, 1), nodes(1, 2), 'p');
grid;

trans_order(1, :) = randi(N-1, 1, Num_tx);
trans_order(1, :) = trans_order(1, :) + 1;    %Increments all by 1 to
exclude the sink node
trans_order(2, 1) = 0;
for i = 2:length(trans_order)   %Generate times nodes will try to send
    trans_order(2, i) = trans_order(2, i-1) + rand*10; %10
end
sim_time = trans_order(2, Num_tx);  %Time of last transmission
%trans_order(1, length(trans_order)) = 2;
trans_order_bak = trans_order;  %Save original trans_order for second
run

%Times assume 6LowPAN header (25 B) + encryption overhead (21 B) +
LOADng header
%127 byte packet takes 4.064 msec at 250 kbps
t_RREQ = 608/Data_rate;  %time to transmit RREQ (76 bytes)
t_RREP = 640/Data_rate;  %time to transmit RREP (80 bytes)
t_Data = 1016/Data_rate; %Full 127 byte data packet
t_RREP_ACK = 512/Data_rate; %time to transmit RREP_ACK (64 bytes)
t_RERR = 608/Data_rate;  %time to transmit RRER (76 bytes)
t_ACK = 512/Data_rate; %time to transmit ACK (64 bytes)

%Times to transmit packets. Index corresponds to type # in txmitting
txmit_times = [t_RREQ, t_RREP, t_Data, t_RREP_ACK, t_RERR, t_ACK];
seq_nos = randi(65536, 1, N); %Generate random 16-bit initial seq_nos
for nodes
seq_nos_bak = seq_nos;  %Save original seq_nos for second run
tx_times = zeros(2, N); %Stores the time each node has been
transmitting
trans_num = zeros(2, N);%Stores total number of transmissions for each
node
trans_num_comp = zeros(2, N);
```

```
trans_tries = zeros(2, N);
retries = zeros(1, N);
latency = [0 0; 0 0];
trans_tot = [0 0; 0 0];
collisions = zeros(2, 11);
out_RREP_ratio = zeros(4, N);

while run < 2  %Loop limits to two runs

route_table = zeros(3, N, N);
route_table_sink = zeros(N, N);
RREQ_fwd = zeros(N, N+2);
neigh = zeros(N,N);
sink_RREQ = zeros(1, N);
success = zeros(1, N);
node_latency = zeros(1, N);
time_inc = 0.0001; %Time increment (0.0001 = 100 usec)
timer = 0;  %Initialize global timer
time_out = 2;
tx_fin = [];

build_neigh_assoc(N, Range, nodes);   % Build layer-2 neighbor
associations

fprintf('Beginning simulation run %d;\n', run);

while ~isempty(trans_order) || ~isempty(pending_txmit) ||
~isempty(txmitting) || ~isempty(pending_response)
    %Main simulation loop. Will run for modified then normal routing.
    %Add trans_order nodes with times <= 0 to pending_txmit
    while ~isempty(trans_order) && trans_order(2, 1) <= 0    %While the
first node has transmit time <= 0
        pending_txmit = [pending_txmit; [trans_order(1, 1),
trans_order(1, 1), 1, 3, 256, seq_nos(trans_order(1)), 0, 0, 0]];
        sink_flag = find(route_table(3, :,
pending_txmit(size(pending_txmit, 1), 1)));
        if run == 0 && ~isempty(sink_flag) %If first run and there is
route to sink
            pending_txmit(size(pending_txmit, 1), [3 9]) = [sink_flag
1]; %Dest is node with sink flag
        elseif run == 0
            pending_txmit(size(pending_txmit, 1), 4) = 1; %Type is RREQ
            %fprintf('%f: RREQ sent, node: %d;\n', timer,
pending_txmit(size(pending_txmit, 1), 1));
        end
        if run == 1 && route_table(1, pending_txmit(size(pending_txmit,
1), 3), pending_txmit(size(pending_txmit, 1), 1)) == 0
        %If second run and no route to sink
            pending_txmit(size(pending_txmit, 1), 4) = 1; %Type is RREQ
        end
        seq_nos(trans_order(1)) = seq_nos(trans_order(1))+1;
        trans_order(:, 1) = []; %Clear first column of trans_order
    end
```

```
    if ~isempty(txmitting)
        tx_fin = find(txmitting(:, 7) <= 0);    %Get indices of nodes
finished transmitting
    end

    if ~isempty(tx_fin)   %If there are nodes finished transmitting
        if run == 0     %If first run
            tx_complete_mod(tx_fin);    %Modified routing
        elseif run == 1    %If second run
            tx_complete(tx_fin);    %Normal LOADng routing
        end
        tx_fin = [];    %Clear tx_fin
    end

    if (~isempty(find(pending_txmit(:, 7) <= 0)))
        CSMA_CA();  %Checks if the "pending_txmit" nodes can transmit
    end

    if size(txmitting, 1) >= 2  %If there are at least 2 transmitting
nodes
        collision(txmitting(:, 1), neigh);    %Check to see if there is
a collision
    end

    %Locate expired timers in pending_response
    if ~isempty(pending_response) && ~isempty(find(pending_response(:,
4) <= 0))
        pend_resp_exp = find(pending_response(:, 4) <= 0);
        if ~isempty(pend_resp_exp)
            for i = 1:length(pend_resp_exp)
                pr = pending_response(pend_resp_exp(i), :);
                if pr(5) >= 4   %Tried more than 4 times
                    fprintf('%f: Transmission failure, node: %d, type:
%d, dest: %d;\n', timer, pr(1), pr(3), pr(2));
                else            %Tried <= 5 times
                    sf = route_table(3, pr(2), pr(1));  %Sink flag
                    pending_txmit = [pending_txmit; [pr(1), pr(1),
pr(2), pr(3), 256, seq_nos(pr(1)), 0, pr(5)+1, sf]];
                    fprintf('%f: Retransmission, node: %d, type: %d,
dest: %d, attempt: %d;\n', timer, pr(1), pr(3), pr(2), pr(5)+1);
                end
                if pending_response(pend_resp_exp(i), 3) == 1
                    RREQ_fwd(pending_response(pend_resp_exp(i), 1), :)
= 0;
                end
            end
            pending_response(pend_resp_exp, :) = [];
        end
    end

    %Adjust timers by time_inc
    timer = timer + time_inc;
    pending_txmit(:, 7) = pending_txmit(:, 7) - time_inc;   %Backoff -
time_inc
```

```
    txmitting(:, 7) = txmitting(:, 7) - time_inc;   %t_tx - time_inc
    trans_order(2, :) = trans_order(2, :) - time_inc;
    node_latency(:) = node_latency(:) + time_inc;
    if ~isempty(pending_response)     %If pending_response not empty
        pending_response(:, 4) = pending_response(:, 4) - time_inc;
%Decrement pending timeouts
    end
    RREQ_fwd(:, 2) = RREQ_fwd(:, 2) - time_inc;

    %Track the times that the nodes have been transmitting
    tx_ind = txmitting(:, 1);
    tx_times(run+1, tx_ind) = tx_times(run+1, tx_ind) + time_inc;
end

disp('****************************************************************'
);
fprintf('Simulation run %d complete.\n', run);
disp('****************************************************************'
);

run = run + 1;
route_len(run) = mean(mean(nonzeros(route_table(2, :, :))));
trans_order = trans_order_bak;  %Restore trans_order for second run
seq_nos = seq_nos_bak;          %Restore seq_nos for second run

end

sink_nbrs = neigh(1, 1:2:size(neigh, 2));
sink_nbrs = sink_nbrs(sink_nbrs>0);
not_sink_nbrs = linspace(1, N, N);
temp = ~ismember(not_sink_nbrs, sink_nbrs);
not_sink_nbrs = not_sink_nbrs .* temp;
not_sink_nbrs = not_sink_nbrs(not_sink_nbrs>0);
out_sink_trans_num = trans_num_comp(:, sink_nbrs);
out_node_trans_num = trans_num_comp(:, not_sink_nbrs);

%Measure power usage (mWh)
out_power = tx_times * (0.0000008 / 0.0000032);
out_power = out_power + (0.000006 * trans_num);
out_power = out_power + (trans_tries * 0.00004766);

%% Get node distances from sink
node_dist = zeros(N,2);
trans_hist = zeros(2,N);
trans_hist_sink = zeros(2,N);
for i = 1:N
    node_dist(i, 1) = i;
    node_dist(i, 2) = sqrt((nodes(1,1)-nodes(i,1))^2 + (nodes(1,2)-
nodes(i,2))^2);
end
node_dist = sortrows(node_dist, 2);

trans_hist_sink(1, N/2) = trans_num_comp(1, 1);
trans_hist_sink(2, N/2) = trans_num_comp(2, 1);
```

```
for i = 1:N/2-1
    trans_hist(1, N/2-i) = trans_num_comp(1, node_dist(2*i, 1));
    trans_hist(1, N/2+i) = trans_num_comp(1, node_dist(2*i+1, 1));
    trans_hist(2, N/2-i) = trans_num_comp(2, node_dist(2*i, 1));
    trans_hist(2, N/2+i) = trans_num_comp(2, node_dist(2*i+1, 1));
end
figure;
bar(trans_hist(1, :));
hold;
bar(trans_hist_sink(1, :), 'r');

figure;
bar(trans_hist(2, :));
hold;
bar(trans_hist_sink(2, :), 'r');
disp('Simulation complete');

**************************************************************************
function build_neigh_assoc(N, Range, nodes)
%This function builds the neigh matrix, which is a 2D array which
%contains the neighbors and distance of all nodes within transmit range
of
%each node.

global neigh;

fprintf('Building node neighbor associations\n');
for i = 1:N     %i is address of node
    ind = 1;
    for j = 1:N     %j is address of neighbor
        D = Range + 1;  %Initial condition: D > Range
        if i~=j     %Prevents node being compared to self
            %Compute distance from node i to node j
            D = sqrt((nodes(i,1)-nodes(j,1))^2 + (nodes(i,2)-
nodes(j,2))^2);
        end
        if D <= Range   %Create neighbor array.
            neigh(i, ind) = j;
            neigh(i, ind+1) = D;
            ind = ind + 2;
        end
    end
end
fprintf('Neighbor associations created\n');
end

**************************************************************************
function CSMA_CA()
%This function checks all of the nodes in the "pending_txmit" matrix.
%If the node has a backoff timer <= 0, and there are no other neighbor
%nodes transmitting, the node is moved to the "txmitting" matrix with
%tx_time equal to the transmit time for the type of packet.
%If one or more neighbor nodes are transmitting, the "pending_txmit"
%node's backoff is set to the highest tx_time for the transmitting
```

```
%neighbor(s), simulating waiting for the medium to become idle.

global pending_txmit;
global txmitting;
global txmit_times;
global neigh;
global trans_num;
global run;
global trans_tries;

ind = 1;
while (size(pending_txmit, 1) >= ind) %While ind <= to size of
pending_txmit

    if pending_txmit(ind, 7) > 0 %If node is in backoff
        ind = ind + 1;
        continue;
    end
    trans_tries(run+1, pending_txmit(ind, 1)) = trans_tries(run+1,
pending_txmit(ind, 1)) + 1;
    tx_temp = 0;
    idle = 1;       %Medium idle flag initially set to 1
    neigh_temp = neigh(pending_txmit(ind, 1), 1:2:size(neigh, 2));
%Get neighbor addresses

    if ~isempty(txmitting) && ismember(pending_txmit(ind, 1),
txmitting(:, 1))
    %If the node is already transmitting
        x = find(txmitting(:, 1) == pending_txmit(ind, 1));
        %Set backoff = to the current transmit time
        pending_txmit(ind, 7) = txmitting(x, 7);
        ind = ind + 1;
        continue;   %Go to next loop iteration
    end

    if ~isempty(txmitting)    %If txmitting is not empty
        tx_temp = ismember(txmitting(:, 1), neigh_temp);  %Find
transmitting nodes that are neighbors
    end

    if sum(tx_temp) > 0 %If there are transmitting nodes in range
        idle = 0;       %Set idle flag to 0
        tx_times_temp = tx_temp .* txmitting(:, 7);  %Array of transmit
times for neighbor nodes
        pending_txmit(ind, 7) = max(tx_times_temp);  %Set backoff to
largest transmit time
    end

    if (idle && (pending_txmit(ind, 7) <= 0))   %Medium is idle &&
backoff <= 0
        trans_num(run+1, pending_txmit(ind, 1)) = trans_num(run+1,
pending_txmit(ind, 1))+1;    %Track transmission
        txmitting(size(txmitting, 1)+1, [1:6, 8:9]) =
pending_txmit(ind, [1:6, 8:9]);  %Move to txmitting queue
```

```
        txmitting(end, 7) = txmit_times(pending_txmit(ind, 4)); %Set
transmit time based on type size(txmitting, 1)
        pending_txmit(ind, :) = [];    %Erase from pending_xmit
    else        %Medium not idle or backoff >= 0
        ind = ind + 1;  %Increment "ind" here and not if idle
    end
end
end

**************************************************************************
function collision(txmt_nodes, nbrs)
%This function tests to see if there is a collision due to the hidden-
node
%problem. It checks if any of the transmitting nodes have common
%neighbors. If they do, the common neighbor signals a collision and all
%transmititng nodes in its range are moved back into pending_txmit.

global txmitting;
global pending_txmit;
global retries;
global sim_time;
global timer;
global collisions;
global run;

BEB = [0 0.0041 0.0082 0.0143 0.0184 0.0225];
nbr_coll = [];
nbrs = nbrs(:, 1:2:size(nbrs, 2));  %Clear ranges
coll_ind = ceil(timer/(sim_time/10));

for i = 1:length(txmt_nodes)
    for j = i+1:length(txmt_nodes)
        node1 = nbrs(txmt_nodes(i), :);
        node1 = node1(node1 ~= 0);
        node2 = nbrs(txmt_nodes(j), :);
        node2 = node2(node2 ~= 0);
        memb_test = ismember(node1, node2); %Check for common neighbors
        if sum(memb_test)  %If there are shared neighbors
            coll_nodes = memb_test .* node1;      %Get node(s) where
collision occurred
            coll_nodes = coll_nodes(coll_nodes ~= 0);  %Clear zeros
            for k = 1:length(coll_nodes)
                fprintf('%f: Collision, node: %d;\n', timer,
coll_nodes(k));
                %Get index(es)in txmt_nodes of transmitting neighbor
nodes
                nbr_tx_nodes = find(ismember(txmt_nodes,
nbrs(coll_nodes(k), :)));
                new_node = ~ismember(nbr_tx_nodes, nbr_coll); %Find
node index(es) not already in nbr_coll
                new_node = new_node .* nbr_tx_nodes;
                new_node = new_node(new_node ~= 0);      %Clear zeros
                collisions(run+1, coll_ind) = collisions(run+1,
coll_ind) + 1;
```

```
                if new_node
                    %Add new_node nodes to nbr_coll
                    nbr_coll = vertcat(nbr_coll, new_node); %Index of
txmitting nodes who had collisions
                end
            end
        end
    end
end
for m =1:length(nbr_coll) %For all transmitting neighbor nodes
    %Move transmitting neighbors to pending_txmit
    retries(txmitting(nbr_coll(m), 1)) = retries(txmitting(nbr_coll(m),
1)) + 1;
    backoff = BEB(randi(retries(txmitting(nbr_coll(m), 1)) + 1));
    %backoff = (0.0082*rand + 0.0041)*retries(txmitting(nbr_coll(m),
1));
    %Alternate backoff algorithm
    if retries(txmitting(nbr_coll(m), 1)) < 5
        pending_txmit = [pending_txmit; [txmitting(nbr_coll(m), 1:6),
backoff, txmitting(nbr_coll(m), 8:9)]];
    else
        retries(txmitting(nbr_coll(m), 1)) = 0;
        disp('collision failure'); %Track failure due to collisions
    end
end
txmitting(nbr_coll, :) = [];
end

**************************************************************************
function tx_complete(tx_fin)
%This function is called when nodes have completed transmitting in
normal
%LOADng, i.e., when their transmit timers reached 0. It processes the
packet
%according to the node that is receiveing.  'tx_fin' holds the
index(es) of
%the node(s) in 'txmitting' that have finished transmitting.

global txmitting;
global route_table;
global route_table_sink;
global pending_response;
global time_out;
global RREQ_fwd;
global node_latency;
global trans_num_comp;
global out_RREP_ratio;

for i = 1:length(tx_fin)
    if txmitting(tx_fin(i), 4) == 1 %If type == RREQ
        %Count RREQs forwarded
        out_RREP_ratio(4, (txmitting(tx_fin(i), 1))) =
out_RREP_ratio(4, (txmitting(tx_fin(i), 1))) + 1;
    end
```

```
    trans_num_comp(2, txmitting(tx_fin(i), 1)) = trans_num_comp(2,
txmitting(tx_fin(i), 1)) + 1;
    if txmitting(tx_fin(i), 1) == txmitting(tx_fin(i), 2) &&
txmitting(tx_fin(i), 4) > 1
    %If txmitting node is source and the type ~= RREQ
        %Start measuring latency
        node_latency(txmitting(tx_fin(i), 1)) = 0;  %Reset the latency
time for node
        if txmitting(tx_fin(i), 4) == 2 %If type == RREP
            %Count RREPs initiated
            out_RREP_ratio(3, (txmitting(tx_fin(i), 1))) =
out_RREP_ratio(3, (txmitting(tx_fin(i), 1))) + 1;
        end
    end
    txmitting(tx_fin(i), 5) = txmitting(tx_fin(i), 5) - 1; %Dec hop
limit
    if txmitting(tx_fin(i), 1) == txmitting(tx_fin(i), 2) &&
txmitting(tx_fin(i), 4) <= 3
    %If txmitting node is source and the type == RREQ, RREP, or Data
        %Put into pending response array
        if ~isempty(pending_response)
            pr_ind = find(pending_response(:, 1) ==
txmitting(tx_fin(i), 1));
            if isempty(pr_ind) || ~ismember(txmitting(tx_fin(i), 3),
pending_response(pr_ind, 2))
            %If node not already in pending_response for this
transmission
                pending_response = [pending_response;
[txmitting(tx_fin(i), [1, 3, 4]), time_out, txmitting(tx_fin(i), 8)]];
            end
        else
            pending_response = [txmitting(tx_fin(i), [1, 3, 4]),
time_out, txmitting(tx_fin(i),8)];
        end
    end

    %Get next hop
    if (txmitting(tx_fin(i), 1) == 1) && (txmitting(tx_fin(i), 3) ~= 1)
    %If sink node, choose lowest metric
        route_list = route_table_sink(:, txmitting(tx_fin(i), 3));
        next_hop = find(route_list);    %Get non-zero element indexes
        if ~isempty(next_hop)
            next_hop = find(route_list == min(route_list(next_hop)));
%Get lowest non-zero metric index
            if length(next_hop) > 1
                next_hop = next_hop(randi(length(next_hop))); %In case
of multiple with same metric, randomly choose one
            end
        end
    else    %Not the sink node
        next_hop = route_table(1, txmitting(tx_fin(i), 3),
txmitting(tx_fin(i), 1));
    end

    if txmitting(tx_fin(i), 4) == 1     %Type is RREQ
```

```
        flood(txmitting(tx_fin(i), :), tx_fin(i));
    else    %Type is RREP, data, RREP_ACK, RERR, or ACK
        unicast(next_hop, txmitting(tx_fin(i), :), tx_fin(i));
        if txmitting(tx_fin(i), 1) == 1 && txmitting(tx_fin(i), 4) == 2
        %If this is sink node and RREP
            ind = find(RREQ_fwd(txmitting(tx_fin(i), 3), :) == 0, 1);
%Index of first zero in row
            RREQ_fwd(txmitting(tx_fin(i), 3), ind) =
txmitting(tx_fin(i), 1);
        end
    end
end
txmitting(tx_fin, :) = [];   %Remove tx_fin nodes from txmitting
end

**************************************************************************
function tx_complete_mod(tx_fin)
%This function is called when nodes have completed transmitting in
modified
%LOADng, i.e., when their transmit timers reached 0. It processes the
packet
%according to the node that is receiveing.  'tx_fin' holds the
index(es) of
%the node(s) in 'txmitting' that have finished transmitting.

global txmitting;
global route_table;
global route_table_sink;
global pending_response;
global time_out;
global sink_RREQ;   %Tracks the times the sink received RREQ from each
node
global node_latency;
global trans_num_comp;
global out_RREP_ratio;

for i = 1:length(tx_fin)
    if txmitting(tx_fin(i), 4) == 1 %If type == RREQ
        %Count RREQs forwarded
        out_RREP_ratio(2, (txmitting(tx_fin(i), 1))) =
out_RREP_ratio(2, (txmitting(tx_fin(i), 1))) + 1;
    end
    trans_num_comp(1, txmitting(tx_fin(i), 1)) = trans_num_comp(1,
txmitting(tx_fin(i), 1)) + 1;
    if txmitting(tx_fin(i), 1) == txmitting(tx_fin(i), 2) &&
txmitting(tx_fin(i), 4) > 1
    %If txmitting node is source and the type ~= RREQ
        %Start measuring latency
        node_latency(txmitting(tx_fin(i), 1)) = 0;  %Reset the latency
time for node
        if txmitting(tx_fin(i), 4) == 2 %If type == RREP
            %Count RREPs initiated
            out_RREP_ratio(1, (txmitting(tx_fin(i), 1))) =
out_RREP_ratio(1, (txmitting(tx_fin(i), 1))) + 1;
        end
```

```
    end
    txmitting(tx_fin(i), 5) = txmitting(tx_fin(i), 5) - 1; %Dec hop
limit
    if txmitting(tx_fin(i), 1) == txmitting(tx_fin(i), 2) && ...
            txmitting(tx_fin(i), 4) <= 3 && txmitting(tx_fin(i), 3) ~=
65535
    %If txmitting node is source && type == RREQ, RREP, or Data && not
    %broadcast
        %Put into pending response array
        if ~isempty(pending_response)
            pr_ind = find(pending_response(:, 1) ==
txmitting(tx_fin(i), 1));
            if isempty(pr_ind) || ~ismember(txmitting(tx_fin(i), 3),
pending_response(pr_ind, 2))
            %If node not already in pending_response for this
transmission
                pending_response = [pending_response;
[txmitting(tx_fin(i), [1, 3, 4]), time_out, txmitting(tx_fin(i), 8)]];
            end
        else
            pending_response = [txmitting(tx_fin(i), [1, 3, 4]),
time_out, txmitting(tx_fin(i),8)];
        end
    end

    if txmitting(tx_fin(i), 3) == 65535 %If dest is broadcast address
        broadcast(txmitting(tx_fin(i), :));
        continue;  %Go to next loop iteration
    end

    %Get next hop
    if (txmitting(tx_fin(i), 1) == 1) && (txmitting(tx_fin(i), 3) ~= 1)
    %If sink node, choose lowest metric
        route_list = route_table_sink(:, txmitting(tx_fin(i), 3));
        next_hop = find(route_list);    %Get non-zero element indexes
        if ~isempty(next_hop)
            next_hop = find(route_list == min(route_list(next_hop)));
%Get lowest non-zero metric index
            if length(next_hop) > 1
                next_hop = next_hop(randi(length(next_hop))); %In case
of multiple with same metric, randomly choose one
            end
        end
    else    %Not the sink node
        next_hop = route_table(1, txmitting(tx_fin(i), 3),
txmitting(tx_fin(i), 1));
    end

    if txmitting(tx_fin(i), 4) == 1     %Type is RREQ
        if txmitting(tx_fin(i), 1) == 1 %If txmitting node is sink node
            %Dec hop limit to make return route metric through sink
higher.
            txmitting(tx_fin(i), 5) = txmitting(tx_fin(i), 5) - 2;
            if sink_RREQ(txmitting(tx_fin(i), 2)) == 1
```

```
                %If this is not the first RREQ received from this
source node
                sink_flood(txmitting(tx_fin(i), :), tx_fin(i));
                sink_RREQ(txmitting(tx_fin(i), 2)) = 2;  %Prevent
another RREP to this node
            else
                %Flood packet pretending not to be the sink
                flood_mod(txmitting(tx_fin(i), :), tx_fin(i));
                sink_RREQ(txmitting(tx_fin(i), 2)) = 1;
            end
        else    %Txmitting node not the sink node
            flood_mod(txmitting(tx_fin(i), :), tx_fin(i));
        end
    else    %Type is RREP, data, RREP_ACK, RERR, or ACK
        unicast(next_hop, txmitting(tx_fin(i), :), tx_fin(i));
    end
end
txmitting(tx_fin, :) = [];   %Remove tx_fin nodes from txmitting
end

**************************************************************************
function flood(RREQ_pkt, fin_add)
%This function is called upon a node's completion of transmitting a
RREQ
%packet. It finds its neighbor nodes that have not forwarded it this
%particular RREQ packet and adds them to pending_txmit (type = RREQ).
%It is passed the row from 'txmitting' of the node which has completed
%transmitting and the row index. Standard LOADng.

global pending_txmit;
global neigh;
global RREQ_fwd;
global time_out;
global seq_nos;
global txmit_times;
global txmitting;

if RREQ_pkt(1) == RREQ_pkt(2)   %If node is the source
    RREQ_fwd(RREQ_pkt(2), 1:3) = [RREQ_pkt(6) time_out RREQ_pkt(1)];
%Add this RREQ to 'RREQ_fwd'
end

if RREQ_fwd(RREQ_pkt(2), 2) <=0    %If timeout exceeded
    RREQ_fwd(RREQ_pkt(2), :) = 0;  %Clear RREQ_fwd for this RREQ
    %Find all pending_txmit nodes for this RREQ (same seq num)
    RREQ_exp = find(pending_txmit(:, 6) == RREQ_pkt(6));
    %Clear all pending_txmit nodes for this RREQ
    pending_txmit(RREQ_exp, :) = [];
    txmitting(RREQ_exp, :) = [];
    return;     %Exit the function
end

%Add tx_complete node to RREQ_fwd
```

```
ind = find(RREQ_fwd(RREQ_pkt(2), :) == 0, 1);   %Index of first zero in
row
RREQ_fwd(RREQ_pkt(2), ind) = RREQ_pkt(1);

%Find all neighbors not in RREQ_fwd
neigh_temp = neigh(RREQ_pkt(1), 1:2:size(neigh, 2));   %Get neighbor
addresses
%Find neighbor nodes that haven't forwarded RREQ
RREQ_neigh = ~ismember(neigh_temp, RREQ_fwd(RREQ_pkt(2),
3:size(RREQ_fwd, 2)));
RREQ_neigh = RREQ_neigh .* neigh_temp;  %Only choose neighbors that
haven't forwarded RREQ
RREQ_neigh = RREQ_neigh(RREQ_neigh ~= 0); %Clear zero elements

if ~isempty(RREQ_neigh)
    delay = 0;
    for i = 1:length(RREQ_neigh)    %Neighbor nodes to pending_trans
(type = RREQ)
        if ~ismember(RREQ_neigh(i), pending_txmit(:, 1))
        %If the neighbor node is not already in pending_txmit for this
RREQ
            if RREQ_neigh(i) == RREQ_pkt(3) %If neighbor node is
destination
                pending_txmit = [pending_txmit; [RREQ_neigh(i),
RREQ_neigh(i), RREQ_pkt(2), 2, 256, seq_nos(RREQ_neigh(i)), 0, 0,
RREQ_pkt(9)]];
                seq_nos(RREQ_neigh(i)) = seq_nos(RREQ_neigh(i))+1;
            else
                pending_txmit = [pending_txmit; [RREQ_neigh(i),
RREQ_pkt(2:6), delay, 0, RREQ_pkt(9)]];
            end
            %update_route_table(RREQ_neigh(i), fin_add);    %Update the
route table of receiving node
            delay = randi(size(RREQ_neigh)) * txmit_times(3);
            %delay = delay + txmit_times(1);
        end
        update_route_table(RREQ_neigh(i), fin_add);    %Update the
route table of receiving node
    end
end

end

**************************************************************************
function flood_mod(RREQ_pkt, fin_add)
%This function is called upon a node's completion of transmitting a
RREQ
%packet. It finds its neighbor nodes that have not forwarded it this
%particular RREQ packet and adds them to pending_txmit (type = RREQ).
%It is passed the row from 'txmitting' of the node which has completed
%transmitting and the row index. Modified LOADng.

global pending_txmit;
global neigh;
```

```
global RREQ_fwd;
global time_out;
global seq_nos;
global txmit_times;
global txmitting;

if RREQ_pkt(1) == RREQ_pkt(2)   %If node is the source
    RREQ_fwd(RREQ_pkt(2), 1:3) = [RREQ_pkt(6) time_out RREQ_pkt(1)];
%Add this RREQ to 'RREQ_fwd'
end

if RREQ_fwd(RREQ_pkt(2), 2) <=0    %If timeout exceeded
    RREQ_fwd(RREQ_pkt(2), :) = 0;  %Clear RREQ_fwd for this RREQ
    %Find all pending_txmit nodes for this RREQ (same seq num)
    RREQ_exp = find(pending_txmit(:, 6) == RREQ_pkt(6));
    %Clear all pending_txmit and txmitting nodes for this RREQ
    pending_txmit(RREQ_exp, :) = [];
    RREQ_exp = find(txmitting(:, 6) == RREQ_pkt(6));
    txmitting(RREQ_exp, :) = [];
    return;     %Exit the function
end

%Add tx_complete node to RREQ_fwd
ind = find(RREQ_fwd(RREQ_pkt(2), :) == 0, 1);   %Index of first zero in
row
RREQ_fwd(RREQ_pkt(2), ind) = RREQ_pkt(1);

%Find all neighbors not in RREQ_fwd
neigh_temp = neigh(RREQ_pkt(1), 1:2:size(neigh, 2));   %Get neighbor
addresses
%Find neighbor nodes that haven't forwarded RREQ
RREQ_neigh = ~ismember(neigh_temp, RREQ_fwd(RREQ_pkt(2),
3:size(RREQ_fwd, 2)));
RREQ_neigh = RREQ_neigh .* neigh_temp;  %Only choose neighbors that
haven't forwarded RREQ
RREQ_neigh = RREQ_neigh(RREQ_neigh ~= 0); %Clear zero elements
if ismember(1, neigh_temp) && ~ismember(1, RREQ_neigh)
%If the sink node is a neighbor and has previously forwarded the RREQ
    update_route_table(1, fin_add);
end

if ~isempty(RREQ_neigh)
    delay = 0;
    for i = 1:length(RREQ_neigh)    %Neighbor nodes to pending_trans
(type = RREQ)
        if ~ismember(RREQ_neigh(i), pending_txmit(:, 1))
        %If the neighbor node is not already in pending_txmit for this
RREQ
            if RREQ_neigh(i) == RREQ_pkt(3) && RREQ_neigh(i) ~= 1 &&
RREQ_pkt(9) ~= 1 %Difference from flood routine
            %If neighbor node is destination && not the sink node &&
not
            %sink flood
```

```
                pending_txmit = [pending_txmit; [RREQ_neigh(i),
RREQ_neigh(i), RREQ_pkt(2), 2, 256, seq_nos(RREQ_neigh(i)), 0, 0,
RREQ_pkt(9)]];
                seq_nos(RREQ_neigh(i)) = seq_nos(RREQ_neigh(i))+1;
            else
                pending_txmit = [pending_txmit; [RREQ_neigh(i),
RREQ_pkt(2:6), delay, 0, RREQ_pkt(9)]];
            end
            %update_route_table(RREQ_neigh(i), fin_add);    %Update the
route table of receiving node
            delay = randi(size(RREQ_neigh)) * txmit_times(3);
            %delay = delay + txmit_times(3);     %Add jitter
        end
        update_route_table(RREQ_neigh(i), fin_add);    %Update the
route table of receiving node
    end
end

end

**************************************************************************
function sink_flood(RREQ_pkt, fin_add)
%This function is called upon the sink node's completion of
transmitting a
%RREQ packet. It first finds all of its neighbor nodes that have not
%forwarded it this particular RREQ packet. It then adds itself to the
%list, randomly chooses a node from the list, alters the RREQ to
reflect
%this new source, and continues flooding the RREQ. It then adds the
%neighbor nodes to pending_txmit (type = RREQ). It is passed the row
from
%'txmitting' of the node which has completed transmitting and the row
index.

global pending_txmit;
global neigh;
global RREQ_fwd;
global time_out;
global route_table_sink;
global seq_nos;
global txmit_times;
global txmitting;

if RREQ_pkt(1) == RREQ_pkt(2)   %If node is the source
    RREQ_fwd(RREQ_pkt(2), 1:3) = [RREQ_pkt(6) time_out RREQ_pkt(1)];
%Add this RREQ to 'RREQ_fwd'
end

if RREQ_fwd(RREQ_pkt(2), 2) <=0    %If timeout exceeded
    RREQ_fwd(RREQ_pkt(2), :) = 0;  %Clear RREQ_fwd for this RREQ
    RREQ_exp = find(pending_txmit(:, 6) == RREQ_pkt(6));
    %Clear all pending_txmit nodes for this RREQ
    pending_txmit(RREQ_exp, :) = [];
    txmitting(RREQ_exp, :) = [];
```

```
    return;     %Exit the function
end

%Add tx_complete node to RREQ_fwd
ind = find(RREQ_fwd(RREQ_pkt(2), :) == 0, 1);   %Index of first zero in
row
RREQ_fwd(RREQ_pkt(2), ind) = RREQ_pkt(1);

%Find all neighbors not in RREQ_trans
neigh_temp = neigh(RREQ_pkt(1), 1:2:size(neigh, 2));   %Get neighbor
addresses
%Find neighbor nodes that haven't forwarded RREQ
RREQ_neigh = ~ismember(neigh_temp, RREQ_fwd(RREQ_pkt(2),
3:size(RREQ_fwd, 2)));
RREQ_neigh = RREQ_neigh .* neigh_temp;  %Only choose neighbors that
haven't forwarded RREQ
RREQ_neigh = RREQ_neigh(RREQ_neigh ~= 0); %Clear zero elements
RREQ_neigh = RREQ_neigh(route_table_sink(RREQ_neigh, RREQ_pkt(2)) > 0);
%Make sure nodes had route to source
if isempty(RREQ_neigh) || randi(2) == 1    %Choose to add sink 33% of
the time
    RREQ_neigh= horzcat(RREQ_neigh, 1);    %Add sink node to the array
end
%Choose random neighbor
rand_neigh = RREQ_neigh(randi(size(RREQ_neigh, 2)));
%Alter RREQ packet (new dest and sink bit == 1)
RREQ_pkt([3 9]) = [rand_neigh 1];
%Remove sink node to prevent loading sink RREQ back into pending_txmit
RREQ_neigh = RREQ_neigh(RREQ_neigh>1);

if ismember(rand_neigh, pending_txmit(:, 1))    %If the chosen node is
in pending_txmit
    %Remove the node from pending_txmit
    pending_txmit(find(pending_txmit(:, 1) == rand_neigh), :) = [];
end

%Put into pending_txmit
if rand_neigh == 1  %If sink chose self
    pending_txmit = [pending_txmit; [1, 1, RREQ_pkt(2), 2, 256,
seq_nos(1), 0, 0, RREQ_pkt(9)]];
    return;     %Leave routine. Don't want to flood anymore
end

%Decrement hop-limit to lower chance of sink node being used in return
route
RREQ_pkt(5) = RREQ_pkt(5) - 1;

if ~isempty(RREQ_neigh)
    delay = 0;
    for i = 1:length(RREQ_neigh)    %Neighbor nodes to pending_trans
(type = RREQ)
        if RREQ_neigh(i) == RREQ_pkt(3) %If neighbor node is
destination
```

```
            pending_txmit = [pending_txmit; [RREQ_neigh(i),
RREQ_neigh(i), RREQ_pkt(2), 2, 256, seq_nos(RREQ_pkt(3)), delay, 0,
RREQ_pkt(9)]];
            seq_nos(RREQ_pkt(3)) = seq_nos(RREQ_pkt(3))+1;
        elseif ~ismember(RREQ_neigh(i), pending_txmit(:, 1))
            %If not the destination and the node is not in
pending_txmit
            pending_txmit = [pending_txmit; [RREQ_neigh(i),
RREQ_pkt(2:6), delay, 0, RREQ_pkt(9)]];
            delay = delay + txmit_times(3);
        end
        update_route_table(RREQ_neigh(i), fin_add);    %Update the
route table of receiving node
    end
end

end

**************************************************************************
function unicast(next_hop, txmit_pkt, fin_add)
%This function is called when the packet is a type that is sent via a
%unicast.  "next_hop" is the next hop.  "txmit_pkt" is the line from
%"txmitting" for the node in question.

global pending_txmit;
global seq_nos;
global pending_response;
global timer;
global txmit_times;
global success;
global node_latency;
global latency;
global trans_tot;
global txmitting;
global run;

delay = 0;
if txmit_pkt(1) == txmit_pkt(2)
    trans_tot(2, run+1) = trans_tot(2, run+1) + 1;
end
if next_hop == txmit_pkt(3) %If next hop is dest
    latency(run+1) = latency(run+1) + node_latency(txmit_pkt(2)); %Add
packet latency to total
    trans_tot(1, run+1) = trans_tot(1, run+1) + 1;  %Increment
trans_tot
    if ~isempty(pending_response)
        resp = find(pending_response(:, 1) == next_hop);    %Check if
dest is pending a response
        if ~isempty(resp)   %If node pending response
            pending_response(resp, :) = [];   %Remove from
pending_response
        end
    end
    sink_flag = txmit_pkt(9);
```

```
    switch txmit_pkt(4)
        case 2  %RREP, respond with RREP_ACK then data
            pending_txmit = [pending_txmit; [next_hop, next_hop,
txmit_pkt(2), 4, 256, seq_nos(next_hop), 0, 0, txmit_pkt(9)]];
            seq_nos(next_hop) = seq_nos(next_hop) + 1;
            fprintf('%f: Packet received, node: %d, from: %d, type:
RREP;\n', timer, next_hop, txmit_pkt(2));
            pending_txmit = [pending_txmit; [next_hop, next_hop,
txmit_pkt(2), 3, 256, seq_nos(next_hop), txmit_times(4), 0,
txmit_pkt(9)]];
            seq_nos(next_hop) = seq_nos(next_hop) + 1;
        case 3  %Data, respond with ACK
            if sink_flag
                pending_txmit = [pending_txmit; [next_hop,
txmit_pkt(2), 65535, txmit_pkt(4:9)]];
                pending_txmit(end, 3) = 65535;
                delay = txmit_times(3);
            end
            pending_txmit = [pending_txmit; [next_hop, next_hop,
txmit_pkt(2), 6, 256, seq_nos(next_hop), delay, 0, txmit_pkt(9)]];
            seq_nos(next_hop) = seq_nos(next_hop) + 1;
            fprintf('%f: Packet received, node: %d, from: %d, type:
Data;\n', timer, next_hop, txmit_pkt(2));
        case 4  %RREP_ACK
            fprintf('%f: Packet received, node: %d, from: %d, type:
RREP_ACK;\n', timer, next_hop, txmit_pkt(2));
        case 5  %RERR
            fprintf('%f: Packet received, node: %d, from: %d, type:
RERR;\n', timer, next_hop, txmit_pkt(2));
        case 6  %ACK
            fprintf('%f: Packet received, node: %d, from: %d, type:
ACK;\n', timer, next_hop, txmit_pkt(2));
            success(next_hop) = success(next_hop) + 1;
    end
else    %Next hop not dest
    if isempty(pending_txmit)   %If pending_txmit = []
        pending_txmit = [next_hop, txmit_pkt(2:6), 0, 0, txmit_pkt(9)];
    else
        pending_txmit = [pending_txmit; [next_hop, txmit_pkt(2:6), 0,
0, txmit_pkt(9)]];
    end
end
if run == 0 && txmit_pkt(1) == 1 && txmit_pkt(2) ~= 1
    txmitting(fin_add, 5) = txmitting(fin_add, 5) - 1;
end
update_route_table(next_hop, fin_add);    %Update the route table of
receiving node
end

*************************************************************************
function broadcast(txmit_pkt)
%This function broadcasts the data packet.

global timer;
global neigh;
```

```
global latency;
global run;
global txmit_times;

neigh_temp = neigh(txmit_pkt(1), 1:2:size(neigh, 2)); %Get node
neighbor addresses
fprintf('%f: Broadcast, node: %d, source: %d, type: %d;\n', timer,
txmit_pkt(1), txmit_pkt(2), txmit_pkt(4));

if txmit_pkt(1) == 1 || ismember(1, neigh_temp)
    fprintf('%f: Packet received, dest: %d;\n', timer, txmit_pkt(3));
    latency(run+1) = latency(run+1) + txmit_times(txmit_pkt(4)); %Add
time to latency
else
    fprintf('%f: Packet NOT received, dest: %d, out of range;\n',
timer, txmit_pkt(3));
end
end

**************************************************************************
function update_route_table(node, ind)
%This function updates that routing tables based on the transmitted
packet.
%'node' is the node address that is being updated.  'ind' is used to
%reference the 'txmitting' array to get info to update the route
tables.

global txmitting;
global route_table;
global route_table_sink;

if node ~= 1    %If not the sink node
    %Update neighbor route
    route_table(1, txmitting(ind, 1), node) = txmitting(ind, 1);
    route_table(2, txmitting(ind, 1), node) = 1;
    if node == txmitting(ind, 3) && txmitting(ind, 4) == 2 &&
txmitting(ind, 1) == txmitting(ind, 2)
    %If rvc node is dest AND type == RREP AND neighbor == source
        route_table(3, txmitting(ind, 1), node) = txmitting(ind, 9);
    end
    if (route_table(1, txmitting(ind, 2), node)==0) || ((256-
txmitting(ind, 5)) < route_table(2, txmitting(ind, 2), node))
    %If no existing route to source OR route is less hops than the
current route
        %Update source route
        route_table(1, txmitting(ind, 2), node) = txmitting(ind, 1);
        route_table(2, txmitting(ind, 2), node) = 256-txmitting(ind,
5);
    end
    if node == txmitting(ind, 3) && txmitting(ind, 4) == 2 &&
txmitting(ind, 9) == 1
    %If rcv node is dest AND type == RREP AND packet sink flag is set
        route_table(3, txmitting(ind, 2), node) = txmitting(ind, 9);
    end
```

```
else        %Sink node
    %Update neighbor route
    route_table_sink(txmitting(ind, 1), txmitting(ind, 1)) = 1;
    if (route_table_sink(txmitting(ind, 1), txmitting(ind, 2))==0) ||
(route_table_sink(txmitting(ind, 1), txmitting(ind, 2)) > 256-
txmitting(ind, 5));
    %If no existing route to source OR route is less hops than the
current route
        route_table_sink(txmitting(ind, 1), txmitting(ind, 2)) = 256-
txmitting(ind, 5);
    end
end
end
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] M. P. Đurišić, Z. Tafa, G. Dimić and V. Milutinović, "A survey of military applications of wireless sensor networks," in *Proc. of Mediterranean Conference on Embedded Computing*, Bar, Montenegro, 2012, pp. 196–199.

[2] *Marine Corps Reference Publication 2–10A.5, Remote Sensor Operations*, United States Marine Corps, Washington, DC, 2016.

[3] S. H. Yang, *Wireless Sensor Networks, Signals and Communication Technology*, London, UK: Springer-Verlag, 2014.

[4] *IEEE Standard for Local and Metropolitan Area Networks—Part 15.4: Low-Rate Wireless Personal Area Networks*, IEEE Standard 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011), 2015.

[5] M. Pradhan, F. Gökgöz, N. Bau and D. Ota, "Approach towards application of commercial off-the-shelf Internet of Things devices in the military domain," in *Proc. of IEEE 3rd World Forum on Internet of Things*, Reston, VA, 2016, pp. 245–250.

[6] *Federal Information Processing Standards Publication 140–2, Security Requirements for Cryptographic Modules*, National Institute of Standards and Technology. Gaithersburg, MD, 2002.

[7] T. Clausen, J. Yi and A. C. de Verdiere, "LOADng: Towards AODV Version 2," in *Proc. of IEEE Vehicular Technology Conference*, Quebec City, QC, 2012, pp. 1–5.

[8] J. Olsson. (2014). 6LoWPAN demystified. [Online]. Available: http://www.ti.com/lit/wp/swry013/swry013.pdf

[9] T. Clausen, A. C. de Verdiere, J. Yi, A. Niktash, Y. Igarashi, H. Satoh, and U. Herberg, "The LLN on-demand ad hoc distance-vector routing protocol–next generation," *The Internet Engineering Task Force*, July 2016, Internet Draft, work in progress, draft-clausen-lln-loadng-15.

[10] M. Er-Rouidi, H. Moudni, H. Mouncif and A. Merbouha, "An energy consumption evaluation of reactive and proactive routing protocols in mobile ad-hoc network," in *Proc. of 13th International Conference on Computer Graphics, Imaging and Visualization*, Beni Mellal, Morocco, 2016, pp. 437–441.

[11] V. P. Patil, "Reactive and proactive routing protocol energy efficiency performance analysis in wireless ad hoc networks," *International Journal of Electronics and Computer Science Engineering*, pp. 2333–2341, 2012.

[12] J. Yi, T. Clausen and Y. Igarashi, “Evaluation of routing protocol for low power and Lossy Networks: LOADng and RPL,” in *Proc. of IEEE Conference on Wireless Sensor*, Kuching, Malaysia, 2013, pp. 19–24.

[13] J. Deng, R. Han, and S. Mishra, “Decorrelating wireless sensor network traffic to inhibit traffic analysis attacks,” in *Elsevier Pervasive and Mobile Computing Journal*, vol. 2, no 2, pp. 159–186, 2006.

[14] V. P. V. Gottumukkala, V. Pandit, H. Li and D. P. Agrawal, “Base-station Location Anonymity and Security Technique (BLAST) for wireless sensor networks,” in *Proc. of IEEE International Conference on Communications*, Ottawa, ON, 2012, pp. 6705–6709.

[15] H. Shen and L. Zhao, “ALERT: An Anonymous Location-based Efficient Routing Protocol in MANETs,” in *IEEE Transactions on Mobile Computing*, June 2013, vol. 12, no. 6, pp. 1079–1093.

[16] A. Callanan and P. Thulasiraman, “Achieving sink node anonymity under energy constraints in tactical wireless sensor networks,” in *Proc. of IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support*, Orlando, FL, 2015, pp.186–192.

[17] S. Alsemairi and M. Younis, “Forming a cluster-mesh topology to boost base-station anonymity in wireless sensor networks,” in *Proc. of IEEE Wireless Communications and Networking Conference*, Doha, Qatar, 2016, pp. 1–6.

[18] L. Sweeney, “k-anonymity: A model for protecting privacy,” in *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.

[19] M. Siekkinen, M. Hiienkari, J. K. Nurminen, and J. Nieminen, “How low energy is Bluetooth low energy? Comparative measures with ZigBee/802.15.4,” in *Proc. of IEEE WCNC Workshop on Internet of Things Enabling Technologies, Embracing Machine-To-Machine Communications and Beyond*, 2012, pp. 232–237.

[20] J. Lee, K. Kapitanova, and S. H. Son, “The price of security in wireless sensor networks,” in *Computer Networks*, vol. 54, no. 17, pp. 2967–2978, Dec. 2010.

[21] J. A. Cordero, J. Yi and T. Clausen, “Optimization of jitter configuration for reactive route discovery in wireless mesh networks,” in *Proc. of 11th International Symposium and Workshops on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, Tsukuba Science City, 2013, pp. 452–459.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California