REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
09-05-2011		

4. TITLE AND SUBTITLE
Sinkhole Avoidance Routing in Wireless Sensor Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Stephenson, Andrew John

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 401 (2011)

12. DISTRIBUTION/AVAILABILITY STATEMENT
This document has been approved for public release; its distribution is UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Wireless sensor networks, or WSNs, are an emerging commercial technology that may have practical applications on the modern battlefield. A wireless sensor network consists of individual sensor nodes that work cooperatively to collect and communicate environmental data. In a surveillance role, a WSN could be deployed across a geographic area of interest, allowing military commanders to monitor enemy troop positions and movements. Wireless sensor networks have enormous potential as an information gathering tool, but they also present many unique challenges to security engineers. An adversary can easily capture and tamper with one of the many unguarded sensor nodes to disrupt or significantly degrade the quality of surveillance that the WSN provides. This project examined potential attacks against WSNs and developed a modified routing protocol that increases the overall data integrity and reliability of wireless sensor networks.

15. SUBJECT TERMS
wireless sensor networks, sinkhole attack, routing protocol

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE		51	19b. TELEPHONE NUMBER *(Include area code)*

Sinkhole Avoidance Routing in Wireless Sensor Networks

MIDN 1/C Andrew J. Stephenson
Trident Research Project
Information Technology Major

Adviser: Dr. Eric Harder

May 9, 2011

The purpose of computing is insight, not numbers.
–Richard Hamming

Abstract

Wireless sensor networks, or WSNs, are an emerging commercial technology that may have practical applications on the modern battlefield. A wireless sensor network consists of individual sensor nodes that work cooperatively to collect and communicate environmental data. In a surveillance role, a WSN could be deployed across a geographic area of interest, allowing military commanders to monitor enemy troop positions and movements. Wireless sensor networks have enormous potential as an information gathering tool, but they also present many unique challenges to security engineers. An adversary can easily capture and tamper with one of the many unguarded sensor nodes to disrupt or significantly degrade the quality of surveillance that the WSN provides. This project examined potential attacks against WSNs and developed a modified routing protocol that increases the overall data integrity and reliability of wireless sensor networks.

Due to battery limitations of individual sensor nodes, many WSN protocols seek to conserve power by simplifying computations and reducing the number of radio transmissions required for communication. These practices allow the WSN to have a longer life expectancy; however, such protocols are easy targets for enemy exploitation. In what is known as a sinkhole attack, a comprised sensor node is maliciously used to alter the wireless mesh of a sensor network for the purpose of disrupting the logical

flow of information across the network. The purpose of this project is to minimize the disruption from such an attack. We have proposed modifications to an existing tree based routing protocol so that it attempts to avoid sinkholes and increase the overall data throughput of the network by sacrificing some of the networks transmission efficiency. The efficacy of the project's proposed sinkhole avoidance strategy is also supported through the use of software based WSN network simulations.

Acknowledgements

I would like to acknowledge the following people for their help in support of my Trident research project. Without them this project would not have been possible:

Dr. Harder – For his guidance, wisdom, and patience as my project advisor.

CDR Vincent, USN – For his encouragement to pursue a WSN research project.

Dr. Brown – For his advice and suggestions throughout the progression of my project.

Dr. Dillner – For her input and encouragement during my project review.

Dr. Wick – For his support of the Trident Scholar program.

The Trident Committee – For allowing me the opportunity to perform a rewarding undergraduate research project on behalf of the United States Naval Academy.

Contents

1 Motivation For Research

In business, a misinformed decision may lead to falling stock prices. In war, a misinformed decision may lead to death. A warrior does not deal in dollars, euros, or yen. He or she deals in the currency of human life. Command decisions are made based on known information. Accurate and timely information can lead a commander to make the correct decision under the severest of time constraints. In a modern war zone, seconds can dictate the difference between success and failure. An emerging technology – wireless sensor networks – may some day provide reliable battlefield information to commanders in real time, reducing risk and saving lives.

2 Introduction

A wireless sensor network, or WSN, refers to a group of small battery powered sensors. An individual sensor, commonly referred to as a node, consists of five major parts: a processor, digital memory, a radio, a sensor suite, and a battery. Additionally, a sensor node can be fitted with actuators that allow it to generate power, move about its environment, or perform some specific task. At its most basic level, a single WSN node is designed to be a sensor. Typical sensor suites are capable of detecting changes in light, sound, temperature, pressure, or acceleration. More sophisticated sensors can be used to detect seismic activity, chemicals, or even radiation [1]. Wireless sensor networks can be used in a variety of peaceful applications, for example: equipment monitoring in industrial facilities, pollution monitoring outside of power plants, or allergen monitoring inside of hospitals. Wireless sensor networks also have the potential for many military applications. Hundreds or even thousands of wireless sensors could be dropped from aircraft and spread over a wide geographic area. These sensors would be able to set up a surveillance network used to monitor enemy troop and equipment movement. In addition, a wireless sensor network could be strategically deployed by special forces near points of interest. Considering the small size of sensor network nodes, a covertly deployed WSN would be an excellent way to secretly monitor a hostile force or installation without need for maintenance or personnel. Such covert networks could be tied into a satellite data link, providing constant and instantaneous information to command centers anywhere on the globe [2].

3 Background

Wireless sensor networks have gained a wide range of attention in the past decade due to their promise to provide reliable, low maintenance, and relatively low cost sensors that can be quickly deployed into a wide variety of applications. Wireless sensor networks can generally be broken up into two categories: structured and ad-hoc. Structured WSNs consist of WSN networks with a planned deployment of each sensor node with regard to its location. Such deployments might be seen in industrial applications where Wireless Sensor Networks are replacing traditional wired sensors (such as safety valve monitoring at an oil refinery). Ad-hoc WSNs do not have planned deployments. The sensor nodes are distributed across an area of interest and are allowed to set up their own routing structure with respect to the base station(s). Ad-hoc WSNs typically consist of many more nodes than a structured WSN, as a higher density of nodes is required to ensure that fault tolerant wireless communication is possible between all nodes in the network and the base station. This project will focus on ad-hoc wireless sensor networks, as ad-hoc WSNs are more suited to military applications. An ad-hoc wireless sensor network is more conducive to surveillance over harsh terrain in remote geographic locations. An ad-hoc WSN gives the user the ability to deploy the network quickly; such networks could be quickly deployed by fast moving ground forces or military aircraft [1].

There are a number of different protocols and hardware sets that can be utilized for Wireless Sensor Networks. One of the first operating systems developed specifically for Wireless Sensor Networks is the 'Tiny' operating system, known as TinyOS. It was developed at the University of California at Berkeley beginning in 1999. TinyOS is a Linux based operating system that is significantly parsed down so that it can be utilized by resource limited WSN nodes. It is written in nesC, which is a variant of the C programming language and is 'event driven,' meaning it does not behave like many other operating systems when dealing with system processes. The entire operating system is only capable of performing one process at a time, and it does not provide a means to prioritize the order in which processes run. TinyOS utilizes many short programs, known as 'event handlers,' to handle large tasks that the node may be asked to perform, to include data routing [18].

4 Related Work

Security in Wireless Sensor Networks is an issue of critical importance to the development of WSN technology as a whole. A significant amount of research has been invested into solving some of the security issues that Wireless Sensor Networks face, to include intrusion detection, host authentication, and data sinkhole mitigation. In [11], authors I. Krontiris *et al.* propose a WSN implementation that would be able to detect sinkhole attacks in Wireless Sensor Networks that utilize the MintRoute protocol (a routing protocol that is similar to the Collection Tree Protocol). Such a system could enable a WSN to quickly detect an attack and trigger its defense mechanisms in order to reduce the volume of data. In [6], authors U. Colesanti and S. Santini have performed an in depth evaluation of the Collection Tree Protocol. Their research explains the inner workings of the CTP protocol in depth and tests the Collection Tree Protocol under several different conditions.

In [10], authors J. Deng *et al.* enumerate a "Intrusion-tolerant routing protocol for Wireless Sensor Networks: INSENS." The INSENS protocol aims to reduce the impact an adversary could have on a WSN by utilizing a number of security features to include light cryptography, positive host identification, and network analysis at the base station. In a similar line of research, T Shu *et al.*[8] proposes a method to defeat sinkhole attacks through the utilization of "randomized dispersive routes." Under this method, network messages are broken up into many 'shares' that are distributed throughout the network before they converge on the base station. Once a set number of shares successfully arrive at the base station, the data from the origin node can be reassembled. Neither of the routing schemes enumerated above utilize a purely routing based approach to mitigating the sinkhole problem in WSNs. Our approach uses only changes to network routing in order to avoid sinkholes.

5 Preliminary Project Work

To gain a better understanding of Wireless Sensor Networks, a portion of project time was spent working with actual WSN hardware. Two types of Wireless Sensor Network nodes were studied. The first was the MICA2 Wireless Measurement System. The MICA2 is commercially available through Crossbow Technology. The MICA2 platform is small, measuring only 2.25 x 1.25 x 1.0 inches with its sensor board and battery pack attached. The unit is powered by two standard AA batteries and has a battery life of up to one year

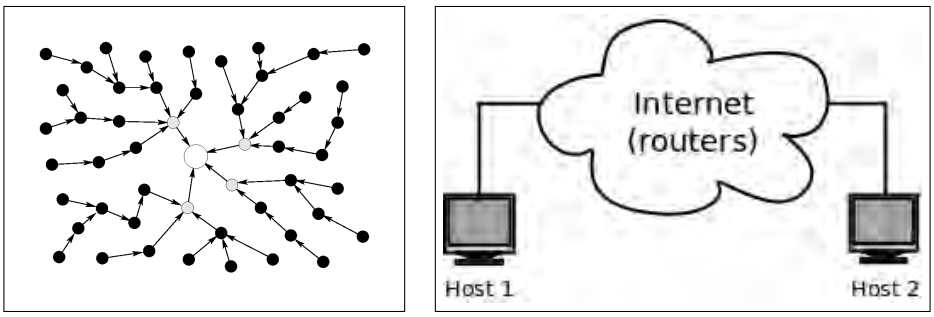
under continuous operation, given that it is calibrated to use a power saving 'sleep' mode that reduces the number of transmissions and computations that the node performs [17]. The MICA2 has only 128 KB of program memory and the processor only draws 8 mA of current while active and only 15 μA in sleep mode. The transmitter draws 27 mA of current when transmitting and 10 mA while in receive mode. It is important to note the high cost of radio transmission in terms of battery drain. The second WSN hardware platform used in project work was the IRIS Wireless Measurement System. The IRIS WSN is essentially an improvement on the MICA2 node, and is very similar in terms of size and capabilities [16]. Both WSN node platforms run the TinyOS operating system and are capable of implementing the Collection Tree Protocol (CTP). TinyOS applications were built, compiled, installed, and run on both the MICA2 and IRIS hardware platforms. Due to the limited memory and minimalist premises behind Wireless Sensor Networks, the installation of new programs on WSN nodes becomes a somewhat difficult task. In order to implement a new application on the MICA2 or IRIS platform, the entire operating system must be recompiled and then reloaded individually on to each sensor node. This process is very time consuming and would generally only be performed as part of a major operating system upgrade.

6 Security in Wireless Sensor Networks

Despite the immense potential of sensor networks, the low cost and small size of a single sensor node severely restricts an individual node's computing power and memory. On top of this, a single unit's lifetime is dictated by its least sophisticated technology – the battery. Any activity on the part of a node, in terms of computation and transmission, directly affects the lifetime of the entire unit. In addition, while the computing power of a comparable sized unit may increase following Moore's law[1], it is likely that more inexpensive productions of the same unit will be chosen over more capable components [3]. Thus, security in sensor networks must be designed and implemented with energy and computing efficiency in mind.

In addition to battery and computational limitations, wireless sensor networks face another unique problem – node capture. In typical network security schemes, it is assumed that the individual hosts (nodes) are safe from physical capture or tampering. In WSNs, nodes may need to be placed in hostile and or easily accessible locations, meaning that the system must

[1]Moore's law is the observation that the number of transistors that can be placed on a circuit doubles roughly every two years



(a) WSN many-to-one tree structured routing

(b) Internet host-to-host routing

Figure 1: Comparision of Internet routing and WSN routing

be designed so that the capture or destruction of a node will not disrupt the overall data transfer capability of the network [4]. Additionally, it can be assumed than an adversary will be able to extract critical data from a captured node. She may then be able to use this data to deny service to the network, or otherwise exploit the network's security system [5]. Threats on WSNs present a unique challenge in that many traditional computer network security solutions do not directly apply to a WSN.

On the Internet, hosts generally communicate in a one to one fashion, that is, one host uses the network to communicate with only one other specifically addressed host. Wireless sensor networks, on the other hand, do not communicate in a one to one fashion. Communication patterns in a WSN can be broken down into three basic types: many to one, one to many, and many to many [3]. The pattern of many to one communication stems from the layout of a typical wireless sensor network: a single 'base station' is responsible for collecting data from many different nodes. The base station is interested in aggregating data from a network composed of n nodes, where $n \gg 1$. The reverse is also true, resulting in a one to many communication pattern. This type of communication occurs when the base station wishes to send out configuration information to every node within the network, resulting in a multicast message that is generated at the base station of the network and is desseminated to every node in the network. Lastly, nodes within the network may want to exchange information with other nodes within the network (communication that excludes the base station). Such communication may occur when nodes are participating in the exchange of local routing information, aggregating data within a neighborhood of nodes, or 'voting' in an effort to detect an illegitimate node in the network [1]. For the purpose of this research, we will focus on WSN communication that is many to one.

The host to host nature of Internet communication generally follows a simple one to one communication model, where a host is capable of sending out messages with a foreign address and receiving data that has been addressed to it. The host is able to hand off a packet to a router that has a wider knowledge of the network topology. The router then sends the packet to a series of other routers that guide the message datagram to its eventual destination. Under this model, a host is only expected to know the address of the host that the message will be sent to. Conversely, in the role of a receiver, the host only has to listen for packets that are addressed to it. Routing across the network is not handled by hosts; it is handled instead by routers that are built and configured specifically for that task. In a wireless sensor network, routing is handled differently. Every node within a wireless sensor network can be expected to act as a receiving node, a transmitting node, and a routing node. Depending on the layout of the sensor network at deployment time, the topology of the network can leave any sensor in the network in one or a combination of all three of the roles mentioned above. This difference between Internet routing and WSN routing further complicates the application of Internet based networking protocols to WSN topologies.

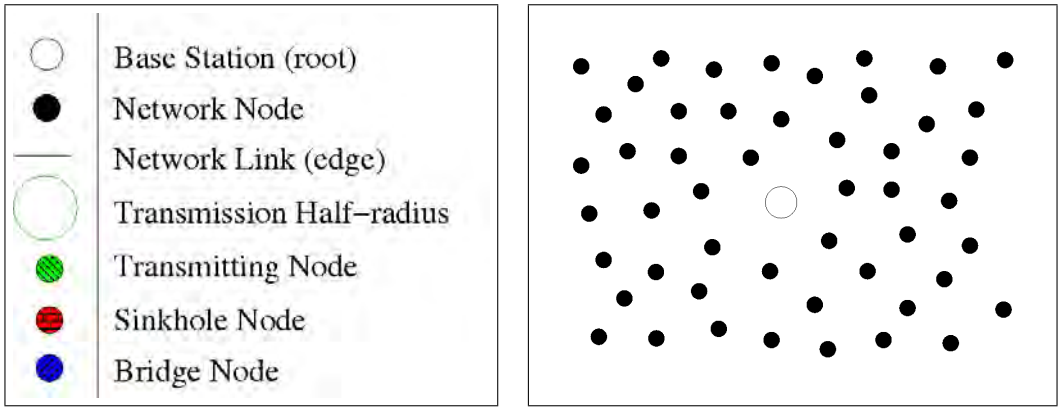
A majority of secure traffic over the Internet utilizes the Transmission Control Protocol. The Transmission Control Protocol utilizes two features that are generally not implemented in WSN protocols in order to save battery life (too many extra transmissions). The first feature is that of 'three way handshaking.' Before data is transmitted between two hosts on the Internet, TCP ensures that there is a valid and reliable connection between the two hosts. The sender initiates the 'handshake' by first querying the receiver. Upon receipt of the query, the receiving host will then send back an acknowledgement (known as an ACK), which lets the originating host know that a data exchange session has been set up between the sender and the receiver. Finally, the sending host sends its data to the receiving host (which also implicitly serves as an acknowledgement – the third part of the handshake). Three way handshaking allows a data transmission session to be set up between two hosts before actual information is exchanged. The second feature of TCP is an extension of the first – the use of acknowledgments. The Transmission Control Protocol performs accounting on the information that is transfered between hosts. Each datagram message that is exchanged between the two hosts is acknowledged in an ACK process that is similar to the three way handshake described above. If transmitted data is incomplete or lost, TCP will retransmit the data to ensure its accurate and complete arrival. Unfortunately, most Wireless Sensor Network protocols do not implement acknowledgements due to the extra transmissions that are required to ensure data delivery [15].

Internet	Wireless Sensor Networks
One to One communication	Many to One Communication
Assumption of host's physical security	No assumption of node security
Strong cryptography	Weak or no cryptography
Transmission Acknowledgements	Little or no acknowledgements

Table 1: Comparison of differences between Internet and WSN communication protocols

7 Wireless Sensor Network Topology

In order to understand the establishment of routing protocols in wireless sensor networks, we must first understand the properties of the network graph. We define the network graph to be the set of all WSN nodes, to include the base station (the base station is also commonly referred to as the 'root' or 'sink' of the network). We assume that the uninitialized network graph is a connected, undirected graph, meaning that every node in the network is adjacent to at least one other node in the network (they can communicate wirelessly). Please reference Figure 2 and Figure 3 for a further visual explanation.

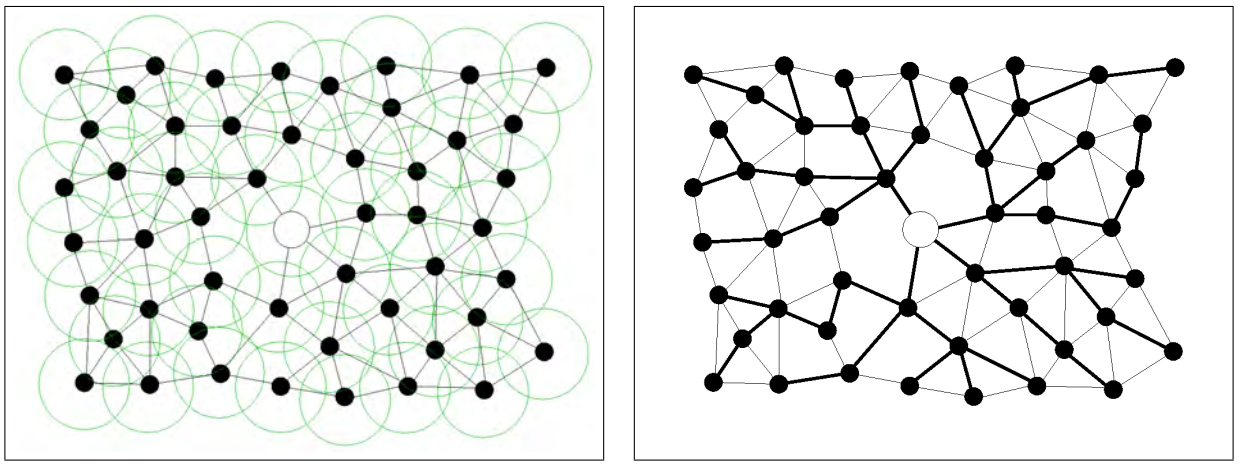


(a) Key for network diagrams

(b) An uninitialized WSN network with nodes and a base station

Figure 2

Adjacency in a wireless sensor network is dictated by the radius of communication R_c and is best described as a *unit disk graph*: each node lies at the center of a unit disk (with unit radius r), and an edge is defined whenever two disks overlap. For sensor networks, we then have $r = R_c/2$ [9]. This requirement on the edges imposes a sense of distance in the graph. It is important to note that arbitrary edge sets are not possible; in particular, edges that



(a) Unit disk graph with radii 'r.' Overlapping 'r' indicates a possible network edge

(b) All possible connections of the network graph, with higher quality network connections represented in bold

Figure 3: An arbitrary network graph with possible network connection edges given the 'r' of each node

exceed $2r$ in length cannot occur, so widely distributed edges cannot be connected directly, and hence must be connected by a path through the graph as shown in Figure 3.

The tree structured routing utilized by wireless sensor networks is established through the utilization of some sort of path metric. In general, the selection of edges is based on the 'best path' between two nodes, given the path metric. The Collection Tree Protocol (CTP) seeks to find the best path to the root node by transmitting as few times as possible (it seeks to transmit over the most reliable path to the base station). This path metric in CTP is known as the ETX value which stands for 'Expected Transmissions'). The ETX value represents the predicted number of node transmissions that will need to occur for a message to reach the base station. The ETX value is established during network setup and is based on the number of transmissions that are necessary to successfully transmit a routing query message between two nodes. An optimal ETX value between any set of nodes is '1,' as only one transmission is required to successfully transmit the data. The base station is a special case, as it does not need to transmit in order to communicate with itself, thus, the base station has an ETX value of '0.' Every other node in the network will have an ETX value that is greater than '1,' and the ETX value of any node in the network will the be the sum of the ETX value of its parent and the ETX value of the link between the aforementioned node and its parent. A valid data transmission over a routing tree using CTP is shown in Figure 4

We model a sensor network as a unit disk graph $G(V, E)$, composed of the set $V, |V| = N$ of vertices (sensors) and the set E of edges determined by radio reception.

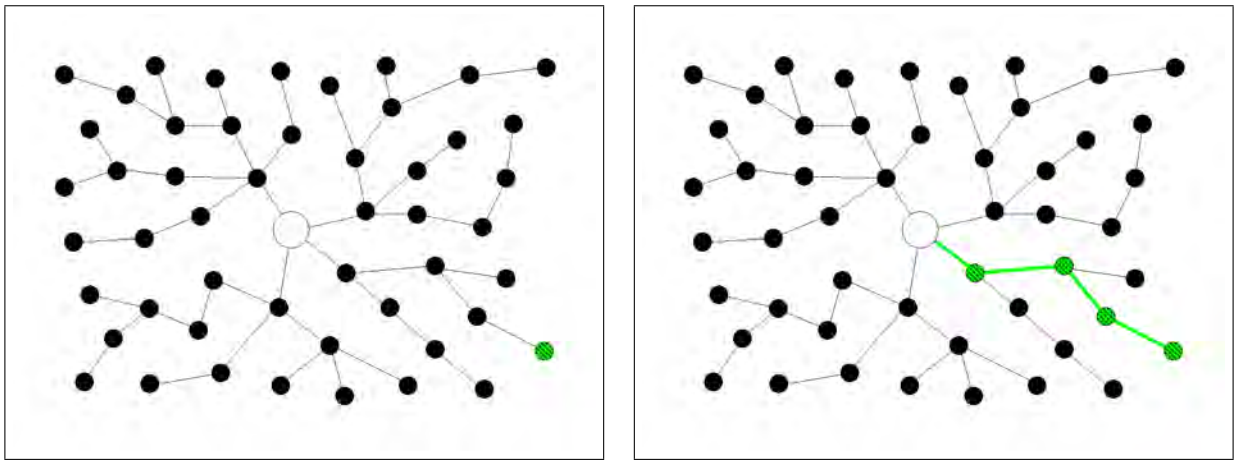
The tree $T(V, E') \subseteq G(V, E)$ is defined by the TinyOS Collection Tree Protocol, rooted at the base station. We will usually denote this tree simply as T_{CTP}.

We make the following assumptions about the sensor network:

- Each sensor node is identical in terms of initial battery life, transmitter and receiver capacity.
- Only one base station is placed.
- Once placed, the sensors are fixed.
- The sensors are uniformly distributed; for any sensor v and neighborhood

$$N_v \triangleq \{v_i \mid d_r(v, v_i) < r_{\text{radio}}\}$$

  we have, on average, $|N_v| = k$ for any $v \in G$, with sufficient small variance to allow for simple analysis.



(a) The green node is attempting to transmit data to the base station

(b) The data follows the path directed by the network tree and successfully reaches the base station

Figure 4: Normal data routing in a WSN

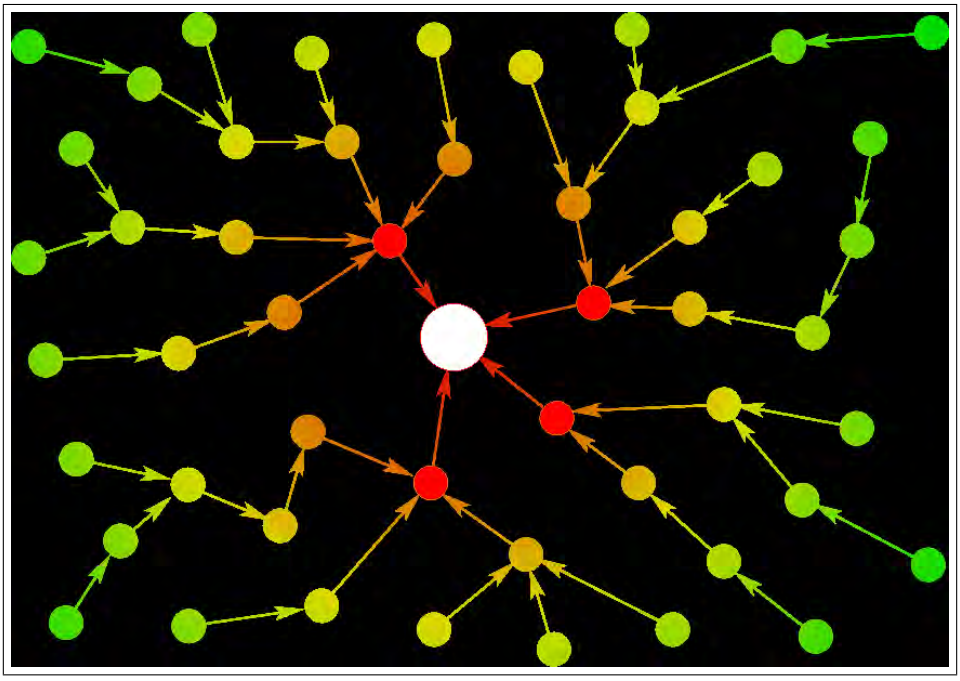


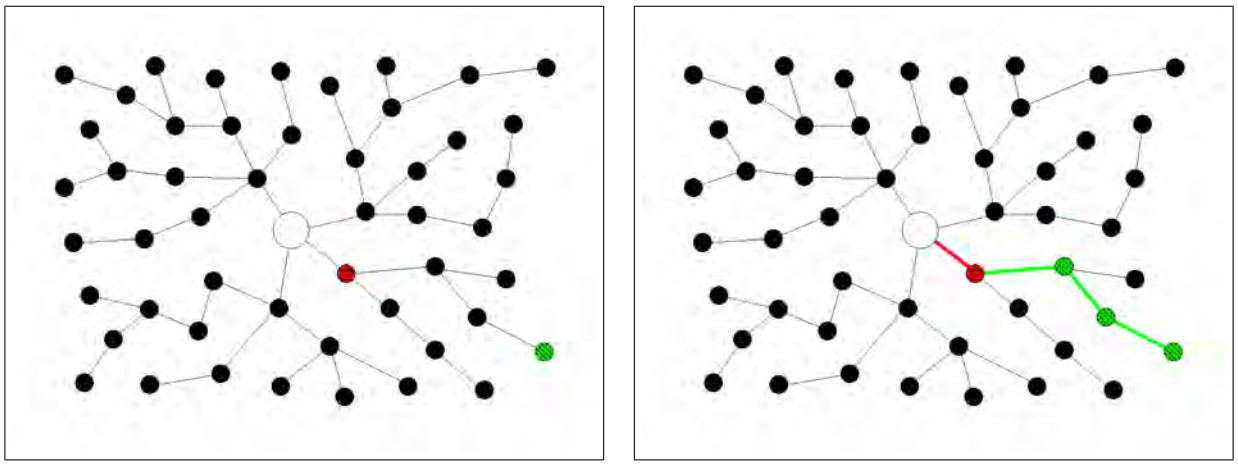
Figure 5: A depiction of the risk gradient, where nodes in green represent a low risk of data loss,and nodes in red represent a high risk of data loss

8 Threat Model

The essential function of a sensor network is to report data. The data flows across the deployment region, routed via the minimum weight spanning tree (MST). Each sensor routes data along the minimum weight path to a base station that acts as a root node (we will use 'node' and 'sensor' interchangeably when there is no confusion). For the purpose of our research, we assume that the base station is secure and has not been tampered with by an adversary. This is a reasonable assumption, as the compromise of the base station would allow the adversary to control the entire WSN and serves to trivialize the avoidance of a routing sinkhole, which is our project focus.

A sensor node that has been compromised by an attacker can act as a 'sinkhole' and manipulate all network data that is forwarded to it. Our research focuses on an adversary that controls a sinkhole in the network and chooses to drop all network data that is forwarded to it. In particular, the adversary influences the routing so that it maximizes the amount of traffic flowing to it. This is the 'sinkhole' attack and is the focus of this work. We assume that the adversary is able to completely compromise one sensor. The adversary will then

have the ability to influence how the minimum spanning tree is established. A compromised sensor would be able to advertise a favorable metric so as to be included in the tree as a routing node instead of a leaf node. We assume that the adversary may enhance the sensor to support this metric. In this way, the compromised sensor will receive traffic from downstream nodes for examination (whereas a leaf node does not have downstream nodes). The adversary may achieve her goal through a combination of positioning herself close to the base, or influencing how the minimum spanning tree is established through fraudulent route costs, wherein a compromised sensor will advertise a favorable metric so as to be included in the tree as a routing node instead of a leaf node. We assume that the adversary only drops traffic. This means that the adversary will try to influence as much traffic as possible by positioning herself close to the base, resulting in a "risk gradient" where nodes that are far from the base are less likely to be compromised, as shown in Figure 5.



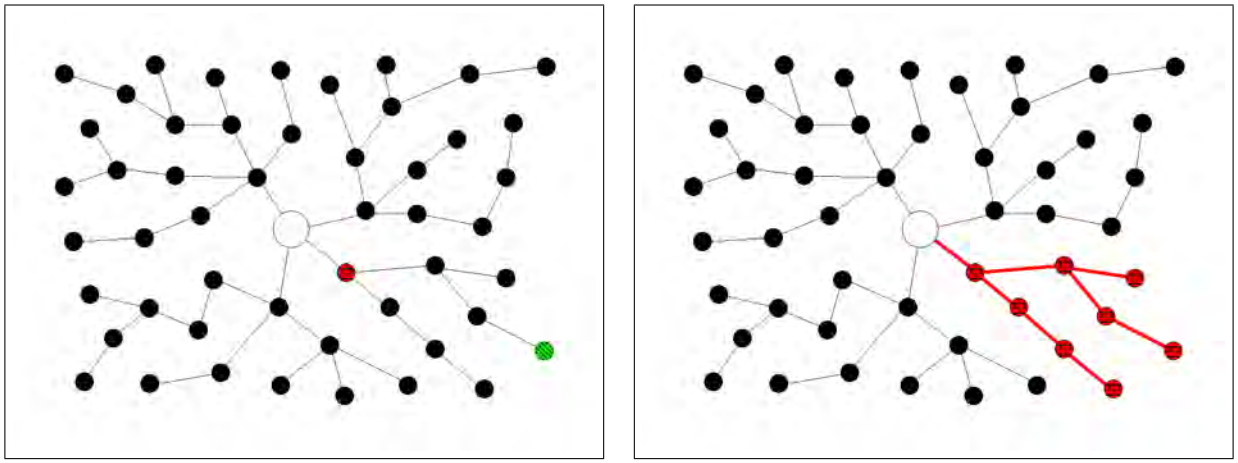
(a) The green node is attempting to transmit data to the base station in the presence of a sinkhole

(b) The data follows the path directed by the network tree and is intercepted by the sinkhole before it reaches the base station

Figure 6: WSN Routing Example

The position of a sinkhole node in the network will affect the impact that the sinkhole is able to have on the network as a whole. In a network that utilizes tree structured routing, every node in the network must rely on its parent to forward data closer to the base station. In a sense, tree structured routing creates a communication chain that can be broken by removing a single link. The closer the broken link is to the base station, the more links that will be severed from the base station. This idea ties into the risk gradient depicted in Figure 5. If a sinkhole node happens to be positioned at the root of a subtree, then the

sinkhole will be able to disconnect that entire subtree from the base station. Depending on the distribution of nodes in the sensor network, this could mean that a large geographic area of the sensor network will be unable to report its data.



(a) A WSN with a sinkhole, given the green transmitting node and the red sinkhole

(b) The sinkhole is at the root of the subtree, which effectively cuts off the rest of that subtree from the network

Figure 7: The effect of a sinkhole in a WSN

9 Description of Protocols

9.1 Generic Protocol Description

In order to better describe the complex distributed protocols enumerated in this project, we will first define an abstract distributed protocol. Our abstract scenario will describe barking dogs in a neighborhood. Our distributed protocol will give instructions for a dog to execute once an event occurs. Each dog in the neighborhood executes the same protocol.

When (I hear another dog bark)
Then
1 I bark once

This abstract protocol can be used to imagine how a WSN initializes the routing infrastructure of the network. A dog that is in the center of his neighborhood would initiate a bark. This bark would then move outward towards the edges of the neighborhood until every dog

in the neighborhood is barking. The CTP protocol initiates itself in a similar manner, except that beacons are used instead of barks, and that many of the events are triggered by timers in order to reduce the number of transmissions required by each node.

9.2 The Collection Tree Protocol

The Collection Tree Protocol (CTP) is the standard protocol that our project aims to improve upon. CTP attempts to transmit data over the lowest cost path. In simple terms, the 'cost' of a spanning tree is the sum over all edges of the transmission quality. This quality is captured by the Expected Transmissions value (ETX), and is used by CTP for the construction of the minimal spanning tree. From the description: "CTP is a tree-based collection protocol. Some number of nodes in a network advertise themselves as tree roots. Nodes form a set of routing trees to these roots. CTP is address-free in that a node does not send a packet to a particular root; instead, it implicitly chooses a root by choosing a next hop. Nodes generate routes to roots using a routing gradient." [7] We seek to expand CTP so that it supports bridge discovery. The challenge is to implement our changes to the source code in such a way that does not break the protocol or significantly add to the complexity of the protocol in terms of battery cost (extra transmissions and computations). We are most concerned with taking advantage of the distributed nature of the protocol so as to keep the routing complexity on an individual node to a minimum.

Our modification of CTP will occur inside the routing engine. We will focus on four functions within the code that have important roles in initializing routing within CTP. We refer to these functions as *event handlers*, as they are executed when a specific routing related event occurs. These event handlers allow CTP to build and update the routing tables[2] necessary for the protocol to successfully implement tree structured routing. The four event handlers we will modify are: the Send Beacon Handler, the Beacon Message Received Handler, the Table Update Handler, and the Update Route Handler. We also describe the Beacon Timer Handler, as it directly influences two of the other event handlers that we modify within the routing engine.

[2]CTP stores the information of neighboring nodes in a data structure know as a *Routing Table*. By default, this data structure is limited to 10 entries.

Beacon Timer Handler

When (The send beacon timer expires)
Then
1 Reset send beacon timer
2 Send event to *Send Beacon Handler*
3 Send event to *Update Route Handler*

Source code[3]:

```
void CtpRoutingEngine::event_BeaconTimer_fired(){
    if(radioOn && running){
        if(!tHasPassed){
            post_updateRouteTask();
            post_sendBeaconTask();
            trace()<<"Beacon_timer_fired.";
            remainingInterval();
        }
        else{
            decayInterval();
        }
    }
}
```

Send Beacon Handler

When (A send beacon event is received)
Then
1 Send a beacon with my information: ID, Parent ID, ETX

Source code:

```
void CtpRoutingEngine::sendBeaconTask(){
        error_t eval;
        if(sending){
                return;
        }

        beaconMsg->setOptions(0) ;

        if(cfe->command_CtpCongestion_isCongested()){
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_ECN);
        }

        beaconMsg->setParent(routeInfo.parent);
        if(state_is_root){
                beaconMsg->setEtx(routeInfo.etx);
        }
```

[3]A complete listing of all source code is printed in the appendix

```
        else if(routeInfo.parent == INVALID_ADDR){
                beaconMsg->setEtx(routeInfo.etx);
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_PULL);
        } else{
                beaconMsg->setEtx(routeInfo.etx + le->command_LinkEstimator_getLinkQuality(
                    routeInfo.parent));
        }

        trace()<<"sendBeaconTask - parent: "<<(int)beaconMsg->getParent()
    <<" etx: "<<(int)beaconMsg->getEtx();

        beaconMsg->getRoutingInteractionControl().lastHop = self ; // ok
        eval = le->command_Send_send(AM_BROADCAST_ADDR, beaconMsg->dup());

        if(eval == SUCCESS){
                //statistics
                collectOutput("Ctp_Beacons","Tx") ;
                sending = true;
        } else if(eval == EOFF){
                radioOn = false;
                trace()<<"sendBeaconTask - running: "<<running<<" radioOn: "<<radioOn;
        }
}
```

Beacon Message Received Handler

When (A beacon message is received)

Then

1. Read neighbor beacon information: ID, Parent ID, ETX
2. Check parent ID
3. If parent ID matches my own ID, then stop
4. Calculate message ETX using the link estimator
5. Calculate new ETX by adding neighbor ETX to message ETX
6. Send event to *Table Update Handler*

Source code:

```
void CtpRoutingEngine::event_BeaconReceive_receive(cPacket* msg){
    Enter_Method("event_BeaconReceive_receive");
    am_addr_t from;
    bool congested;

    //statistics
    collectOutput("Ctp_Beacons","Rx");

    from = command_AMPacket_source(msg);
    CtpBeacon* rcvBeacon = check_and_cast<CtpBeacon*>(msg);
    congested = command_CtpRoutingPacket_getOption(msg,CTP_OPT_ECN);
```

```
    trace()<<"BeaconReceive.receive - from "<<(int)from<<" [parent: "
    <<(int)rcvBeacon->getParent()<<" etx: "<<(int)rcvBeacon->getEtx()
    <<"]";

    //update neighbor table
    if(rcvBeacon->getParent() != INVALID_ADDR){
        // If this node is a root, request a forced insert in the link
        // estimator table and pin the node.
        if(rcvBeacon->getEtx() == 0){
            trace()<<"from a root, inserting if not in table "
            <<"my_ll_addr: "<<my_ll_addr;
            le->command_LinkEstimator_insertNeighbor(from);
            le->command_LinkEstimator_pinNeighbor(from);
        }
        routingTableUpdateEntry(
                from,rcvBeacon->getParent(),
                rcvBeacon->getEtx();

        command_CtpInfo_setNeighborCongested(from,congested) ;
    }

    if(command_CtpRoutingPacket_getOption(msg, CTP_OPT_PULL))
        resetInterval();

    delete msg ;
}
```

Table Update Handler

When	(A table update event is received)
Then	**1** Read new neighbor information: ID, Parent ID, ETX
	2 If routing table is full, then stop
	3 Add new neighbor information to the routing table

Source code:

```
error_t CtpRoutingEngine::routingTableUpdateEntry(
    am_addr_t from,
    am_addr_t parent,
    uint16_t etx{

    uint8_t idx;
    uint16_t  linkEtx;
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(from));

    idx = routingTableFind(from);
    if(idx == routingTableSize){
        trace()<<"routingTableUpdateEntry - FAIL, table full";
        return FAIL;
    }
```

```
    else if(idx == routingTableActive){
        if(passLinkEtxThreshold(linkEtx)){
            routingTable[idx].neighbor = from;
            routingTable[idx].info.parent = parent;
            routingTable[idx].info.etx = etx;
            routingTable[idx].info.haveHeard = 1;
            routingTable[idx].info.congested = false;
            routingTableActive++;
            trace()<<"routingTableUpdateEntry - OK, new entry";
        }
        else
            trace()<<"routingTableUpdateEntry - Fail, link quality("
            <<(int)linkEtx<<") below threshold";

    }
    else{
        //found, just update
        routingTable[idx].neighbor = from;
        routingTable[idx].info.parent = parent;
        routingTable[idx].info.etx = etx;
        routingTable[idx].info.haveHeard = 1;
        trace()<<"routingTableUpdateEntry - OK, updated entry";
    }
    return SUCCESS;
}
```

Update Route Handler:

> **When** (An update route event is received)
>
> **Then**
>
> **1** Read all neighbor information from the table: ID, Parent ID, ETX
>
> **2** Pick the neighbor with the best ETX
>
> **3** Set that neighbor as the new parent

Source code:

```
void CtpRoutingEngine::updateRouteTask(){

    uint8_t i;
    routing_table_entry* entry;
    routing_table_entry* best;
    uint16_t minEtx;
    uint16_t currentEtx;
    uint16_t linkEtx, pathEtx;

    if (state_is_root)
        return;

    best = NULL;
    /* Minimum etx found among neighbors, initially infinity */
    minEtx = MAX_METRIC;
```

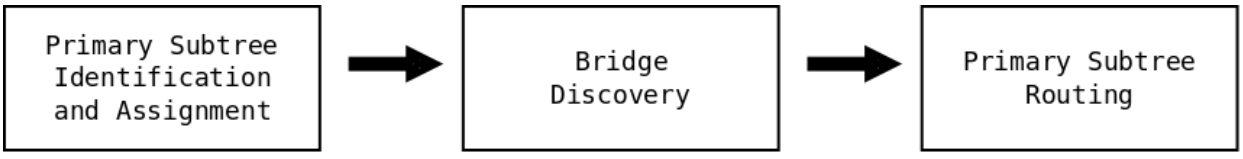
```
/* Metric through current parent, initially infinity */
currentEtx = MAX_METRIC;

trace()<<"updateRouteTask";

/* Find best path in table, other than our current */
for(i = 0; i < routingTableActive; i++){
    entry = &routingTable[i];

    // Avoid bad entries and 1-hop loops
    if(entry->info.parent == INVALID_ADDR ||
       entry->info.parent == my_ll_addr){

        trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "
        <<(int)entry->neighbor<<" parent: "<<entry->info.parent
        <<" etx: NO_ROUTE]";

        continue;
    }

    // Compute this neighbor's path metric
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(entry->neighbor));

    trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor
    <<" parent: "<<entry->info.parent<<" etx: "<<(int)linkEtx
    <<"]";

    pathEtx = linkEtx + entry->info.etx;
    /* Operations specific to the current parent */
    if(entry->neighbor == routeInfo.parent){
        trace()<<"already parent";

        currentEtx = pathEtx;

        /* update routeInfo with parent's current info */
        routeInfo.etx = entry->info.etx;
        routeInfo.congested = entry->info.congested;
        continue;
    }

    /* Ignore links that are congested */
    if(entry->info.congested)
        continue;
    /* Ignore links that are bad */
    if(!passLinkEtxThreshold(linkEtx)){
        trace()<<"did not pass threshold.";
        continue;
    }

    if(pathEtx < minEtx){
        minEtx = pathEtx;
        best = entry;
```

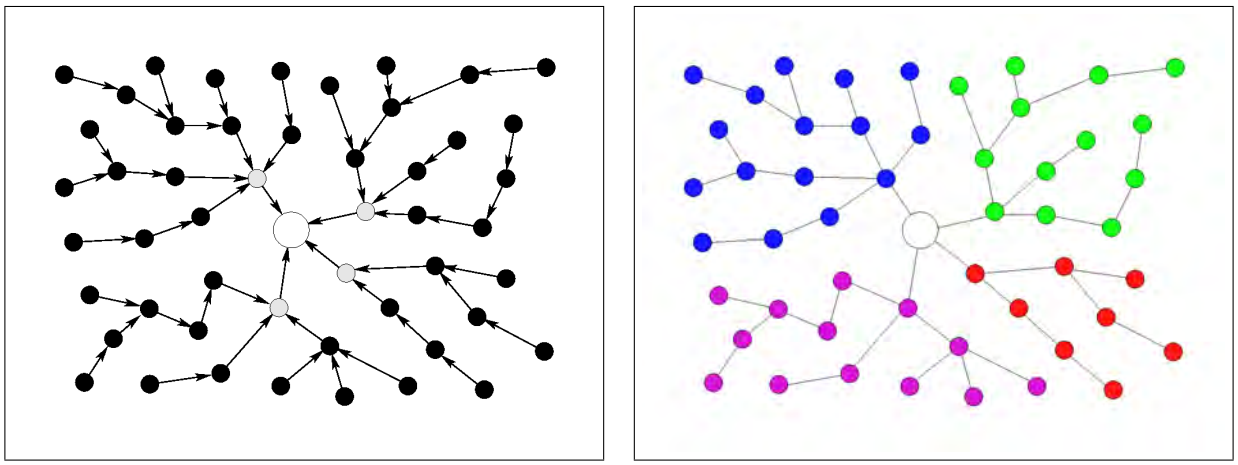
```
        }
    }
    if(minEtx != MAX_METRIC){
        if(currentEtx == MAX_METRIC ||
           (routeInfo.congested && (minEtx < (routeInfo.etx + 10))) ||
           minEtx + PARENT_SWITCH_THRESHOLD < currentEtx){

            parentChanges++;

            trace()<<"Changed parent. from "<<(int)routeInfo.parent<<" to "
            <<(int)best->neighbor;

            le->command_LinkEstimator_unpinNeighbor(routeInfo.parent) ;
            le->command_LinkEstimator_pinNeighbor(best->neighbor) ;
            le->command_LinkEstimator_clearDLQ(best->neighbor) ;

            routeInfo.parent = best->neighbor;
            routeInfo.etx = best->info.etx;
            routeInfo.congested = best->info.congested;
        }
    }
    else if(!justEvicted &&
            currentEtx == MAX_METRIC &&
            minEtx != MAX_METRIC)
        signal_Routing_routeFound();

    justEvicted = false;
}
```

9.3 Protocol Modification

The Collection Tree Protocol (CTP), which is described in detail above, is used as the basis for our routing protocol. Our work has added the awareness of principle subtrees to CTP. Our changes to CTP are described below.

Our protocol:





(a) Directed tree utilized by the network protocol for data forwarding

(b) The different subtrees of the entire network tree are highlight in different colors

Figure 8: Network tree topology that illustrates the multiple principle subtrees of the entire network

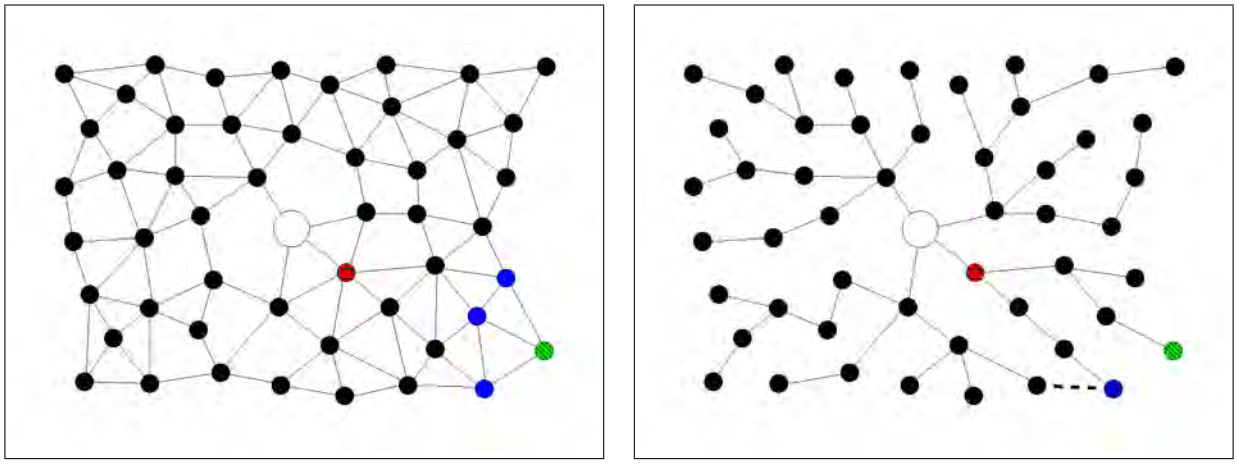
Principle Subtree Identification and Assignment

- After CTP network setup is complete, determine which nodes have the base station as a parent
- Identify these nodes as the roots of the principle subtrees
- Each of these root nodes receives an index value assigned by the base station
- The subtree roots then disseminate their principle subtree index (PST ID) to the members of their principle subtree

Bridge Discovery - reference Figure 9

- Each node in the network queries its list of neighbors and obtains their PST ID
- If a node has a neighbor with a PST ID value different from its own, then the neighbor node is a bridge
- Each bridge in a principle subtree T_j, with index j, that contains the sinkhole 're-runs' the CTP tree establishment protocol by advertising a very low-cost route to the base. This advertisement is limited to only those sensor nodes in T_j, and results in multiple CTP tree roots with routes to the base. Sensor node routing tables are modified to keep track of whether next hops (to parents) are routed towards bridges, giving nodes the

chance to pick a next hop at random from among the available bridges. This approach is feasible since we have implemented the ability for sensor nodes to determine their subtree index as part of the bridge discovery simulations.



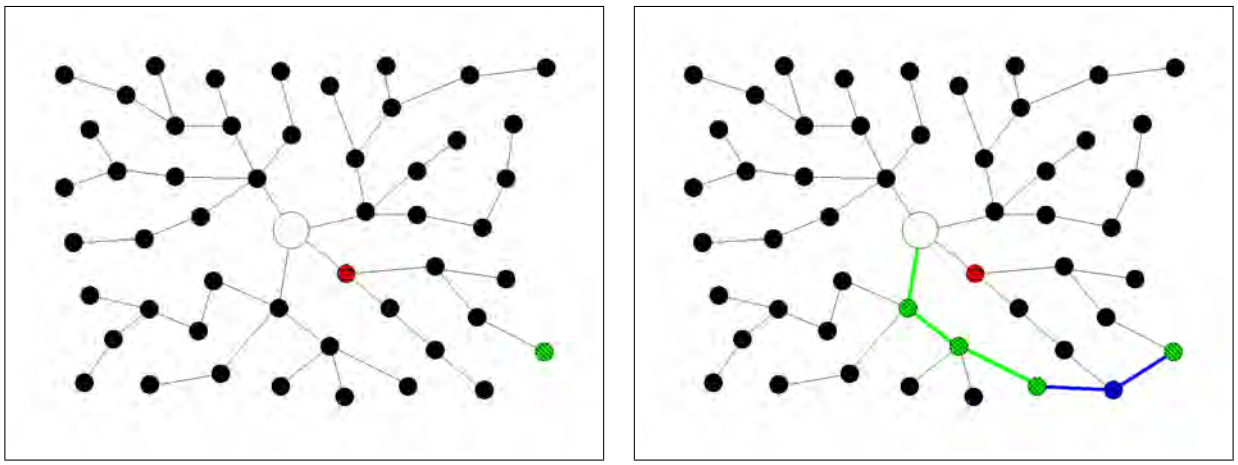
(a) The transmitting node has multiple neighbors that it can potentially forward data to, represented by the blue nodes

(b) One neighbor of the transmitting node is a bridge to another subtree, which is indicated by the blue node and its dashed line to the alternate subtree

Figure 9: Bridge node identification

Principle Subtree Routing - reference Figure 10

- If a node suspects that there is a sinkhole in its subtree, it chooses a bridge node and routes in that direction (adding the bridge node ID to the packet to ensure proper downstream routing)
- A bridge node that receives data from a different subtree will forward the data to the base station via the Collection Tree Protocol



(a) The transmitting node is attempting to transmit data to the base station in the presence of a sinkhole

(b) The transmitting node routes data to a bridge node, who then routes the data to a different subtree. Once in a different subtree, the data is routed to the base station via the normal directed network tree

Figure 10: Principle Subtree Routing using a bridge between principle subtrees

Our changes to the CTP protocol occur within four event handlers that are a part of the CTP routing engine: the Send Beacon Handler, the Beacon Message Received Handler, the Table Update Handler, and the Update Route Handler. We modify these functions so as to implement bridge descovery through the addition of a priniciple subtree index (PST_ID). Inclusion of the PST_ID in the protocol will only add two bytes[4] to each beacon message.

Beacon Timer Handler[5]

When	(The send beacon timer expires)
Then	**1** Reset send beacon timer
	2 Send event to *Send Beacon Handler*
	3 Send event to *Update Route Handler*

[4]One byte is equal to 8 bits.

[5]We do not modify this event handler; however, it is important to understand it's function as it is responsible for triggering the *Send Beacon Handler* and the *Update Route Handler*.

Send Beacon Handler[6]

> **When** (A send beacon event is received)
> **Then**
> **1** Send a beacon with my information: ID, Parent ID, ETX, *PST_ID*

Source code:

```
void CtpRoutingEngine::sendBeaconTask() {
        error_t eval;
        if(sending){
                return;
        }

        beaconMsg->setOptions(0);

        if(cfe->command_CtpCongestion_isCongested()){
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_ECN);
        }

        beaconMsg->setParent(routeInfo.parent);
        if(state_is_root){
                beaconMsg->setEtx(routeInfo.etx);
        }
        else if(routeInfo.parent == INVALID_ADDR){
                beaconMsg->setEtx(routeInfo.etx);
                beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_PULL);
        } else{
                beaconMsg->setEtx(routeInfo.etx + le->command_LinkEstimator_getLinkQuality(
                    routeInfo.parent));
        }

        beaconMsg->setPstId(my_pstId);

        trace()<<"sendBeaconTask - parent: "<<(int)beaconMsg->getParent()
    <<" etx: "<<(int)beaconMsg->getEtx();

        beaconMsg->getRoutingInteractionControl().lastHop = self ; // ok
        eval = le->command_Send_send(AM_BROADCAST_ADDR, beaconMsg->dup());

        if(eval == SUCCESS) {
                //statistics
                collectOutput("Ctp_Beacons","Tx") ;
                sending = true;
        } else if(eval == EOFF) {
                radioOn = false;
                trace()<<"sendBeaconTask - running: "<<running<<" radioOn: "<<radioOn;
        }
}
```

[6]Changes to the original CTP protocol are printed in *italics*

Beacon Message Received Handler

When (A beacon message is received)

Then

1 Read neighbor beacon information: ID, Parent ID, ETX,*PST_ID*

2 Check parent ID

3 If parent ID matches my own ID, then stop

4 *If parent ID is root, set my PST_ID to my ID*

5 *If parent is not root, set my PST_ID to my parent's PST_ID*

6 Calculate message ETX using the link estimator

7 Calculate new ETX by adding neighbor ETX to message ETX

8 Send event to *Table Update Handler*

Source code:

```
void CtpRoutingEngine::event_BeaconReceive_receive(cPacket* msg){
    Enter_Method("event_BeaconReceive_receive");
    am_addr_t from;
    bool congested;

    //statistics
    collectOutput("Ctp Beacons","Rx");

    from = command_AMPacket_source(msg);
    CtpBeacon* rcvBeacon = check_and_cast<CtpBeacon*>(msg);
    congested = command_CtpRoutingPacket_getOption(msg,CTP_OPT_ECN);

    trace()<<"BeaconReceive.receive - from "<<(int)from<<" [parent: "
    <<(int)rcvBeacon->getParent()<<" etx: "<<(int)rcvBeacon->getEtx()
    <<" pstId: "<<(int)rcvBeacon->getPstId()<<"]";

    //update neighbor table
    if(rcvBeacon->getParent() != INVALID_ADDR){
        // If this node is a root, request a forced insert in the link
        // estimator table and pin the node.
        if(rcvBeacon->getEtx() == 0){
            trace()<<"from a root, inserting if not in table "
            <<"my_ll_addr: "<<my_ll_addr;
            le->command_LinkEstimator_insertNeighbor(from);
            le->command_LinkEstimator_pinNeighbor(from);
            // since i hear root, i'm the root of a principle subtree
            my_pstId = my_ll_addr;
        }

        if(rcvBeacon->getEtx() == 0)
            routingTableUpdateEntry(
                from,rcvBeacon->getParent(),
                rcvBeacon->getEtx(),
```

```
                my_pstId);
        else
            routingTableUpdateEntry(
                from,rcvBeacon->getParent(),
                rcvBeacon->getEtx(),
                rcvBeacon->getPstId());

        command_CtpInfo_setNeighborCongested(from,congested) ;
    }

    if(command_CtpRoutingPacket_getOption(msg, CTP_OPT_PULL))
        resetInterval();

    delete msg ;
}
```

Table Update Handler

> **When** (A table update event is received)
>
> **Then**
>
> **1** Read new neighbor information: ID, Parent ID, ETX, *PST_ID*
>
> **2** If routing table is full, then stop
>
> **3** *If neighbor PST_ID is not equal to my PST_ID, I am a bridge*
>
> **4** Add new neighbor information to the routing table

Source code:

```
error_t CtpRoutingEngine::routingTableUpdateEntry(
    am_addr_t from,
    am_addr_t parent,
    uint16_t etx,
    am_addr_t pstId){

    uint8_t idx;
    uint16_t  linkEtx;
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(from));

    idx = routingTableFind(from);
    if(idx == routingTableSize){
        trace()<<"routingTableUpdateEntry - FAIL, table full";
        return FAIL;
    }
    else if(idx == routingTableActive){
        if(passLinkEtxThreshold(linkEtx)){
            routingTable[idx].neighbor = from;
            routingTable[idx].pstId = pstId;
            routingTable[idx].info.parent = parent;
            routingTable[idx].info.etx = etx;
            routingTable[idx].info.haveHeard = 1;
            routingTable[idx].info.congested = false;
```

```
            routingTableActive++;
            trace()<<"routingTableUpdateEntry - OK, new entry";
            if (pstId != my_pstId)
                trace()<<"## my_pstId: "<<(int)my_pstId<<" neighbor pstId: "
                <<pstId;
        }
        else
            trace()<<"routingTableUpdateEntry - Fail, link quality("
            <<(int)linkEtx<<") below threshold";

    }
    else{
        //found, just update
        routingTable[idx].neighbor = from;
        routingTable[idx].info.parent = parent;
        routingTable[idx].info.etx = etx;
        routingTable[idx].info.haveHeard = 1;
        trace()<<"routingTableUpdateEntry - OK, updated entry";
    }
    return SUCCESS;
}
```

Update Route Handler:

When (An update route event is received)

Then

1. Read all neighbor information from the table: ID, Parent ID, ETX, *PST_ID*
2. Pick the neighbor with the best ETX
3. Set that neighbor as the new parent

Source code:

```
void CtpRoutingEngine::updateRouteTask(){

    uint8_t i;
    routing_table_entry* entry;
    routing_table_entry* best;
    uint16_t minEtx;
    uint16_t currentEtx;
    uint16_t linkEtx, pathEtx;

    if (state_is_root)
        return;

    best = NULL;
    /* Minimum etx found among neighbors, initially infinity */
    minEtx = MAX_METRIC;
    /* Metric through current parent, initially infinity */
    currentEtx = MAX_METRIC;

    trace()<<"updateRouteTask";
```

```
/* Find best path in table, other than our current */
for(i = 0; i < routingTableActive; i++){
    entry = &routingTable[i];

    // Avoid bad entries and 1-hop loops
    if(entry->info.parent == INVALID_ADDR ||
       entry->info.parent == my_ll_addr){

        trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "
        <<(int)entry->neighbor<<" parent: "<<entry->info.parent
        <<"  etx: NO_ROUTE]";

        continue;
    }

    // Compute this neighbor's path metric
    linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(entry->neighbor));

    trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor
    <<" parent: "<<entry->info.parent<<"  etx: "<<(int)linkEtx
    <<" pstId: "<<(int)entry->pstId<<"]";

    pathEtx = linkEtx + entry->info.etx;
    /* Operations specific to the current parent */
    if(entry->neighbor == routeInfo.parent){
        trace()<<"already parent";

        currentEtx = pathEtx;

        /* update routeInfo with parent's current info */
        routeInfo.etx = entry->info.etx;
        routeInfo.congested = entry->info.congested;
        continue;
    }

    /* Ignore links that are congested */
    if(entry->info.congested)
        continue;
    /* Ignore links that are bad */
    if(!passLinkEtxThreshold(linkEtx)){
        trace()<<"did not pass threshold.";
        continue;
    }

    if(pathEtx < minEtx){
        minEtx = pathEtx;
        best = entry;
    }
}
if(minEtx != MAX_METRIC){
    if(currentEtx == MAX_METRIC ||
```

```
            (routeInfo.congested && (minEtx < (routeInfo.etx + 10))) ||
            minEtx + PARENT_SWITCH_THRESHOLD < currentEtx){

              parentChanges++;

              trace()<<"Changed parent. from "<<(int)routeInfo.parent<<" to "
              <<(int)best->neighbor<<" with pstId "<<(int)best->pstId;

              le->command_LinkEstimator_unpinNeighbor(routeInfo.parent) ;
              le->command_LinkEstimator_pinNeighbor(best->neighbor) ;
              le->command_LinkEstimator_clearDLQ(best->neighbor) ;

              routeInfo.parent = best->neighbor;
              routeInfo.etx = best->info.etx;
              routeInfo.congested = best->info.congested;
              my_pstId = best->pstId;
          }
      }
      else if(!justEvicted &&
              currentEtx == MAX_METRIC &&
              minEtx != MAX_METRIC)
          signal_Routing_routeFound();

      justEvicted = false;
}
```

10 Simulations

Our research has utilized simulations in order to increase our understanding of how the CTP works, and also to show some principles of the network. Our network simulations utilize the Castalia v3.0 wireless sensor networks simulator, which operates on top of the OMNeT++ v4.1 network simulator. In the Castalia simulator, we have modified an implementation of the CTP protocol adapted for the C++ language used by Castalia. The original C++ code for the CTP protocol has been provided to us by the authors of [6].

In our project, we simulated networks of 50, 150, and 300 MICA2 WSN nodes with a single base station. The simulated WSN networks were programmed to use the CTP protocol for routing. After successfully simulating a network of 50 nodes, we moved on to simulating larger networks. In addition, we implemented a sinkhole node in the simulation network. The sinkhole actively sends and receives network beacons but does not forward actual application (sensor) data to the base station.

Data from sixty simulation runs are shown in Appendix A.1. The graph shows the effect a sinkhole can have on a WSN that utilizes CTP. The graph shows simulations on a 300 node network. In these simulation runs, thirty individual network topologies were used. Each topology was generated using a random uniform distribution. Within each distribution, two simulations were run: one that tested the network with a sinkhole, and one that tested the network without a sinkhole. The graph shows the individual topologies as two columns: the red column represents packet loss[7] (as a percentage) with a sinkhole, while the blue column represents the packet loss without a sinkhole. It is important to note that there can be packet loss when there is no sinkhole in the network – these losses can be attributed to simulated data collisions[8] or errors.

Simulations were also conducted using our modified protocol. Through the trace file generated during each simulation run, we were able to determine that bridge nodes were successfully discovered by our protocol.

11 Conclusion

This project has developed a novel strategy to mitigate sinkhole attacks in WSNs that utilize tree structured routing. Our work has shown that an implementation of this strategy in the CTP protocol is feasible based on simulations that show the existence of 'bridge nodes' in ad-hoc WSNs that are initialized using CTP. Future work on this area might include implementing our sinkhole mitigation strategy into a working version of the Collection Tree Protocol (CTP) and testing its effectiveness with respect to the overall data delivery ratio in the Castalia wireless sensor networks simulator.

References

[1] J. Yick, B. Mukherjee and D. Ghosal. Wireless Sensor Network Survey. *Department of Computer Science, University of California at Davis*, 2008.

[7] Packet loss refers to the percentage of data that was transmitted by a node in the network but was not successfully received by the base station.

[8] Data collisions occur when a node receives two or more data packets simultaneously and the node is forced to drop some or all of the data.

[2] M. Saraogi. Security in Wireless Sensor Networks. *Department of Computer Science, University of Tennessee, Knoxville*, 2005.

[3] C. Karlof and D. Wagner. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. *University of California, Berkeley.*

[4] A. Perrig, J. Stankovic, and D. Wagner. Security in Wireless Sensor Networks. In *Communications of the ACM* Vol 47 No 6, 2004.

[5] A. Vaseashta and S. Vaseashta. A Survey of Sensor Network Security. In *Sensors & Transducers Journal* Vol 94 Issue 7, 2008.

[6] U. Colesanti and S. Santini. A Performance Evaluation Of The Collection Tree Protocol Based On Its Implementation For The Castalia Wireless Sensor Networks Simulator. *Department of Computer Science, ETH Zurich* Tech Rep 681, 2010.

[7] O. Gnawali, et al. The Collection Tree Protocol. *http://sing.stanford.edu/gnawali/ctp/.*

[8] T. Shu, M. Krunz, and S. Liu. Secure data Collection in Wireless Sensor Networks Using Randomized Dispersive Routes. In *IEEE Transactions on Mobile Computing* Vol 9 No 7, 2010.

[9] G. Tel. *Introducion to Distributed Algorithms.* New York: Cambridge University Press, 2000.

[10] J. Deng, R. Han, and S. Mishra. INSENS: Intrusion–Tolerant Routing For Wireless Sensor Networks. *Computer Communications* 29, 2006.

[11] I. Krontiris, et al. Intrusion Detection of Sinkhole Attacks in Wireless Sensor Networks. *Anthens Information Technology*, 2007.

[12] D. Dolev and A. Yao. On the Security of Public Key Protocols. In *IEEE Transactions on Information Theory 29*(2) 1983.

[13] W. Pestman. *Mathematical Statistics: An Introduction.* Berlin: Walter De Gruyter Co., 1998.

[14] J. Gentle. *Elements of Computational Statistics,* New York: Springer-Verlag, 2002.

[15] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring The Internet* New York: Addison Wesley, 2005.

[16] "IRIS Datasheet." The IRIS Wireless Measurement System.
Crossbow Technology, Inc.

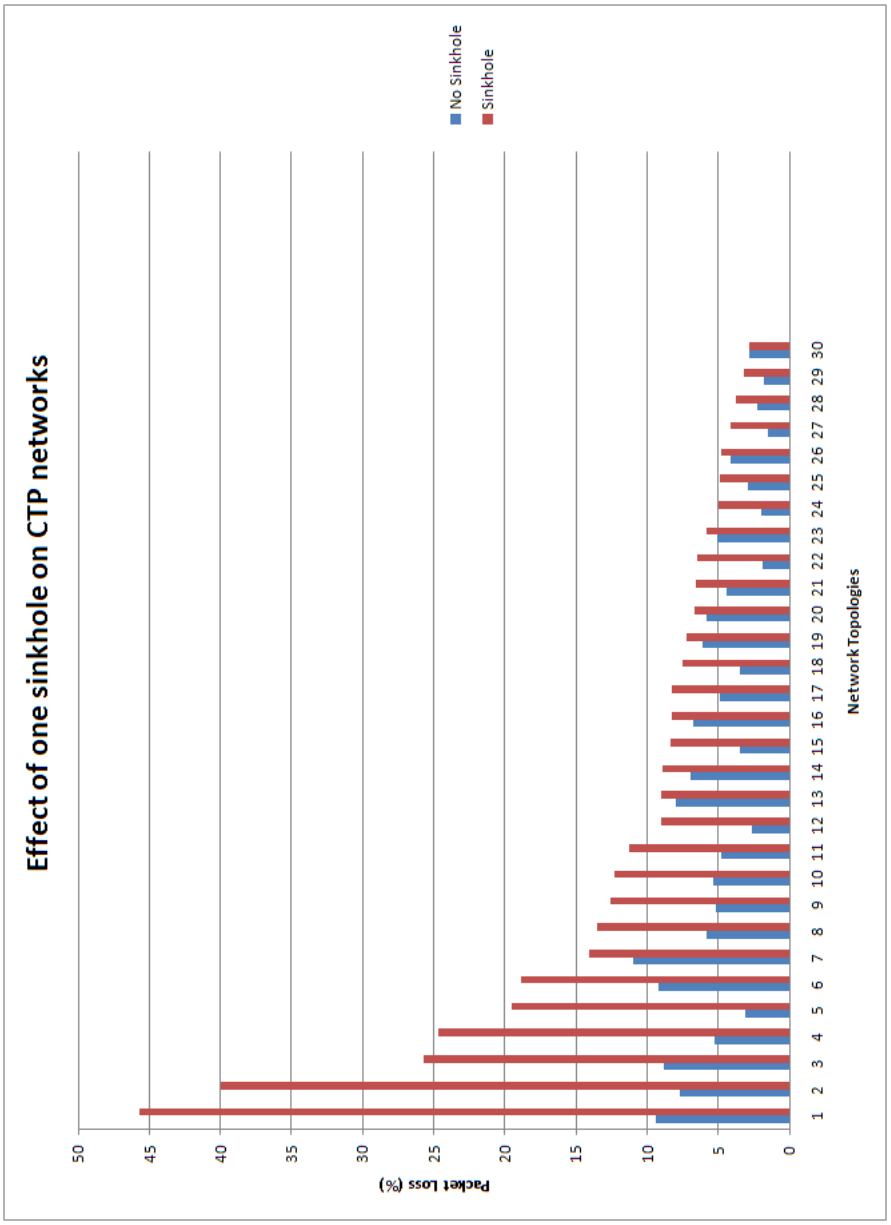
[17] "MICA2 Datasheet." The MICA2 Wireless Measurement System.
Crossbow Technology, Inc.

[18] "TinyOS." *http://www.tinyos.net/.*

A Appendix

A.1 Effect of Sinkhole in Simulation

Note: This graph is described in the Simulations section



A.2 Source Code

CTP Routing Engine:

```
#include "CtpRoutingEngine.h"
#include "CtpForwardingEngine.h"
#include "LinkEstimator.h"
Define_Module(CtpRoutingEngine);

void CtpRoutingEngine::initialize(){
        ////////////////////////// Castalia Implementation ////////////////////
        ////////////////////////////////////////////////////////////////////////

        //Pointers to other modules for direct function calls.
        cfe = check_and_cast<CtpForwardingEngine*>(getParentModule()->getSubmodule("CtpForwardingEngine"))
              ;
        le = check_and_cast<LinkEstimator*>(getParentModule()->getSubmodule("LinkEstimator")) ;

        //Id of the node (like TOS_NODE_ID)
        self = getParentModule()->getParentModule()->getParentModule()->getIndex();

        // The default values are set in CtpRoutingEngine.ned
        // but they can be overwritten in omnetpp.ini
        routingTableSize = par("routingTableSize"); // default 10 entries
        minInterval = par("minInterval"); // default 128
        maxInterval = par("maxInterval") ; // default 512000
        ctpReHeaderSize = par("ctpReHeaderSize") ; // default header size: 5 bytes
        isRoot = par("isRoot") ; // sets this node as root


        // Clock drift simulation (it is present at each layer)
        if (getParentModule()->getParentModule()->getParentModule()->findSubmodule("ResourceManager") !=
             -1) {
                resMgrModule = check_and_cast <ResourceManager*>(getParentModule()->getParentModule()->
                     getParentModule()->getSubmodule("ResourceManager"));
        } else {
                opp_error("\n[Mac]:\n Error in geting a valid reference to ResourceManager for direct
                     method calls.");
        }
        setTimerDrift(resMgrModule->getCPUClockDrift());

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        ////////////////////// CtpForwardingEngine (default) /////////////////
        ////////////////////////////////////////////////////////////////////////

        ECNOff = true;
        radioOn = true ; // TO IMPLEMENT ------ radioOn in stdcontrol
        running = false ;
        sending = false ;
        justEvicted = false ;

        routingTable = new routing_table_entry[routingTableSize] ;

        currentInterval = minInterval;

        ////////////////////////////////////////////////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        ////////////////////////// Init.init() ////////////////////////////////
        ////////////////////////////////////////////////////////////////////////

        routeUpdateTimerCount = 0;
        parentChanges = 0;
        state_is_root = 0;

        routeInfoInit(&routeInfo);
```

```
61          routingTableInit();
62          my_ll_addr = command_AMPacket_address() ;
63
64          beaconMsg = new CtpBeacon() ;
65          beaconMsg->setByteLength(ctpReHeaderSize) ;
66
67          // Call the corresponding rootcontrol command
68          isRoot? command_RootControl_setRoot() : command_RootControl_unsetRoot() ;
69
70          ////////////////////////////////////////////////////////////////////
71          ////////////////////////////////////////////////////////////////////
72
73
74          ////////////////////////////////// Statistics /////////////////////////////////
75          ///////////////////////////////////////////////////////////////////////////////
76
77          declareOutput("Ctp_Beacons") ;
78
79          ///////////////////////////////////////////////////////////////////////////////
80          ///////////////////////////////////////////////////////////////////////////////
81  }
82
83  void CtpRoutingEngine::handleMessage(cMessage* msg){
84          int msgKind = msg->getKind();
85                  switch (msgKind) {
86          case TIMER_SERVICE:{
87                  handleTimerMessage(msg);
88                  break;
89          }
90          default:{
91                  opp_error("Unkown message type.") ;
92          }
93                  }
94                  delete msg ;
95
96  }
97
98  void CtpRoutingEngine::timerFiredCallback(int timer)
99  {
100         trace() << "CtpRE - TimerFiredCallback, value: "<<timer;
101         switch (timer) {
102
103                 case ROUTE_TIMER:{
104                         setTimer(ROUTE_TIMER, tosMillisToSeconds(BEACON_INTERVAL)) ; // because it's a
                                periodic timer.
105                         event_RouteTimer_fired() ;
106                         break;
107                 }
108                 case BEACON_TIMER:{
109                         event_BeaconTimer_fired() ;
110                         break;
111                 }
112                 case POST_UPDATEROUTETASK:{
113                         updateRouteTask() ;
114                         break ;
115                 }
116
117                 case POST_SENDBEACONTASK:{
118                         sendBeaconTask() ;
119                         break ;
120                 }
121
122
123                 default:{
124                         opp_error("Unexpected message!");
125                 }
126         }
127 }
128
129 CtpRoutingEngine::~CtpRoutingEngine(){
```

```
130         delete beaconMsg ;
131         beaconMsg = NULL ;
132 
133         delete [] routingTable ;
134         routingTable = NULL ;
135 }
136 
137 void CtpRoutingEngine::chooseAdvertiseTime() {
138         t = currentInterval;
139         t /= 2;
140         t += command_Random_rand32(1) % t;
141         tHasPassed = false;
142         setTimer(BEACON_TIMER, tosMillisToSeconds(t)) ;
143 }
144 
145 void CtpRoutingEngine::resetInterval() {
146   currentInterval = minInterval;
147   chooseAdvertiseTime();
148 }
149 
150 void CtpRoutingEngine::decayInterval() {
151         currentInterval *= 2;
152         if (currentInterval > maxInterval) {
153                 currentInterval = maxInterval;
154         }
155         chooseAdvertiseTime();
156 }
157 
158 void CtpRoutingEngine::remainingInterval() {
159    uint32_t remaining = currentInterval;
160    remaining -= t;
161    tHasPassed = true;
162    setTimer(BEACON_TIMER, tosMillisToSeconds(remaining)) ;
163 }
164 
165 error_t CtpRoutingEngine::command_StdControl_start() {
166         Enter_Method("command_StdControl_start") ;
167         //start will (re)start the sending of messages
168         if (!running) {
169                 running = true;
170                 resetInterval();
171                 setTimer(ROUTE_TIMER, tosMillisToSeconds(BEACON_INTERVAL)) ;
172                 trace()<<"stdControl.start -- running "<<running<<" radioOn: "<<radioOn ;
173         }
174         return SUCCESS;
175 }
176 
177 error_t CtpRoutingEngine::command_StdControl_stop() {
178         Enter_Method("command_StdControl_stop") ;
179         running = false ;
180         trace()<<"stdControl.stop -- running "<<running<<" radioOn: "<<radioOn ;
181         return SUCCESS;
182 }
183 
184 void CtpRoutingEngine::event_RadioControl_startDone(error_t error) {
185         Enter_Method("event_RadioControl_startDone") ;
186         radioOn = true;
187         trace()<<"radioControl.startDone -- running "<<running<<" radioOn: "<<radioOn ;
188         if (running) {
189                 uint16_t nextInt;
190                 nextInt = command_Random_rand16(0) % BEACON_INTERVAL ;
191                 nextInt += BEACON_INTERVAL >> 1;
192                 setTimer(BEACON_TIMER, tosMillisToSeconds(nextInt));
193         }
194 }
195 
196 void CtpRoutingEngine::event_RadioControl_stopDone(error_t error) {
197         Enter_Method("event_RadioControl_stopDone") ;
198         radioOn = false;
199         trace()<<"radioControl.stopDone -- running "<<running<<" radioOn: "<<radioOn ;
```

```
200
201 }
202
203 /* Is this quality measure better than the minimum threshold? */
204 // Implemented assuming quality is EETX
205 bool CtpRoutingEngine::passLinkEtxThreshold(uint16_t etx) {
206         return true;
207         //      return (etx < ETX_THRESHOLD);
208 }
209
210 /* Converts the output of the link estimator to path metric
211  * units, that can be *added* to form path metric measures */
212 uint16_t CtpRoutingEngine::evaluateEtx(uint16_t quality) {
213         trace()<<"evaluateEtx - "<<(int) quality<<" -> "<<(int)(quality+10);
214         return (quality + 10);
215 }
216
217
218 /* updates the routing information, using the info that has been received
219  * from neighbor beacons. Two things can cause this info to change:
220  * neighbor beacons, changes in link estimates, including neighbor eviction */
221 void CtpRoutingEngine::updateRouteTask() {
222         uint8_t i;
223         routing_table_entry* entry;
224         routing_table_entry* best;
225         uint16_t minEtx;
226         uint16_t currentEtx;
227         uint16_t linkEtx, pathEtx;
228
229         if (state_is_root)
230                 return;
231
232         best = NULL;
233         /* Minimum etx found among neighbors, initially infinity */
234         minEtx = MAX_METRIC;
235         /* Metric through current parent, initially infinity */
236         currentEtx = MAX_METRIC;
237
238         trace()<<"updateRouteTask" ;
239
240         /* Find best path in table, other than our current */
241         for (i = 0; i < routingTableActive; i++) {
242                 entry = &routingTable[i];
243
244                 // Avoid bad entries and 1-hop loops
245                 if (entry->info.parent == INVALID_ADDR || entry->info.parent == my_ll_addr) {
246                         trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor<<"
                                parent: "<<entry->info.parent<<"  etx: NO_ROUTE]" ;
247                         continue;
248                 }
249                 /* Compute this neighbor's path metric */
250                 linkEtx = evaluateEtx(/*call LinkEstimator.getLinkQuality(entry->neighbor) changed with:
                        */ le->command_LinkEstimator_getLinkQuality(entry->neighbor));
251                 trace()<<"routingTable["<<(int)i<<"]: neighbor: [id: "<<(int)entry->neighbor<<" parent: "
                        <<entry->info.parent<<"  etx: "<<(int)linkEtx
252                           <<" pstId: "<<(int)entry->nonepstId<<"]";
253                 pathEtx = linkEtx + entry->info.etx;
254                 /* Operations specific to the current parent */
255                 if (entry->neighbor == routeInfo.parent) {
256                         trace()<<"already parent";
257                         currentEtx = pathEtx;
258                         /* update routeInfo with parent's current info */
259                         routeInfo.etx = entry->info.etx;
260                         routeInfo.congested = entry->info.congested;
261                         continue;
262                 }
263                 /* Ignore links that are congested */
264                 if (entry->info.congested)
265                         continue;
266                 /* Ignore links that are bad */
```

```
267                     if (!passLinkEtxThreshold(linkEtx)) {
268                             trace()<<"did not pass threshold.";
269                             continue;
270                     }
271
272                     if (pathEtx < minEtx) {
273                             minEtx = pathEtx;
274                             best = entry;
275                     }
276             }
277
278
279             /* Now choose between the current parent and the best neighbor */
280             /* Requires that:
281                 1. at least another neighbor was found with ok quality and not congested
282                 2. the current parent is congested and the other best route is at least as good
283                 3. or the current parent is not congested and the neighbor quality is better by
284                    the PARENT_SWITCH_THRESHOLD.
285               Note: if our parent is congested, in order to avoid forming loops, we try to select
286                     a node which is not a descendent of our parent. routeInfo.ext is our parent's
287                     etx. Any descendent will be at least that + 10 (1 hop), so we restrict the
288                     selection to be less than that.
289              */
290             if (minEtx != MAX_METRIC) {
291                     if (currentEtx == MAX_METRIC ||
292                                     (routeInfo.congested && (minEtx < (routeInfo.etx + 10))) ||
293                                     minEtx + PARENT_SWITCH_THRESHOLD < currentEtx) {
294                             // routeInfo.metric will not store the composed metric.
295                             // since the linkMetric may change, we will compose whenever
296                             // we need it: i. when choosing a parent (here);
297                             //             ii. when choosing a next hop
298                             parentChanges++;
299
300                             trace()<<"Changed parent. from "<<(int)routeInfo.parent<<" to "<<(int)best->
                                    neighbor<<" with pstId "<<(int)best->nonepstId;
301                             le->command_LinkEstimator_unpinNeighbor(routeInfo.parent) ;
302                             le->command_LinkEstimator_pinNeighbor(best->neighbor) ;
303                             le->command_LinkEstimator_clearDLQ(best->neighbor) ;
304
305                             routeInfo.parent = best->neighbor;
306                             routeInfo.etx = best->info.etx;
307                             routeInfo.congested = best->info.congested;
308                             nonemy_pstId = best->nonepstId;
309                     }
310             }
311
312             /* Finally, tell people what happened:  */
313             /* We can only loose a route to a parent if it has been evicted. If it hasn't
314              * been just evicted then we already did not have a route */
315             if (justEvicted && routeInfo.parent == INVALID_ADDR)
316                     signal_Routing_noRoute() ;
317             /* On the other hand, if we didn't have a parent (no currentEtx) and now we
318              * do, then we signal route found. The exception is if we just evicted the
319              * parent and immediately found a replacement route: we don't signal in this
320              * case */
321             else if (!justEvicted &&
322                             currentEtx == MAX_METRIC &&
323                             minEtx != MAX_METRIC)
324                     signal_Routing_routeFound() ;
325             justEvicted = false;
326     }
327
328     /* send a beacon advertising this node's routeInfo */
329     // only posted if running and radioOn
330     void CtpRoutingEngine::sendBeaconTask() {
331             error_t eval;
332             if (sending) {
333                     return;
334             }
335
```

```
336         beaconMsg->setOptions(0) ;
337
338         /* Congestion notification: am I congested? */
339         if (cfe->command_CtpCongestion_isCongested()) {
340                 beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_ECN) ;
341         }
342
343         beaconMsg->setParent(routeInfo.parent) ;
344         if (state_is_root) {
345                 beaconMsg->setEtx(routeInfo.etx) ;
346         }
347         else if (routeInfo.parent == INVALID_ADDR) {
348                 beaconMsg->setEtx(routeInfo.etx) ;
349                 beaconMsg->setOptions(beaconMsg->getOptions() | CTP_OPT_PULL) ;
350         } else {
351                 beaconMsg->setEtx(routeInfo.etx + le->command_LinkEstimator_getLinkQuality(routeInfo.
                        parent)) ;
352         }
353
354         beaconMsg->nonesetPstId(nonemy_pstId); // hmm; i should have set my_pstId to that of my parent
355
356         trace()<<"sendBeaconTask - parent: "<<(int)beaconMsg->getParent()<<" etx: "<<(int)beaconMsg->
                getEtx();
357
358         beaconMsg->getRoutingInteractionControl().lastHop = self ; // ok
359         eval = le->command_Send_send(AM_BROADCAST_ADDR, beaconMsg->dup()); // the duplicate will be deleted
                in the LE module, we keep a copy here that is reused each time.
360
361         if (eval == SUCCESS) {
362                 //statistics
363                 collectOutput("Ctp Beacons","Tx") ;
364                 sending = true;
365         } else if (eval == EOFF) {
366                 radioOn = false;
367                 trace()<<"sendBeaconTask - running: "<<running<<" radioOn: "<<radioOn ;
368         }
369 }
370
371 void CtpRoutingEngine::event_BeaconSend_sendDone(cMessage* msg, error_t error) {
372         Enter_Method("event_BeaconSend_sendDone") ;
373         if (!sending) {
374                 //something smells bad around here
375                 opp_error("something smells bad around here");
376                 return;
377         }
378         sending = false;
379 }
380
381 void CtpRoutingEngine::event_RouteTimer_fired() {
382         if (radioOn && running) {
383                 post_updateRouteTask() ;
384         }
385 }
386
387 void CtpRoutingEngine::event_BeaconTimer_fired() {
388         if (radioOn && running) {
389                 if (!tHasPassed) {
390                         post_updateRouteTask() ; // always the most up to date info
391                         post_sendBeaconTask() ;
392                         trace()<<"Beacon timer fired.";
393                         remainingInterval();
394                 }
395                 else {
396                         decayInterval();
397                 }
398         }
399 }
400
401 /*
402  * We don't need a pointer to the header, we can use the methods of cPacket
```

```
403  * instead, we return a pointer to CtpBeacon and that's it.
404  */
405 CtpBeacon* CtpRoutingEngine::getHeader(cPacket* msg){
406         return check_and_cast<CtpBeacon*>(msg) ;
407 }
408
409 /* Handle the receiving of beacon messages from the neighbors. We update the
410  * table, but wait for the next route update to choose a new parent */
411 void CtpRoutingEngine::event_BeaconReceive_receive(cPacket* msg) {
412         Enter_Method("event_BeaconReceive_receive") ;
413         am_addr_t from;
414         bool congested;
415
416         //statistics
417         collectOutput("Ctp_Beacons","Rx") ;
418
419         // we skip the check of beacon length.
420
421         //need to get the am_addr_t of the source
422         from = command_AMPacket_source(msg) ;
423         CtpBeacon* rcvBeacon = check_and_cast<CtpBeacon*>(msg) ;
424
425         congested = command_CtpRoutingPacket_getOption(msg,CTP_OPT_ECN) ;
426
427         trace()<<"BeaconReceive.receive__from_"<<(int)from<<"_[parent:_"<<(int)rcvBeacon->getParent()<<"_
              etx:_"<<(int)rcvBeacon->getEtx()
428           <<"_pstId:_"<<(int)rcvBeacon->nonegetPstId()<<"]";
429         //update neighbor table
430         if (rcvBeacon->getParent() != INVALID_ADDR) {
431
432                 /* If this node is a root, request a forced insert in the link
433                  * estimator table and pin the node. */
434                 if (rcvBeacon->getEtx() == 0) {
435                         trace()<<"from_a_root,_inserting_if_not_in_table_"<<"my_ll_addr:_"<<my_ll_addr;
436                         le->command_LinkEstimator_insertNeighbor(from) ;
437                         le->command_LinkEstimator_pinNeighbor(from) ;
438                         nonemy_pstId = my_ll_addr;  // since i hear root, i'm the root of a principle
                              subtree
439                 }
440                 //TODO: also, if better than my current parent's path etx, insert
441
442                 if (rcvBeacon->getEtx() == 0)
443                         routingTableUpdateEntry(from, rcvBeacon->getParent(), rcvBeacon->getEtx(),
                              nonemy_pstId);
444                 else
445                         routingTableUpdateEntry(from, rcvBeacon->getParent(), rcvBeacon->getEtx(),
                              rcvBeacon->nonegetPstId());
446
447                 command_CtpInfo_setNeighborCongested(from,congested) ;
448         }
449
450         if (command_CtpRoutingPacket_getOption(msg, CTP_OPT_PULL)) {
451                 resetInterval();
452         }
453         delete msg ;
454         // we do not return the message, we delete it.
455 }
456
457 /* Signals that a neighbor is no longer reachable. need special care if
458  * that neighbor is our parent */
459 void CtpRoutingEngine::event_LinkEstimator_evicted(am_addr_t neighbor) {
460         Enter_Method("event_LinkEstimator_evicted") ;
461         routingTableEvict(neighbor);
462         trace()<<"LinkEstimator.evicted" ;
463         if (routeInfo.parent == neighbor) {
464                 routeInfoInit(&routeInfo);
465                 justEvicted = true;
466                 post_updateRouteTask() ;
467         }
468 }
```

```
469
470 /*
471  * UnicastNameFreeRouting Inteface ----------------------------------------
472  */
473 /* Simple implementation: return the current routeInfo */
474 am_addr_t CtpRoutingEngine::command_Routing_nextHop() {
475         Enter_Method("command_Routing_nextHop") ;
476         return routeInfo.parent;
477 }
478 bool CtpRoutingEngine::command_Routing_hasRoute() {
479         Enter_Method("command_Routing_hasRoute") ;
480         return (routeInfo.parent != INVALID_ADDR);
481 }
482 // ----------------------------------------------------------------------
483
484 /*
485  * CtpInfo Interface (Part 1) -------------------------------------------------------------------
486  */
487 error_t CtpRoutingEngine::command_CtpInfo_getParent(am_addr_t* parent) {
488         if (parent == NULL)
489                 return FAIL;
490         if (routeInfo.parent == INVALID_ADDR)
491                 return FAIL;
492         *parent = routeInfo.parent;
493         return SUCCESS;
494 }
495
496 error_t CtpRoutingEngine::command_CtpInfo_getEtx(uint16_t* etx) {
497         Enter_Method("command_CtpInfo_getEtx") ;
498         if (etx == NULL)
499                 return FAIL;
500         if (routeInfo.parent == INVALID_ADDR)
501                 return FAIL;
502         if (state_is_root == 1) {
503                 *etx = 0;
504         } else {
505                 // path etx = etx(parent) + etx(link to the parent)
506                 *etx = routeInfo.etx + evaluateEtx(le->command_LinkEstimator_getLinkQuality(routeInfo.
                        parent) );
507         }
508         return SUCCESS;
509 }
510
511 void CtpRoutingEngine::command_CtpInfo_recomputeRoutes() {
512         Enter_Method("command_CtpInfo_recomputeRoutes") ;
513         post_updateRouteTask() ;
514 }
515
516 void CtpRoutingEngine::command_CtpInfo_triggerRouteUpdate() {
517         Enter_Method("command_CtpInfo_triggerRouteUpdate") ;
518         resetInterval();
519 }
520
521 void CtpRoutingEngine::command_CtpInfo_triggerImmediateRouteUpdate() {
522         Enter_Method("command_CtpInfo_triggerImmediateRouteUpdate") ;
523         resetInterval();
524 }
525
526 void CtpRoutingEngine::command_CtpInfo_setNeighborCongested(am_addr_t n, bool congested) {
527         Enter_Method("command_CtpInfo_setNeighborCongested") ;
528         uint8_t idx;
529         if (ECNOff)
530                 return;
531         idx = routingTableFind(n);
532         if (idx < routingTableActive) {
533                 routingTable[idx].info.congested = congested;
534         }
535         if (routeInfo.congested && !congested)
536                 post_updateRouteTask() ;
537         else if (routeInfo.parent == n && congested)
```

```
538                 post_updateRouteTask() ;
539 }
540
541 bool CtpRoutingEngine::command_CtpInfo_isNeighborCongested(am_addr_t n) {
542         Enter_Method("command_CtpInfo_isNeighborCongested") ;
543         uint8_t idx;
544
545         if (ECNOff)
546                 return false;
547
548         idx = routingTableFind(n);
549         if (idx < routingTableActive) {
550                 return routingTable[idx].info.congested;
551         }
552         return false;
553 }
554 // -----------------------------------------------------------------------
555
556 /*
557  *  RootControl Interface ------------------------------------------------
558  */
559 /** sets the current node as a root, if not already a root */
560 /*  returns FAIL if it's not possible for some reason      */
561 error_t CtpRoutingEngine::command_RootControl_setRoot() {
562         Enter_Method("command_RootControl_setRoot") ;
563         bool route_found = false ;
564         route_found = (routeInfo.parent == INVALID_ADDR);
565         state_is_root = 1;
566         nonemy_pstId  = 0; // a root is no principle subtree
567         routeInfo.parent = my_ll_addr; //myself
568         routeInfo.etx = 0;
569         if (route_found)
570                 signal_Routing_routeFound() ;
571         trace()<<"RootControl.setRoot - I'm a root now!"<<(int) routeInfo.parent  ;
572         return SUCCESS;
573 }
574
575 error_t CtpRoutingEngine::command_RootControl_unsetRoot() {
576         Enter_Method("command_RootControl_unsetRoot") ;
577         state_is_root = 0;
578         routeInfoInit(&routeInfo);
579         trace()<<"RootControl.unsetRoot - I'm not a root now!" ;
580         post_updateRouteTask() ;
581         return SUCCESS;
582 }
583
584 bool CtpRoutingEngine::command_RootControl_isRoot() {
585         Enter_Method("command_RootControl_isRoot") ;
586         return state_is_root;
587 }
588 // -----------------------------------------------------------------------
589
590 // default events Routing.noRoute and Routing.routeFound skipped -> useless
591
592 /* This should see if the node should be inserted in the table.
593  * If the white_bit is set, this means the LL believes this is a good
594  * first hop link.
595  * The link will be recommended for insertion if it is better* than some
596  * link in the routing table that is not our parent.
597  * We are comparing the path quality up to the node, and ignoring the link
598  * quality from us to the node. This is because of a couple of things:
599  *    1. because of the white bit, we assume that the 1-hop to the candidate
600  *       link is good (say, etx=1)
601  *    2. we are being optimistic to the nodes in the table, by ignoring the
602  *       1-hop quality to them (which means we are assuming it's 1 as well)
603  *       This actually sets the bar a little higher for replacement
604  *    3. this is faster
605  *    4. it doesn't require the link estimator to have stabilized on a link
606  */
607 bool CtpRoutingEngine::event_CompareBit_shouldInsert(cPacket *msg, bool white_bit) {
```

```
608         Enter_Method("event_CompareBit_shouldInsert") ;
609         bool found = false;
610         uint16_t pathEtx;
611         uint16_t neighEtx;
612         int i;
613         routing_table_entry* entry;
614         CtpBeacon* rcvBeacon ;
615 
616         // checks if it is a CtpBeacon
617         if(dynamic_cast<CtpBeacon*>(msg) == NULL){
618                 delete msg ;
619                 return false ;
620         }
621 
622         /* 1.determine this packet's path quality */
623         rcvBeacon = check_and_cast<CtpBeacon*>(msg); // we don't need a pointer to header, we use cPacket
                methods.
624 
625         if (rcvBeacon->getParent() == INVALID_ADDR){
626                 delete msg ;
627                 return false;
628         }
629 
630         /* the node is a root, recommend insertion! */
631         if (rcvBeacon->getEtx() == 0) {
632                 delete msg ;
633                 return true;
634         }
635 
636         pathEtx = rcvBeacon->getEtx() ;
637 
638         /* 2. see if we find some neighbor that is worse */
639         for (i = 0; i < routingTableActive && !found; i++) {
640                 entry = &routingTable[i];
641                 //ignore parent, since we can't replace it
642                 if (entry->neighbor == routeInfo.parent)
643                         continue;
644                 neighEtx = entry->info.etx;
645                 //neighEtx = evaluateEtx(call LinkEstimator.getLinkQuality(entry->neighbor));
646                 found |= (pathEtx < neighEtx);
647         }
648         delete msg ;
649         return found;
650 }
651 
652 /*****************************************************/
653 /* Routing Table Functions                           */
654 
655 /* The routing table keeps info about neighbor's route_info,
656  * and is used when choosing a parent.
657  * The table is simple:
658  *   - not fragmented (all entries in 0..routingTableActive)
659  *   - not ordered
660  *   - no replacement: eviction follows the LinkEstimator table
661  */
662 
663 void CtpRoutingEngine::routingTableInit() {
664         routingTableActive = 0;
665 }
666 
667 /* Returns the index of parent in the table or
668  * routingTableActive if not found */
669 uint8_t CtpRoutingEngine::routingTableFind(am_addr_t neighbor) {
670         uint8_t i;
671         if (neighbor == INVALID_ADDR)
672                 return routingTableActive;
673         for (i = 0; i < routingTableActive; i++) {
674                 if (routingTable[i].neighbor == neighbor)
675                         break;
676         }
```

```
677         return i;
678 }
679
680
681 error_t CtpRoutingEngine::routingTableUpdateEntry(am_addr_t from, am_addr_t parent, uint16_t etx,
        am_addr_t nonepstId)    {
682         uint8_t idx;
683         uint16_t  linkEtx;
684         linkEtx = evaluateEtx(le->command_LinkEstimator_getLinkQuality(from));
685
686         idx = routingTableFind(from);
687         if (idx == routingTableSize) {
688                 //not found and table is full
689                 //if (passLinkEtxThreshold(linkEtx))
690                 //TODO: add replacement here, replace the worst
691                 //}
692                 trace()<<"routingTableUpdateEntry - FAIL, table full";
693                 return FAIL;
694         }
695         else if (idx == routingTableActive) {
696                 //not found and there is space
697                 if (passLinkEtxThreshold(linkEtx)) {
698                         routingTable[idx].neighbor = from;
699                         routingTable[idx].nonepstId = nonepstId;
700                         routingTable[idx].info.parent = parent;
701                         routingTable[idx].info.etx = etx;
702                         routingTable[idx].info.haveHeard = 1;
703                         routingTable[idx].info.congested = false;
704                         routingTableActive++;
705                         trace()<<"routingTableUpdateEntry - OK, new entry";
706                         if (nonepstId != nonemy_pstId)
707                                 trace()<<"## my_pstId: "<<(int)nonemy_pstId<<" neighbor pstId: "<<nonepstId
                                        ;
708                 } else {
709                         trace()<<"routingTableUpdateEntry - Fail, link quality ("<<(int)linkEtx<<") below
                                threshold" ;
710                 }
711         } else {
712                 //found, just update
713                 routingTable[idx].neighbor = from;
714                 routingTable[idx].info.parent = parent;
715                 routingTable[idx].info.etx = etx;
716                 routingTable[idx].info.haveHeard = 1;
717                 trace()<<"routingTableUpdateEntry - OK, updated entry";
718         }
719         return SUCCESS;
720 }
721
722 /* if this gets expensive, introduce indirection through an array of pointers */
723 error_t CtpRoutingEngine::routingTableEvict(am_addr_t neighbor) {
724         uint8_t idx,i;
725         idx = routingTableFind(neighbor);
726         if (idx == routingTableActive)
727                 return FAIL;
728         routingTableActive--;
729         for (i = idx; i < routingTableActive; i++) {
730                 routingTable[i] = routingTable[i+1];
731         }
732         return SUCCESS;
733 }
734 /********** end routing table functions **************/
735
736 // Collection Debug skipped -> not useful in our implementation
737
738 /*
739  * CtpRoutingPacket Interface ------------------------------------------------------------------
740  */
741 bool CtpRoutingEngine::command_CtpRoutingPacket_getOption(cPacket* msg, ctp_options_t opt) {
742         return ((getHeader(msg)->getOptions() & opt) == opt) ? true : false;
743 }
```

```
744
745 void CtpRoutingEngine::command_CtpRoutingPacket_setOption(cPacket* msg, ctp_options_t opt) {
746         getHeader(msg)->setOptions(getHeader(msg)->getOptions() | opt) ;
747 }
748
749 void CtpRoutingEngine::command_CtpRoutingPacket_clearOption(cPacket* msg, ctp_options_t opt) {
750         getHeader(msg)->setOptions(getHeader(msg)->getOptions() & ~opt) ;
751 }
752
753 void CtpRoutingEngine::command_CtpRoutingPacket_clearOptions(cPacket* msg) {
754         getHeader(msg)->setOptions(0) ;
755 }
756
757 am_addr_t CtpRoutingEngine::command_CtpRoutingPacket_getParent(cPacket* msg) {
758         return getHeader(msg)->getParent();
759 }
760 void CtpRoutingEngine::command_CtpRoutingPacket_setParent(cPacket* msg, am_addr_t addr) {
761         getHeader(msg)->setParent(addr);
762 }
763
764 uint16_t CtpRoutingEngine::command_CtpRoutingPacket_getEtx(cPacket* msg) {
765         return getHeader(msg)->getEtx();
766 }
767
768 void CtpRoutingEngine::command_CtpRoutingPacket_setEtx(cPacket* msg, uint8_t etx) {
769         getHeader(msg)->setEtx(etx);
770 }
771 // ------------------------------------------------------------------------------------------------------
772
773 /*
774  * CtpInfo Interface (Part 2) ------------------------------------------------------------------------
775  */
776 uint8_t CtpRoutingEngine::command_CtpInfo_numNeighbors() {
777         return routingTableActive;
778 }
779
780 uint16_t CtpRoutingEngine::command_CtpInfo_getNeighborLinkQuality(uint8_t n) {
781         return (n < routingTableActive)? le->command_LinkEstimator_getLinkQuality(routingTable[n].neighbor
              ):0xffff;
782 }
783
784 uint16_t CtpRoutingEngine::command_CtpInfo_getNeighborRouteQuality(uint8_t n) {
785         return (n < routingTableActive)? le->command_LinkEstimator_getLinkQuality(routingTable[n].neighbor
              ) + routingTable[n].info.etx:0xfffff;
786 }
787
788 am_addr_t CtpRoutingEngine::command_CtpInfo_getNeighborAddr(uint8_t n) {
789         return (n < routingTableActive)? routingTable[n].neighbor:AM_BROADCAST_ADDR ;
790 }
791 // ------------------------------------------------------------------------------------------------------
792
793
794 //////////////////////////// Custom functions /////////////////////////////////////
795 ///////////////////////////////////////////////////////////////////////////////////
796
797 // These functions trigger an event in the module where they should signal it.
798 void CtpRoutingEngine::signal_Routing_routeFound(){
799         cfe->event_UnicastNameFreeRouting_routeFound() ;
800 }
801 void CtpRoutingEngine::signal_Routing_noRoute(){
802         cfe->event_UnicastNameFreeRouting_routeFound() ;
803 }
804
805 /*
806  * AMPacket Interface (just what we need) ------------------------------------
807  */
808 am_addr_t CtpRoutingEngine::command_AMPacket_source(cMessage* msg){
809         RoutingPacket* rPkt = check_and_cast<RoutingPacket*>(msg) ;
810         return (uint16_t) rPkt->getRoutingInteractionControl().lastHop ;
811 }
```

```
812
813 am_addr_t CtpRoutingEngine::command_AMPacket_address(){
814         return self;
815 }
816 // -----------------------------------------------------------------------
817
818 // these functions simulate the post command of TinyOs
819 void CtpRoutingEngine::post_updateRouteTask(){
820         setTimer(POST_UPDATEROUTETASK,0); // cannot call the updateRouteTask directly. By this way it is
                more similar to the post command in TinyOs.
821 }
822
823 void CtpRoutingEngine::post_sendBeaconTask(){
824         setTimer(POST_SENDBEACONTASK,0) ;
825 }
826
827 ////////////////////////////////////////////////////////////////////////////////
828 ////////////////////////////////////////////////////////////////////////////////
```

CTP Routing Engine Header:

```
1  #ifndef _CTPROUTINGENGINE_H_
2  #define _CTPROUTINGENGINE_H_
3
4  #include "Ctp.h"
5
6  using namespace std;
7
8
9  ////////////////////////// TreeRouting.h //////////////////////////////
10 ////////////////////////////////////////////////////////////////////////
11
12 enum {
13         AM_TREE_ROUTING_CONTROL = 0xCE,
14         BEACON_INTERVAL = 8192,
15         INVALID_ADDR  = 0xffff,
16         ETX_THRESHOLD = 50,       // link quality=20% -> ETX=5 -> Metric=50
17         PARENT_SWITCH_THRESHOLD = 15,
18         MAX_METRIC = 0xFFFF,
19 };
20
21
22 typedef struct {
23         am_addr_t parent;
24         uint16_t etx;
25         bool haveHeard;
26         bool congested;
27 } route_info_t;
28
29 typedef struct {
30         am_addr_t neighbor;
31         am_addr_t nonepstId;
32         route_info_t info;
33 } routing_table_entry;
34
35 inline void routeInfoInit(route_info_t *ri) {
36         ri->parent = INVALID_ADDR;
37         ri->etx = 0;
38         ri->haveHeard = 0;
39         ri->congested = false ;
40 }
41
42 ////////////////////////////////////////////////////////////////////////
43 ////////////////////////////////////////////////////////////////////////
44
45
46 ////////////////////////// Custom values //////////////////////////////
47 ////////////////////////////////////////////////////////////////////////
```

```
48
49  enum{
50          BEACON_TIMER = 1,
51          ROUTE_TIMER = 2,
52          POST_UPDATEROUTETASK = 3,
53          POST_SENDBEACONTASK = 4,
54  };
55
56  ////////////////////////////////////////////////////////////////////
57  ////////////////////////////////////////////////////////////////////
58
59  class CtpRoutingEngine ;
60  class LinkEstimator ;
61  class CtpRoutingEngine: public CastaliaModule , public TimerService{
62   protected:
63
64          ////////////////////// CtpRoutingEngineP.nc //////////////////////////
65          ////////////////////////////////////////////////////////////////////
66
67          bool ECNOff ;
68          bool radioOn ;
69          bool running ;
70          bool sending ;
71          bool justEvicted ;
72
73          route_info_t routeInfo ;
74          bool state_is_root;
75          am_addr_t my_ll_addr;
76
77          am_addr_t nonemy_pstId; // my pstID is that of my parent'
78
79          cPacket beaconMsgBuffer ;
80          CtpBeacon* beaconMsg ; // we don't need a pointer to the header, we use methods of cPacket instead
                .
81
82          /* routing table -- routing info about neighbors */
83          routing_table_entry* routingTable ;
84          uint8_t routingTableActive;
85
86          /* statistics */
87          uint32_t parentChanges;
88          /* end statistics */
89
90          uint32_t routeUpdateTimerCount;
91
92          uint32_t currentInterval ;
93          uint32_t t;
94          bool tHasPassed;
95
96          ////////////////////////////////////////////////////////////////////
97          ////////////////////////////////////////////////////////////////////
98
99          ////////////////////// Custom Variables //////////////////////////
100         ////////////////////////////////////////////////////////////////////
101
102         // Pointers to other modules.
103         CtpForwardingEngine *cfe ;
104         LinkEstimator *le ;
105         ResourceManager* resMgrModule;
106
107         // Beacon Frame size.
108         int ctpReHeaderSize ;
109
110         // Node Id.
111         int self;
112
113         // Sets a node as root from omnetpp.ini
114         bool isRoot ;
115
116         // Sets a node as a sinkhole from omnetpp.ini
```

```
117         bool isSink;
118
119         // Arguments of generic module CtpRoutingEngineP
120         uint32_t minInterval ;
121         uint32_t maxInterval ;
122         uint8_t routingTableSize ;
123
124         ////////////////////////////////////////////////////////////////////////
125         ////////////////////////////////////////////////////////////////////////
126
127
128         ////////////////// Castalia Functions //////////////////////////////
129         ////////////////////////////////////////////////////////////////////
130
131         virtual void initialize();
132         virtual void handleMessage(cMessage* msg) ;
133         void timerFiredCallback(int timer) ;
134         virtual ~CtpRoutingEngine() ;
135
136         ////////////////////////////////////////////////////////////////////////
137         ////////////////////////////////////////////////////////////////////////
138
139         ////////////////// CtpRoutingEngine functions //////////////////////
140         ////////////////////////////////////////////////////////////////////
141
142         void chooseAdvertiseTime();
143         void resetInterval();
144         void decayInterval();
145         void remainingInterval() ;
146
147         bool passLinkEtxThreshold(uint16_t etx) ;
148         uint16_t evaluateEtx(uint16_t quality) ;
149
150         void updateRouteTask() ;
151         void sendBeaconTask() ;
152
153         void event_RouteTimer_fired() ;
154         void event_BeaconTimer_fired() ;
155
156         CtpBeacon* getHeader(cPacket* msg) ;
157
158         void routingTableInit() ;
159         uint8_t routingTableFind(am_addr_t) ;
160         error_t routingTableUpdateEntry(am_addr_t, am_addr_t, uint16_t, am_addr_t);
161         error_t routingTableEvict(am_addr_t);
162
163         // CtpRoutingPacket Interface ---------------------------------------------
164         bool command_CtpRoutingPacket_getOption(cPacket* msg, ctp_options_t opt);
165         void command_CtpRoutingPacket_setOption(cPacket* msg, ctp_options_t opt);
166         void command_CtpRoutingPacket_clearOption(cPacket* msg, ctp_options_t opt);
167         void command_CtpRoutingPacket_clearOptions(cPacket* msg);
168         am_addr_t command_CtpRoutingPacket_getParent(cPacket* msg);
169         void command_CtpRoutingPacket_setParent(cPacket* msg, am_addr_t addr);
170         uint16_t command_CtpRoutingPacket_getEtx(cPacket* msg);
171         void command_CtpRoutingPacket_setEtx(cPacket* msg, uint8_t etx);
172         // ------------------------------------------------------------------------
173
174         ////////////////////////////////////////////////////////////////////
175         ////////////////////////////////////////////////////////////////////
176
177
178         ////////////////// Custom functions ////////////////////////////////
179         ////////////////////////////////////////////////////////////////////
180
181         // generates an event in the module where they should signal it.
182         void signal_Routing_routeFound() ;
183         void signal_Routing_noRoute() ;
184
185         // AMPacket Interface (just what we need) ---------------------
186         am_addr_t command_AMPacket_source(cMessage* msg);
```

```
187         am_addr_t command_AMPacket_address();
188         // ------------------------------------------------------------------
189
190         // functions that simulate the post command of TinyOs
191         void post_sendBeaconTask() ;
192         void post_updateRouteTask() ;
193
194         ////////////////////////////////////////////////////////////////////
195         ////////////////////////////////////////////////////////////////////
196
197
198  public:
199
200         ////////////////// CtpRoutingEngine functions //////////////////////
201         ////////////////////////////////////////////////////////////////////
202
203         error_t command_StdControl_start();
204         error_t command_StdControl_stop() ;
205         void event_RadioControl_startDone(error_t error) ;
206         void event_RadioControl_stopDone(error_t error) ;
207
208         void event_BeaconSend_sendDone(cMessage* msg, error_t error) ;
209
210         void event_BeaconReceive_receive(cPacket* msg);
211
212         void event_LinkEstimator_evicted(am_addr_t neighbor) ;
213
214         // UnicastNameFreeRouting Interface ----------------------------------
215         bool command_Routing_hasRoute() ;
216         am_addr_t command_Routing_nextHop() ;
217         // ------------------------------------------------------------------
218
219         // CtpInfo Interface (Part 1) ----------------------------------------
220         error_t command_CtpInfo_getParent(am_addr_t*) ;
221         uint8_t command_CtpInfo_getEtx(uint16_t* etx) ;
222         void command_CtpInfo_recomputeRoutes() ;
223         void command_CtpInfo_triggerRouteUpdate() ;
224         void command_CtpInfo_triggerImmediateRouteUpdate() ;
225         void command_CtpInfo_setNeighborCongested(uint16_t n,bool congested) ;
226         bool command_CtpInfo_isNeighborCongested(uint16_t addr) ;
227         // ------------------------------------------------------------------
228
229         // RootControl Interface ---------------------------------------------
230         bool command_RootControl_isRoot();
231         error_t command_RootControl_setRoot() ;
232         error_t command_RootControl_unsetRoot() ;
233         // ------------------------------------------------------------------
234
235         bool event_CompareBit_shouldInsert(cPacket *msg, bool white_bit) ;
236
237         // CtpInfo Interface (Part 2) ----------------------------------------
238         uint8_t command_CtpInfo_numNeighbors() ;
239         uint16_t command_CtpInfo_getNeighborLinkQuality(uint8_t) ;
240         uint16_t command_CtpInfo_getNeighborRouteQuality(uint8_t) ;
241         am_addr_t command_CtpInfo_getNeighborAddr(uint8_t) ;
242         // ------------------------------------------------------------------
243
244         ////////////////////////////////////////////////////////////////////
245         ////////////////////////////////////////////////////////////////////
246
247 };
248 #endif
```