

**Jimma University**  
**Jimma Institute of Technology**  
**Faculty of Computing and Informatics**

**Improving The Performance of Lightweight On demand Ad hoc Distance Vector next generation (LOADng) Protocol on Wireless Sensor Network using Agglomerative Hierarchical Clustering**

**A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science**

**In**  
**Computer Networking**

**By: HAILE HADARO**

**Advisor: FISSEHA BAYU (PhD Cand.)**

**Co-advisor: NEGASA BERHANU (MSc.)**

Jimma, Ethiopia

August, 2021

## Declaration

I the undersigned, hereby declare that this thesis is my original work performed under the supervision of Fisseha Bayu (PhD candidate) and Mr. Negasa Berhanu, has not been presented as a thesis for a MSc degree program in any other university and all sources of materials used for a thesis are fully acknowledged.

**Name:** Haile Hadaro

**Signature:** \_\_\_\_\_

**Date of Submission:** \_\_\_\_\_

This thesis has been submitted for examination with my approval as university advisors.

1. Fisseha Bayu (PhD candidate)  
Main Advisor



Signature

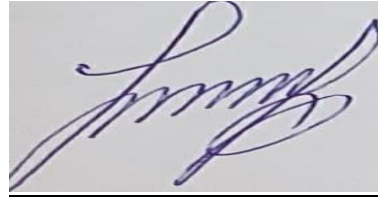
2. Mr. Negasa Berhanu (MSc.)  
Co advisor



Signature

## Examiners Approval

We the Examiners board approve that, this thesis has passed through the defense and review process.



1. Melkamu Deressa (PhD)  
External Examiner

Signature

2. Temesgen Derejaw (MSc.)  
Internal Examiner

---

Signature

3. Sofi Alemu (MSc.)  
Chairperson

---

Signature

4. Kibebewu Ababu (MSc.)  
Head of Department

---

Signature

## **Acknowledgement**

First, I would like to say thank you to God who has given me the strength to endure all the difficulties and challenges of life. Secondly, I would like to thank my advisor Fisseha Bayu (PhD Cand.) for his endless patience in reading, correcting and giving me insightful ideas to improve the work and the constant motivation and support during the course of the work. I truly appreciate and value his guidance and encouragement from the commencement to the end of this thesis. I also thanks my co-advisor Negasa Berhanu for his great supports and advice for successful accomplishment of the thesis.

Finally, I must thank my amazing family and friends. They are the greatest blessing in my life.

## Abstract

*Sensor technologies are becoming increasingly important in acquiring information about nearby environments, and their application in wireless sensor networks is becoming increasingly prevalent. A large number of sensor nodes that are installed in the field to observe certain events defines these networks. Due to the limited battery capacity in sensor nodes, energy efficiency is a major and challenging problem in such power constrained networks. To extend the lifetime of wireless sensor networks as well as conserving its power, some network parameters have been considered, which play an important role in the reduction of power consumption. These parameters are as battery capacity, communication radius, node density and query period. They have a direct impact on the network's lifetime. These parameters have to be chosen in such a way that the network use its energy resources efficiently. To enhance these parameters effectively, different routing mechanisms are used. Routing has great role in maintaining effective path in between nodes. There are several routing strategies or route discovery mechanisms. In this thesis, the energy effective Lightweight On demand Ad hoc Distance Vector next generation (LOADng) routing mechanisms are discovered by using agglomerative hierarchical algorithms. The routing processes starting from route discovery to packet transmission were studied. The control messages used in LOADng protocols such as (RREQ, RREP, RERR, RREP\_Ack), generating to forwarding are briefly examined. Introducing an energy-efficient data forwarding strategy based on node clustering to improve network stability and lifetime while lowering energy consumption is the contribution of this work. To achieve this contribution, a special simulation tool that helps in analyzing the effects of the parameters on sensor network lifetime has been designed and implemented by NS3 and GNU plot is used for plotting the graph. The results are simulated and compared in three different metrics such as packet delivery ratio (PDR), Average end to end delay (EED), Normalized routing overhead.*

# Table of Contents

|  |      |
|--|------|
| Declaration .....  | i    |
| Acknowledgement .....  | iii  |
| Abstract .....   | iv   |
| Table of Figures .....   | viii |
| List of Tables .....   | ix   |
| List of Acronyms/Abbreviations .....                                 | x    |
| Chapter One .....  | 1    |
| Introduction.....  | 1    |
| 1.1. Statement of the Problem .....                                  | 2    |
| 1.2. Objectives.....   | 4    |
| 1.2.1. General objectives .....                                      | 4    |
| 1.2.2. Specific Objective .....                                      | 4    |
| 1.3. Methodology.....  | 4    |
| 1.4. Scope .....   | 6    |
| 1.5. Thesis Organization .....                                       | 6    |
| Chapter Two.....   | 7    |
| Literature Review.....   | 7    |
| 2.1 Architectures of WSN.....  | 7    |
| 2.2 Applications of WSN.....   | 8    |
| 2.2.1 Health/medical Applications .....                              | 9    |
| 2.2.2 Industrial Applications .....                                  | 9    |
| 2.2.3 Military Applications .....                                    | 10   |
| 2.2.4 Flora and Fauna Application.....                               | 11   |
| 2.2.5 Environmental Application.....                                 | 11   |
| 2.2.6 Urban Applications .....                                       | 12   |
| 2.3. Proactive routing protocols .....                               | 14   |
| 2.3.1 Distributed Energy-Efficient Clustering (DEEC) .....           | 14   |
| 2.3.2 Lower Energy Adaptive Cluster Hierarchy (LEACH) .....          | 14   |
| 2.3.3 Energy Efficient Protocol with Static Clustering (EEPSC) ..... | 15   |
| 2.3.4 Directed Diffusion (DD) .....                                  | 15   |
| 2.3.5 Sensor Protocol for Information via Negotiation (SPIN) .....   | 15   |
| 2.3.6 Geographic and Energy Aware routing (GEAR).....                | 15   |

|  |    |
|--|----|
| 2.3.7 Sequential Assignment Routing (SAR).....                                   | 16 |
| 2.3.8 SPEED.....   | 17 |
| 2.4. Reactive routing protocols.....   | 17 |
| 2.4.1 Ad Hoc on Demand Distance Vector Routing (AODV).....                       | 17 |
| 2.4.2 Dynamic Source Routing (DSR).....  | 18 |
| 2.4.3 Energy Efficient Clustering Algorithm for Event-Driven EECED.....          | 18 |
| 2.4.4 Quadrant Based Lower Energy Adaptive Clustering (Q-LEACH).....             | 18 |
| 2.4.5 Power-Efficient Gathering in Sensor Information Systems (PEGASIS).....     | 19 |
| 2.4.6 Minimum Energy Communication Network (MECN).....                           | 19 |
| 2.4.7 Small Minimum Energy Communication Network (SMECN).....                    | 19 |
| 2.4.8 Threshold-sensitive energy efficient sensor network (TEEN).....            | 20 |
| 2.5 Hybrid Protocol.....   | 22 |
| 2.5.1 Geographic Adaptive Fidelity (GAF).....                                    | 22 |
| 2.5.2 Routing Rumor (RR).....  | 22 |
| 2.5.3 Adaptive Threshold-sensitive energy efficient sensor network (APTEEN)..... | 22 |
| Chapter Three.....   | 26 |
| Related Work.....  | 26 |
| Chapter Four.....  | 35 |
| Proposed Work.....   | 35 |
| 4.1. Architectures of Proposed Work.....   | 39 |
| 4.2. Types of Nodes In Clustering.....   | 40 |
| 4.2.1. Homogeneous Clustering.....   | 40 |
| 4.2.2. Heterogeneous Clustering.....   | 41 |
| 4.3 Agglomerative Hierarchical Clustering.....                                   | 41 |
| 4.3.1 Cluster Head selection.....  | 43 |
| 4.3.2 Defining Proximity between clusters.....                                   | 43 |
| Chapter Five.....  | 51 |
| Implementation.....  | 51 |
| 5.1. Route Requests (RREQs).....   | 51 |
| 5.1.1. RREQ Generation.....  | 51 |
| 5.1.2. RREQ Processing.....  | 52 |
| 5.1.3. RREQ Forwarding.....  | 52 |
| 5.1.4. RREQ Transmission.....  | 53 |

|   |    |
|---|----|
| <b>5.2. Route Replies (RREPs)</b> .....                   | 53 |
| <b>5.2.1. RREP Generation</b> .....                       | 53 |
| <b>5.2.2. RREP Processing</b> .....                       | 54 |
| <b>5.2.3. RREP Forwarding</b> .....                       | 55 |
| <b>5.2.4. RREP Transmission</b> .....                     | 55 |
| <b>5.3. Route Errors (RERRs)</b> .....                    | 55 |
| <b>5.3.1. RERR Generation</b> .....                       | 56 |
| <b>5.3.2. RERR Processing</b> .....                       | 56 |
| <b>5.3.3. RERR Forwarding</b> .....                       | 57 |
| <b>5.3.4. RERR Transmission</b> .....                     | 58 |
| <b>5.4. Route Reply Acknowledgments (RREP_ACKs)</b> ..... | 58 |
| <b>5.4.1. RREP_ACK Generation</b> .....                   | 59 |
| <b>5.4.2. RREP_ACK Processing</b> .....                   | 59 |
| <b>5.4.3. RREP_ACK Forwarding</b> .....                   | 59 |
| <b>5.4.4. RREP_ACK Transmission</b> .....                 | 60 |
| <b>Chapter Six</b> .....                                  | 66 |
| <b>Simulation and Result Analysis</b> .....               | 66 |
| <b>6.1 Model for WSN Simulations</b> .....                | 66 |
| <b>6.1.1 Network Model</b> .....                          | 67 |
| <b>6.1.2 Node Model</b> .....                             | 68 |
| <b>6.2 NS3 (Network Simulator-3)</b> .....                | 69 |
| <b>6.3. Result Analysis</b> .....                         | 71 |
| <b>6.3.1. Packet Delivery Ratio (PDR)</b> .....           | 71 |
| <b>6.3.2. Average End to End Delay</b> .....              | 72 |
| <b>6.3.3. Normalized Routing Overhead</b> .....           | 72 |
| <b>Chapter Seven</b> .....                                | 74 |
| <b>Conclusions and Recommendations</b> .....              | 74 |
| <b>7.1 Introduction</b> .....                             | 74 |
| <b>7.2 Conclusion</b> .....                               | 74 |
| <b>7.3 Future Work</b> .....                              | 75 |
| <b>Reference</b> .....                                    | 76 |
| <i>Appendices</i> .....                                   | 80 |



## Table of Figures

|   |    |
|---|----|
| Figure 1 Original LOADng protocol .....                             | 3  |
| Figure 2 Smart route LOADng protocol .....                          | 3  |
| Figure 3 The proposed LOADng protocol .....                         | 4  |
| Figure 4 The overall Methodology used in this work.....             | 5  |
| Figure 5 Physical architecture of Single Sensor in WSN .....        | 8  |
| Figure 6 Applications of WSN.....                                   | 8  |
| Figure 7 The applications of WSN in Health/Medical .....            | 9  |
| Figure 8 The applications of WSN in Industries .....                | 10 |
| Figure 9 The applications of WSN in Military .....                  | 11 |
| Figure 10 The applications of WSN in Flora and Fauna .....          | 11 |
| Figure 11 The applications of WSN in Environmental monitoring ..... | 12 |
| Figure 12 The applications of WSN in Urban planning .....           | 13 |
| Figure 13 Flow Chart of proposed model .....                        | 42 |
| Figure 14 Points to calculate Distance matrix.....                  | 45 |
| Figure 15 Wireless Sensor Network Model .....                       | 68 |
| Figure 16 Tier based Node model .....                               | 69 |
| Figure 17 How C++ and Python codes are integrated in NS3 .....      | 70 |
| Figure 18 Packet Delivery ratio .....                               | 71 |
| Figure 19 Average end-to-end delay .....                            | 72 |
| Figure 20 Route Overhead .....                                      | 73 |

## List of Tables

|  |    |
|--|----|
| <b>Table 1 Comparisons of different protocols</b> .....                        | 24 |
| <b>Table 2 Comparisons of LOADng protocols in different scenarios</b> .....    | 33 |
| <b>Table 3 RREQ and RREP messages fields and RREP_ACK message fields</b> ..... | 37 |
| <b>Table 4 The fields of Routing Set and Pending Ack Set</b> .....             | 38 |
| <b>Table 5 Two-dimensional points to calculate distance Matrix</b> .....       | 46 |
| <b>Table 6 The calculated distance matrix</b> .....                            | 46 |

## List of Acronyms/Abbreviations

| <b>Acronym/Abbreviation</b> | <b>Description</b>   |
|-----------------------------|--|
| <b>AODV</b>                 | Ad Hoc on Demand Distance Vector Routing   |
| <b>BEENISH</b>              | Balanced Energy Efficient Network Integrated Super Heterogeneous   |
| <b>DCFR</b>                 | Dynamic Connectivity Factor Routing  |
| <b>DEEC</b>                 | Distributed Energy Efficient Clustering  |
| <b>DSR</b>                  | Dynamic Source Routing   |
| <b>DVRP</b>                 | Distance Vector Routing Vector   |
| <b>EDAL</b>                 | Energy Efficient Delay Aware Lifetime balancing  |
| <b>EDDEEC</b>               | Enhanced Developed Distributed Energy Efficient Clustering   |
| <b>EELBC</b>                | Energy Efficient Load Balanced Clustering  |
| <b>EEPSC</b>                | Energy Efficient Protocol with Static Clustering   |
| <b>EHGUC-OAPR</b>           | Energy Harvesting Genetic Based Unequal Clustering Optimal Adoptive Performance Routing  |
| <b>ESRPSDC</b>              | Efficient and Secure Routing Protocol for Wireless sensor network through Signal-to-noise-ratio based Dynamic Clustering mechanism |
| <b>LEACH</b>                | Lower Energy Adaptive Clustering Hierarchy   |
| <b>LOADng</b>               | Lightweight On demand Ad hoc Distance vector next generation protocol  |
| <b>LLNs</b>                 | Low Power and Lossy Network  |
| <b>M-ATTEMPT</b>            | Mobility supporting Adoptive Threshold based Thermal aware Energy efficient Multi-Hop Protocol                                     |
| <b>Q-LEACH</b>              | Quadrature based Lower Energy Adaptive Clustering Hierarchy  |
| <b>RREQ</b>                 | Route Request  |
| <b>RREP</b>                 | Route Reply  |
| <b>RERR</b>                 | Route Error  |
| <b>RREP_ACK</b>             | Route Reply Acknowledge  |
| <b>TEEN</b>                 | Threshold-sensitive Energy Efficient sensor Network  |
| <b>WSN</b>                  | Wireless Sensor Network  |

# Chapter One

## Introduction

Wireless Sensor Networks (WSNs) are infrastructureless wireless networks to control physical and environmental situations, like (temperature, pressure, motion, sound, vibration, or pollutions and to collectively allow to pass data through in the given network to a base station or sink [1]. The base station in the network can act as a linkage between users and the network. Anyone should access the required message/information from the network by requesting the queries and gathering required results from the base station. A wireless sensor network can contain so many sensor nodes; may up to hundreds or thousands of sensor nodes. The sensor nodes should communicate each other through radio signals. Sensing and processing devices, radio transceivers, and power components can all be found in wireless sensor nodes. Each node in a wireless sensor network (WSN) is constantly limited in terms of resources. The processing/computing speed, storage capacity, and communication bandwidth of the WSN node are all constrained. The sensor nodes are responsible for managing themselves on appropriate network infrastructure, such as in a multi-hop communication scenario, once they have been put in the proper location. The onboard sensors then start collecting data that is of relevance to you. Wireless sensor devices frequently respond to inquiries made from a control site in order to carry out specific instructions or deliver sensing samples. The sensor nodes can operate in a continuous or event-driven mode. The Local positioning and Global Positioning System (GPS) algorithms can be utilized to gather positioning and location information.

Thousands of resource-constrained sensors are used in Wireless Sensor Networks (WSNs) to monitor their surroundings, collect data, and send it to remote computers for further processing. Despite the fact that WSNs are regarded extremely adaptable ad-hoc networks, network management has been a major difficulty in these networks because to the large deployment size and associated quality concerns including resource management, scalability, and dependability. Topology management is thought to be a possible solution to these issues. Clustering is the most well-known topology management strategy in WSNs, and it involves grouping nodes to manage them and/or performing distributed activities like resource management. Although clustering techniques are most commonly used to reduce energy usage, they can also be used to achieve a variety of quality-driven goals.

## 1.1. Statement of the Problem

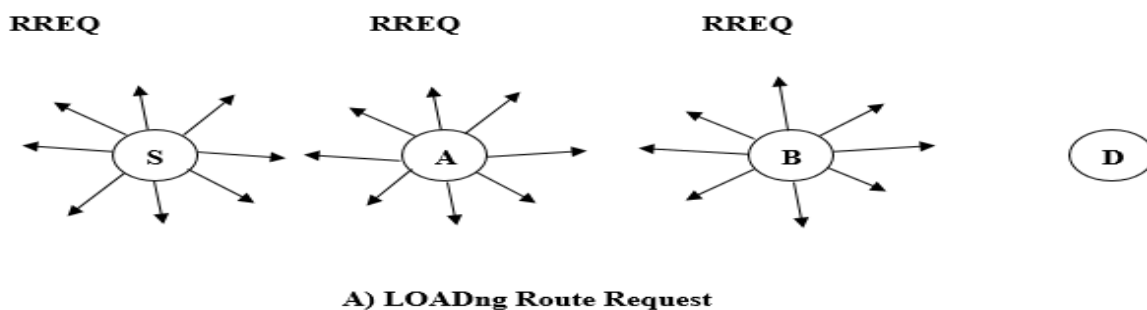
There are many different challenges in wireless sensor network. Of this, Energy is scarcest resource that must be utilized properly because it is impossible to recharge each node, so it must be energy efficient as much as possible. Another problem is the storage problem. While using multi-hop techniques to communicate with other nodes, it needs much energy and storage because each intermediate node can carry the packets to be transferred to its neighbor node. To overcome the routing protocols and select efficient routing path, there are several protocols are used. Of this, this work is on Lightweight On demand Distance Vector next generation. (LOADng). The main problem realized in LOADng protocol is:-

The route discovery delay is one of the great challenge in LOADng Protocol. Outgoing packets are buffered during the discovery phase, which may cause losses on memory-constrained systems. Furthermore, because flooding is inefficient in terms of energy, nodes may experience energy depletion. Another concern of LOADng is control message collisions caused by floods, which can result in unnecessary retransmissions. All the problems raised above is shortlisted as follows.

- ❖ High memory usage because of flooding
- ❖ High energy is required while route discovery because the RREQ is broadcasted among the nodes
- ❖ Using broadcast message to discover route

The solution that realized for the problems raised above is using clustering mechanisms, which saves energy as well as storage.

The overall problems and solutions are described in the figure below. The following figures (a) and (b) shows original work of LOADng protocol request and reply respectively[2].

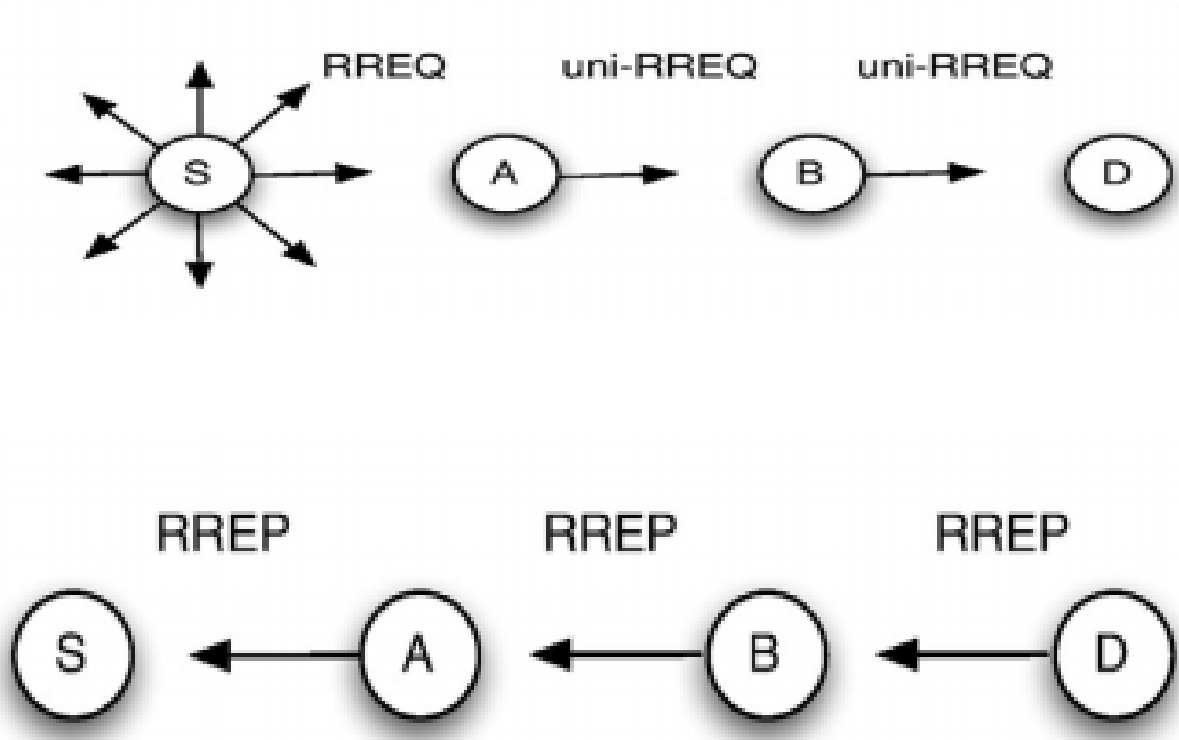




**B) LOADng Route Reply**

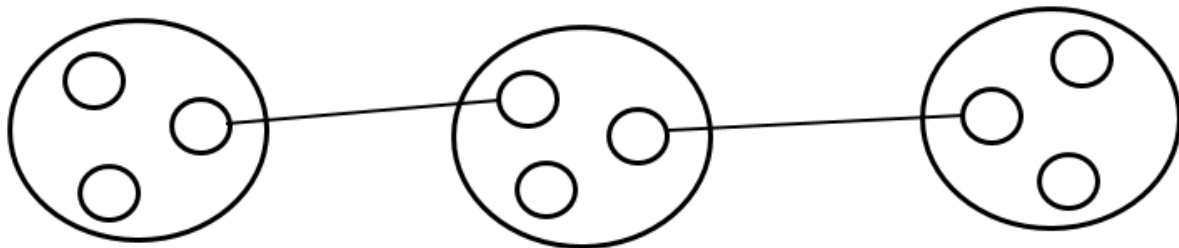
*Figure 1 Original LOADng protocol*

The following figures illustrates the current LOADng protocols



*Figure 2 Smart route LOADng protocol*

The following graph shows the proposed work



*Figure 3 The proposed LOADng protocol*

## **1.2. Objectives**

The general and specific objectives of the thesis are described below.

### **1.2.1. General objectives**

The general objectives of this work is improving the performance of Lightweight On demand Ad hoc Distance Vector next generation (LOADng) protocol on Wireless Sensor Network by clustering of nodes.

### **1.2.2. Specific Objective**

To achieve the General objectives, the following specific tasks should be performed

- ❖ Analyzing the existing system
- ❖ Route discovery: - while making the route the sender sends the route request message only to the cluster heads because the Cluster Head (CH) has all the information about its cluster member
- ❖ Route caching: - stores the routing information for next use temporarily
- ❖ Modeling the proposed system
- ❖ Simulating the proposed system
- ❖ Test and evaluate the system

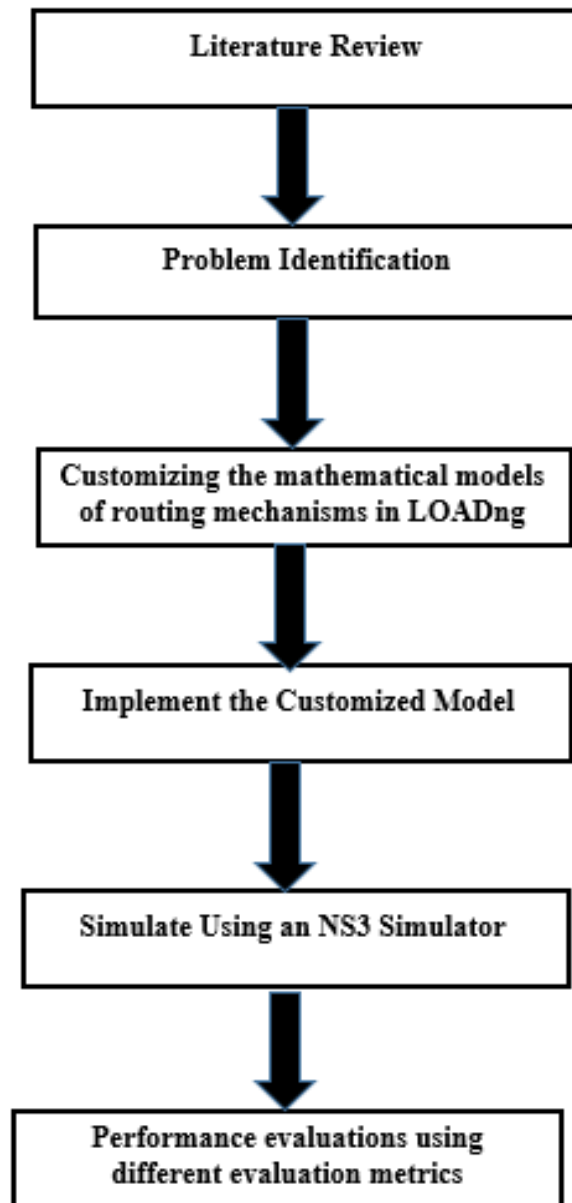
## **1.3. Methodology**

Several methodologies are used for the success accomplishment of this work. The following are some of them.

**Document review:** - different documents related to wireless sensor networks and its routing algorithms has been reviewed. There are many documents exist in different servers including IEEE, which has published in different years. In addition, peer review is another method tried to improve the skills of self-review of any documents.

**Design Methodology:** different methodologies used in computing, like Experimental method, Simulation method and Theoretical method. However, in this work simulation method could be used. Simulation is employed because it allows researchers to explore systems or regimes outside of the experimental domain, as well as systems that are under development or innovation. Many

different simulation tools are used in performance evaluation of networks. The tools being used here are NS3, GNU plot.



*Figure 4 The overall Methodology used in this work*



## **1.4. Scope**

There are several different types of protocols in wireless sensor networks, which different researchers tried to find out at different times. Some researchers done at energy minimizations, whereas others on efficient utilization of storage for WSN. This work focused on LOADng protocol only limited to the Energy minimizations.

## **1.5. Thesis Organization**

Chapter one contains the overall introductory sections such as introduction, statement of the problems, objectives of the thesis, methodologies, literature review, scope of the thesis, and thesis organization. Chapter Two contains the detailed explanations of different types of routing protocols WSN, such as proactive, reactive and hybrid as well as advantages and drawbacks of those protocols on each type. Chapter Three contains detailed explanations about related works on LOADng protocols and advantages and drawbacks of routing parameters used in different scenarios. Chapter Four explains details of design and architectures of the proposed work. Chapter Five is about the implementations details and different algorithms used in proposed work. Chapter Six contains simulation tools and result analysis as well as the last and seventh chapter contains Conclusion and recommendation.

## Chapter Two

### Literature Review

Wireless sensor networks (WSNs) allow new applications to be connected. Because of several constraints in WSN, it requires non-traditional protocol design paradigms. A proper balance between communication and data/signal processing capacities must be due to the necessity of low system complexity along with low energy consumption.

#### 2.1 Architectures of WSN

Architecture of WSN may include the organization of simple node, sink/base station node and other nodes on global perspectives of the entire network. The autonomy and adaptability of the architecture must be taken into account during the design process. Autonomy states that sensors can be put in an unattended or physically inaccessible area, and as a result, they must operate with minimal assistance from base stations or human administrators. When there is no major change in sensor readings, sensor networks should promote adaptability in the way they operate by responding to the environmental changes that sensors monitor. For example, sensors may lower their duty cycles to reduce power consumption in the sensors. Sensor networks are used in a variety of applications and some of which need mobility. Sensor network communication is entirely dependent on the application and method of data collection; it mostly consists of data transmitted to nodes and data captured by nodes about the environment. Sensing, memory, CPU, transceiver (transmitter and receiver), and battery are the five main components of a sensor node [3]. The figure below shows the architectures of node.

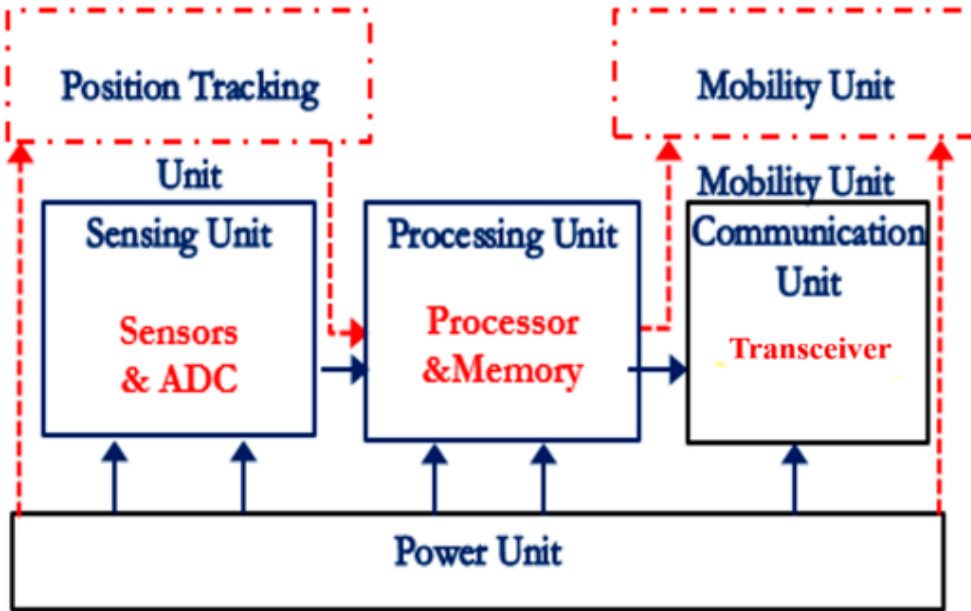


Figure 5 Physical architecture of Single Sensor in WSN

## 2.2 Applications of WSN

There are different areas in which WSN should be applied. Currently WSN can be applied either in mature use or still in infant stages of development. Of different applications, few of them are parts of this study. These are – Health/medical usage, Industrial, Military, Flora and Fauna, Environmental monitoring and urban control[4].



Figure 6 Applications of WSN

### 2.2.1 Health/medical Applications

Diagnostics, investigatory, and administration of drugs are some of the medical/health benefits of WSNs. WSN were applied on supporting interfaces for the incapacitated, integrated patient monitoring and management, tele monitoring of human physiological information, and tracking and monitoring medical practitioners or patients inside the medical facility. WSNs in the health domain use advanced medical sensors to monitor patients in a healthcare institution, such as a hospital or at home, as well as providing real-time monitoring of patient vitals via wearable devices. The main categories of WSN on health applications are patient wearable monitoring, home assistance systems, and hospital patient monitoring, are depicted, along with the most regularly used sensors.

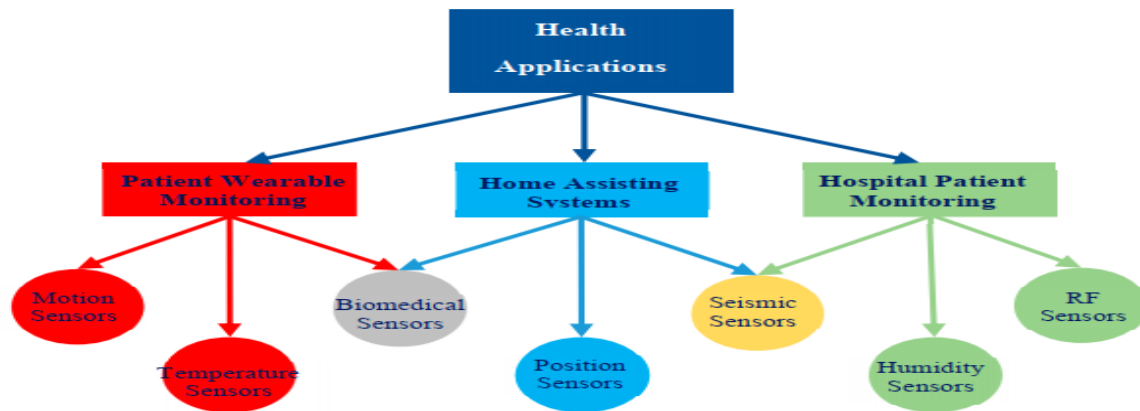
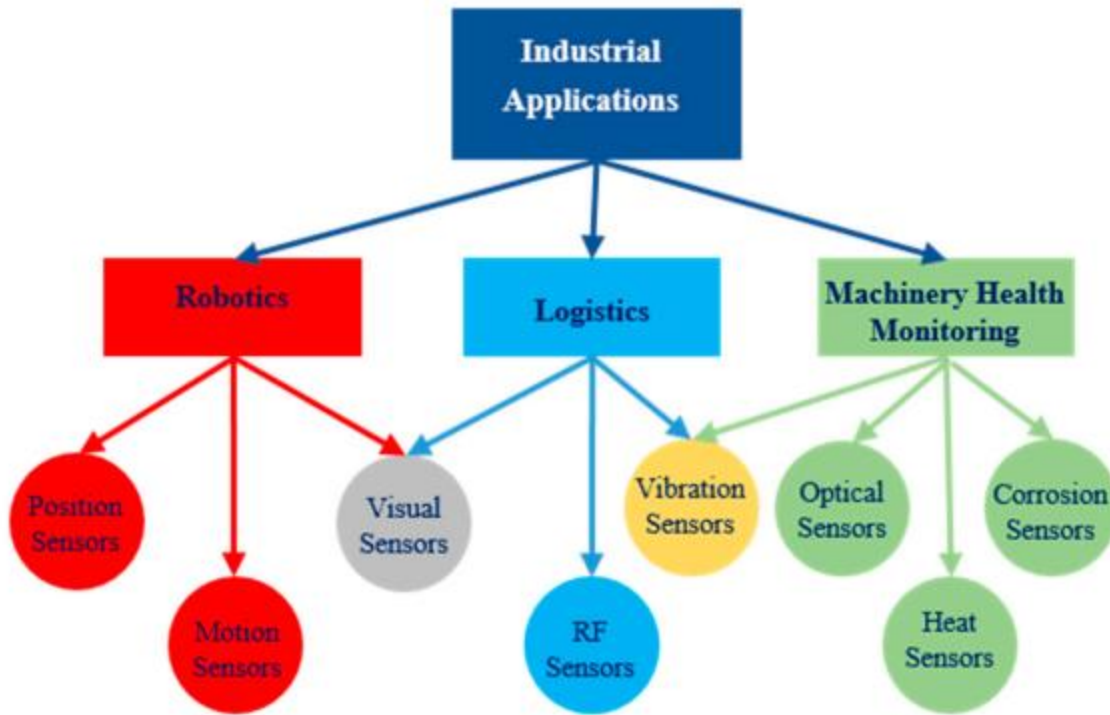


Figure 7 The applications of WSN in Health/Medical

### 2.2.2 Industrial Applications

WSNs can be used in a variety of industrial settings to tackle a variety of challenges. WSNs have been proposed for Technological Condition-based Maintenance since they could result in significant cost savings/investments as well as new features. The installation of suitable sensors in wired classes is frequently limited by the amount of wiring involved. Logistics, robotics, and machinery health monitoring are the three main subcategories of industrial WSN applications. The figure below shows some of the categories of industrial applications of the WSN.



*Figure 8 The applications of WSN in Industries*

### **2.2.3 Military Applications**

Military was not only the first field of human activity to adopt WSNs, but it is also thought to be the catalyst for sensor network development. Smart Dust is a good illustration of these early research efforts, which took place in the late 1990s in order to construct sensor nodes that, despite their small size, could perform spying operations. WSNs are likely to be an important component of military intelligence, facilities, control, communications, computing, frontline surveillance, investigation, and targeting systems.

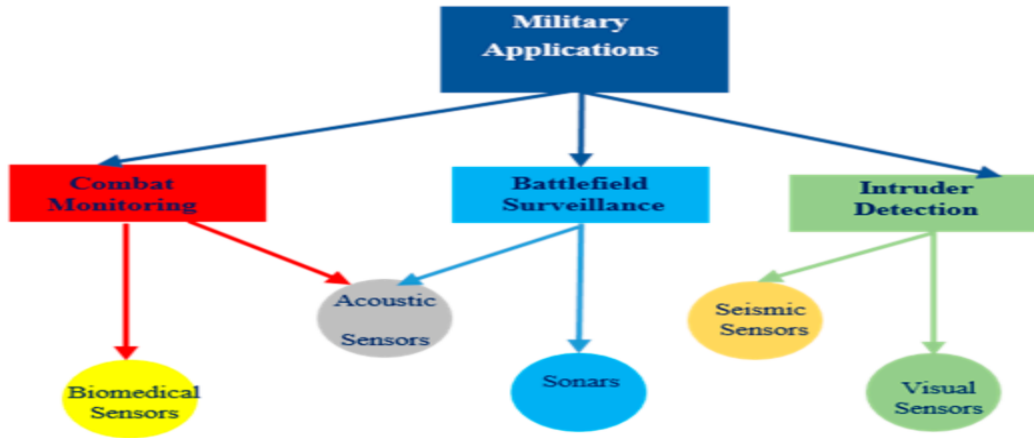


Figure 9 The applications of WSN in Military

### 2.2.4 Flora and Fauna Application

Every country requires both flora and fauna domains. Greenhouse monitoring, crop monitoring, and livestock farming are the three main subcategories of WSN in flora and fauna applications. The following figure describes various types of flora and fauna applications in WSNs.

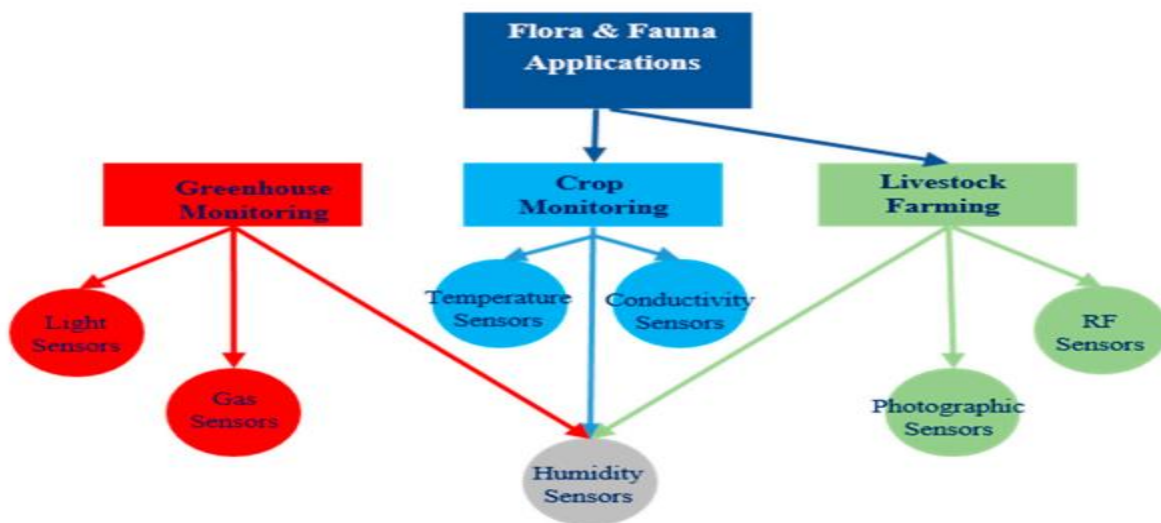


Figure 10 The applications of WSN in Flora and Fauna

### 2.2.5 Environmental Application

The use of WSNs can improve environmental applications that require continuous monitoring of ambient conditions in hostile and remote locations. Environmental Sensor Networks has grown to encompass a wide range of WSN applications in environmental and earth science research. This includes oceans, seas, glaciers, the atmosphere, volcanoes, and forests. The main categories of

environmental applications of WSNs are - water monitoring, air monitoring and emergency alerting, are depicted along with the types of sensors that are typically used in them. The figure below shows the categories of WSNs in environmental applications.

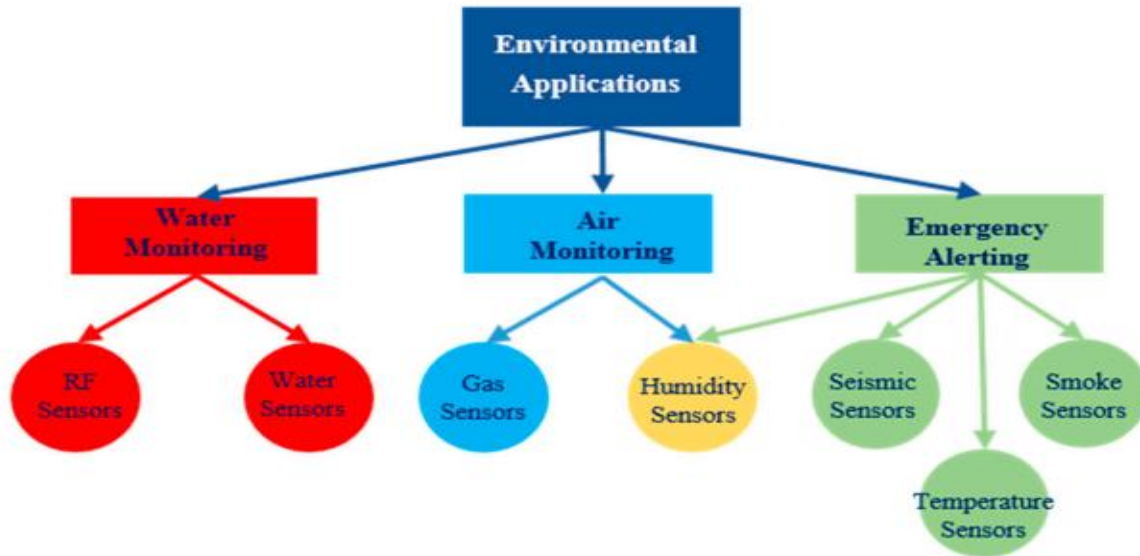


Figure 11 The applications of WSN in Environmental monitoring

### 2.2.6 Urban Applications

WSNs can be used to monitor the movement of various structural projects such as buildings and other infrastructural projects such as flyovers, bridges, roads, embankments, tunnels. The infrastructural projects, allowing manufacturing/engineering practices to monitor possessions remotely without having to visit the sites, saving money that would have been spent on site visits. WSNs' wide range of sensing skills also allows them to gather unprecedented amounts of data on a target location, whether it's a room, a building, or the outdoors. Smart homes, smart cities, transportation systems, and structural health monitoring are among the most prominent WSN applications in the urban area.

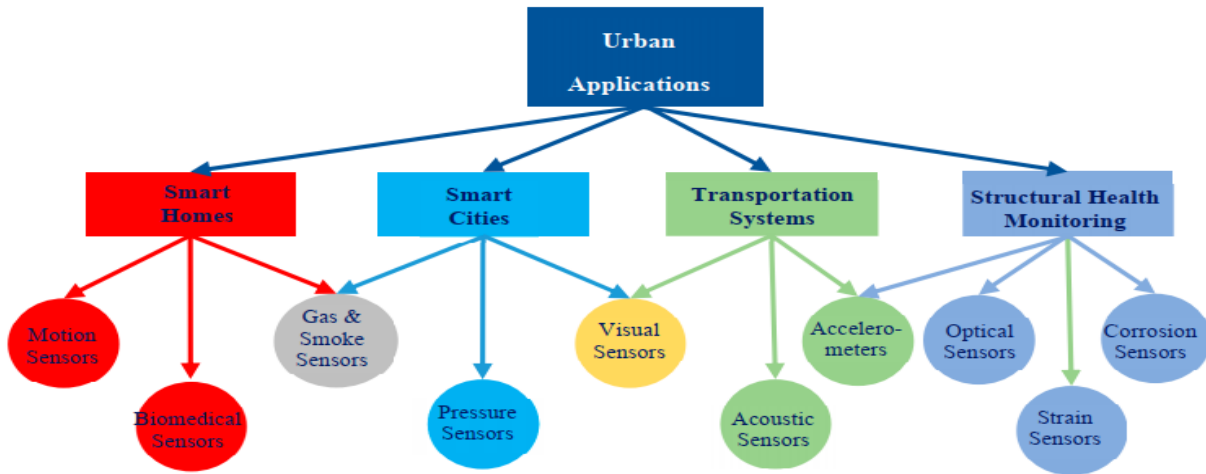


Figure 12 The applications of WSN in Urban planning

Multi-hop data transmission is used to broadcast data packets due to a number of issues, including limited transmission and computation power and a high density of sensor nodes in some areas. As a result, routing in wireless sensor networks has been a topic of attenuation in recent research over the last few years. Since the sensor nodes use non-rechargeable batteries, energy efficient use of their sources, as well as efficient network routing is a significant research concern [5].

A group of several number of sensors make up a wireless sensor network (WSN) that are strategically placed in an area to track and communicate with one another over a wireless medium. Due to issues such as routing protocols, energy usage, security, and data aggregation, this advanced technology is still limited [6].

In the area of wireless sensor network, several researches were hold. The focuses of the researchers are on energy minimization because of its nodes has low battery and low computing powers. After successful configuration of wireless sensor network, the nodes might need to communicate each other. To communicate and transmit the data among the network, there is a manner, in which how the communication will occur, which is protocol. There are about three different manners or approaches of communication or route discovery. The reactive, proactive and hybrid. Reactive approaches of protocols are used when the data is needed to transfer or the route is needed to pass. Under the reactive approaches there are different types of protocols and each of them have its own advantages and drawback. Proactive approaches are used when the routes have been maintained standby always, even if it is not the time to transfer or transmit the data and update the routes



periodically. Hybrid are formed using both protocols at the same time and its details are described below.

### **2.3. Proactive routing protocols**

It is also known as a table-driven technique because it follows a set route through life. Each node in proactive protocols maintains routing information for the entire network in the individual routing table. The routing table information is correct and up-to-date, and the update is permanent since all network nodes send control messages to update their routing tables on a regular basis. Link-state routing algorithms that constantly flood their neighbor's link information should be used in proactive routing protocols. The major drawback of proactive routing protocol is that all the nodes on the network still require keeping their routing table up to date and this cause overwhelming in control messages.

#### **2.3.1 Distributed Energy-Efficient Clustering (DEEC)**

The distributed energy-efficient clustering algorithm is a cluster-based algorithm that chooses cluster heads based on the probability of the residual energy ratio of each node in the network after a certain round and average total network energy in order to adapt to multi-level heterogeneous networks. In this algorithm, the nodes with the most energy have more chances to become the cluster head. The network's lifespan is extended.

#### **2.3.2 Lower Energy Adaptive Cluster Hierarchy (LEACH)**

The Low-Energy Adaptive Clustering Hierarchy Protocol (LEACH), which is self-adaptive and self-organized and uses a TDMA/CDMA MAC to reduce inter-cluster and intra-cluster collisions, was the first hierarchical or clustering-based protocol in which cluster heads are randomly chosen. Data processing, on the other hand, is centralized and done on a daily basis. The LEACH Protocol distributes nodes with the ability to collect and process data in the sector at random. The LEACH protocol is divided into several rounds, each of which is divided into two stages (setup phase and steady state phase).

LEACH is a fully distributed approach that is both efficient and fast, but it has several drawbacks. For example, CH selection in any round is random and does not take into account node energy level, which can lead to drainage of a specific node[7], and it assumes that sensor networks are homogeneous networks, resulting in poor performance in heterogeneous settings.

### **2.3.3 Energy Efficient Protocol with Static Clustering (EEPSC)**

The Energy Efficient Protocol with Static Clustering is hierarchical and uses static clustering routing. It partitions the entire network into a few static clusters and attempts to spread the load by selecting high-energy sensor nodes as CHs to avoid the overhead of dynamic clustering.

### **2.3.4 Directed Diffusion (DD)**

In the design of communication protocols for Wireless Sensor Network architecture, energy efficiency is a major consideration. In random and mesh topology networks, Direct Diffusion allows sink and source nodes to communicate. This routing protocol takes a data-centric approach, with intermediate nodes aggregating data before sending it to a sink node. Researchers have successfully used DD in conjunction with Passive Clustering (PC) to increase energy efficiency. The network is divided into small clusters in this approach, and the DD protocol is implemented at the application layer, resulting in lower energy costs, reduced delay, and increased data delivery rate. The network's infrastructure is made up of three parts: a Source that starts data transmission, intermediate nodes that detect and track events in the area, and a sink where data is sent to its final destination. DD is a data-centric routing protocol that selects all routes based on application-level information. Furthermore, network nodes send and receive messages as well as create attribute-value pairs. Interests, gradients, cache memory, and reinforcement path are the four core features of DD [8].

### **2.3.5 Sensor Protocol for Information via Negotiation (SPIN)**

SPIN is a negotiation-based protocol that was one of the first attempts to develop a data-centric routing mechanism. The SPIN concept is to label data using high-level descriptors or meta-data. The data advertising mechanism, which is a key feature of SPIN, is used to exchange meta-data among sensors prior to transmission. When a node receives new data, it broadcasts to its neighbors, and interested neighbors i.e. those who do not have the data, send a request message to retrieve it. Three types of messages are identified in SPIN to exchange data between nodes. These are: The ADV message allows a sensor to advertise a certain meta-data, while the REQ message requests the appropriate data, and the DATA message contains the actual data [9].

### **2.3.6 Geographic and Energy Aware routing (GEAR)**

The sensor networks will be made up of a large number of densely dispersed sensors and actuators. The fact that the nodes in these networks are untethered and unsupervised is a crucial feature. As

a result, energy efficiency is a key consideration in the design of these networks. Because of sensor network queries are frequently geographical, the design and evaluation of an energy-efficient routing algorithm that propagates a query to the appropriate geographical region without flooding is necessary. The Geographic and Energy Aware Routing (GEAR) algorithm routes the packets to the target region using energy aware neighbor selection and then disseminates it within the destination region using Recursive Geographic Forwarding or Restricted Flooding. The GEAR protocol was tested using simulation and found to have a significantly longer network lifetime than non-energy aware geographic routing algorithms, especially for non-uniform traffic distribution [10].

### **2.3.7 Sequential Assignment Routing (SAR)**

It is a collection of algorithms for sensor network organization and mobility management. The sequential assignment routing (SAR) technique generates numerous trees, each with a one-hop neighbor to the sink as its root. Each tree branches outward from the sink, avoiding nodes with low throughput or a long delay. Most nodes belong to numerous trees at the end of the procedure. This lets a node to select one of several paths to send its message to the sink. The SAR algorithm selects a path with high predicted energy resources, with accommodations provided to accept packets of various priorities. To handle prioritized packets, a weighted QoS metric is utilized, which is calculated as a product of priority level and delay.

The routing ensures that the weighted QoS measure remains consistent. As a result, higher priority packets use lower delay paths, whereas lower priority packets must utilize higher delay paths. Over the network's lifespan, SAR minimizes the average weighted QoS metric. After some transmissions, a metric update is triggered by the sink to reflect changes in available energy resource. Sequential Assignment Routing (SAR) is the only protocol for sensor networks that considers QoS when making routing decisions. The SAR protocol constructs trees routed from one-hop neighbors of the sink, taking into account the QoS metric, energy resource on each link, and packet priority level. Multiple paths from the sink to the sensors can be established utilizing the trees that have been constructed. The energy resources and QoS on each path are used to choose one of these paths.

### **2.3.8 SPEED**

SPEED is a routing protocol for large-scale sensor networks that allows for soft real-time communications. End-to-end soft real-time is accomplished by using feedback management and non-deterministic regional forwarding to establish a desired distribution speed across the network. Each node in the network must be georeferenced in use geographic location for packet forwarding. The protocol provides three types of real-time communication services:

*Real-time unicast:* a packet is sent to a particular network node known by its geographic location and global network address;

*Real-time multicast:* This service helps you to send a data packet to all nodes within a destination area defined by its radius and center location.

*Real-time any cast:* a packet is sent to at least one node inside an area specified by its center and radius [11].

## **2.4. Reactive routing protocols**

This routing protocol, also known as demand approach, makes route decisions based on current network conditions, allowing the route path to be dynamically altered. When a source wants to send packets to a specific sink in a reactive routing protocol, it must first find the route to the sink. Mechanisms for route discovery Unless the sink is unavailable or the route is no longer required, the route remains valid. Unlike other approaches, there is no requirement for all nodes to keep routing information up to present-day

### **2.4.1 Ad Hoc on Demand Distance Vector Routing (AODV)**

AODV is a reactive protocol, when information has to be sent, reactive approaches find the best way. It may be used for both unicast and multicast routing. It maintains the courses as long as the source needs it. It uses a grouping number to indicate that the course is new. By leveraging succession number it, ensure the course is without circle. When a node decides to send data to another node and the route is not available, it broadcasts an RREQ (Route Request) message.

Nodes receiving this RREQ message check whether they are the destination node to which the source node needs to send data. If they are, they will respond with RREP (route reply). By storing the source IP address and broadcast ID, nodes keep track of course demand. If they get a related RREQ in the future, they will discard it because they have already planned it. Nodes that pass the

RREQ message will stamp a regressive pointer to the node that sent it. Nodes label forward pointers to the destination node as RREP engenders back to the source node. If a source node receives an RREP with a higher grouping number or the same arrangement number but a lower bounce check, the source will switch to this new route. Connections will naturally break and be erased from the transitional node steering table when the source node stops sending information in the course.

If a connection failure occurs while the route is dynamic, the source node will receive an RERR (Route Error) indicating that the goal is inaccessible, and if the source requires the route, the course will be revealed again.

#### **2.4.2 Dynamic Source Routing (DSR)**

DSR is on demand or responsive routing protocol. It utilizes the source steering, aggregating the location of the relative multitude of nodes between source and objective during route discovery. It comprises of two stages, route discovery and route maintenance. At the point when a node needs to send an information bundle to another node, it starts a route discovery. Node floods route request(RREQ) which contains sender's location, objective's location and demand ID. The node getting RREQ checks either it is the objective or not or it checks route is accessible with it for destination node. In route support stage on the off chance that node find connect disappointment, it sends course mistake (RRER) message to the source node, nodes accepting the RRER message refreshes their reserve.

#### **2.4.3 Energy Efficient Clustering Algorithm for Event-Driven EECED**

The protocol [12] Energy Efficient Clustering Algorithm for Event-Driven Wireless Sensor Networks improves lifetime by balancing node energy consumption. The base station is in the middle of the area and has ample memory to handle messages. Only after an occurrence happens when data is transmitted from nodes to CHs.

#### **2.4.4 Quadrant Based Lower Energy Adaptive Clustering (Q-LEACH)**

In [13] an energy efficient algorithm based on quadrant based routing protocol called as Q-LEACH introduced which divides the whole network into quadrants. Those nodes, which are nearby the sink node, will broadcast the message. Since it uses a reactive routing mechanism and hence all nodes maintain the destination node information before finding the path to destination or sink node.

It contains advantage of both location and hierarchical based routing protocols. To find the path to the destination, this protocol uses a route request packet.

#### **2.4.5 Power-Efficient Gathering in Sensor Information Systems (PEGASIS)**

PEGASIS is energy efficient because data is only forwarded to the next node on the chain, which is likely the shortest distance due to the greedy approach used to build the chain. Furthermore, each node receives data from no more than two other nodes, which saves energy when receiving packets. Furthermore, rather than sending all received data, each node aggregates it with its sensed data. Because of the transfer of aggregated data, less transmission energy is used. However, owing to the selfish approach to chain forming in PEGASIS, on the chain, there can only be a few long routes.

As a result, only a few nodes will be required to transmit over long distances, and as a result, they will die out faster. Furthermore, in PEGASIS, as each node takes turns transmitting data to the sink, the nodes farther away from the sink will die out faster due to the long distance traveled. In PEGASIS, there is also data flow in the opposite direction. Data from nodes near the sink may be transmitted backwards to a different leader node further away from the sink, and then the leader node transmits it to the sink. All of these problems would lead to network partitioning and a shorter network lifespan [14].

#### **2.4.6 Minimum Energy Communication Network (MECN)**

This protocol is based on location. The basic principle behind MECN is to create a sub-network in which number of nodes is fewer and less power is used to relay a data between the nodes. For a sensor network, it creates a low-energy subnetwork. Location information is based on GPS. The MECN routing scheme is based on the idea of a relay zone, which is made up of relay nodes. The area around the nodes is built into a relay zone. Relay nodes are intermediary nodes present between the source and the destination node. MECN uses these relay nodes to reach the destination. An area is chosen where there are fewer nodes and the amount of energy available for transmission is minimal. MECN works in two steps: [15] Sparse Graph Construction and Optimal links search.

#### **2.4.7 Small Minimum Energy Communication Network (SMECN)**

SMECN is essentially a tweaked version of the MECN with less complexity and being more power efficient. The algorithm takes into account the struggles between nodes and developing smaller

sized sub- graphs. The main goal of SMCEN is to find the enclosure graph as result to find the shortest energy paths. In [16] contrast to the energy required to transmit data to neighboring nodes in MECN, the energy required to transfer data to neighboring nodes in SMECN is lower.

#### **2.4.8 Threshold-sensitive energy efficient sensor network (TEEN)**

Threshold sensitive Energy Efficient Sensor Network protocol [17] it follows a hierarchical algorithm with the use of a data-centric mechanism. In TEEN, the cluster head broadcasts two thresholds to its members for sensed operations, hard and soft thresholds. TEEN performs badly in programs that require quarterly feedback, and if the thresholds are not met, the customer will not receive any results at all. TEEN is a reactive clustering routing protocol, which is improved by LEACH. The cluster head (CH) of each cluster collects data from its cluster members. The CHs combine and process data before sending it to the BS or a higher-level CH. Clustering routing protocols provide the advantage of requiring all nodes to only transmit data to their CH, and only the CHs to aggregate data. It conserves energy. [18] To ensure that energy consumption is distributed uniformly, each node takes turns as the CH. The random selection function is used in the CH voting. After clusters are created, CHs assign cluster members a time slot during which they can transmit their data.

In TEEN routing protocol, unlike LEACH, CHs broadcast hard and soft thresholds to their participants to monitor the amount of data transmitted. The nodes will only transmit sensed data to CH in the current round if the current value of the sensed attribute is greater than the value of hard threshold, according to the hard threshold defined based on the range of interest value of users. The sensed attribute is then saved as part of the sensed value. This is a vector that exists inside the system. Nodes continually detect the attribute; it is only transmitted when the next value varies from the sensed value by an amount equal to or greater than the soft threshold. By ignoring small changes in the perceived attribute, the soft threshold minimizes the frequency of data transmission. The soft threshold value can be adjusted to meet the needs of the users. Setting a lower soft threshold improves network accuracy at the expense of higher energy usage. As a result, users must control the trade-off between energy efficiency and accuracy by adjusting the size of the soft threshold.

Recently, in addition to the above-mentioned protocols and algorithms there has been many researches focused on investigating the energy-efficient routing protocols. In [19], The EDAL

protocol, that can minimize overhead computing, has been proposed. That protocol has been found proposed to achieve a substantial reduction in overall traffic costs for collecting sensor readings under loose delay limits. In [20], a protocol for M-ATTEMPT has been suggested. In contrast to multi-hop communication, it is suggested that the this protocol has lower energy consumption and is more efficient.

In [21], with an effective nodes encoding scheme and a multi-objective fitness feature, the routing protocol was developed. This protocol has been suggested as it perform better in terms of network lifespan, energy usage and data packet transmission to the base station. In [22], a new energy-efficient routing protocol has been suggested using the success rate of messages. It is found that, in terms of communication reliability and energy consumption, the protocol will outperform existing schemes. In [23], A DVRP protocol has been proposed in which the transmission of packets is based on the angle of the flooding zone from the sender nodes towards the surface sink. It could work better that in terms of: end-to-end delays, energy usage and data delivery ratios.

In [24], The DCFR protocol has been proposed to implement a distributed run time recovery of sensor nodes due to the sudden failure of the CHs. The DCFR protocol has been found to be energy efficient and fault-tolerant. In [25], A cluster based routing protocol with heterogeneous node distribution in wireless sensor networks has been suggested. This protocol is found to be able to balance the energy usage between nodes and greatly increase the network lifetime. In [26], The BEENISH protocol, that assumes the WSN comprises four node energy levels, was proposed. It is noticed that longer stability, lifespan and more efficient messages can be accomplished by the proposed protocol.

In [27], The EDDEEC protocol was proposed to select CH on the basis of a dynamically evolving probability. This protocol is found to be able to offer longer life, stability duration and more efficient messages to BS than DEEC. In [28], The ESRPSDC routing protocol has been proposed to implement error recovery to prevent end-to-end error recovery. This protocol has been found to be able to substantially increase the energy consumption and reception rate of packets. In [29], An EELBC algorithm has been proposed that tackles energy conservation as well as load balancing. This algorithm has been found to be able to perform better in terms of load balancing, energy consumption and execution time. In [30], The EHGUC-OAPR algorithm has been proposed to combine the genetic-based unequal clustering algorithm for energy harvesting and the



optimal adaptive efficiency routing algorithm. This algorithm has been found to significantly boost the energy balance and data distribution ratio of the network.

## **2.5 Hybrid Protocol**

This protocol uses clustering techniques to make the network stable and scalable, and it combines proactive and reactive technologies. The network cloud is divided into many clusters, each of which is dynamically maintained as nodes join or leave a cluster [31].

### **2.5.1 Geographic Adaptive Fidelity (GAF)**

GAF is a routing protocol that is both location and energy sensitive. It was originally designed for ad hoc networks, but WSNs can now use it as well. It is based on the assumption that in terms of routing, all adjacent nodes are equal [32]. The whole network is separated into a virtual grid in GAF. The grid size is determined by the fact that any node in one grid will connect with any other node in another grid. Each node in the network is allocated a cell, and each cell has only one active node. The highest remaining energy of a node is used to select an active node. Each node uses GPS-based location information to associate itself with the grid.

### **2.5.2 Routing Rumor (RR)**

Rumor Routing is a variation of directed diffusion. It is also a hybrid data-centric protocol. It is designed to be used in situations where the nodes are not sure of their position. Instead of overwhelming the whole network with a message, the Rumor Routing (RR) protocol directs the query to certain nodes that have knowledge about the incident. Rumor Routing is a kind of middle ground between query and case flooding. The RR protocol uses long-lived agents to disseminate event information across the network. When a node notices an event, it records it in its table and creates an agent. Agents traverse through the network for disseminating of information about events to different nodes [33]. When a query reaches a node, which has the inquired event in its event table, the query is routed towards the event by referring to the event table. In this way, only one path between source and destination is there in the RR protocol. This avoids the overhead of flooding the entire network with queries.

### **2.5.3 Adaptive Threshold-sensitive energy efficient sensor network (APTEEN)**

APTEEN is a common wireless sensor network routing technique, but when clustering, cluster heads are chosen at random, making it easy to select nodes with low residual energy as cluster heads, resulting in network holes. At the same time, due to the enormous volume of data being

forwarded, the cluster head near the sink node is overburdened in multi hop transmission. The advantages of the clustering routing protocol include easy topology management, great energy efficiency, and benefits for data fusion and transmission processing. A popular clustering routing protocol, Adaptive Threshold-sensitive Energy Efficient Sensor Network Protocol (APTEEN), defines two thresholds, hard threshold and soft threshold, which determine the range of observed values and the amount of change between the two measured values before and after. The usage of two criteria can help to limit data transfers that are both unneeded and recurrent.

APTEEN has a high routing efficiency since it uses clustering and data fusion. It can collect data on a regular basis and respond promptly in an emergency. APTEEN is often used in the environment monitoring, and monitor forest fires. The APTEEN routing protocol is used in [34] to monitor the temperature, humidity, and moisture of the land, and sensor nodes are used to monitor these indications on a regular basis. The LEACH routing protocol enhances the APTEEN routing protocol. LEACH's original clustering mode and cluster head selection are used in the protocol. During the cluster formation step, all nodes are separated into clusters, and each node generates a number between  $[0, 1]$  at random; if the number is less than  $T(n)$ , the node is chosen as the cluster head. When picking the cluster heads in the cluster, however, the APTEEN protocol ignores the energy and position of the selected nodes.

Choosing the cluster heads from the nodes with the lowest energy, leading the nodes to die prematurely, causing energy holes, and directly harming the network stability. Because of the aforementioned flaws, the APTEEN has been greatly improved both at home and abroad. In APTEEN, several protocols optimize the cluster heads selection approach, primarily to limit cluster heads selection in terms of energy and location, in order to select the best cluster heads to increase overall network performance and reduce overall network energy consumption.

It may be contended that the talk about of reactive versus proactive protocols has been broadly inquired about in the context of traditional ad hoc networks, and it could be a well-known reality that reactive protocols perform much better in systems with low traffic. We in any case contend that LLNs are distinctive from conventional ad hoc systems not only due to hub capacity, but too within the nature of traffic generation and directing prerequisites, making it worth to revisit the talk about within the setting of LLNs.

The overall Comparison of protocols realized above described below in table.

*Table 1 Comparisons of different protocols*

| <b>Protocol</b> | <b>Energy Consumption</b> | <b>Scalability</b> | <b>Mobility</b>    | <b>Route Selection</b> | <b>Classification</b> | <b>Data aggregation</b> |
|-----------------|---------------------------|--------------------|--------------------|------------------------|-----------------------|-------------------------|
| DD              | Limited                   | Limited            | Limited            | Proactive              | Data Centric          | Yes                     |
| RR              | Low                       | Good               | Base Station Fixed | Hybrid                 | Data Centric          | Yes                     |
| SPIN            | Limited                   | Limited            | Possible           | Proactive              | Data Centric          | Yes                     |
| COUGAR          | Limited                   | Limited            | Base Station Fixed | Reactive               | Data Centric          | Yes                     |
| AQUIRE          | Low                       | Limited            | Limited            | Reactive               | Data Centric          | Yes                     |
| LEACH           | High                      | Good               | Base Station Fixed | Proactive              | Hierarchical          | Yes                     |
| PEGASIS         | Maximum                   | Good               | Base Station Fixed | Reactive               | Hierarchical          | No                      |
| MECN            | Maximum                   | No                 | Base Station Fixed | Reactive               | Location Based        | No                      |
| SMECN           | Maximum                   | No                 | Base Station Fixed | Reactive               | Location Based        | No                      |
| TEEN            | High                      | Good               | Base Station Fixed | Reactive               | Hierarchical          | Yes                     |

|        |         |         |                          |           |                   |     |
|--------|---------|---------|--------------------------|-----------|-------------------|-----|
| APTEEN | High    | Good    | Base<br>Station<br>Fixed | Hybrid    | Hierarchical      | Yes |
| GAF    | Limited | Good    | Possible                 | Hybrid    | Location<br>Based | No  |
| GEAR   | Limited | Limited | Limited                  | Proactive | Location<br>Based | No  |
| SAR    | High    | Limited | Limited                  | Proactive | QOS-based         | Yes |
| SPEED  | Low     | Limited | Base<br>Station<br>Fixed | Proactive | QOS-based         | No  |

## Chapter Three

### Related Work

Routing protocols are in charge of establishing and maintaining connections between network nodes. As a result, routing protocol behavior has a significant impact on network performance. On-demand routing systems are better suited to dynamic networks since they adapt fast to the network's changing nature. On-demand routing protocol has a minimal processor overhead, memory overhead, and network utilization as compared to proactive routing protocol.

Although there are various protocols and upgrades for wireless networks in recent literature, only a small number of them are specifically developed to meet the needs of IoT applications. The IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) and the Lightweight On-Demand Ad hoc Distance-vector Routing Protocol – Next Generation (LOADng) are two examples. While RPL is the industry standard for IoT routing, LOADng is still in the works and has emerged as a more lightweight and less sophisticated alternative. RPL's Mode of Operation allows it to transmit multicast messages, however it does so at the cost of a lot of memory and complicated implementations with a vague explanation.

Unlike RPL, the LOADng does not provide functionality for multicast message forwarding. As previously stated, multicast data communication is an important feature for IoT applications since it enables for the transmission of similar data messages to a small set of nodes with fewer transmissions. As a result, a routing protocol developed for LLN must provide efficient and reliable multicast support. Although newer techniques have improved LOADng functionality and introduced additional capabilities to the primary protocol core, the state-of-the-art does not currently handle multicast messages on the LOADng, to the best of the authors' knowledge. [35] Multicast LOADng (M-LOADng) is a novel multicast route discovery technology that enables multicast group leaders (also known as sink nodes or gateways) to establish a routing tree for a collection of network nodes that can decide to join a multicast group. The created tree is used to forward multicast data messages, but it can also be used by all other nodes sending data to the sink. The M-LOADng also has techniques for reducing control message utilization while maintaining the route finding process. Finally, the work provides for the use of multiple forwarding modes,

allowing it to cater to the needs of a variety of applications. Thus, the main contributions of this work are the following:

- ✓ Support for the LOADng protocol in constructing a multicast routing tree for efficient multicast message routing;
- ✓ Allow data message forwarding from the nodes to the sink to be performed using the built-in multicast routing tree instead of expensive new route discovery process;
- ✓ Reduce the number of control messages sent and increase the route discovery process's dependability;
- ✓ Different multicast data message forwarding options are available to meet the needs of various IoT applications.
- ✓ Present a complete performance evaluation of the suggested solution using a real-world testbed.

In the Internet of Things (IoT), machine type communication (MTC) allows devices to talk with one another to create an intelligent environment. From the standpoint of heterogeneous network integration. IoT devices have a diverse set of requirements and specifications. Sensor nodes are limited-resource devices having limited energy, memory, and bandwidth, as well as non rechargeable batteries. The node's lifetime is determined by how well it uses its energy. When a route is overused, the energy of the nodes along the route is depleted significantly faster.

Choosing an inconsistent routing statistic at random can result in unpredictable routing and the failure to find the best path to the destination. The suitable routing measure is chosen based on the network's intended use. All routing measures, in any event, should meet the three conditions of consistency, optimality, and loop-freeness. A composite routing measure can be created by combining two or more compatible metrics. The routing metrics are divided into two categories: link-based and node-based. The link attributes, such as transmission power or projected number of transmission counts, are taken into account by a link-based routing measure. A node-based routing measure takes into account node characteristics such as residual energy and congestion. The number of active paths across a node has scarcely been investigated in terms of node congestion. The protocol should avoid overloading a node, as this may result in the node's energy being depleted sooner.

There have been substantial works on LOADng:

- (i) To broaden the scope of its deployment ability in other topologies (for example, data collection networks);
- (ii) To make it more efficient in LLNs by lowering its control overhead;
- (iii) In order to improve its performance in medium-to-heavy traffic situations;
- (iv) To improve protocol scalability, interoperability, and security, as well as path quality.

Despite this, little effort has been done on developing alternative route metrics to improve data transmission reliability, node energy efficiency, network resource utilization efficiency, and network longevity. In addition, there is very little research on the impact of multipath routing in LOADng-based LLNs. Furthermore, to the best of the authors' knowledge, no research has been done on the impact of data forwarding strategies in such networks when combined with MPR. In this research, a novel composite routing metric based on hop count (HC), node residual energy (RE), and total number of live routes (LR) in a node is suggested [36] for the LOADng. Then, for node-disjoint multipath discovery, the new composite metric is combined with an expanded version of LOADng. We also present a unique weighted forwarding (WF) technique to improve the network's dependability, load balancing, and energy economy. the network's resilience, load balancing, and energy efficiency

The main contributions of this paper are the following.

- ✓ Propose LRRE, a composite routing statistic that combines remaining energy (RE), live route count (LR), and hop count.
- ✓ Compare the performance of LOADng single path routing with the route metrics LRRE, LR, RE, and HC.
- ✓ Develop a ns-3 simulation module for LOADng multipath routing using the LRRE route metric.
- ✓ Introduce an unique MPR data forwarding strategy to improve network stability and longevity while lowering energy consumption.
- ✓ For the simple example of a single source-destination (S-D) pair, derive analytical equations to compare the load-balancing capabilities of different data forwarding methods.
- ✓ Compare and contrast LOADng MPR's performance with the LRRE route metric and other forwarding algorithms.

The use of wireless communications to provide a ubiquitous and pervasive network is fundamental to smart city concepts. The wireless network devices in use are nodes with a variety of hardware, energy, and communication constraints. Due to these limitations, selecting an effective routing protocol is a critical step in achieving a high-performance network. Despite the fact that the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) is utilized in a wide range of smart city applications, new research has shown some of the protocol's flaws and limits. The Lightweight On-demand Ad hoc Distance-vector Routing Protocol (LOADng) could be a viable alternative to RPL in this situation. Although the LOADng reactive routing protocol is constantly utilized and tested in numerous network contexts, its performance in a smart city network has yet to be investigated. [37] As a result, this paper presents a performance evaluation study of LOADng in a smart city scenario. The routing protocol was tested using various routing metrics in order to determine which one was best suited for a smart city application.

The main contributions of this work are the following:

- (i) identify the advantages and disadvantages of using LOADng in smart city applications;
- (ii) give detailed analysis of the advantages and drawbacks of several LOADng routing metrics for smart city applications.

Power line communications (PLC) is a prominent technology that provides infrastructure for IoT, smart grids, smart cities, and in-home networking applications, and it has been tested for internet access. This technology is strongly related to sensor networks and automatic meter reading applications since it provides free infrastructure and meets data rate requirements. The application under consideration is the deployment of the G3-PLC LOADng routing protocol in sensor/meter network nodes that all share the same media. G3-PLC is a power line communication standard that uses OFDM at the physical layer and is aimed at smart grid applications. The Logical Link Control uses LOADng routing, while the Medium Access Control uses CSMA/CA [38]. By synthetically training an AI to infer the number of hops required to reach any destination by looking at topological and geometrical properties of the network, the goal of this work is to show how a supervised learning regression-based ANN can be used to infer the routing set in the central element of our network. We believe that using machine learning to create smart, fast, next-generation networking is the way to go.



The LOADng Collection Tree Protocol (LOADng-CTP) is one of the LOADng protocol extensions. This module enables the creation of a bi-directional collection tree that is anchored in a single router and spans the entire network. [39] LOADng-CTP allows you to reduce the number of RREQ flooding operations from  $(n-1)$  to 2 in order to build bi-directional paths between the root and all other routers in the network, but it has the same state requirements as before. This addition is useful in scenarios where a central controller or monitoring entity is present and controlling the network, such as smart grid management or home/building/factory automation.

Because many devices on an Internet of Things (IoT) network have significant computing resource limits in terms of memory, processing, and power, these networks are sometimes referred to as Low Power and Lossy Networks (LLNs). [40] Device mobility has various detrimental effects on the network in these cases, particularly in terms of connection, because node movement changes the routing protocol's paths. The neighboring routing protocol has a strong influence on network performance. The protocol that has become the industry standard for IoT networks was intended for static networks and, among other things, has limited reactivity in mobility scenarios. Currently, the predominant reactive protocol for LLNs is LOADng. Given the importance of mobility in the future of IoT, this paper suggests Extended Kalman Filter (EKF)-LOADng, a LOADng-based solution for LLNs in the mobility context. The solution's goal is to make mobile nodes aware of their location and, using the EKF, anticipate their nonlinear course. The retrieved location is utilized to improve mobile node connectivity and shorten pathways while messages are exchanged on the network. Under two separate scenarios in the Contiki OS, EKF-LOADng was compared against the normal version and an upgraded version of the LOADng protocol.

Although some research has been done on LOADng, these studies do not compare the protocol's performance with multiple routing measures. During the route creation process, a routing metric is in charge of defining values for paths. Furthermore, a routing protocol will choose the way to forward a message depending on the metrics information. [41] As a result, the goal of this study is to conduct a performance evaluation of LOADng using several routing metrics. Different traffic patterns and network sizes are considered in the scenarios under investigation.

The routing metrics that are employed have an impact on the performance of a routing protocol. A routing metric specifies how a routing protocol should weight each path and choose the optimal one. As a result, this work presents a performance evaluation research that evaluates the usage of

several routing measures on LOADng, based on the necessity of understanding the protocol's true potential. The hop count measure is used by default in the LOADng protocol to pick the path (in this case, the shortest path) between two nodes. However, alternative information can be used to calculate the weight of the routes. The routing measure in use has a significant impact on the performance of a routing system. The LOADng uses information from control messages or calculated at the time of the signal received to compute the weight of each path according to the routing metric during the route building process (transmission of RREQs and RREPs). The calculated values are used to update the routing table with the most efficient route to a given location. Then, using control messages, the values are sent to the other nodes.

As a result, based on the routing metric information, the routing table records the optimum path to a destination. New routing metrics are introduced, such as the Minimum Battery Cost Routing (MBCR) and Min-Max Battery Cost Routing (MMBCR). The Minimum Battery Cost Routing (MBCR) is a routing statistic that takes into account the energy of the nodes. When computing the optimum path between two nodes, the MBCR takes into account the node's remaining battery capacity. The MBCR's objective is to avoid routes with low residual energy in order to reduce packet loss and reduce total network power consumption. MMBCR In the same vein as MBCR. However, to address MBCR's basic problem, MMBCR's approach is to avoid using a route when the nodes' remaining battery is low.

The requirement for effective communication in low power lossy networks has gotten a lot of attention recently as the Internet of Things (IoT) has progressed. The LOADng routing protocol is used to offer a novel multipath-enhanced routing method for low power lossy networks are proposed. [42]The energy overheads incurred as well as the amount and size of control messages generated by three alternative multipath routing methods are analyzed using a theoretical framework for node disjoint multipath schemes. The two AODV-based multipath routing methods for use with the LOADng protocol have been changed. The essential properties of the network are supposed to be derived using a geometric random graph model based on the Log-normal shadowing radio model.

The deployment of smart grid applications necessitates the use of customized communication methods. Because of the existing communication infrastructure in the energy grid, power line communication (PLC) is becoming a more appealing choice. G3-PLC is a 2012 standard that

outlines certain strategies for dealing with the power line channel's poor transmission performance. G3-PLC network nodes, in particular, must act as relays in order to execute hop-by-hop packet transfer. WSNs inspired the development of LOADng. In LOADng, the G3-PLC standard added four new mechanisms: weak link counts, smart RREQs, a route repair mechanism, and another significant change to the acknowledgment mechanism.[43] Each packet has a number of weak links. Each packet that uses a weak link adds to this total. The Link Quality Indicator (LQI), which ranges from 0 to 256, determines how weak a link is. The LQI is calculated from the physical properties of the transmission when a packet is received. The link is considered weak if the LQI is less than or equal to 52 (52 corresponds to a Signal to Noise Ratio (SNR) of 3 dB).

In traditional LOADng, the metric value is used to judge whether or not the identified route is better. The hop count is the default metric value. Before the metric value is examined in G3-PLC LOADng, the weak link count is checked first. In practice, higher-quality routes are given priority over shorter routes. When a node receives an RREQ packet, it can verify its routing table to see if it contains a path to the destination using the Smart RREQ technique. If this is the case, the RREQ is merely unicast to this path rather than broadcasting to all neighbors. When an active link is broken, the node shortly before the broken link generates a local RREQ to discover an alternate route to the target using the route repair mechanism. It sends an RERR packet if it can't find it. All unicasted packets, such as RREPs, Smart RREQs, and RERRs, are acknowledged using the unique acknowledgement mechanism. However, because these two lasts were not used, only RREPs, as in the old LOADng protocol, were recognized.

The overall related works related to LOADng are described as follows

**Table 2 Comparisons of LOADng protocols in different scenarios**

| <b>Title</b>  | <b>Authors</b>   | <b>Parameter used for path selection</b>   | <b>Strength</b>   | <b>weakness</b>  |
|---|--|--|---|--|
| Performance analysis of the RPL and LOADng Routing Protocols in a Home Automation Scenario                  | Malisa Vucinic, Bernard Tourancheau et al (2014)         | Centralized architecture                   | RPL provides decent overall performance, but LOADng may be better suited to sparse LLN deployments with low-priority traffic where the Route Hold Time can reach a high value.      | The results indicates that LOADng is not the ideal solution for Home Automation applications when response speed is critical.  |
| Performance Assessment of the LOADng Routing Protocol in Smart City Scenarios                               | Jose V. V. Sobral , Joel J. P. C. Rodrigues et al (2017) | Link Quality Indicator Weak links (LQI WL) | To differentiate links, it used a threshold. The optimum route between a sender and a destination node is thus the one with the smallest weak links among those that are available. | LQI WL examines the quality of the link between nodes and attempts to avoid low-quality pathways. This may need the use of more hops.  |
| Performance Evaluation of LOADng Routing Protocol in IoT P2P and MP2P Applications                          | Jose V. V. Sobral, Joel J. P. C. Rodrigues et al (2016)  | Point to point and Multi point to point    | In IOT scenarios using MP2P apps has better performance   | It was observed that when the size of the network grows, the performance of protocol reduced dramatically.   |
| Improving Network Lifetime and Reliability for Machine Type Communications based on LOADng Routing Protocol | Deepthi Sasidharan, Lillykutty Jacob (2018)              | Single Path Routing (SPR)                  | Introduce a unique data forwarding strategy to improve network stability and lifetime by reducing energy consumption.   | Only nodes that are within transmission range of each other can connect directly, and a source node may need to make numerous intermediate hops to reach the destination node. |

|   |   |  |   |   |
|---|---|--|---|---|
| An Enhanced Routing Protocol for Internet of Things Applications over Low Power Networks                        | Ricardo A. L. Rabêlo and Kashif Saleem et al (2019)         | Internet routes                          | It minimizes the number of control messages needed to build routes between nodes.   | Rather than looking for a specific destination address, it employs the Internet route discovery mechanism, which looks for any Interconnected Nodes (IN).                 |
| A mobility solution for low power and lossy networks using the LOADng protocol                                  | Allan J. R. Gonçalves I Ricardo A. L. Rabêlo I et al (2019) | Extended Kalman Filter (EKF)             | Makes mobile nodes aware of their position, which improves mobile node connectivity and shortens pathways while messages are exchanged on the network.  | Trilateration, which calculates the current mobile node location based on the RSSI of three surrounding static nodes, is used for RF-based localization.                  |
| Artificial-Intelligence-Based Performance Enhancement of the G3-PLC LOADng Routing Protocol for Sensor Networks | Francesco Marcuzzi and Andrea M. Tonello (2019)             | supervised-learning regression-based ANN | By synthetically teaching an AI to estimate the amount of hops required to reach any destination by looking at topological and geometrical aspects of the network, it may be used to infer the routing set in the network's central element, enabling smart, rapid, next-generation networking. | To begin the learning process and devise a statistical trend, ML requires a large enough database of issue solutions (i.e. desirable behavior for the solving algorithm). |
| Path Accumulation Extensions for the LOADng Routing Protocol in Sensor Networks                                 | Thomas Heide Clausen, Jiazi Yi (2019)                       | Path Accumulation                        | Minimizes the number of RREQ flooding operations from (n-1) to 2 to create bidirectional pathways between the root and all other routers in the network.  | Only applicable for deployments where a central controller, or monitoring entity is present and operating the network   |

## Chapter Four

### Proposed Work

The acronym LOADng refers to the (Lightweight On Demand Ad hoc Distance vector next generation) protocol. The basic operations of LOADng protocol is [44], When a LOADng Router (originator) discovers a route to a destination, it generates Route Requests (RREQs) and floods them around the network, as well as Route Replies (RREPs) generated by the sought destination and sent to the originator through unicasts. When an intermediate router forwards an RREQ, it creates a temporary routing table entry for the RREQ's originator to forward direction from the destination to the originator. When the sought destination receives an RREQ, it responds with a unicast RREP, which is forwarded along the installed reverse route and whose forwarding serves to establish a forward route from the originator to the destination. For each bidirectional path via a LOADng router, four entries are held in the routing table: one for the next hop and another for the destination from that next hop for directions.

One of the most appealing aspects of LOADng is its ability to combine simplicity and extensibility. The core protocol establishes mechanisms for establishing and maintaining bidirectional routes between router pairs. But it also allows for the development of functional extensions, which can improve the protocol's behavior and performance for specific deployments, topologies, and traffic patterns. LOADng is an AODV-based protocol tailored to the needs of LLNs [45].

The information required for hop-by-hop data packet routing is provided by LOADng. A router will know the next hop towards the destination for a data packet, but not the full path that the packet will take. To achieve this, only the destination address is included in the header of a data packet, relying on intermediate routers to make forwarding decisions based on their local knowledge of the routing topology. This allows each router to make a decision on the fly (e.g., if local connectivity changes more frequently than routing updates can be propagated globally through the network) and, *e.g.*, when a data packet arrives at a router without, or with obsolete, topological knowledge, this allows fast re-routing mechanisms to engage and enhance data packet distribution [46].

LOADng is one of routing protocol in sensor networks whose aim is to discover bi-directional paths. Each LOADng Router generates and processes RREQ, RREP, RREP-ACK and RERR messages. As a reactive protocol, established routes are only when there is data to be sent and there is any path towards destination. Routes are maintained for as long as there is traffic using this path. When a LOADng router attempts to send a data packet to a LOADng router for which it has no matching entry in its Routing Set, it starts the LOADng route exploration. A RREQ packet is then created and flooded in the network. Nodes routers receiving such RREQ install or upgrade their route towards the RREQ originator if it corresponds during this phase.

When the destination receives the RREQ, it produces an RREP packet that is sent unicast hop-by-hop along the reverse route to the RREQ originator. If the RREP-ACK flag is set, nodes receiving the RREP will unicast a proper RREP-ACK to the neighbor from which received the RREP, with the purpose to notify the neighbor sending the RREP that their link is bidirectional. When the originator of the RREQ receives the correspondent RREP, the route is installed in the Routing Set. Only the destination is allowed to respond to an RREQ during the route discovery procedure. This eliminates the need for AODV's artifact, Destination Sequence Numbers, allowing the message size to be reduced in comparison to other protocols. In this way, no gratuitous RREP are not sent whilst loop freedom is retained.

When a route towards destination is formed, a LOADng router does not maintain a precursor list as AODV does, and it only takes care of the next hop to forward the packet towards the final destination. A route error (RERR) packet is sent only to the data packet's originator when forwarding a data packet to the next hop towards destination fails. Different address lengths are allowed, including those of IPv6. The only constraint is that in a given network each device has a unique address, and that all addresses of the network are of the same length. Each LOADng router must have at least one port, each of them equipped with one or more network addresses. This protocol is layer-independent. It can be used as a route over protocol at layer 3 or as a mesh under routing protocol at layer 2.

The LOADng routing protocol was created for wireless networks that have limited hardware capacity. The protocol is based on the well-known AODV and features a reactive route discovery mechanism that creates a path between nodes on-demand [47]. The LOADng's core structure is made up of four control messages and three data structures that each network computer is

responsible for maintaining. Route Request (RREQ), Route Reply (RREP), Route Reply Acknowledgement (RREP ACK), and Route Error are the control messages that are used to create and maintain paths between nodes (RERR).

**Table 3 RREQ and RREP messages fields and RREP\_ACK message fields**

| Field                                | Size(bits) | Description   |
|--------------------------------------|------------|---|
| <b>RREQ and RREP messages fields</b> |            |   |
| type                                 | 8          | type of message   |
| addr-length                          | 4          | length of originator and destinations addresses                                   |
| seq-num                              | 16         | sequence number that uniquely identifies each message of an originator            |
| metric-type                          | 8          | metric type used to construct the route   |
| Route_metric                         | 32         | value computed based on the used metric type                                      |
| hop-count                            | 8          | number of times that message was transmitted                                      |
| hop-limit                            | 8          | maximum number of hops permitted  |
| originator                           | variable   | address of the message originator   |
| destination                          | variable   | address of the message destination  |
| Ack-required                         | 1          | if flagged, indicates the need of an acknowledgment message (only used with RREP) |
| <b>RREP_ACK message fields</b>       |            |   |
| addr-length                          | 4          | length of originator and destinations addresses                                   |
| seq-num                              | 16         | sequence number of the RREP that will be confirmed                                |
| destination                          | variable   | address of the message destination, i.e., the RREP originator address             |



**Table 4 The fields of Routing Set and Pending Ack Set**

| <b>Field</b>           | <b>Size(bits)</b> | <b>Description</b>  |
|------------------------|-------------------|---|
| <b>Routing Set</b>     |                   |   |
| R_dest_addr            | Variable          | address of route destination  |
| R_next_addr            | Variable          |   |
| R_metric_type          | 8                 | metric type used to compute the route metric                          |
| R_metric               | 32                | value computed to the route based on the metric type                  |
| R_hop_count            | 8                 | number of hops to reach the destination                               |
| R_seq_num              | 16                | sequence number of the message used to create/refresh the route entry |
| R_bidirectional        | 1                 | Boolean flag that if TRUE, indicates a bidirectional route            |
| R_local_iface_addr     | Variable          | address of the interface used to communicate with the destination     |
| R_valid_time           | 16                | valid time of the route   |
| <b>Pending Ack Set</b> |                   |   |
| P_next_hop             | Variable          | address of the node which the RREP was sent                           |
| P_originator           | Variable          | address of the RREP originator  |
| P_seq_num              | 16                | value of the seq_num field of the sent RREP                           |
| P_ack_received         | 1                 | Boolean flag set as TRUE when the corresponding RREP_ACK is received  |
| P_ack_timeout          | 16                | time to expire the entry  |

When a node wants to find a route to a destination, it should send an RREQ message to initiate the route discovery process. As a result, the created RREQ is broadcast, which includes the intended destination's address. When a node receives an RREQ, it must update some message attributes, save information about the message originator and previous hop node to its Routing Set, and validate the message destination. Before broadcasting a message to its neighbors, the node should check the message's maximum hop limit if the node address does not match the RREQ destination. Otherwise, in response to the received route request, the node could send an RREP message to the RREQ originator.

The RREP is prepared and sent in unicast over the same path as the RREQ. Each node that receives an RREP message should refresh its Routing Set and update the message fields. Using the information reported in the Routing Set during RREQ propagation, the message should then be forwarded to the next hop in the path to the message destination. This procedure is repeated until the RREP achieves its objective. When the RREQ originator receives the RREP, the route discovery process is complete, and data messages can be sent along the built path.

The LOADng optionally provides the use of the RREP ACK message to validate the receipt of RREP messages in order to increase the reliability of the built routes. As a result, when an RREP is created, the node may choose to set the Ack-required flag in the message. When an RREP message with the Ack needed flag set is sent, the sender node should add an entry to the Pending Acknowledgement Set containing information about the RREP next hop node and a valid time to wait for the RREP reply. When a node receives an RREP with the Ack-required flag set, it should send the message sender an RREP ACK to validate the RREP reception. If the Pending Acknowledgement Set entry does not receive an RREP ACK before it expires, the delivered RREP's next hop should be put to the Blacklisted Neighbor Set, and all further messages received from it should be disregarded.

#### **4.1. Architectures of Proposed Work**

WSNs are frequently referred to as infrastructure-less networks [48], in which nodes must work together to build a network and collect and send data. Because sensor nodes have limited resources, resource management is a major concern, especially in WSNs with a large number of nodes, many of which may operate as data forwarding nodes. Because wireless data transmission consumes the majority of the energy in sensor platforms, resource use in WSNs, particularly energy

consumption, requires careful monitoring. Clustering approaches are effective and useful for reducing energy consumption and extending the network's lifetime. Clustering is the process of grouping nodes or data points into groups such that data points in the same group are more comparable to data points in other groups than data points from other groups. To put it another way, the aim is to separate groups with similar characteristics and assign them to clusters. Clustering is divided into two stages. These are the Cluster Head (CH) node selection and cluster creation. In step one, a Cluster Head (CH) is chosen from among the network nodes and in step two, nodes that are not chosen as CHs are determined to simply bind to a cluster node in the network. The clustering algorithm that is preferably used in this work is agglomerative hierarchical clustering.

## **4.2. Types of Nodes In Clustering**

Based on the computing power, there are two types of nodes in clustering, homogenous and heterogeneous nodes. In homogenous clustering all nodes have equal power to operate the task whereas heterogeneous nodes have different computing resource, bandwidth and computing power to operate. Some nodes in heterogeneous clustering have special power.

### **4.2.1. Homogeneous Clustering**

Sensors in homogeneous networks start with the same amount of energy, have the same communication range, and have the same sensing range. As a result, sensors react similarly to similar conditions and consume the same amount of energy. Each node may be a CH in these networks, which are the most common in most applications. Additionally, the CH function can be rotated among nodes on a regular basis to improve load balancing and uniform energy consumption. A homogeneous sensor network is made up of BS and sensor nodes that have the same capabilities, such as processing power and memory space. In these types of networks, data processing is focused on the data dissemination mechanism.

Flat and hierarchical topologies are two well-known architectures that have been commonly used in homogeneous networks for data distribution and collection. Sensor nodes in a hierarchical network are arranged in clusters, with CHs acting as simple data relays. Since CHs have the same transmission capacity as sensor nodes, the upper bound of throughput can be used to calculate the minimum number of clusters available. Of course, the increased throughput obtained by clustering comes at the expense of additional nodes that serve as CHs. In a hierarchical network, data

aggregation entails merging data in CHs to minimize the number of transmitted messages to BS. As a result, network reliability in terms of energy consumption improves. The work is based on homogenous nodes.

#### **4.2.2. Heterogeneous Clustering**

In this form of clustering, not all sensors are in the same situations and which are known as heterogeneous networks. There are two types of sensors in this networks. The first type of sensor is a super node, which has a higher processing capacity and more sophisticated hardware; CH are chosen from among super nodes and receive data from sensor nodes. Standard nodes with lower capabilities make up the second type, which are used to sense certain environmental parameters. In such networks, mobile base stations are randomly located in the network area and collect data directly from ordinary sensor nodes or replay data using certain sensor nodes. Sensor nodes may often be dispersed, and the distance between two sensor nodes can be considerable. The longer the distance between sensor nodes, the more energy is used for communication. Sensor nodes must, in the meantime, be able to sense and communicate with each other for longer periods.

#### **4.3 Agglomerative Hierarchical Clustering**

The most common method of hierarchical clustering used to group objects in clusters based on their similarity is agglomerative clustering. AGNES is another name for it (Agglomerative Nesting). Each object is first treated as a singleton cluster by the algorithm. Then, one by one, pairs of clusters are merged until all artifacts are contained in a single giant cluster. A dendrogram, which is a tree-based representation of the elements, is the result. Agglomerative clustering starts at the bottom and works its way up. To put it another way, each entity is conceived as a single-element cluster at beginning leaf. The two clusters that are the most comparable are joined into a new larger cluster nodes at each phase of the procedure nodes. This process is repeated until all points belong to a single large cluster.

There are a few possible ways to perform agglomerative hierarchical clustering here. However they generally follow the following major steps:

- Calculating the proximity matrix for the initial clusters
- Searching for the minimal distance in the matrix
- Combining the two clusters with the minimal distance

- Updating the proximity matrix by calculating the distances between the new cluster with the other clusters
- Repeating the previous three steps if more than one cluster remains

Flow chart of the proposed algorithm

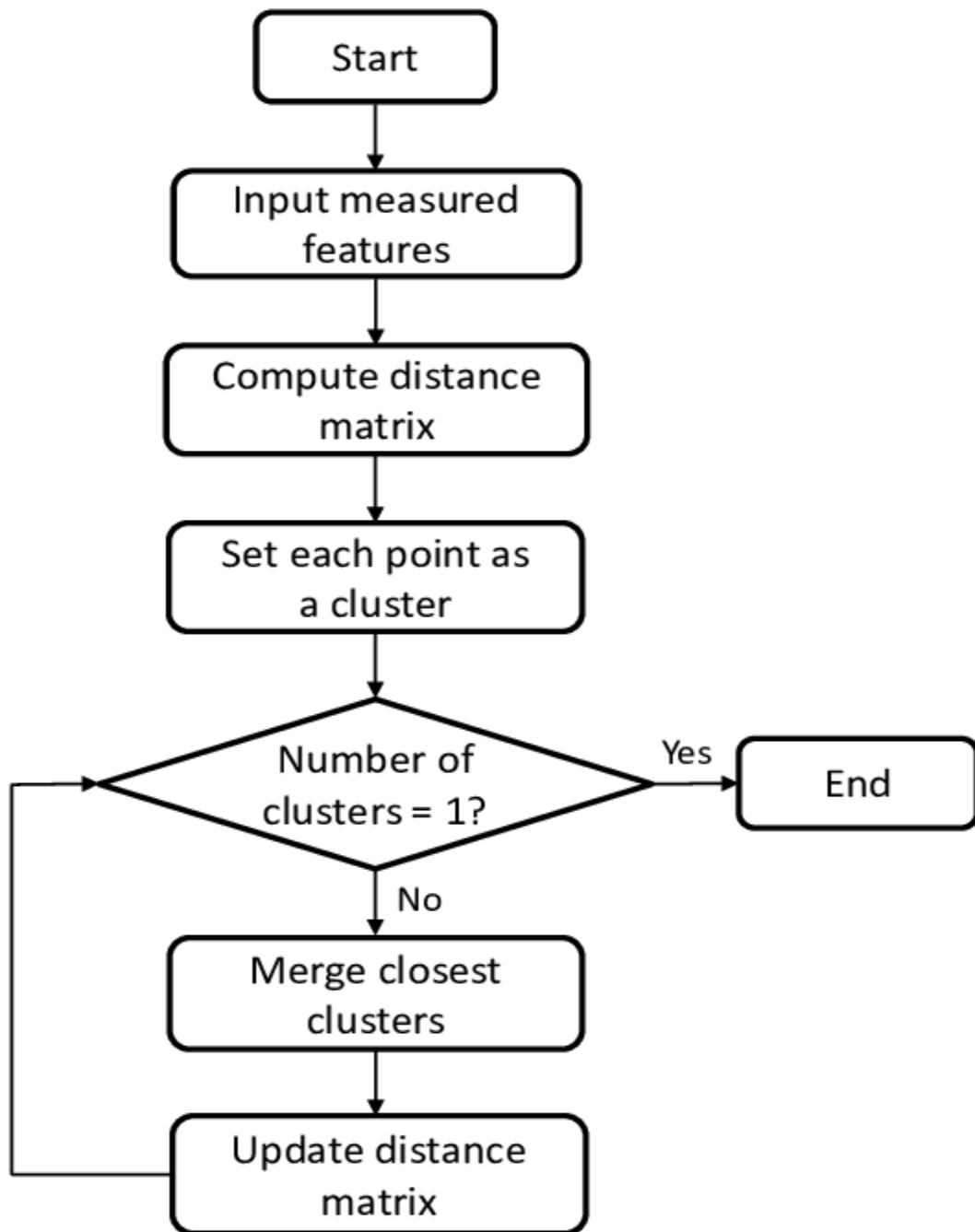


Figure 13 Flow Chart of proposed model

### **4.3.1 Cluster Head selection**

The Cluster Head selection has a significant effect on the clustering algorithm's efficiency and simplicity, as well as the network lifetime. Choosing the right CH will help you save a lot of money on electricity. The BS will do the CH selection, which can be centralized. It can also be spread when nodes take on the CH function on their own [49]. The CH selection can be performed deterministically or arbitrarily. The energy parameter will play a role in CH selection, with higher energy nodes having a better chance of being chosen as the CH. The CH selection can also be influenced by the distance parameter. Other factors, such as cluster size or the number of neighbors, can influence CH selection.

When choosing a CH becomes a time-consuming and complicated operation, it may add overhead to the network, resulting in energy consumption. When choosing a CH, a balance should be struck between the optimization of the CH and the energy usage resulting from the imposed overhead. CH clutter is avoided by distributing the CH evenly. The distance between cluster member nodes and their CH can be large in cluttered CH. To put it another way, intra-cluster communications use a lot of power. The delay period is the time it takes to choose a channel. It can also be used to describe the amount of time it takes for a cluster to grow. The success of clustering is influenced by this parameter.

### **4.3.2 Defining Proximity between clusters**

The calculation of cluster proximity is the core operation of the proximity algorithm, and it is the concept of cluster proximity that distinguishes the various agglomerative hierarchical techniques like MIN, MAX, and Group average. A graph-based view of a cluster is the source of several agglomerative hierarchical clustering techniques like MIN, MAX, and Group average. The distance between the nearest two locations in separate clusters, or the shortest edge between two nodes in separate subsets of nodes in graph terms, is defined by MIN. MAX defines cluster proximity as the distance between the farthest two points in separate clusters, or the longest edge between two nodes in separate subsets of nodes.

### **Cluster formation**

Clusters have been fully formed and all nodes have specified their status during the cluster forming process. Some nodes serve as the cluster head, while others serve as cluster members. The following principles should be considered when forming clusters.

## Cluster count

The CH selection and cluster forming method in most recent probabilistic and randomized clustering algorithms naturally result in the development of various cluster counts. The collection of CH is however, calculated in advance in some works. Thus, cluster counts are predefined.

## Intra-cluster communications

The communication between a sensor and its designated CH was assumed direct in some of the early clustering approaches (one-hop). However, in cases where the communication range of sensors is limited or the number of sensor nodes is very large and the number of CH is bounded, multi-hop intra-cluster communication is now a requirement.

The location of each node is calculated using Euclidian distance. Based on Euclidean distance, Euclidean matrix should be formed and store the distance of each nodes. Ones the Euclidean matrix formed each node knows the distance of its nearest node. On this way the communication should be started.

## Inter-cluster communications

CH can either directly or indirectly send data gathered from member nodes to the BS. In direct mode, CH uses one-hop transmission to send data to the BS. In the indirect mode, CH sends data via multi-hop transmission and a CH closer to the BS. Member nodes have different parameters to select an appropriate cluster including:

- The distance between the node and the CH: Each node calculates its Euclidean distance to the CH and connects to the CH with the shortest distance. The number of hops a node must make to hit the CH: Nodes are either directly or indirectly related to the CH (using two-hops or multi-hops). The number of hops has an effect on the CH selection.

$$\text{dist}(\overrightarrow{XY}) = \sqrt{(x - x_0)^2 + (y - y_0)^2}$$

Where: -

- $\overrightarrow{XY}$  the Euclidean distance of two points in two dimensional i.e. x and y plane
- $x - x_0$  and  $y - y_0$  Are the respective points of in x and y plane.

- The size of the cluster: The number of nodes in a cluster represents the cluster's energy density. As a result, cluster size is a critical consideration in clustering. The formation of the cluster will result in the emergence of overhead that is imposed on the network. The amount of overhead applied to the network is determined by the parameters that influence CH selection as well as the complexity of the cluster creation algorithms. Clearly, lower overhead equates to greater performance. The selected cluster may have a large number of clusters, and its distance from the central station is long and energy is small, so this cluster will die and access will be difficult. Therefore, overhead rises, and the window for selecting a sink in a suitable cluster closes.

Intercommunication is accomplished using MIN (single link) techniques. The maximum distance (maximum similarity) between any two points in two different clusters is known as the MIN or Single link proximity of two clusters. Starting with all points as singleton clusters, add ties between points one at a time, shortest links first, then these single links merge the points into clusters, using graph terminology.

### Distance Matrix

The Distance Matrix determines the fastest or shortest routes between two points and solves the commonly known Travelling Salesman problem. To calculate the distance matrix of the following points some steps are applied.

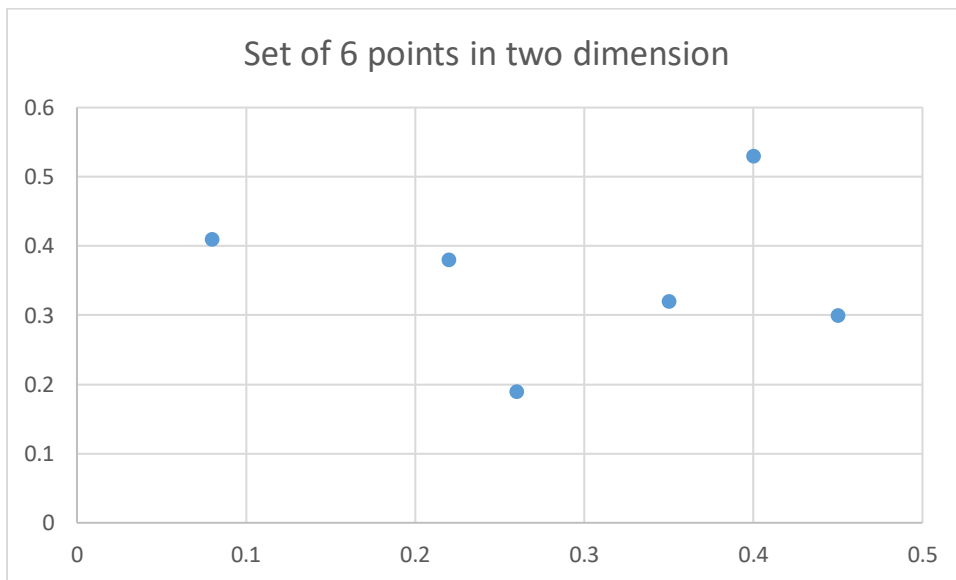


Figure 14 Points to calculate Distance matrix



The table below shows numeric representation of the points above. All the distance matrixes of the points are represented also.

**Table 5 Two-dimensional points to calculate distance Matrix**

| Points | x coordinates | y coordinate |
|--------|---------------|--------------|
| P1     | 0.40          | 0.53         |
| P2     | 0.22          | 0.38         |
| P3     | 0.35          | 0.32         |
| P4     | 0.26          | 0.19         |
| P5     | 0.08          | 0.41         |
| P6     | 0.45          | 0.30         |

The Euclidean distance between each points are calculated and put in the following manner. The formula used to find Euclidean distance is  $\text{dist}(\overrightarrow{XY}) = \sqrt{(x - x_0)^2 + (y - y_0)^2}$

**Table 6 The calculated distance matrix**

|    | P1   | P2   | P3   | P4   | P5   | P6   |
|----|------|------|------|------|------|------|
| P1 | 0.00 | 0.24 | 0.22 | 0.37 | 0.34 | 0.23 |
| P2 | 0.24 | 0.00 | 0.15 | 0.20 | 0.14 | 0.25 |
| P3 | 0.22 | 0.15 | 0.00 | 0.15 | 0.28 | 0.11 |
| P4 | 0.37 | 0.20 | 0.15 | 0.00 | 0.29 | 0.22 |
| P5 | 0.34 | 0.14 | 0.28 | 0.29 | 0.00 | 0.39 |
| P6 | 0.23 | 0.25 | 0.11 | 0.22 | 0.39 | 0.00 |

Once the distance matrix is calculated, every nodes have distance of its neighboring nodes. Therefore, it is very important while routing.

### **Balance**

In terms of the number of member nodes, distance to the BS, and cluster balance in the environment, formed clusters differ from one another. Clusters are balanced in terms of member nodes and standing position in some clustering algorithms, resulting in a balanced distribution of clusters in the environment. However, there are several other algorithms in which there is no

equilibrium in the number of member nodes, the distance between the cluster and the BS, the number of neighbors and other parameters.

### **Clustering parameters**

In clustering, several parameters can be used as the basic means for comparing and classifying clustering protocols. Some of the parameters are described as follows.

#### **Energy efficiency**

The battery is the most significant weakness in sensor networks. Sensors are powered by a limited-capacity battery that cannot be recharged in most cases. As a result, the first concept that clustering algorithms can remember is optimum energy consumption.

#### **Location awareness**

In order to determine their distance from their neighbors, sensor nodes must be aware of their position, their neighbors, and in some cases, the entire network. Installing the Global Positioning System (GPS) or calculating the frequency of the returned signal are also common ways to do this. Both approaches necessitate the use of energy. Some algorithms are built in such a way that nodes do not even need to know their own physical locations or the locations of their neighbors. Many strategies for positioning in WSNs have been suggested in recent years. These methods can be classified into two categories: There are two types of techniques: anchor-based techniques and non-anchor techniques. Before starting the positioning procedure, an anchor node is a sensor node that knows where it is by using manual settings or GPS.

In anchor-based techniques, the sensor network is believed to have a fixed number of anchor nodes. Since all nodes can estimate their definite location in the global coordinate system, the aim of these techniques is to make use of their capabilities. There is no need for an anchor node in non-anchor techniques because sensor nodes estimate their relative location in the network graph at the end of the positioning process. While anchor-based techniques can measure the precise location of nodes in the global coordinate system, having an anchor node necessitates additional equipment, positioning, and manual settings that are not feasible due to sensor node limitations and an unsuitable network environment. Techniques that do not use anchors are less expensive than anchor-based techniques.

## **Advantages of Clustering**

In addition to supporting scalability of the network and reducing the energy consumption by aggregating data and decreasing transmissions, clustering has many other advantages. Some clustering advantages were described below.

*Data Fusion:* - CHs collect data from various nodes and send it to the base station (BS) during data fusion. Data fusion also removes redundant data at the CH stage, which relieves the sensor nodes of additional communication load. As a result, data fusion improves total network lifetime and conserves all network capacity [50].

*Data Load Management:* - Clustering allows for effective data load control and a consistent network lifetime. In order to transmit data from upper layers, CHs closer to the BS experience increased data loads. To deal with this problem, CHs that are closer to the BS keep fewer member nodes to reduce load. As a result, all nodes deplete their resources at the same rate, and the network lifetime becomes consistent.

*Efficient Energy Saving:* - Data is transmitted via flooding in flat networks, but data is aggregated on the CH level and sent to the BS through multi-hop routing in cluster-based networks. In a cluster-based network, multi-hop routing reduces the number of transmission routes, saving energy exponentially [51].

*Relay Node:* - When nodes fail to communicate, the network is partitioned or disconnected. The relay node is used to reconnect the partitions and reestablish the route. The relay node may be either static or mobile. A static node's initial job, on the other hand, is to locate the disjoint portion and then deploy a relay node there. However, a mobile relay node is a unique type of node that operates in a disjointed region [52].

*Robustness:* - The critical step after forming a cluster-based WSN is cluster maintenance. Cluster maintenance is essential to keep the network running smoothly. It can handle a variety of situations, including network size changes, node movement, and unforeseen operational flaws. Managing these differences within each cluster is all that clustering algorithms need. As a result, cluster maintenance makes the network more stable and easy to manipulate topologically.

*Collision Avoidance:* - When a single channel is considered in a sensor network, it is shared among sensor nodes. As a result, when a large number of nodes send data at the same time, the network's output suffers. This can be efficiently solved in a cluster-based WSN, where the CH uses scheduling to assign a specific time slot to each member node [53].

*Latency Reduction:* - The cumulative time it takes for a message to move from source to destination node is referred to as latency. By maintaining a routing table at the CH level to make efficient routing decisions, cluster-based WSN improves packet delivery efficiency. Furthermore, cluster-based networks based on the linked dominant set form a predefined communication pathway known as the backbone tree, which allows for fast and efficient multi-hop routing.

*Secure Data Communication:* - Malicious nodes can attack to alter or hack data because CH performs data aggregation. Powerful authentication schemes are developed in cluster-based WSNs to prevent malicious nodes from entering the network. These schemes help to ensure data security and privacy.

*Fault Tolerance:* - Hardware failure, delay, interference, energy exhaustion, and other factors may all affect sensor nodes. Cluster-based protocols are ideal for such limitations, where nodes are not replaceable in a harsh setting. WSNs, in particular in harsh conditions and remote areas, must be able to reconfigure themselves without human intervention. Fault tolerance techniques must be considered during the protocol design stage in order to protect aggregated data. When CH fails, cluster repair and CH backup are more practical techniques for securing the whole network reconstruction [54].

*Data Communication Assurance:* - CH uses single-hop or multi-hop routing to return the aggregated data to the base station. Because of its high likelihood of incidence, the probability of data loss in mobile networks has piqued researchers' interest in recent studies. To deal with such issues, the mobile node sends a joint request to its CH before initiating data contact. If the sender receives the acknowledgment code, it begins data transmission; if it does not, the sender node considers itself to be no longer a member of the network and must reconnect. The network then starts transmitting data to the parent node after the node is rejoined. As a result, ensuring compatibility between member nodes and their CH is critical for efficient data delivery [55].

*Deadlock Prevention:* - Data is sent to the base station via intermediate nodes in multi-hop communication. Different nodes transmit data to the base station in this criterion. As a result, the node closest to the sink node is overburdened with more information than the nodes further out. As a result, nodes closer to the BS deplete energy faster, resulting in deadlock near the BS. This can result in the network being divided into groups. Because of the restricted range, the far nodes may not be able to reach BS. Other nodes, on the other hand, continue to have resources. To address these issues, load balanced clusters were examined, in which a cluster closer to the base station maintains less member nodes than a cluster farther away. As a result, a closer CH keeps enough energy for intercluster contact. As a result, using clusters of unequal size will effectively manage deadlock prevention.

*Network Lifetime:* - Since nodes have limited capacity, bandwidth, and processing capacities, increasing network lifetime is a critical factor. Optimizing a few problems in WSNs, such as intra-cluster communication expense, redundant data gathering, and uniform cluster loads, is usually a critical task. Such considerations are taken into account during CH voting, extending the network's lifespan. Furthermore, higher energy routes are prioritized for data transmission, with this requirement resulting in uniformed energy loss in the network and extending network lifespan [56].

*Efficient Quality of Services:* - WSN's functionalities and network implementations necessitate the requirement of service efficiency. End-to-end delay, reliability, throughput, jitter, and bandwidth are common effective QoS parameters. In cluster-based protocols, it is difficult to meet all of the requirements for QoS parameters. To accept one or more QoS parameters based on application requirements, a trade-off is required. The energy efficiency of state-of-the-art cluster-based protocols is prioritized over QoS. For real-time application domains such as healthcare, frontline applications, and event observing, QoS problems are taken into account [57].

# Chapter Five

## Implementation

RREQ packets are sent by a LOADng router to obtain routes. When receiving LOADng packets, the routing table may also be modified. LOADng packet reception, encoding, and forwarding for various message types.

### 5.1. Route Requests (RREQs)

When a LOADng Router has data packets to deliver to a destination and the data packet source is local to that LOADng Router i.e., an address in that LOADng Router's Local Interface Set or Destination Address Set. But if no matching tuple is in the Routing Set, it creates Route Requests (RREQs). After launching an RREQ, a LOADng Router waits for a corresponding RREP. If no such RREP is received within the requested period, the LOADng Router may send a new RREQ for the sought destination with an incremented sequence number up to a limit of RREQ retries times.

#### 5.1.1. RREQ Generation

RREQ Message has the following message contents

**Addr-length** is an integer field, which encodes the length of the originator/source and destination addresses. Addr-length is calculated as the length of an address in octets minus 1

**Seq-num** is an unsigned integer field that holds the LOADng Router's sequence number for sending the RREQ message.

**Metric-type** is a field and specifies the type of metric requested by this RREQ.

**Route-metric** is a field, of length defined by RREQ.metric-type, which specifies the route metric of the route (the sum of the link metrics of the links), through which this RREQ has traveled.

**hop-count** is a field and specifies the total number of hops which the message has crossed from the originator.

**hop-limit** is a field and specifies the number of hops that the message is allowed to traverse.

An RREQ message is generated according to the message fields

- RREQ.addr-length set to the length of the address,

- RREQ.metric-type set to the desired metric type;
- RREQ.route-metric := 0.
- RREQ.seq-num set to the next unused sequence number, maintained by this LOADng Router;
- RREQ.hop-count := 0;
- RREQ.hop-limit := MAX\_HOP\_LIMIT;
- RREQ.destination := the address for which a route is being sought;
- RREQ.originator := One of the addresses of the LOADng Router's in which LOADng Interface generates the RREQ. The source address of the data packet provided by that host is used, if the LOADng Router is generating RREQ on behalf of a host connected to this LOADng Router.

### 5.1.2. RREQ Processing

RREQ uses the variables hop-count and hop-limit, which have been modified when processing the message. When a LOADng Router receives an RREQ message, it must process it as described below:

- ❖ If a message is ineligible for processing, it must be discarded without further processing. The message is ineligible for forwarding.
- ❖ Otherwise, the message is processed
- ❖ If the destination is given in any local interface tuple or corresponds to the destination address of any Destination Address Tuple of this LOADng Router, the RREP generating technique must be utilized. When forwarding, the RREQ is not taken into account..
- ❖ Otherwise, if the hop-count is less than the maximum hop count and the hop-limit is greater than 0, the message is evaluated for forwarding.

### 5.1.3. RREQ Forwarding

When processing the message, the variables utilized-metric type, hop-count, hop-limit, and route-metric were altered, and they are used in this part to update the content of the message to be forwarded. Prior to transmission, an RREQ that is being considered for forwarding must be changed as follows:

1. RREQ.metric-type := used-metric-type
2. RREQ.route-metric := route-metric

3. RREQ.hop-count := hop-count
4. RREQ.hop-limit := hop-limit

An RREQ must be transmitted in accordance with the network's flooding operation. This could be accomplished through traditional flooding, a decreased relay set mechanism, or any other kind of information dispersion. It is important to make sure that network traversal time is set to the maximum time that an RREQ can traverse the network, taking into account any in-router delays caused by or imposed by such algorithms.

#### **5.1.4. RREQ Transmission**

RREQs are sent to all neighboring LOADng Routers through all interfaces in the Local Interface Set, whether they are created or forwarded. When an RREQ message is broadcast, all receiving LOADng Routers will process it and consider it for forwarding at the same time, or practically at the same time. RREQ messages should be jittered if employing data link and physical layers that are prone to packet loss due to collisions.

### **5.2. Route Replies (RREPs)**

A LOADng Router generates Route Replies (RREPs) in response to an RREQ, which are transmitted by the LOADng router with the RREQ.destination address in its Destination Address Set or Local Interface Set. RREPs are sent to the originator of the RREQ in response to which the RREP was formed in unicast, hop by hop, along the Reverse Route built by that RREQ. The Forward Route is established towards the RREP.destination when a LOADng Router forwards an RREP.

Forwarding of RREQs installing the Reverse Route and forwarding of RREPs installing the Forward Route offer bidirectional pathways between the RREQ.originator and RREQ.destination.

#### **5.2.1. RREP Generation**

The RREP must be created in response to RREQ messages that have been issued in the past (RREP.originator, RREP.seqnum). An RREP can be generated immediately in response to each RREQ processed to provide the shortest possible route establishment delays, or after a certain delay after the arrival of the first RREQ to use the best received RREQ (e.g., received over the lowest-cost route) but at the cost of longer route establishment delays. A LOADng router can



generate extra RREPs at the cost of more control traffic if subsequent RREQs with the same (RREP.originator, RREP.seq-num) pairs indicate a better path.

An RREP's content is as follows:

- RREP.addr-length set to the length of the address,
- RREP.seq-num set to the next unused sequence number, maintained by this LOADng Router;
- RREP.metric-type set to the same value as the RREQ.metric-type in the corresponding RREQ if the router knows the metric-type. Otherwise, RREP.metric-type is set to HOP\_COUNT;
- RREP.route-metric := 0
- RREP.hop-count := 0;
- RREP.hop-limit := maximum hop limit;
- RREP.destination := the address to which this RREP message will be sent; this corresponds to the RREQ.originator from the RREQ message, which this RREP message was created in response to;
- RREP.originator:= the LOADng Router's address for generating the RREP. If the LOADng Router is creating an RREP on behalf of the hosts connected to it, or one of the addresses in the LOADng Router's Destination Address Set, the host address is used.

### 5.2.2. RREP Processing

While processing the post, the variables hop-count and hop-limit were changed, and they are used.

When a LOADng Router receives an RREP message, it must process it as follows:

1. The message must be discarded without further processing if it is ineligible for processing. The message is not qualified to be forwarded..
2. Otherwise, the message is processed
3. If RREP.acknowledgement is required is set, the previous-hop must receive an RREP ACK message.
4. The message is not considered for forwarding if hop-count equals maximum hop count or hop-limit equals 0.

5. Otherwise, the RREP message is considered for forwarding if RREP.destination is not mentioned in any local interface tuple and does not relate to the destination address of any Destination Address Tuple of this LOADng Router.

### **5.2.3. RREP Forwarding**

When processing the message, the variables used-metric sort, hop-count, hop-limit, and route-metric were modified, and they were used to update the content of the message to be forwarded. The following is an RREP message that should be revised before being forwarded.

1. RREP.metric-type := used-metric-type
2. RREP.route-metric := route-metric
3. RREP.hop-count := hop-count
4. RREP.hop-limit := hop-limit
5. The RREP is transmitted,

After then, the RREP message is unicast to next hop, RREP.destination.

### **5.2.4. RREP Transmission**

An RREP is sent to the LOADng Router, which has the address specified in the RREP.destination field in either its Local Interface Set or its Destination Address Set. The RREP is forwarded to that LOADng Router in unicast mode. However, the RREP must be sent in order for it to be processed in each intermediate LOADng Router to set up appropriate forward route and Allow for an update to RREP.hop-count to reflect the route.

## **5.3. Route Errors (RERRs)**

If a LOADng Router fails to deliver a data packet to a next hop or a destination, and the data packet's source or destination addresses do not match the LOADng Router's destination address set, a Route Error must be generated (RERR). If the data packet was originated by a host behind that LOADng Router, this RERR must be transmitted down the Reverse Route to the source of the data packet for which delivery was failed to the last LOADng Router along the Reverse Route.

### **Identifying Invalid RERR Messages**

If any of the following conditions are true, a received RERR is invalid and must be deleted without further processing:

- The length of this message's address differs from the length of this LOADng Router's addresses.
- The RERR.originator address contains the address of this LOADng Router.

A LOADng Router may recognize additional reasons for determining that an RERR message is invalid for processing that are not covered by this specification, such as allowing a security protocol to do signature verification and preventing the protocol from processing unverifiable RERR messages.

### **5.3.1. RERR Generation**

The LOADng Router generates a packet with an RERR message when it detects a link breakdown with the following content:

- RERR.error-code := the error code associated with the occurrence that resulted in the RERR being generated,
- RERR.addr-length := the length of the address
- RERR.unreachableAddress := the destination address of a data packet that was not delivered correctly.
- RERR.originator := one address of the LOADng Interface of the LOADng Router that generates the RERR.
- RERR.destination := the source address from the unsuccessfully delivered data packet, towards which the RERR is to be sent.
- RERR.hop-limit := maximum hop limit;

### **5.3.2. RERR Processing**

The following notation is used during the RERR processing:

The address of the LOADng Router from which the RERR was received is known as previous-hop. Hop-limit is a variable that represents the hop-limit as specified in the RERR message that was received. A LOADng Router must take the following actions after receiving an RERR:

1. The RERR must be discarded without further processing if it is invalid for processing. The message is not eligible to be forwarded.
2. Included TLVs are processed/updated according to their specification.
3. Set the variable hop-limit to RERR.hop-limit - 1.

4. Find the Routing Tuple in the Routing Set where:
  - R\_dest\_addr = RERR.unreachableAddress
  - R\_next\_addr = previous-hop
5. If no matching Routing Tuple is found, the RERR is not processed further, but is considered for forwarding.
6. Otherwise, if one matching Routing Tuple is found:
  - If RERR.errorcode is 0 ("No available route"), this matching Routing Tuple is updated as follows:
    - + R\_valid\_time:= EXPIRED
 Extensions to this specification may define additional error codes in the Error Code IANA registry, and may insert processing rules here for RERRs with that error code.
  - If hop-limit is greater than 0, the RERR message is considered for forwarding

### 5.3.3. RERR Forwarding

An RERR is, ultimately destined either for the LOADng Router, which has, in its Destination Address Set or in its Local Interface Set, the address from RERR originator.

An RERR, considered for forwarding is therefore processed as follows:

1. RERR.hop-limit := hop-limit
2. Find the Destination Address Tuple (henceforth, matching Destination Address Tuple) in the Destination Address Set where:
  - \* D\_address = RERR.destination
3. If one or more matching Destination Address Tuples are found, the RERR message is discarded and not retransmitted, as it has reached the final destination.
4. Otherwise, find the Local Interface Tuple (henceforth, matching Local Interface Tuple) in the Local Interface Set where:
  - \* I\_local\_iface\_addr\_list contains RERR.destination.
5. If a matching Local Interface Tuple is found, the RERR message is discarded and not retransmitted, as it has reached the final destination.
6. Otherwise, if no matching Destination Address Tuples or Local Interface Tuples are found, the RERR message is transmitted

#### 5.3.4. RERR Transmission

An RERR is eventually sent to the LOADng Router with the address listed in the RERR.destination field in either its Local Interface Set or its Destination Address Set. The RERR is forwarded to that LOADng Router in unicast. The RERR, on the other hand, must be sent in order for it to be processed by each intermediary LOADng Router, allowing them to update their Routing Sets and remove tuples for this destination.

RERR Transmission is accomplished by the following procedure:

1. Find the Routing Tuple in the Routing Set, where:  
R\_dest\_addr = RERR.destination
2. Find the Local Interface Tuple, where:  
I\_local\_iface\_addr\_list contains R\_local\_iface\_addr from the Matching Routing Tuple
3. The RERR is transmitted over the LOADng Interface, identified by the Matching Interface Tuple to the neighbor LOADng Router, identified by R\_next\_addr from the Matching Routing Tuple.

#### 5.4. Route Reply Acknowledgments (RREP\_ACKs)

The RREP.ackrequired flag in a sent RREP must be set by a LOADng Router to indicate that it is anticipating an RREP ACK. When doing so, the LOADng Router must also add a tuple to the Pending Acknowledgment Set (P\_next hop, P\_originator, P\_seq num, P\_ack timeout) and set P\_ack timeout to current time + RREP\_ACK\_TIMEOUT.

##### Identifying Invalid RREP\_ACK Messages

If any of the following conditions are true, a received RREP ACK is invalid and must be deleted without further processing: The length of the address(es) of this LOADng Router differs from the length of the address(es) given by this message (i.e., RREP ACK.addrlength + 1).

A LOADng Router may recognize additional reasons for identifying that an RREP ACK message is invalid for processing that are not covered by this specification, such as allowing a security protocol to perform signature verification and preventing the protocol from processing unverifiable RREP ACK messages.

### 5.4.1. RREP\_ACK Generation

When a LOADng Router receives an RREP message with the RREP.ackrequired flag set, it must produce at least one RREP ACK and deliver it in unicast to the neighbor who sent the RREP. An RREP ACK message is generated by a LOADng Router with the following content:

- RREP\_ACK.addr-length: = the length of the address.
- RREP\_ACK.seq-num := the value of the RREP.seq-num field of the received RREP;
- RREP\_ACK.destination:= RREP.originator of the received RREP.

### 5.4.2. RREP\_ACK Processing

A LOADng Router must do the following when it receives an RREP ACK from a LOADng neighbor LOADng Router:

1. The RREP ACK must be destroyed without further processing if it is invalid for processing.
2. Find the Routing Tuple (henceforth, Matching Routing Tuple) where:
  - \* R\_dest\_addr = previous-hop;The Matching Routing Tuple is updated as follows:
  - \* R\_bidirectional:= TRUE
3. If a Pending Acknowledgement Tuple (henceforth, Matching Pending Acknowledgement Tuple) exists, where:
  - \* P\_next\_hop is the address of the LOADng Router from which the RREP\_ACK was received.
  - \* P\_originator = RREP\_ACK.destination
  - \* P\_seq\_num = RREP\_ACK.seq-num

Then the RREP has been acknowledged. The Matching Pending Acknowledgement Tuple is updated as follows:

\* P\_ack\_received := TRUE

\* P\_ack\_timeout := EXPIRED

### 5.4.3. RREP\_ACK Forwarding

An RREP\_ACK is intended only for a specific direct neighbor, and must not be forwarded.

#### 5.4.4. RREP\_ACK Transmission

An RREP\_ACK is transmitted in unicast to the neighbor LOADng Router from which the RREP was received.

#### Common rules for RREQ and RREP messages

The structure of RREQ and RREP messages is identical, and the processing steps are similar. The algorithms explain how to send this type of message in a standard way. The LOADng code can be significantly simplified by sharing some of the processing algorithms.

#### Identifying valid RREQ and RREP Messages

When a LOADng router receives an RREQ or RREP packet, it must first determine if the message is legitimate for processing. In certain cases, a received RREQ is invalid for processing and must be discarded. Additional reasons can be recognized by a LOADng router to determine that an RREQ is invalid for processing.

#### Identifying valid RREQ and RREP Messages

*procedure isValidMessage(loadng\_packet)*

```
if <originator> contains an address of this router then  
return false  
end if  
repeat //  
until R_dest_addr = <originator> and R_seq_num > <seq-num>  
if matching Routing Tuple found then  
return false  
end if  
if <metrics> ≠ interface metrics then  
return false  
end if  
if some TLVs required by the metric are absent then  
return false  
end if  
if <type> = RREQ_TYPE and previous-hop is blacklisted then  
return false  
end if  
return true  
end procedure
```

## RREQ and RREP Processing algorithm

**Require:** *isValidMessage(loadng\_packet)* = **true**

**procedure** *CommonProcess\_RREQ\_or\_RREP(loadng\_packet)*

*Process\_TLV(<tlv\_block>)*

**if** *packet received over weak link* **then**

*<weak\_links> <weak\_links> + 1*

**end if**

**repeat**

**until** *R\_dest\_addr = <originator>*

**if** *no matching Routing Tuple found* **then**

*R\_dest\_addr <originator>*

*R\_next\_addr previous-hop*

*R\_dist MAX\_DIST*

*R\_seq\_num -1*

*R\_valid\_time current time + R\_HOLD\_TIME*

**end if**

**if**

*{ (<route-cost>, <weak-links>, (<tlv>)\*) < R\_dist and R\_seq\_num = <seq-num> }*

**or** *R\_seq\_num > <seq-num>* **then**

*R\_next\_addr previous-hop*

*R\_dist (<route-cost>, <weak-links>, (<tlv>)\*)*

*R\_seq\_num <seq-num>*

*R\_valid\_time current time + R\_HOLD\_TIME*

**repeat**

**until** *R\_dest\_addr = previous-hop*

**if** *no matching Routing Tuple found* **then**

*R\_dest\_addr previous-hop*

*R\_next\_addr previous-hop*

*R\_dist MAX\_DIST*

*R\_seq\_num -1*

*R\_valid\_time current time + R\_HOLD\_TIME*

**end if**

**return true**

**else**

**return false**

**end if**

**end procedure**

## LOADng Packet Processing

When receiving a LOADng packet, it is processed according to the packet type.



## **RREQ Message Processing algorithm**

```
procedure Process_RREQ(loadng_packet)  
  
if isValidMessage(loadng_packet)= false  
then  
return false  
else if CommonProcess_RREQ_or_RREP(loadng_packet)= false  
then  
return false  
end if  
if <destination> ≠ an address of this router then  
return true  
else  
Generate_RREP(loadng_packet)  
return false  
end if  
end procedure
```

A RREP is created when the RREQ arrives at its destination. The function Generate RREP considers as an RREP is generated in response to the LOADng packet and then unicast to the next hop along the reverse route towards the originator of the RREQ.

## **RREP Message Processing algorithm**

```
procedure Process_RREP(loadng_packet)  
  
if isValidMessage(RREP)= false then  
return false  
else if CommonProcess_RREQ_or_RREP(loadng_packet)= false  
then  
return false  
end if  
if ACK-REQUIRED flag = 1 then  
Send_RREP_ACK(loadng_packet)  
end if  
if <destination> ≠ an address of this router then  
return true  
end if  
end procedure
```

The function Send\_RREP\_ACK sends by unicast a RREP\_ACK to the previous hop neighbor from which received the RREP.

## **RERR Message Processing algorithm**

*procedure Process\_RERR(loadng\_packet)*

*Process\_TLV(loadng\_packet)*

**repeat**

*Read Routing Tuple in the Routing Set*

**until**

*R\_dest\_addr = <destination> and R\_next\_addr = previous-hop*

**if** *no matching Routing Tuple found* **then**

**return false**

**else if** *matching Routing Tuple found* **then**

*R\_valid\_time expired*

**return true**

**end if**

**end procedure**

## **RREP-ACK Message processing**

On receiving a RREP-ACK from a LOADng neighbor router, a LOADng router should run the algorithms and in which it updates its routing set if necessary.

The function CheckPending checks whether a corresponding RREP is pending, i.e. if the Pending Acknowledgment Set contains a tuple (P\_next\_hop, P\_originator, P\_seq\_num, P\_ack\_timeout) such as:

- ✓ P\_next\_hop is the address of the LOADng neighbor router from which the RREP-ACK was received;
- ✓ P\_originator corresponds to the <originator> field of the RREP-ACK;
- ✓ P\_seq\_num corresponds to the <seq-num> field of the RREP-ACK.

## **RREP-ACK Processing algorithm**

*procedure Process\_RREP-ACK(loadng\_packet)*

*Process\_TLV(loadng\_packet)*

*CheckPending(neighbor,originator,seq-num)*

**if** *matching Tuple is found* **then**

**end if**

**end procedure**

## **LOADng Packet Forwarding**

After message processing, and if it still continues being processed (i.e., if the correspondent processing function returns **true** value), the message is considered to be forwarded.

### **RREQ Forwarding**

A Route Request (RREQ), considered for forwarding, must be updated as follows and prior to it being transmitted. The function `UpdateRouteCost` updates `<route-cost>` field according to the cost associated with the interface over which the RREQ is transmitted, and according to the specification of the `<metrics>` included in the RREQ

The function `Forward_RREQ` forwards the RREQ message. RREQ forwarding may be undertaken using classic flooding, may employ a reduced relay set mechanism such as Simplified Multicast Forwarding or any other information diffusion mechanism such as the Trickle Algorithm.

### **RREQ Forwarding algorithm**

***Require:** `Process_RREQ(loadng_packet)= true`*

***procedure** `Consider_Forwarding_RREQ(loadng_packet)`*

*`UpdateRouteCost(interface, <metrics>)`*

*`Forward_RREQ(loadng_packet)`*

***end procedure***

### **RREP Forwarding**

A Route Reply (RREP) message, considered for forwarding, must be updated as follows, prior to it being transmitted:

### **RREP Forwarding algorithm**

***Require:** `Process_RREP(loadng_packet)= true`*

***procedure** `Consider_Forwarding_RREP(loadng_packet)`*

*`Update_RouteCost(interface, metrics)`*

***if** `interface uses RREP-ACKs` **then***

*`ACK_REQUIRED flag ← 1`*

*end if*

*Forward\_RREP(loadng\_packet)*

*end procedure*

Note that if this interface of the LOADng router uses RREP-ACKs to check the bi-directionality of the links, the ACK\_REQUIRED flag must be set to 1. The Forward RREP function unicasts the RREP message to the next hop towards the RREP's specified destination.

### **RERR Forwarding**

The LOADng router, which contains the address from the source field in its Destination Address Set, is the final destination for an RERR. As a result, an RERR that is being considered for forwarding is processed as follows. The Forward RERR function unicasts the RERR message to the next hop, which is the source specified in the RERR.

### **RERR Forwarding algorithm**

**Require:** *Process\_RERR (loadng\_packet) = true*

*procedure Consider\_Forwarding\_RERR(loadng\_packet)*

*repeat*

*Read Routing Tuple in the Routing Set*

*until D\_address = <source>*

*if no matching Routing Tuple found then*

*discard the RERR*

*else*

*Forward\_RERR(loadng\_packet)*

*end if*

*end procedure*

## Chapter Six

### Simulation and Result Analysis

Simulation is a significant technology in today's world. Computer simulations can be used to model hypothetical and real-life objects so that they can be examined. On the computer, the network is likewise emulated. A network simulator is a program that simulates a network on a computer. The network's behavior is determined in one of two ways: by interconnecting network components using mathematical formulas, or by capturing and replaying observations from a production network.

Researchers can use the network simulator to test scenarios that are difficult or expensive to mimic in the real world. It's especially handy for testing new networking protocols or making changes to current ones in a controlled and repeatable manner. Diverse types of nodes, such as hosts, hubs, bridges, routers, and mobile units, can be used to create various network topologies.

There are various types of network simulators that can be compared. Based on ranging from the very simple to the very complex, specifying the nodes and the links between those nodes as well as the traffic between the nodes, specifying everything about the protocols used to handle traffic in a network, and specifying everything about the protocols used to handle traffic in a network. Users can easily visualize the workings of their simulated environment with graphical applications, while text-based applications allow for more advanced customization. Programming-oriented tools provide a programming framework that can be customized to create an application that simulates the networking environment to be assessed.

There are different network simulators with different features. Some of the network simulator are OPNET, NS2, NS3, NetSim, OMNeT++, REAL, J-Sim and QualNet. Of this, the NS3 simulators can be used.

#### 6.1 Model for WSN Simulations

Their relevant models have been introduced in tandem with the development of WSN simulation tools. New components, including as detailed power and energy consumption models and environment models, are included in the models that are not included in traditional network simulators.

### 6.1.1 Network Model

The network model is a database model designed to represent objects and their relationships in a flexible fashion. The schema is its unique feature. Basically the WSN model contains Nodes, Environments, Radio channel, Sink nodes and Agents. The models of WSN is shown in the figure below.

*Nodes:* Each node is a physical device that continuously monitors a set of physical parameters. A common radio channel is used to communicate between nodes. A protocol stack manages communications on the inside. Sensor nodes, unlike traditional network models, feature a second set of components: the physical node tier, which is connected to the outside world. Nodes are frequently seen in a two-dimensional or three-dimensional environment. Node coordinates may be controlled by a separate topology component. A WSN might have anywhere from a few to hundreds of nodes, depending on the application and deployment circumstances.

*Environments:* The extra environment component is the fundamental distinction between traditional and WSN models. This component simulates the generation and propagation of events sensed by the nodes that cause sensor actions, i.e. network communication. The events of interest are usually of a physical magnitude, such as sound waves, seismic waves, or temperature changes.

*Radio channel:* It describes how radio signals are transmitted between nodes in a network. A terrain component is used in more realistic models, and it is coupled to the environment and radio channel components. The physical magnitude of the radio channel is influenced by the terrain component, which is taken into account while computing the propagation as part of the radio channel.

*Sink nodes:* If present, these special nodes receive data from the network and process it. They might probe sensors for information on a particular occurrence of interest. The use of sinks is determined by the application and the simulator's testing.

*Agents:* For the nodes, a generator of interesting occurrences. The agent may generate a physical magnitude variation that propagates across the surroundings and triggers the sensor. When the behavior of this component can be implemented independently of the environment, such as in a mobile vehicle, it is advantageous. Otherwise, events can be triggered by the environment.

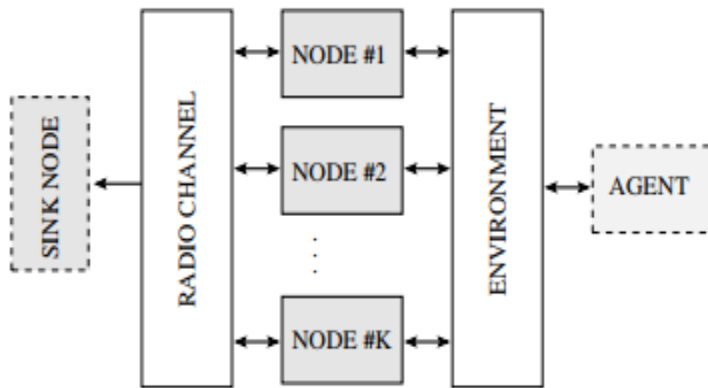


Figure 15 Wireless Sensor Network Model

### 6.1.2 Node Model

Interactions between components produce cross-layer interdependencies, which affect node behavior. The division of a node into abstract tiers is a useful approach to explain it. Protocol tier, Physical node tier, and Media tier are the three main levels.

*Protocol tier:* All communication protocols are included in the Protocol-tier. This tier usually has three layers: a MAC layer, a routing layer, and a specialized application layer. It's worth noting that the protocol tier's functioning is frequently influenced by the status of the physical tier; for example, a routing layer can consider battery limits when deciding on a packet path. As a result, an efficient technique for exchanging tier data must be established.

*Physical node:* The hardware platform and its influence on equipment performance are represented by the physical-node tier. Depending on the application, the actual makeup of this layer may vary. The set of physical sensors, the energy module, and the mobility module are all common elements in this tier. The behavior of the monitoring hardware is described by physical sensors. The energy module replicates power consumption in component hardware, which is an important factor to consider when evaluating WSNs. The sensor position is controlled by the mobility module.

*Media tier:* The media-tier connects the node to the outside world. A node communicates with its surroundings via a radio channel and one or more physical channels. Environmental events are received through physical pathways.

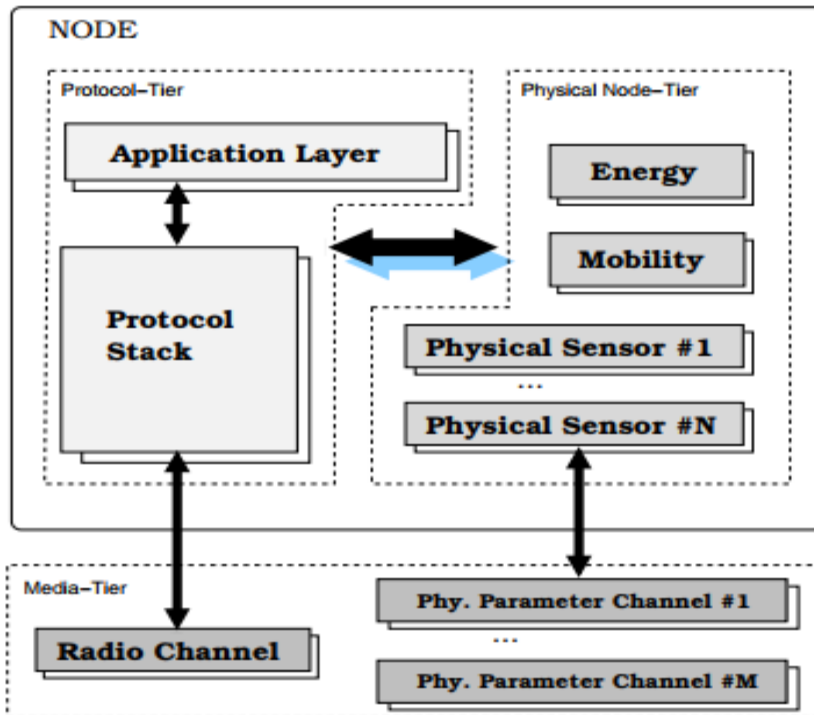


Figure 16 Tier based Node model

## 6.2 NS3 (Network Simulator-3)

The NS-3 is a discrete-event simulator. Its creation was motivated by a desire to advance communication network research. In June 2008, the open source simulator NS-3 was released.

NS-3 is not an expansion of the NS-2 simulator; it is a new simulator that does not support any NS-2 APIs. Because NS-2 programs are written in OTcl and the results can be seen using NAM and XGraph, pure C++ code is not possible. All of the applications in NS-3, on the other hand, are written in pure C++, with optional Python bindings. Although NS-3 lacks a Graphical Tool, graphical results can still be interpreted using the open source NetAnim software.

In terms of wired topology, NS-3 gives a device model of a simple Ethernet network that employs the CSMA/CD protocol scheme with exponentially growing back off to compete for the shared transmission medium. In terms of Wireless Sensor Networks, NS-3 already has modules such as 802.15.4, 6LoWPAN, and RPL. Through the use of Cygwin, NS-3 may run on a variety of operating systems, including Linux and Windows. Basically NS3 simulation contains two programming languages such as C++ and Python.



C++: NS-3 is a library that may be statically or dynamically linked to a C++ main application to implement simulation and core model. These libraries specify the commencement of simulation as well as the topology of simulation.

*Python:* Python wrap C++ programs. Python programs are used to import an ns3 module. The components of ns3 are shown as follows.

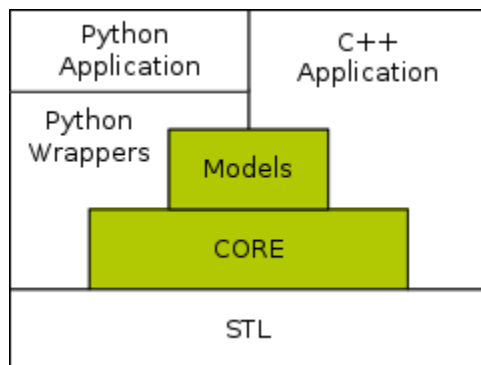


Figure 17 How C++ and Python codes are integrated in NS3

## NetAnim

NetAnim is a Network Animator that comes with ns3 already installed. It's an offline network animator that's included in the ns-allinone-3 package now. It can use an XML trace file generated as an output during simulation to animate the ns-3 network simulation. As a result, all of the essential procedures for creating this XML trace file and configuring its characteristics should be completed within the ns-3 simulation code. NetAnim may fail to compile during the ns3 compilation process.

Using NetAnim is a two-step process.

Step 1: Generate the animation XML trace file during simulation using "ns3::AnimationInterface" in the ns-3 code base

Step 2: Load the XML trace file generated in Step 1 with the offline animator (NetAnim).

### 6.3. Result Analysis

There are several metrics, which can be used while measuring the performances of network. The result of this work is analyzed in some metrics or parameters such as packet delivery ratio, average end-to-end delay and Normalized routing overhead. These metrics have been described and simulated as follows.

#### 6.3.1. Packet Delivery Ratio (PDR)

Throughout the simulation, the Packet Delivery Ratio is the ratio of the total number of successfully received packets by the destination nodes to the total number of packets sent by the source nodes. It also defines the packet loss rate, which has an effect on the network's overall throughput capacity [58]. Formula to calculate Packet Delivery Ratio is as follows:

$$\text{PDR} = (\text{Pr} / \text{Ps}) * 100$$

Where PDR = Packet Delivery Ratio

Pr = Total number of packets received

Ps = Total number of packets sent

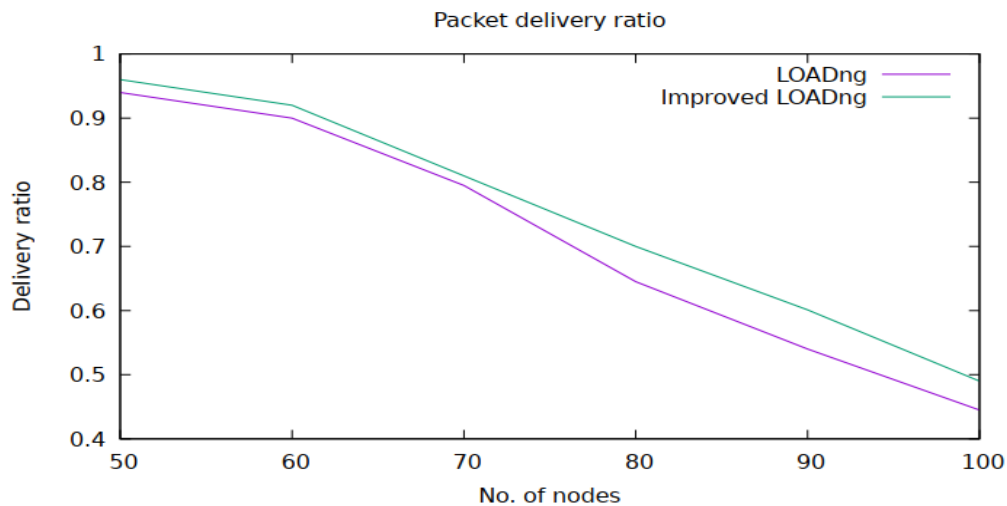


Figure 18 Packet Delivery ratio

### 6.3.2. Average End to End Delay

Average end-to-end is the average delay in packet transmission between both nodes is known as delay. In other word end-to-end delay is the time required, in which the packet to be traversed from source to destination in the network and is measured in seconds [59]. A higher end-to-end delay value indicates that the network is congested and that the routing protocol is performing poorly. Average end-to-end delay is calculated using following formula:

$$AD = \Sigma (T_a - T_s) / n$$

Where AD = Average Delay

$T_a$  = Arrival time of packet

$T_s$  = Start time of packets

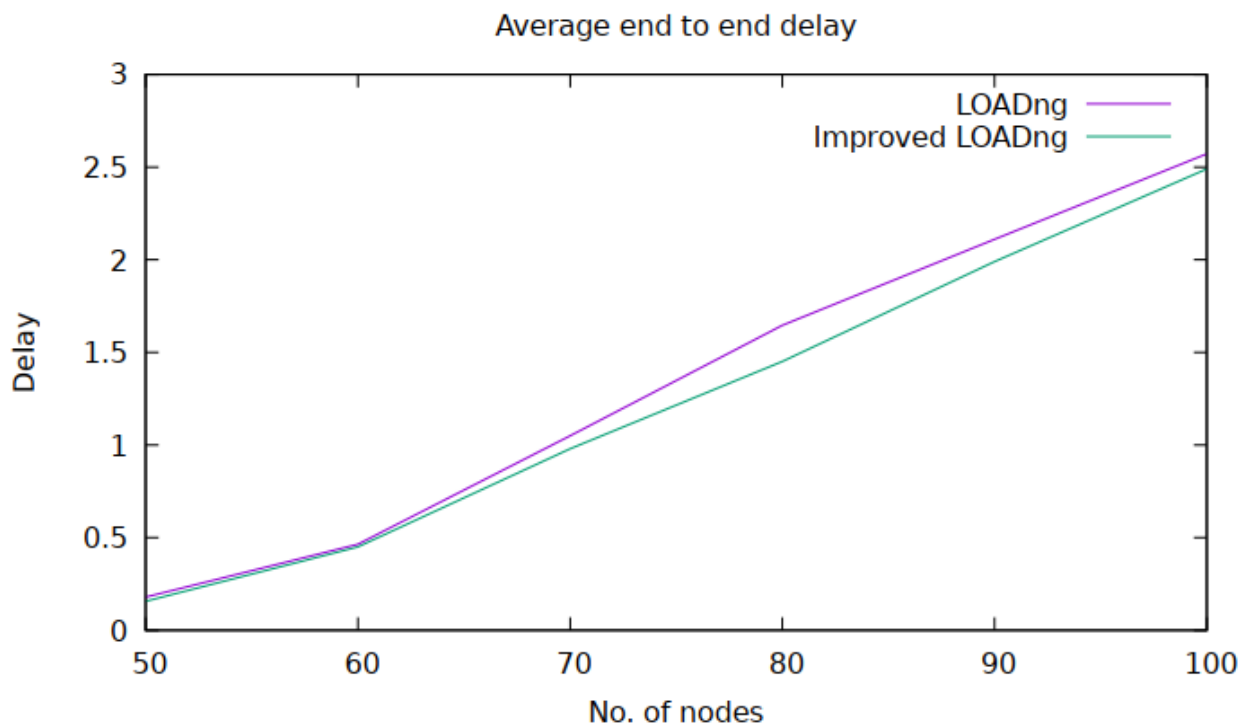


Figure 19 Average end-to-end delay

### 6.3.3. Normalized Routing Overhead

The ratio between the number of routing packets sent and the number of packets actually received is used to measure normalized routing overhead, which is an accounting of dropped packets. The higher the NRO value, the higher the overheads of routing packets and, as a result, the protocol's

efficiency. The total number of transmitted routing packets divided by the number of delivered data packets at destination is known as NRO. A routing packet's hop-by-hop transmission is counted as one transmission. It is the number of all network control packets sent by all nodes in order to discover and maintain a path. The formula to calculate Normalized Routing Overhead is:

$$\text{NRO} = \text{Pro} / \text{Pre}$$

Where NRO = Normalized Routing Overhead

Pro = Routing Packets

Pre = Received Packets

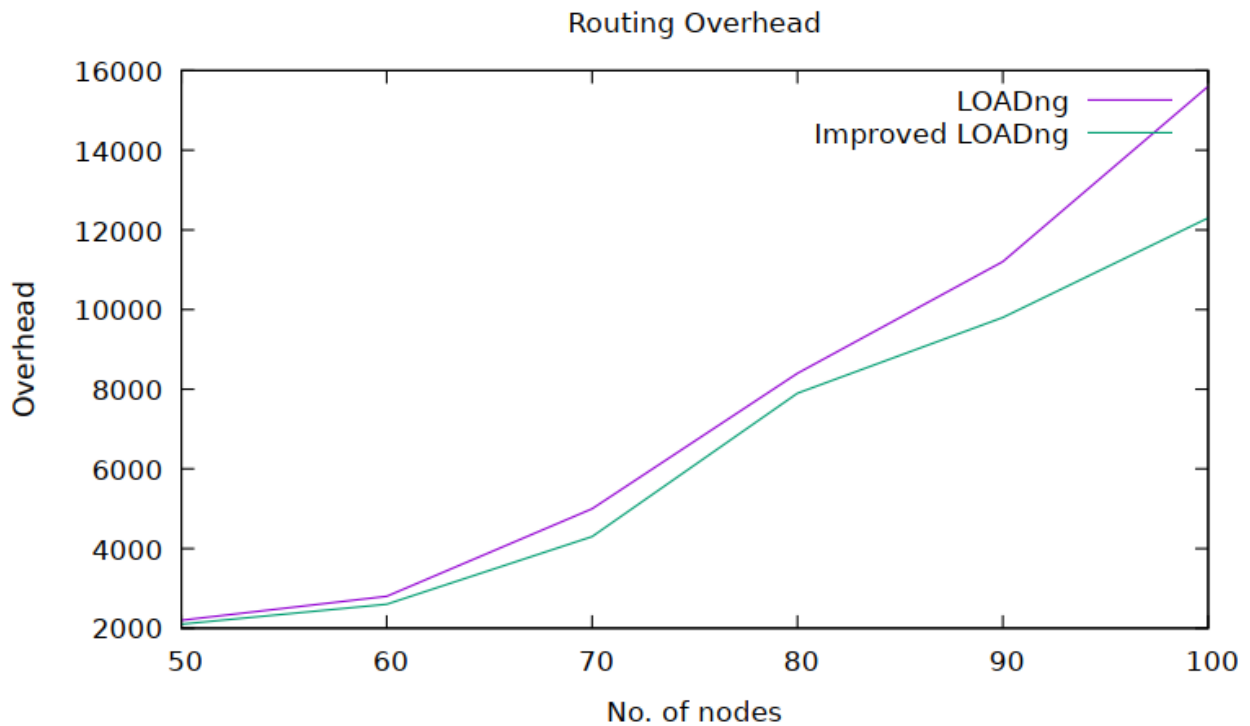


Figure 20 Route Overhead

# Chapter Seven

## Conclusions and Recommendations

### 7.1 Introduction

The RPL protocol was developed by the IETF RoLL (Routing over Low Power and Lossy) working group in order to address relevant routing difficulties (IPv6 Routing Protocol for LLNs). RPL is a proactive routing protocol that uses a directed acyclic network to transmit messages between nodes. LLNs, despite their widespread use, face a number of challenges in providing an effective and dependable service for these new applications. One of the most essential LLN difficulties is to develop a routing protocol capable of meeting the applications' requirements. A routing protocol is in charge of creating and maintaining paths that devices must take for data flow throughout the network. Furthermore, routing protocols should be built to meet a various needs based on network and application specifics. Since the introduction of LOADng as an alternative to RPL, a number of research comparing the two techniques have been published. The LOADng protocol takes a reactive approach, assuming that LLNs remain idle the majority of the time since it can use a proactive approach and would result in extra overhead. The LOADng only establishes a route to a specific destination when there is data to transfer.

### 7.2 Conclusion

In this work the improved LOADng protocols has been analyzed in wireless sensor network scenarios. This protocol is get popularity in recent years because of its simplicity for low power and Lossy networks like wireless sensor networks. To improve the problems in original LOADng protocols this work used the agglomerative hierarchical clustering approach for routing. The work used Euclidean distance to calculate the distance between two nodes and after calculating the Euclidean distance, distance matrix is used to put/store distances in between every nodes. The newly proposed work are more efficient on packet delivery ratio, average end to end delay and routing overhead. The following contributions are added on this work.

- Introducing the use of Agglomerative hierarchical algorithm in LOADng protocol
- Reduce the energy wastage of nodes in the network for route discovery
- Minimizes the memory usages and improving the network life time ignoring broadcasting
- Fast delivery of packets

### **7.3 Future Work**

In this work, the LOADng has been improved using agglomerative hierarchical clustering algorithms for routing, Euclidian distance to calculate the distance between nodes and using distance matrix as routing table. Whenever our work is efficient compared in different metrics with original LOADng, we still recommend other researchers who want to do their research in another clustering algorithms or another algorithms such as Machine Learning or deep learning how to determine the node status whether died, normal or malicious in different threshold.

## Reference

- [1] M. A. Matin and M. M. Islam, "Overview of Wireless Sensor Network," p. 23.
- [2] T. Clausen, J. Yi, and U. Herberg, "Lightweight On-demand Ad hoc Distance-vector Routing - Next Generation (LOADng): Protocol, extension, and applicability," *Computer Networks*, vol. 126, pp. 125–140, Oct. 2017, doi: 10.1016/j.comnet.2017.06.025.
- [3] L. Yong-Min, W. Shu-Ci, and N. Xiao-Hong, "The Architecture and Characteristics of Wireless Sensor Network," in *2009 International Conference on Computer Technology and Development*, Kota Kinabalu, Malaysia, 2009, pp. 561–565. doi: 10.1109/ICCTD.2009.44.
- [4] D. Kandris, C. Nakas, D. Vomvas, and G. Koulouras, "Applications of Wireless Sensor Networks: An Up-to-Date Survey," *ASI*, vol. 3, no. 1, p. 14, Feb. 2020, doi: 10.3390/asi3010014.
- [5] I. Y. Mohammed, "Comparative analysis of proactive & reactive protocols for cluster based routing algorithms in WSNs," p. 12, 2019.
- [6] "H. Aznaoui, S. Raghay, L. Aziz and A. Ait-Mlouk, A comparative study of routing protocols in WSN, 2015 5th International Conference on Information & Communication Technology and Accessibility (ICTA), Marrakech, Morocco, 2015,.pdf."
- [7] P. Maurya and A. Kaur, "A Survey on Descendants of LEACH Protocol," p. 13, 2016.
- [8] N. S. Samaras and F. Triantari, "On Direct Diffusion Routing for Wireless Sensor Networks," p. 6.
- [9] V. S. Patel and C. R. Parekh, "SURVEY ON SENSOR PROTOCOL FOR INFORMATION VIA NEGOTIATION (SPIN) PROTOCOL," p. 4.
- [10] Y. Yu, R. Govindan, and D. Estrin, "Geographical and Energy Aware Routing: a recursive data dissemination protocol for wireless sensor networks," p. 11.
- [11] R. Beniwal, K. Nikolova, and G. Iliev, "MM-SPEED: Multipath Multi-SPEED Routing Protocol in 6LoWPAN Networks," p. 4.
- [12] O. Buyanjargal and Y. Kwon, "An Energy Efficient Clustering Algorithm for Event-Driven Wireless Sensor Networks (EECED)," p. 6.
- [13] J. Gnanambigai, R. Vasanth, and M. R. Gokul, "IMPLEMENTATION OF HYBRID CLUSTERING BASED ROUTING PROTOCOL FOR WIRELESS SENSOR NETWORKS," p. 6, 2014.
- [14] J. Kulshrestha and M. K. Mishra, "DPEGASIS: Distributed PEGASIS for chain construction by the nodes in the network or in a zone without having global network topology information.," p. 5.
- [15] J. Grover and M. Sharma, "Location Based Protocols in Wireless Sensor Network – A Review," p. 5, 2014.
- [16] M. U. Bokhari, Y. K. Tamandani, and Q. Makki, "A comprehensive study of position based routing protocols in WSNs," p. 6.
- [17] A. Manjeshwar and D. P. Agrawal, "TEEN: A Routing Protocol for Enhanced Efficiency in Wireless Sensor Networks," p. 7, 2001.
- [18] Y. Ge, "Optimization on TEEN routing protocol in cognitive wireless sensor network," p. 9, 2018.
- [19] Y. Yao, Q. Cao, and A. V. Vasilakos, "EDAL: An Energy-Efficient, Delay-Aware, and Lifetime-Balancing Data Collection Protocol for Heterogeneous Wireless Sensor Networks," p. 14.

- [20] N. Javaid, "M-ATTEMPT: A New Energy-Efficient Routing Protocol for Wireless Body Area Sensor Networks," *Procedia Computer Science*, p. 8, 2013.
- [21] P. Kuila, "Energy efficient clustering and routing algorithms for wireless sensor networks\_ Particle swarm optimization approach," *Engineering Applications of Artificial Intelligence*, p. 14, 2014.
- [22] M. Yoon, Y.-K. Kim, and J. Chang, "An Energy-efficient Routing Protocol using Message Success Rate in Wireless Sensor Networks," no. 1, p. 8, 2013.
- [23] T. Ali, "Diagonal and Vertical Routing Protocol for Underwater Wireless Sensor Network," p. 8, 2014.
- [24] P. Kuila and P. K. Jana, "Energy efficient fault tolerant clustering and routing algorithms for wireless sensor networks," p. 14, 2014.
- [25] J. Yu, "A cluster-based routing protocol for wireless sensor networks with nonuniform node distribution," p. 8, 2012.
- [26] T. N. Qureshi, "BEENISH: Balanced Energy Efficient Network Integrated Super Heterogeneous Protocol for Wireless Sensor Networks," *Procedia Computer Science*, p. 6, 2013.
- [27] N. Javaid, "EDDEEC: Enhanced Developed Distributed Energy-efficient Clustering for Heterogeneous Wireless Sensor Networks," *Procedia Computer Science*, p. 6, 2013.
- [28] S. Ganesh and R. Amutha, "Efficient and Secure Routing Protocol for Wireless Sensor Networks through SNR Based Dynamic Clustering Mechanisms," *JOURNAL OF COMMUNICATIONS AND NETWORKS*, vol. 15, no. 4, p. 8, 2013.
- [29] P. Kuila and P. K. Jana, "Energy Efficient Load-Balanced Clustering Algorithm for Wireless Sensor Networks," *Procedia Technology*, p. 7, 2012.
- [30] B. Elbhiri, S. Rachid, and D. Aboutajdine, "Developed Distributed Energy-Efficient Clustering (DDEEC) for heterogeneous wireless sensor networks," p. 4.
- [31] M. Abdullah and A. Ehsan, "Routing Protocols for Wireless Sensor Networks: Classifications and Challenges," p. 12.
- [32] Association for Computing Machinery, Ed., *Mobile computing and networking 2001: proceedings of the Seventh Annual International Conference on Mobile Computing and Networking ; July 16 - 21, 2001, Rome, Italy*. New York, NY: ACM Order Dep, 2001.
- [33] A. Manjeshwar and D. P. Agrawal, "APTEEN: A Hybrid Protocol for Efficient Routing and Comprehensive Information Retrieval in Wireless Sensor Networks," p. 8, 2002.
- [34] S. Bhagyashree, S. Prashanthi, and D. K. M. Anandkumar, "Enhancing Network Lifetime in Precision Agriculture using Aptein Protocol," p. 5, 2015.
- [35] J. V. V. Sobral, J. J. P. C. Rodrigues, R. A. L. Rabêlo, K. Saleem, and V. Furtado, "LOADng-IoT: An Enhanced Routing Protocol for Internet of Things Applications over Low Power Networks," *Sensors*, vol. 19, no. 1, p. 150, Jan. 2019, doi: 10.3390/s19010150.
- [36] D. Sasidharan and L. Jacob, "Improving network lifetime and reliability for machine type communications based on LOADng routing protocol," *Ad Hoc Networks*, vol. 73, pp. 27–39, May 2018, doi: 10.1016/j.adhoc.2018.02.007.
- [37] J. V. V. Sobral, J. J. P. C. Rodrigues, and A. Neto, "Performance Assessment of the LOADng Routing Protocol in Smart City Scenarios," in *2017 IEEE First Summer School on Smart Cities (S3C)*, Natal, Aug. 2017, pp. 49–54. doi: 10.1109/S3C.2017.8501394.
- [38] F. Marcuzzi and A. M. Tonello, "Artificial-Intelligence-Based Performance Enhancement of the G3-PLC LOADng Routing Protocol for Sensor Networks," in *2019 IEEE International*

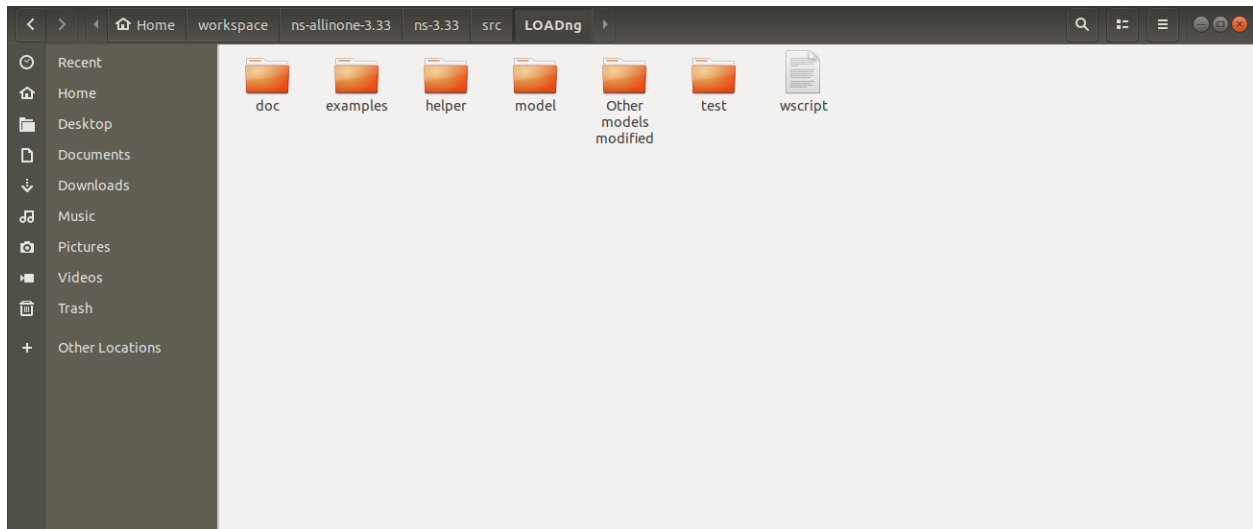


- Symposium on Power Line Communications and its Applications (ISPLC)*, Praha, Czech Republic, Apr. 2019, pp. 1–6. doi: 10.1109/ISPLC.2019.8693390.
- [39] T. Clausen and J. Yi, “Path Accumulation Extensions for the LOADng Routing Protocol in Sensor Networks,” in *Internet of Vehicles – Technologies and Services*, vol. 8662, R. C.-H. Hsu and S. Wang, Eds. Cham: Springer International Publishing, 2014, pp. 150–159. doi: 10.1007/978-3-319-11167-4\_15.
- [40] A. J. R. Gonçalves, R. A. L. Rabêlo, J. J. P. C. Rodrigues, and L. M. L. Oliveira, “A mobility solution for low power and lossy networks using the LOADng protocol,” *Trans Emerging Tel Tech*, vol. 31, no. 12, Dec. 2020, doi: 10.1002/ett.3878.
- [41] J. V. V. Sobral, N. Kumar, C. Zhu, and R. W. Ahmad, “Performance Evaluation of Routing Metrics in the LOADng Routing Protocol,” *JOURNAL OF COMMUNICATIONS SOFTWARE AND SYSTEMS*, vol. 13, no. 2, p. 9, 2017.
- [42] D. Sasidharan and L. Jacob, “Energy and bandwidth efficient multipath-enhanced LOADng routing protocol,” in *2016 Twenty Second National Conference on Communication (NCC)*, Guwahati, India, Mar. 2016, pp. 1–6. doi: 10.1109/NCC.2016.7561118.
- [43] F. Van Trimont, G. B. Katumba, V. Moeyaert, and S. Bette, “Impact of the weak link count mechanism on G3-PLC LOADng routing protocol,” in *2015 IEEE International Symposium on Power Line Communications and Its Applications (ISPLC)*, Austin, TX, USA, Mar. 2015, pp. 107–112. doi: 10.1109/ISPLC.2015.7147598.
- [44] T. H. Clausen, “The LLN On-demand Ad hoc Distance-vector Routing Protocol - Next Generation (LOADng),” p. 35.
- [45] J. Tripathi, “Proactive versus reactive routing in low power and lossy networks: Performance analysis and scalability improvements,” *Ad Hoc Networks*, p. 24, 2014.
- [46] T. Clausen, J. Yi, A. Bas, and U. Herberg, “A Depth First Forwarding (DFF) Extension for the LOADng Routing Protocol,” p. 5.
- [47] J. V. V. Sobral, “Multicast improvement for LOADng in Internet of Things networks,” p. 14, 2019.
- [48] X. Yu, P. Wu, W. Han, and Z. Zhang, “A survey on wireless sensor network infrastructure for agriculture,” p. 6, 2013.
- [49] O. Boyinbode, H. Le, and M. Takizawa, “A survey on clustering algorithms for wireless sensor networks,” p. 7.
- [50] R. Rajagopalan and P. K. Varshney, “Data aggregation techniques in sensor networks: A survey,” p. 31.
- [51] K. Maraiya, K. Kant, and N. Gupta, “Wireless\_Sensor\_Network\_A\_Review\_on\_Data\_Aggregation,” vol. 2, no. 4, p. 6, 2011.
- [52] S. Lee, “Connectivity restoration in a partitioned wireless sensor network with assured fault tolerance,” p. 19, 2014.
- [53] S. H. Lee, S. Lee, H. Song, H. S. Lee, and R. Army, “Gradual Cluster Head Election for High Network Connectivity in Large-Scale Sensor Networks,” p. 5, 2011.
- [54] M. Yu, H. Mokhtar, and M. Merabti, “A Survey of Network Management Architecture in Wireless Sensor Network,” p. 6.
- [55] N. M. Freris and P. R. Kumar, “Fundamentals of Large Sensor Networks: Connectivity, Capacity, Clocks, and Computation,” p. 19.
- [56] Y. Liao, H. Qi, and W. Li, “Load-Balanced Clustering Algorithm With Distributed Self-Organization for Wireless Sensor Networks,” *IEEE SENSORS JOURNAL*, vol. 13, no. 5, p. 9, 2013.

- [57] D. Sahin, "Quality-of-service differentiation in single-path and multi-path routing for wireless sensor network-based smart grid applications," p. 46.
- [58] M. K. U. Khan and K. S. Ramesh, "Effect on Packet Delivery Ratio (PDR) & Throughput in Wireless Sensor Networks Due to Black Hole Attack," vol. 8, no. 12, p. 5, 2019.
- [59] H.-N. Nguyen, T. Begin, A. Busson, and I. G. Lassous, "Evaluation of an End-to-End Delay Estimation in the Case of Multiple Flows in SDN Networks," p. 6.

## Appendices

Some of implementation details are the following



The helper codes used

LOADnghelper.cc

```
1 #include "LOADng-helper.h"
2 #include "ns3/LOADng-routing-protocol.h"
3 #include "ns3/node-list.h"
4 #include "ns3/names.h"
5 #include "ns3/ipv4-list-routing.h"
6
7 namespace ns3 {
8   LOADngHelper::~LOADngHelper ()
9   {
10  }
11
12   LOADngHelper::LOADngHelper () : Ipv4RoutingHelper ()
13   {
14     m_agentFactory.SetTypeId ("ns3::LOADng::RoutingProtocol");
15   }
16
17   LOADngHelper*
18   LOADngHelper::Copy (void) const
19   {
20     return new LOADngHelper (*this);
21   }
22
23   Ptr<Ipv4RoutingProtocol>
24   LOADngHelper::Create (Ptr<Node> node) const
25   {
26     Ptr<LOADng::RoutingProtocol> agent = m_agentFactory.Create<LOADng::RoutingProtocol> ();
27     node->AggregateObject (agent);
28     return agent;
29 }
30
31 void
32 LOADngHelper::Set (std::string name, const AttributeValue &value)
33 {
34   m_agentFactory.Set (name, value);
35 }
36
37 }
38
```

Wireless sensor network helper code

## Wsnhelper.cc

```
1  #include "wsn-helper.h"
2  #include "ns3/inet-socket-address.h"
3  #include "ns3/packet-socket-address.h"
4  #include "ns3/string.h"
5  #include "ns3/data-rate.h"
6  #include "ns3/uinteger.h"
7  #include "ns3/names.h"
8  #include "ns3/random-variable-stream.h"
9  #include "ns3/wsn-application.h"
10
11 namespace ns3 {
12
13 WsnHelper::WsnHelper (std::string protocol, Address address)
14 {
15     m_factory.SetTypeId ("ns3::WsnApplication");
16     m_factory.Set ("Protocol", StringValue (protocol));
17     m_factory.Set ("Remote", AddressValue (address));
18 }
19
20 void
21 WsnHelper::SetAttribute (std::string name, const AttributeValue &value)
22 {
23     m_factory.Set (name, value);
24 }
25
26 ApplicationContainer
27 WsnHelper::Install (Ptr<Node> node) const
28 {
29     return ApplicationContainer (InstallPriv (node));
30 }
31
32 ApplicationContainer
33 WsnHelper::Install (std::string nodeName) const
34 {
35     Ptr<Node> node = Names::Find<Node> (nodeName);
36     return ApplicationContainer (InstallPriv (node));
37 }
38
39 ApplicationContainer
40 WsnHelper::Install (NodeContainer c) const
41 {
42     ApplicationContainer apps;
43     for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
44     {
45         apps.Add (InstallPriv (*i));
46     }
47
48     return apps;
49 }
50
51 Ptr<Application>
52 WsnHelper::InstallPriv (Ptr<Node> node) const
53 {
54     Ptr<Application> app = m_factory.Create<Application> ();
55     node->AddApplication (app);
56
57     return app;
58 }
59
60 void
61 WsnHelper::SetConstantRate (DataRate dataRate, uint32_t packetSize)
62 {
63     m_factory.Set ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1000]"));
64     m_factory.Set ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0]"));
65     m_factory.Set ("DataRate", DataRateValue (dataRate));
66     m_factory.Set ("PacketSize", UIntegerValue (packetSize));
67 }
68
69 }
70
```

## Model codes used

### LOADng-packets.cc

```
1 #include "LOADng-packet.h"
2 #include "ns3/address-utils.h"
3 #include "ns3/packet.h"
4
5 namespace ns3 {
6 namespace LOADng {
7
8 NS_OBJECT_ENSURE_REGISTERED (LOADngHeader);
9
10 LOADngHeader::LOADngHeader (Vector position, Ipv4Address address, Time m)
11 : m_position (position),
12   m_address (address),
13   m_deadline (m)
14 {
15 }
16
17 LOADngHeader::~LOADngHeader ()
18 {
19 }
20
21 TypeId
22 LOADngHeader::GetTypeId (void)
23 {
24   static TypeId tid = TypeId ("ns3::LOADng::LOADngHeader")
25     .SetParent<Header> ()
26     .SetGroupName ("LOADng")
27     .AddConstructor<LOADngHeader> ();
28   return tid;
29 }
30
31 TypeId
32 LOADngHeader::GetInstanceTypeId () const
33 {
34   return GetTypeId ();
35 }
36
37 uint32_t
38 LOADngHeader::GetSerializedSize () const
39 {
40   return sizeof(m_position)+sizeof(m_address)+4+sizeof(m_deadline);
41 }
42
43 void
44 LOADngHeader::Serialize (Buffer::Iterator i) const
45 {
46   i.Write ((const uint8_t*)&m_position, sizeof(m_position));
47   i.Write ((const uint8_t*)&m_address, sizeof(m_address));
48   i.Write ((const uint8_t*)&m_address+4, sizeof(m_address));
49   i.Write ((const uint8_t*)&m_deadline, sizeof(m_deadline));
50 }
51
52 uint32_t
53 LOADngHeader::Deserialize (Buffer::Iterator start)
54 {
55   Buffer::Iterator i = start;
56
57   i.Read ((uint8_t*)&m_position, sizeof(m_position));
58   i.Read ((uint8_t*)&m_address, sizeof(m_address));
59   i.ReadU32();
60   i.Read ((uint8_t*)&m_deadline, sizeof(m_deadline));
61
62   uint32_t dist = i.GetDistanceFrom (start);
63   NS_ASSERT (dist == GetSerializedSize ());
64   return dist;
65 }
66
67 void
68 LOADngHeader::Print (std::ostream &os) const
69 {
70   os << " Position: " << m_position << ", IP: " << m_address << ", Deadline:" << m_deadline << "\n";
71 }
72 }
73 }
74
```

## LOADng-packet-queue

```
1 #include "LOADng-packet-queue.h"
2 #include <algorithm>
3 #include <functional>
4 #include "ns3/ipv4-route.h"
5 #include "ns3/socket.h"
6 #include "ns3/log.h"
7
8 namespace ns3 {
9
10 NS_LOG_COMPONENT_DEFINE ("LOADngPacketQueue");
11
12 namespace LOADng {
13 uint32_t
14 PacketQueue::GetSize ()
15 {
16     return m_queue.size ();
17 }
18
19 bool
20 PacketQueue::Enqueue (QueueEntry & entry)
21 {
22     NS_LOG_FUNCTION ("Enqueing packet destined for" << entry.GetIpv4Header ().GetDestination ());
23
24     // NS_LOG_DEBUG("Packet size while enqueueing "<<entry.GetPacket()->GetSize());
25     m_queue.push_back (entry);
26     return true;
27 }
28
29 void
30 PacketQueue::Drop (uint32_t idx)
31 {
32     m_queue.erase(m_queue.begin()+idx);
33 }
34 bool
35 PacketQueue::Dequeue (Ipv4Address dst, QueueEntry & entry)
36 {
37     NS_LOG_FUNCTION ("Dequeueing packet destined for" << dst);
38     for (std::vector<QueueEntry>::iterator i = m_queue.begin (); i != m_queue.end (); ++i)
39     {
40         if (i->GetIpv4Header ().GetDestination () == dst)
41         {
42             entry = *i;
43             m_queue.erase (i);
44             return true;
45         }
46     }
47     return false;
48 }
49
50 bool
51 PacketQueue::Find (Ipv4Address dst)
52 {
53     for (std::vector<QueueEntry>::const_iterator i = m_queue.begin (); i
54         != m_queue.end (); ++i)
55     {
56         if (i->GetIpv4Header ().GetDestination () == dst)
57         {
58             NS_LOG_DEBUG ("Find");
59             return true;
60         }
61     }
62     return false;
63 }
64
65 uint32_t
66 PacketQueue::GetCountForPacketsWithDst (Ipv4Address dst)
67 {
68     uint32_t count = 0;
69     for (std::vector<QueueEntry>::const_iterator i = m_queue.begin (); i
70         != m_queue.end (); ++i)
71     {
72         if (i->GetIpv4Header ().GetDestination () == dst)
73         {
74             count++;
75         }
76     }
77     return count;
78 }
79
80 }
81 }
```

## LOADng-routing-protocol

```
1 #include "LOADng-routing-protocol.h"
2 #include "ns3/log.h"
3 #include "ns3/inet-socket-address.h"
4 #include "ns3/trace-source-accessor.h"
5 #include "ns3/udp-socket-factory.h"
6 #include "ns3/wifi-net-device.h"
7 #include "ns3/boolean.h"
8 #include "ns3/double.h"
9 #include "ns3/uinteger.h"
10 #include "ns3/vector.h"
11 #include "ns3/udp-header.h"
12
13 #include <iostream>
14 #include <cmath>
15 #include <vector>
16
17 namespace ns3 {
18
19 NS_LOG_COMPONENT_DEFINE ("LOADngRoutingProtocol");
20
21 namespace ImprovedLOADng {
22
23 NS_OBJECT_ENSURE_REGISTERED (ImprovedLOADng); //
24
25 /// UDP Port for LOADng control traffic
26 const uint32_t ImprovedLOADng::LOADng_PORT = 269;
27
28 double max(double a, double b) {
29
30     return (a>b)?a:b;
31 }
32
33 TypeId
34 ImprovedLOADng::GetTypeId (void)
35 {
36     static TypeId tid = TypeId ("ns3::LOADng::ImprovedLOADng")
37     .SetParent<Ipv4ImprovedLOADng> ()
38     .SetGroupName ("LOADng")
39     .AddConstructor<ImprovedLOADng> ()
40     .AddAttribute ("PeriodicUpdateInterval", "Periodic interval between exchange of full routing tab
41     TimeValue (Seconds (15)),
42     MakeTimeAccessor (&ImprovedLOADng::m_periodicUpdateInterval),
43     MakeTimeChecker ())
44     .AddAttribute ("Position", "X and Y position of the node",
45     Vector3DValue (),
46     MakeVectorAccessor (&ImprovedLOADng::m_position),
47     MakeVectorChecker ())
48     .AddAttribute ("Lambda", "Average Packet generation rate",
49     DoubleValue (1.0),
50     MakeDoubleAccessor (&ImprovedLOADng::m_lambda),
51     MakeDoubleChecker <double>())
52     .AddTraceSource ("DroppedCount", "Total packets dropped",
53     MakeTraceSourceAccessor (&ImprovedLOADng::m_dropped),
54     "ns3::TracedValueCallback::Uint32")
55     ;
56     return tid;
57 }
58 void
59
60 ImprovedLOADng::SetPosition (Vector f)
61 {
62     m_position = f;
63 }
64 Vector
65 ImprovedLOADng::GetPosition () const
66 {
67     return m_position;
68 }
69 std::vector<struct msmt>*
70 ImprovedLOADng::getTimeline()
71 {
72     return &timeline;
73 }
74 std::vector<Time>*
75 ImprovedLOADng::getTxTime()
76 {
77     return &tx_time;
78 }
79
80 int64_t
81 ImprovedLOADng::AssignStreams (int64_t stream)
82 {
83     NS_LOG_FUNCTION (this << stream);
84     m_uniformRandomVariable->SetStream (stream);
85     return 1;
86 }
87
88 ImprovedLOADng::ImprovedLOADng ()
89 : Round(0),
```

```

89     isSink(0),
90     m_dropped(0),
91     m_lambda(4.0),
92     timeline(),
93     tx_time(),
94     m_routingTable(),
95     m_bestRoute(),
96     m_queue(),
97     m_periodicUpdateTimer(Timer::CANCEL_ON_DESTROY),
98     m_broadcastClusterHeadTimer(Timer::CANCEL_ON_DESTROY),
99     m_respondToClusterHeadTimer(Timer::CANCEL_ON_DESTROY)
100 {
101     m_uniformRandomVariable = CreateObject<UniformRandomVariable>();
102     for(int i=0; i<1021; i++) m_hash[i] = NULL;
103 }
104
105 ImprovedLOADng::~ImprovedLOADng()
106 {
107 }
108
109 void
110 ImprovedLOADng::DoDispose()
111 {
112     m_ipv4 = 0;
113     for(std::map<Ptr<Socket>, Ipv4InterfaceAddress>::iterator iter = m_socketAddresses.begin();
114         iter != m_socketAddresses.end(); iter++)
115     {
116         iter->first->Close();
117     }
118     m_socketAddresses.clear();
119
120     Ipv4ImprovedLOADng::DoDispose();
121 }
122
123 void
124 ImprovedLOADng::PrintRoutingTable(Ptr<OutputStreamWrapper> stream) const
125 {
126     *stream->GetStream() << "Node: " << m_ipv4->GetObject<Node>()->GetId()
127     << ", Time: " << Now().As(Time::S)
128     << ", Local time: " << GetObject<Node>()->GetLocalTime().As(Time::S)
129     << ", LOADng Routing table" << std::endl;
130
131     m_routingTable.Print(stream);
132     *stream->GetStream() << std::endl;
133 }
134
135 void
136 ImprovedLOADng::Start()
137 {
138     m_scb = MakeCallback(&ImprovedLOADng::Send, this);
139     m_ecb = MakeCallback(&ImprovedLOADng::Drop, this);
140     m_sinkAddress = Ipv4Address("10.1.1.1");
141     ns3::PacketMetadata::Enable();
142     ns3::Packet::EnablePrinting();
143
144     if(m_mainAddress == m_sinkAddress) {
145         isSink = 1;
146     } else {
147         Round = 0;
148         m_routingTable.Setholddowntime(Time(m_periodicUpdateInterval));
149         m_periodicUpdateTimer.SetFunction(&ImprovedLOADng::PeriodicUpdate, this);
150
151         m_broadcastClusterHeadTimer.SetFunction(&ImprovedLOADng::SendBroadcast, this);
152         m_respondToClusterHeadTimer.SetFunction(&ImprovedLOADng::RespondToClusterHead, this);
153         m_periodicUpdateTimer.Schedule(MicroSeconds(m_uniformRandomVariable->GetInteger(10,1000)));
154     }
155 }
156
157 Ptr<Ipv4Route>
158 ImprovedLOADng::RouteOutput(Ptr<Packet> p,
159                             const Ipv4Header &header,
160                             Ptr<NetDevice> oif,
161                             Socket::SocketErrno &sockerr)
162 {
163     NS_LOG_FUNCTION(this << header << (oif ? oif->GetIfIndex() : 0));
164     if(m_socketAddresses.empty())
165     {
166         sockerr = Socket::ERROR_NOROUTETOHOST;
167         NS_LOG_LOGIC("No LOADng interfaces");
168         Ptr<Ipv4Route> route;
169         return route;
170     }
171     Ipv4Address dst = header.GetDestination();
172     RoutingTableEntry rt;
173     NS_LOG_DEBUG("Packet Size: " << p->GetSize()
174                 << ", Packet id: " << p->GetUid() << ", Destination address in Pack");
175 #ifdef DA
176     if(p->GetSize()%56 == 0)
177     {

```



```

178         if (DataAggregation (p))
179         {
180             #endif
181             if (m_routingTable.LookupRoute (dst,rt))
182             {
183                 tx_time.push_back(Simulator::Now());
184                 Ptr<Packet> packet = new Packet(*p);
185                 LOADngHeader hdr;
186                 struct ns3::LOADng::msmt tmp;
187                 packet->RemoveHeader(hdr);
188                 tmp.begin = Simulator::Now();
189                 tmp.end = hdr.GetDeadline();
190                 timeline.push_back(tmp);
191                 return rt.GetRoute();
192             }
193         }
194     #ifdef DA
195     }
196     #endif
197     else if (m_routingTable.LookupRoute (dst,rt))
198     {
199         tx_time.push_back(Simulator::Now());
200         return rt.GetRoute();
201     }
202 #endif
203 return LoopbackRoute (header,oif);
204 }
205
206
207
208 }
209
210 Ptr<Ipv4Route>
211 ImprovedLOADng::LoopbackRoute (const Ipv4Header &hdr, Ptr<NetDevice> oif) const
212 {
213     NS_ASSERT (m_lo != 0);
214     NS_LOG_DEBUG("");
215     Ptr<Ipv4Route> rt = Create<Ipv4Route> ();
216     rt->SetDestination (hdr.GetDestination ());
217
218     std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses.begin ();
219     if (oif)
220     {
221         // Iterate to find an address on the oif device
222         for (j = m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
223         {
224             Ipv4Address addr = j->second.GetLocal ();
225             int32_t interface = m_ipv4->GetInterfaceForAddress (addr);
226             if (oif == m_ipv4->GetNetDevice (static_cast<uint32_t> (interface)))
227             {
228                 rt->SetSource (addr);
229                 break;
230             }
231         }
232     }
233     else
234     {
235         rt->SetSource (j->second.GetLocal ());
236     }
237 }
238
239 NS_ASSERT_MSG (rt->GetSource () != Ipv4Address (), "Valid LOADng source address not found");
240 rt->SetGateway (Ipv4Address ("127.0.0.1"));
241 rt->SetOutputDevice (m_lo);
242 return rt;
243 }
244
245 bool
246 ImprovedLOADng::RouteInput (Ptr<const Packet> p,
247                             const Ipv4Header &header,
248                             Ptr<const NetDevice> idev,
249                             UnicastForwardCallback ucb,
250                             MulticastForwardCallback mcb,
251                             LocalDeliverCallback lcb,
252                             ErrorCallback ecb)
253 {
254     NS_LOG_FUNCTION (m_mainAddress << " received packet " << p->GetUid ()
255                    << " from " << header.GetSource ()
256                    << " on interface " << idev->GetAddress ()
257                    << " to destination " << header.GetDestination ());
258     if (m_socketAddresses.empty ())
259     {
260         NS_LOG_DEBUG ("No LOADng interfaces");
261         return false;
262     }
263     NS_ASSERT (m_ipv4 != 0);
264     // Check if input device supports IP
265     NS_ASSERT (m_ipv4->GetInterfaceForDevice (idev) >= 0);
266     int32_t iif = m_ipv4->GetInterfaceForDevice (idev);
267     Ipv4Address dst = header.GetDestination ();

```

```

268     Ipv4Address origin = header.GetSource ();
269
270     // LOADng is not a multicast routing protocol
271     if (dst.IsMulticast ())
272     {
273         return false;
274     }
275
276     // Deferred route request
277     if (idev == m_lo)
278     {
279         NS_LOG_DEBUG("LoopBackRoute");
280     #ifdef DA
281         Ptr<Packet> pa = new Packet(*p);
282         EnqueuePacket (pa,header);
283         return false;
284     #else
285         RoutingTableEntry toDst;
286         NS_LOG_DEBUG("Deferred: " << dst);
287
288         if (m_routingTable.LookupRoute (dst,toDst))
289         {
290             Ptr<Ipv4Route> route = toDst.GetRoute ();
291             NS_LOG_DEBUG("Deferred forwarding");
292             NS_LOG_DEBUG("Src: " << route->GetSource() << ", Dst: " << toDst.GetDestination() <<
293             ucb(route, p ,header);
294         }
295         else
296         {
297             NS_LOG_DEBUG("Route not found");

```

```

298
299         Ptr<Ipv4Route> rt = Create<Ipv4Route> ();
300         rt->SetDestination (dst);
301         rt->SetSource (origin);
302         rt->SetGateway (Ipv4Address ("127.0.0.1"));
303         rt->SetOutputDevice (m_lo);
304
305         EnqueueForNoDA(ucb, rt, p, header);
306     }
307     return true;
308 #endif
309 }
310 for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
311      m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
312 {
313     Ipv4InterfaceAddress iface = j->second;
314     if (origin == iface.GetLocal ())
315     {
316         return true;
317     }
318 }
319 // LOCAL DELIVERY TO LOADng INTERFACES
320 for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses.begin ();
321      j != m_socketAddresses.end (); ++j)
322 {
323     Ipv4InterfaceAddress iface = j->second;
324     if (m_ipv4->GetInterfaceForAddress (iface.GetLocal ()) == iif)
325     {
326         // Do not deal with broadcast
327         if (dst == iface.GetBroadcast () || dst.IsBroadcast ())

```

```

328     {
329         Ptr<Packet> packet = p->Copy ();
330         if (lcb.IsNull () == false)
331         {
332             NS_LOG_LOGIC ("Broadcast local delivery to " << iface.GetLocal ());
333             lcb (p, header, iif);
334             // Fall through to additional processing
335         }
336         else
337         {
338             NS_LOG_ERROR ("Unable to deliver packet locally due to null callback " << p->GetU
339             ecb (p, header, Socket::ERROR_NOROUTETOHOST);
340         }
341         if (header.GetTtl () > 1)
342         {
343             NS_LOG_LOGIC ("Forward broadcast. TTL " << (uint16_t) header.GetTtl ());
344             RoutingTableEntry toBroadcast;
345             if (m_routingTable.LookupRoute (dst,toBroadcast,true))
346             {
347                 Ptr<Ipv4Route> route = toBroadcast.GetRoute ();
348                 ucb (route,packet,header);
349             }
350             else
351             {
352                 NS_LOG_DEBUG ("No route to forward. Drop packet " << p->GetUid ());
353             }
354         }
355         return true;
356     }
357 }

```

```

358     }
359     // this means arrival
360     if (m_ipv4->IsDestinationAddress (dst, iif))
361     {
362         if (lcb.IsNull () == false)
363         {
364             NS_LOG_LOGIC ("Unicast local delivery to " << dst);
365             lcb (p, header, iif);
366         }
367         else
368         {
369             NS_LOG_ERROR ("Unable to deliver packet locally due to null callback " << p->GetUid () <<
370             ecb (p, header, Socket::ERROR_NOROUTETOHOST);
371         }
372         return true;
373     }
374 }
375
376 // Check if input device supports IP forwarding
377 if (m_ipv4->IsForwarding (iif) == false)
378 {
379     NS_LOG_LOGIC ("Forwarding disabled for this interface");
380     ecb (p, header, Socket::ERROR_NOROUTETOHOST);
381     return true;
382 }
383
384 // enqueue this and not send
385 RoutingTableEntry toDst;
386 if (m_routingTable.LookupRoute (dst,toDst))
387 {

```

```

388     RoutingTableEntry ne;
389     if (m_routingTable.LookupRoute (toDst.GetNextHop (),ne))
390     {
391         Ptr<Ipv4Route> route = ne.GetRoute ();
392         NS_LOG_LOGIC (m_mainAddress << " is forwarding packet " << p->GetUid ()
393         << " to " << dst
394         << " from " << header.GetSource ()
395         << " via nexthop neighbor " << toDst.GetNextHop ());
396     #ifndef DA
397         Ptr<Packet> pa = new Packet(*p);
398         EnqueuePacket(pa, header);
399         return false;
400     #else
401         ucb (route,p,header);
402         return true;
403     #endif
404     }
405 }
406
407 #ifndef DA
408     NS_LOG_DEBUG("Route not found");
409     Ptr<Ipv4Route> rt = Create<Ipv4Route> ();
410     rt->SetDestination (dst);
411     rt->SetSource (origin);
412     rt->SetGateway (Ipv4Address ("127.0.0.1"));
413     rt->SetOutputDevice (m_lo);
414     EnqueueForNoDA(ucb, rt, p, header);
415 }
416
417

```

```

418 #endif
419     return false;
420 }
421
422 void
423 ImprovedLOADng::EnqueueForNoDA(UnicastForwardCallback ucb, Ptr<Ipv4Route> rt, Ptr<const Packet>
424 {
425     struct DeferredPack tmp;
426     tmp.ucb = ucb;
427     tmp.rt = rt;
428     tmp.p = p;
429     tmp.header = header;
430     DeferredQueue.push_back(tmp);
431 }
432 Simulator::Schedule (MilliSeconds (100),&ImprovedLOADng::AutoDequeueNoDA,this);
433 }
434
435 void
436 ImprovedLOADng::AutoDequeueNoDA()
437 {
438     while(DeferredQueue.size())
439     {
440         struct DeferredPack tmp = DeferredQueue.front();
441         tmp.ucb(tmp.rt, tmp.p, tmp.header);
442         DeferredQueue.erase(DeferredQueue.begin());
443     }
444 }
445
446 void
447 ImprovedLOADng::RecvLOADng (Ptr<Socket> socket)

```

```

448 {
449     Address sourceAddress;
450     Ptr<Packet> packet = socket->RecvFrom (sourceAddress);
451     InetSocketAddress inetSourceAddr = InetSocketAddress::ConvertFrom (sourceAddress);
452     Ipv4Address sender = inetSourceAddr.GetIpv4 ();
453     Ipv4Address receiver = m_socketAddresses[socket].GetLocal ();
454     double dist, dx, dy;
455     LOADngHeader LOADngHeader;
456     Vector senderPosition;
457
458
459     packet->RemoveHeader(LOADngHeader);
460
461
462     if(isSink) return;
463     if(LOADngHeader.GetAddress() == Ipv4Address("255.255.255.255")) {
464         NS_LOG_DEBUG("Recv broadcast from CH: " << sender);
465         // Need to update a new route
466         RoutingTableEntry newEntry (
467             /*device=*/ socket->GetBoundNetDevice(), /*dst (sink)*/m_sinkAddress,
468             /*iface=*/ m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0),
469             /*next hop=*/ sender);
470
471         senderPosition = LOADngHeader.GetPosition();
472         dx = senderPosition.x - m_position.x;
473         dy = senderPosition.y - m_position.y;
474         dist = dx*dx + dy*dy;
475         NS_LOG_DEBUG("dist = " << dist << ", m_dist = " << m_dist);
476
477         if(dist < m_dist) {
478
479             m_dist = dist;
480             m_targetAddress = sender;
481             m_bestRoute = newEntry;
482             NS_LOG_DEBUG(sender);
483         }
484         }else {
485             // Record cluster member
486             m_clusterMember.push_back(LOADngHeader.GetAddress());
487         }
488     }
489
490     void
491     ImprovedLOADng::RespondToClusterHead()
492     {
493         Ptr<Socket> socket = FindSocketWithAddress(m_mainAddress);
494         Ptr<Packet> packet = Create<Packet> ();
495         LOADngHeader LOADngHeader;
496         Ipv4Address ipv4;
497         OutputStreamWrapper temp = OutputStreamWrapper(&std::cout);
498
499         // Add routing to routingTable
500         if(m_targetAddress != ipv4) {
501             RoutingTableEntry newEntry, entry2;
502             newEntry.Copy(m_bestRoute);
503             entry2.Copy(m_bestRoute);
504             Ptr<Ipv4Route> newRoute = newEntry.GetRoute();
505             newRoute->SetDestination(m_targetAddress);
506             newEntry.SetRoute(newRoute);
507
508             if(m_bestRoute.GetInterface().GetLocal() != ipv4) m_routingTable.AddRoute (entry2);
509             if(newEntry.GetInterface().GetLocal() != ipv4) m_routingTable.AddRoute (newEntry);
510
511             // m_routingTable.Print(&temp);
512
513             LOADngHeader.SetAddress(m_mainAddress);
514             packet->AddHeader (LOADngHeader);
515             socket->SendTo (packet, 0, InetSocketAddress (m_targetAddress, LOADng_PORT));
516         }
517     }
518
519     void
520     ImprovedLOADng::SendBroadcast ()
521     {
522         Ptr<Socket> socket = FindSocketWithAddress (m_mainAddress);
523         Ptr<Packet> packet = Create<Packet> ();
524         LOADngHeader LOADngHeader;
525         Ipv4Address destination = Ipv4Address ("10.1.1.255");;
526
527         socket->SetAllowBroadcast (true);
528
529         LOADngHeader.SetPosition (m_position);
530         packet->AddHeader (LOADngHeader);
531         socket->SendTo (packet, 0, InetSocketAddress (destination, LOADng_PORT));
532
533         RoutingTableEntry newEntry (
534             /*device=*/ socket->GetBoundNetDevice(), /*dst (sink)*/m_sinkAddress,
535             /*iface=*/ m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (m_mainAddress), 0),
536             /*next hop=*/ m_sinkAddress);
537         m_routingTable.AddRoute (newEntry);

```

```

538 }
539
540 void
541 ImprovedLOADng::PeriodicUpdate ()
542 {
543     double probab = m_uniformRandomVariable->GetValue (0,1);
544     // 10 round a cycle, 100/10=10 cluster heads per round
545     int n = 10;
546     double p = 1.0/n;
547     double t = p/(1-p*(Round%n));
548
549     NS_LOG_DEBUG("PeriodicUpdate!!");
550     // NS_LOG_DEBUG("probab = " << probab << ", t = " << t);
551
552     m_routingTable.DeleteRoute(m_targetAddress);
553     m_routingTable.DeleteRoute(m_sinkAddress);
554
555     /*
556     OutputStreamWrapper temp = OutputStreamWrapper(&std::cout);
557     m_routingTable.Print(&temp);
558     */
559     if(Round%n == 0) valid = 1;
560     Round++;
561     m_dist = 1e100;
562     cluster_head_this_round = 0;
563     m_clusterMember.clear();
564     m_bestRoute.Reset();
565     m_targetAddress = Ipv4Address();
566
567     if(probab < t && valid) {
568         // become cluster head
569
570         // broadcast info
571         NS_LOG_DEBUG(m_mainAddress << " becomes cluster head");
572         valid = 0;
573         cluster_head_this_round = 1;
574         m_targetAddress = m_sinkAddress;
575         m_broadcastClusterHeadTimer.Schedule (MicroSeconds (m_uniformRandomVariable->GetInteger (10000,100000)));
576     }else {
577         m_respondToClusterHeadTimer.Schedule (Milliseconds(100) + MicroSeconds (m_uniformRandomVariable->GetInteger (100,1000)));
578     }
579     m_periodicUpdateTimer.Schedule (m_periodicUpdateInterval + MicroSeconds (m_uniformRandomVariable->GetInteger (100,1000)));
580 }
581
582 void
583 ImprovedLOADng::SetIpv4 (Ptr<Ipv4> ipv4)
584 {
585     NS_ASSERT (ipv4 != 0);
586     NS_ASSERT (m_ipv4 == 0);
587     m_ipv4 = ipv4;
588     // Create lo route. It is asserted that the only one interface up for now is loopback
589     NS_ASSERT (m_ipv4->GetNInterfaces () == 1 && m_ipv4->GetAddress (0, 0).GetLocal () == Ipv4Address ("127.0.0.1"));
590     m_lo = m_ipv4->GetNetDevice (0);
591     NS_ASSERT (m_lo != 0);
592     // Remember lo route
593     RoutingTableEntry rt (
594         /*device=*/ m_lo, /*dst=*/
595         Ipv4Address::GetLoopback (),
596         /*iface=*/ Ipv4InterfaceAddress (Ipv4Address::GetLoopback (), Ipv4Mask ("255.0.0.0")),
597         /*next hop=*/ Ipv4Address::GetLoopback ());
598     rt.SetFlag (INVALID);
599     m_routingTable.AddRoute (rt);
600 }
601
602 Simulator::ScheduleNow (&ImprovedLOADng::Start, this);
603 }
604
605 void
606 ImprovedLOADng::NotifyInterfaceUp (uint32_t i)
607 {
608     NS_LOG_FUNCTION (this << m_ipv4->GetAddress (i, 0).GetLocal ()
609                     << " interface is up");
610     Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
611     Ipv4InterfaceAddress iface = l3->GetAddress (i,0);
612     if (iface.GetLocal () == Ipv4Address ("127.0.0.1"))
613     {
614         return;
615     }
616     // Create a socket to listen only on this interface
617     Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (), UdpSocketFactory::GetTypeId ());
618     NS_ASSERT (socket != 0);
619     socket->SetRecvCallback (MakeCallback (&ImprovedLOADng::RecvLOADng, this));
620     socket->Bind (InetSocketAddress (Ipv4Address::GetAny (), LOADng_PORT));
621     socket->BindToNetDevice (l3->GetNetDevice (i));
622     socket->SetAllowBroadcast (true);
623     socket->SetAttribute ("IpTtl", UintegerValue (1));
624     m_socketAddresses.insert (std::make_pair (socket, iface));
625     // Add local broadcast record to the routing table
626     Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (iface.GetLocal ()));
627     RoutingTableEntry rt (/device=*/ dev, /*dst=*/ iface.GetBroadcast (), /*iface=*/ iface, /*next hop=*/ Ipv4Address::GetBroadcast ());
628     m_routingTable.AddRoute (rt);
629     if (m_mainAddress == Ipv4Address ())
630     {
631         m_mainAddress = iface.GetLocal ();
632     }
633 }

```

```

628     }
629     NS_ASSERT (m_mainAddress != Ipv4Address ());
630 }
631
632 void
633 ImprovedLOADng::NotifyInterfaceDown (uint32_t i)
634 {
635     Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
636     Ptr<NetDevice> dev = l3->GetNetDevice (i);
637     Ptr<Socket> socket = FindSocketWithInterfaceAddress (m_ipv4->GetAddress (i,0));
638     NS_ASSERT (socket);
639     socket->Close ();
640     m_socketAddresses.erase (socket);
641     if (m_socketAddresses.empty ())
642     {
643         NS_LOG_LOGIC ("No LOADng interfaces");
644         m_routingTable.Clear ();
645         return;
646     }
647     m_routingTable.DeleteAllRoutesFromInterface (m_ipv4->GetAddress (i,0));
648 }
649
650 void
651 ImprovedLOADng::NotifyAddAddress (uint32_t i,
652                                 Ipv4InterfaceAddress address)
653 {
654     NS_LOG_FUNCTION (this << " interface " << i << " address " << address);
655     Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
656     if (!l3->IsUp (i))
657     {

```

```

658         return;
659     }
660     Ipv4InterfaceAddress iface = l3->GetAddress (i,0);
661     Ptr<Socket> socket = FindSocketWithInterfaceAddress (iface);
662     if (!socket)
663     {
664         if (iface.GetLocal () == Ipv4Address ("127.0.0.1"))
665         {
666             return;
667         }
668         Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),UdpSocketFactory::GetTypeId ())
669         NS_ASSERT (socket != 0);
670         socket->SetRecvCallback (MakeCallback (&ImprovedLOADng::RecvLOADng,this));
671         // Bind to any IP address so that broadcasts can be received
672         socket->Bind (InetSocketAddress (Ipv4Address::GetAny (), LOADng_PORT));
673         socket->BindToNetDevice (l3->GetNetDevice (i));
674         socket->SetAllowBroadcast (true);
675         m_socketAddresses.insert (std::make_pair (socket,iface));
676         Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (iface.GetLocal ())
677         RoutingTableEntry rt (/*device=*/ dev, /*dst=*/ iface.GetBroadcast (), /*iface=*/ iface, /*ne:
678         m_routingTable.AddRoute (rt);
679     }
680 }
681
682 void
683 ImprovedLOADng::NotifyRemoveAddress (uint32_t i,
684                                    Ipv4InterfaceAddress address)
685 {
686     Ptr<Socket> socket = FindSocketWithInterfaceAddress (address);
687     if (socket)

```

```

688     {
689         m_socketAddresses.erase (socket);
690         Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
691         if (l3->GetNAddresses (i))
692         {
693             Ipv4InterfaceAddress iface = l3->GetAddress (i,0);
694             // Create a socket to listen only on this interface
695             Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),UdpSocketFactory::GetTypeId
696             NS_ASSERT (socket != 0);
697             socket->SetRecvCallback (MakeCallback (&ImprovedLOADng::RecvLOADng,this));
698             // Bind to any IP address so that broadcasts can be received
699             socket->Bind (InetSocketAddress (Ipv4Address::GetAny (), LOADng_PORT));
700             socket->SetAllowBroadcast (true);
701             m_socketAddresses.insert (std::make_pair (socket,iface));
702         }
703     }
704 }
705
706 Ptr<Socket>
707 ImprovedLOADng::FindSocketWithAddress (Ipv4Address addr) const
708 {
709     for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses.begin ();
710          j != m_socketAddresses.end (); ++j)
711     {
712         Ptr<Socket> socket = j->first;
713         Ipv4InterfaceAddress iface = j->second.GetLocal();
714         if (iface == addr)
715         {
716             return socket;
717         }

```



```

718     }
719     return NULL;
720 }
721
722 Ptr<Socket>
723 ImprovedLOADng::FindSocketWithInterfaceAddress (Ipv4InterfaceAddress addr) const
724 {
725     for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses.begin ();
726          j != m_socketAddresses.end (); ++j)
727     {
728         Ptr<Socket> socket = j->first;
729         Ipv4InterfaceAddress iface = j->second;
730         if (iface == addr)
731         {
732             return socket;
733         }
734     }
735     return NULL;
736 }
737
738 void
739 ImprovedLOADng::Send (Ptr<Ipv4Route> route,
740                      Ptr<const Packet> packet,
741                      const Ipv4Header & header)
742 {
743     Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
744     NS_ASSERT (l3 != 0);
745     Ptr<Packet> p = packet->Copy ();
746     l3->Send (p, route->GetSource (), header.GetDestination (), header.GetProtocol (), route);
747 }

```

```

748
749 void
750 ImprovedLOADng::Drop (Ptr<const Packet> packet,
751                      const Ipv4Header & header,
752                      Socket::SocketErrno err)
753 {
754     NS_LOG_DEBUG (m_mainAddress << " drop packet " << packet->GetUid () << " to "
755                  << header.GetDestination () << " from queue. Error " << err);
756 }
757
758 void
759 ImprovedLOADng::EnqueuePacket (Ptr<Packet> p,
760                                const Ipv4Header & header)
761 {
762     NS_LOG_FUNCTION (this << ", " << p << ", " << header);
763     NS_ASSERT (p != 0 && p != Ptr<Packet> ());
764
765     Ptr<Packet> out;
766     UdpHeader uhdr;
767     LOADngHeader LOADngHeader;
768     uint32_t slot = p->GetUid()%1021;
769     struct hash* now = m_hash[slot];
770
771     NS_LOG_DEBUG("IsDontFragment: " << header.IsDontFragment());
772
773     if(header.GetFragmentOffset() == 0) p->RemoveHeader(uhdr);
774
775     while (now != NULL)
776     {
777         if(now->uid == p->GetUid())

```

```

778             break;
779             now = now->next;
780         }
781         if(now != NULL)
782         {
783             NS_LOG_DEBUG("now->p size " << now->p->GetSize() << ", p size " << p->GetSize());
784             now->p->AddAtEnd(p);
785             p = now->p;
786             NS_LOG_DEBUG("after p size " << p->GetSize());
787         }
788
789         while(DeAggregate(p, out, LOADngHeader))
790         {
791             QueueEntry newEntry (out,header);
792             bool result = m_queue.Enqueue (newEntry);
793             struct msmt temp;
794
795             temp.begin = Simulator::Now();
796             temp.end = LOADngHeader.GetDeadline();
797             timeline.push_back(temp);
798             if (result)
799             {
800                 NS_LOG_DEBUG ("Added packet " << out->GetUid () << " to queue.");
801             }
802         }
803     }
804
805     bool
806 ImprovedLOADng::DeAggregate (Ptr<Packet> in, Ptr<Packet>& out, LOADngHeader& lhdr)
807 {

```

```

808     if(in->GetSize() >= 56)
809     {
810         LOADngHeader LOADngHeader;
811         in->RemoveHeader(LOADngHeader);
812         in->RemoveAtStart(16);
813
814         lhdr = LOADngHeader;
815         out = new Packet(16);
816         out->AddHeader(LOADngHeader);
817         NS_LOG_DEBUG("deadline" << LOADngHeader.GetDeadline());
818         return true;
819     }
820     uint32_t slot = in->GetUid()%1021;
821     struct hash* now = m_hash[slot];
822     while(now != NULL)
823     {
824         if(now->uid == in->GetUid()) break;
825         now = now->next;
826     }
827     if(now == NULL)
828     {
829         now = new struct hash;
830         now->uid = in->GetUid();
831         now->next = m_hash[slot];
832         m_hash[slot] = now;
833     }
834     now->p = in;
835     NS_LOG_DEBUG("Size left " << in->GetSize() << ", on UID " << in->GetUid());
836
837     return false;

```

```

838 }
839
840 bool
841 ImprovedLOADng::DataAggregation (Ptr<Packet> p)
842 {
843     // Implement data aggregation policy
844     // and data aggregation function
845
846     #ifdef DA_PROP
847         return Proposal(p);
848     #endif
849     #ifdef DA_OPT
850         return OptTM(p);
851     #endif
852     #ifdef DA_CL
853         return ControlLimit(p);
854     #endif
855     #ifdef DA_SF
856         return SelectiveForwarding(p);
857     #endif
858
859     return true;
860 }
861
862 bool
863 ImprovedLOADng::Proposal (Ptr<Packet> p)
864 {
865     NS_LOG_FUNCTION (this);
866     // pick up those selected entry and send
867     int expired = 0, expected;

```

```

868 //LOADngHeader hdr;
869 Time deadLine = Now();
870
871 // 1.28 = 2*0.64, 0.064 = 64bytes/8kbps
872 // average 10 cluster heads
873 // average 10 members per cluster
874 deadLine += Seconds(m_queue.GetSize()/m_lambda);
875 if(!cluster_head_this_round)
876     // depend on average tx size from cluster member
877     // depend on deadline setting
878     // * average packet_size?
879     deadLine += Seconds(0.064+1.0/m_lambda);
880
881 // NS_LOG_UNCOND("Now: " << Now() << ", Deadline: " << deadLine);
882 for (int i=0; i<(int)m_queue.GetSize(); i++)
883 {
884     NS_LOG_DEBUG("GetDeadline: " << m_queue[i].GetDeadline() << ", UID: " << m_queue[i].GetPacket
885     if(m_queue[i].GetDeadline() < Now())
886     {
887         // drop it
888         NS_LOG_DEBUG("Drop");
889         // NS_LOG_DEBUG("GetLOADngHeader: " << m_queue[i].GetDeadline() << ", Now: " << Now());
890         m_queue.Drop (i);
891         m_dropped++;
892         i--;
893     }
894 }
895
896 for(uint32_t i=0; i<m_queue.GetSize(); i++) {
897 // NS_LOG_UNCOND("GetDeadline: " << m_queue[i].GetDeadline() << ", deadline: " << deadLine);

```



```

898     if(m_queue[i].GetDeadline() < deadLine) {
899         expired++;
900     }
901 }
902 if(cluster_head_this_round) {
903     expected = 1+m_clusterMember.size();
904 }
905 else {
906     expected = 1;
907 }
908
909 // NS_LOG_UNCOND("expired: " << expired << ", expected: " << expected);
910 if(expired >= expected || Now() > Seconds(48.5)) {
911     // merge data
912     QueueEntry temp;
913
914     while(m_queue.Dequeue(m_sinkAddress, temp)) {
915         p->AddAtEnd(temp.GetPacket());
916     }
917
918     return true;
919 }
920 return false;
921 }
922
923 bool
924 ImprovedLOADng::OptTM (Ptr<Packet> p)
925 {
926     Time time = Now();
927     uint32 t rewards[100], maxR = 0;

```

```

928     uint32 t actions[100];
929     static int step = 0;
930
931     for(int i=0; i<100; i++)
932     {
933         actions[i] = 0;
934         rewards[i] = 0;
935         for(uint j=0; j<m_queue.GetSize(); j++)
936         {
937             if(m_queue[j].GetDeadline() >= time) rewards[i] += m_queue[j].GetDeadline().ToInteger(Time::ConvertToInteger(Seconds(1)));
938         }
939         for(int j=1; j<i+step; j++)
940         {
941             rewards[i] += (j<8) ? 30000-j*4000 : 0;
942         }
943         time += Seconds(1/m_lambda);
944     }
945
946     for(int i=0; i<100; i++)
947     {
948         if(rewards[i] > maxR)
949         {
950             maxR = rewards[i];
951         }
952     }
953
954     // wait=1, transmit=2
955     for(int i=98; i>=0; i--)
956     {
957         if(rewards[i] < maxR)

```

```

958         actions[i] = 1;
959     else
960     {
961         double rb[100], rn[100];
962         rb[99] = 1.0;
963         rn[99] = 0.0;
964
965         for(int k=98; k>i; k--)
966         {
967             rn[k] = max(0.0, i*rn[k+1]/(k+1) + rb[k+1]/(k+1));
968             rb[k] = max(rewards[k], rn[k]);
969         }
970
971         if(rewards[i] >= (uint32_t)rb[i+1])
972             actions[i] = 2;
973         else
974             actions[i] = 1;
975     }
976 }
977 step++;
978
979 if(actions[0] > 1 || Now() > Seconds(48.5))
980 {
981     QueueEntry temp;
982     while(m_queue.Dequeue(m_sinkAddress, temp))
983     {
984         p->AddAtEnd(temp.GetPacket());
985     }
986
987     return true;

```

```

988     }
989     return false;
991 }
992
993 bool
994 ImprovedLOADng::ControlLimit (Ptr<Packet> p)
995 {
996     static uint32_t threshold = (1/(log(1/0.1)*(log(1/0.1)+m_lambda)))+2;
997     for(int i=0; i<(int)m_queue.GetSize(); i++)
998     {
999         if(m_queue[i].GetDeadline() < Now())
1000         {
1001             m_queue.Drop(i);
1002             m_dropped++;
1003             i--;
1004         }
1005     }
1006     if(m_queue.GetSize() >= threshold || Now() > Seconds(48.5))
1007     {
1008         QueueEntry temp;
1009         while(m_queue.Dequeue(m_sinkAddress, temp)) {
1010             p->AddAtEnd(temp.GetPacket());
1011         }
1012         return true;
1013     }
1014     return false;
1015 }
1016
1017
1018 bool
1019 ImprovedLOADng::SelectiveForwarding (Ptr<Packet> p)
1020 {
1021     return false;
1022 }
1023 }
1024 }
1025 }
1026

```

## LOADng-routing-table

```

1 #include "LOADng-rtable.h"
2 #include "ns3/simulator.h"
3 #include <iomanip>
4 #include "ns3/log.h"
5
6 namespace ns3 {
7
8     NS_LOG_COMPONENT_DEFINE ("LOADngRoutingTable");
9
10    namespace LOADng {
11        RoutingTableEntry::RoutingTableEntry (Ptr<NetDevice> dev,
12                                             Ipv4Address dst,
13                                             Ipv4InterfaceAddress iface,
14                                             Ipv4Address nextHop)
15        : m_iface (iface),
16          m_flag (VALID)
17        {
18            m_ipv4Route = Create<Ipv4Route> ();
19            m_ipv4Route->SetDestination (dst);
20            m_ipv4Route->SetGateway (nextHop);
21            m_ipv4Route->SetSource (m_iface.GetLocal ());
22            m_ipv4Route->SetOutputDevice (dev);
23        }
24        RoutingTableEntry::~RoutingTableEntry ()
25        {
26        }
27        RoutingTable::RoutingTable ()
28        {
29        }
30
31    bool
32    RoutingTable::LookupRoute (Ipv4Address id,
33                             RoutingTableEntry & rt)
34    {
35        if (m_ipv4AddressEntry.empty ())
36        {
37            // NS_LOG_UNCOND("1");
38            return false;
39        }
40        std::map<Ipv4Address, RoutingTableEntry>::const_iterator i = m_ipv4AddressEntry.find (id);
41        if (i == m_ipv4AddressEntry.end ())
42        {
43            // NS_LOG_UNCOND("2");
44            return false;
45        }
46        rt = i->second;
47        return true;
48    }
49
50    bool
51    RoutingTable::LookupRoute (Ipv4Address id,
52                             RoutingTableEntry & rt,
53                             bool forRouteInput)
54    {
55        if (m_ipv4AddressEntry.empty ())
56        {
57            return false;
58        }
59        std::map<Ipv4Address, RoutingTableEntry>::const_iterator i = m_ipv4AddressEntry.find (id);
60        if (i == m_ipv4AddressEntry.end ())

```

```

61     {
62         return false;
63     }
64     if (forRouteInput == true && id == i->second.GetInterface ().GetBroadcast ())
65     {
66         return false;
67     }
68     rt = i->second;
69     return true;
70 }
71
72 bool
73 RoutingTable::DeleteRoute (Ipv4Address dst)
74 {
75     if (m_ipv4AddressEntry.erase (dst) != 0)
76     {
77         // NS_LOG_DEBUG("Route erased");
78         return true;
79     }
80     return false;
81 }
82
83 uint32_t
84 RoutingTable::RoutingTableSize ()
85 {
86     return m_ipv4AddressEntry.size ();
87 }
88
89 bool
90 RoutingTable::AddRoute (RoutingTableEntry & rt)

```

```

91 {
92     std::pair<std::map<Ipv4Address, RoutingTableEntry>::iterator, bool> result = m_ipv4AddressEntry.in:
93
94     return result.second;
95 }
96
97 bool
98 RoutingTable::Update (RoutingTableEntry & rt)
99 {
100     std::map<Ipv4Address, RoutingTableEntry>::iterator i = m_ipv4AddressEntry.find (rt.GetDestination
101     if (i == m_ipv4AddressEntry.end ())
102     {
103         return false;
104     }
105     i->second = rt;
106     return true;
107 }
108
109 void
110 RoutingTable::DeleteAllRoutesFromInterface (Ipv4InterfaceAddress iface)
111 {
112     if (m_ipv4AddressEntry.empty ())
113     {
114         return;
115     }
116     for (std::map<Ipv4Address, RoutingTableEntry>::iterator i = m_ipv4AddressEntry.begin (); i != m_ip
117     {
118         if (i->second.GetInterface () == iface)
119         {
120             std::map<Ipv4Address, RoutingTableEntry>::iterator tmp = i;

```

```

121             ++i;
122             m_ipv4AddressEntry.erase (tmp);
123         }
124         else
125         {
126             ++i;
127         }
128     }
129 }
130
131 void
132 RoutingTable::GetListOfAllRoutes (std::map<Ipv4Address, RoutingTableEntry> & allRoutes)
133 {
134     for (std::map<Ipv4Address, RoutingTableEntry>::iterator i = m_ipv4AddressEntry.begin (); i != m_ip
135     {
136         if (i->second.GetDestination () != Ipv4Address ("127.0.0.1") && i->second.GetFlag () == VALID)
137         {
138             allRoutes.insert (
139                 std::make_pair (i->first, i->second));
140         }
141     }
142 }
143
144 void
145 RoutingTable::GetListOfDestinationWithNextHop (Ipv4Address nextHop,
146     std::map<Ipv4Address, RoutingTableEntry> & unreachable)
147 {
148     unreachable.clear ();
149     for (std::map<Ipv4Address, RoutingTableEntry>::const_iterator i = m_ipv4AddressEntry.begin (); i
150         != m_ipv4AddressEntry.end (); ++i)

```

```

151     {
152         if (i->second.GetNextHop () == nextHop)
153         {
154             unreachable.insert (std::make_pair (i->first,i->second));
155         }
156     }
157 }
158
159 void
160 RoutingTableEntry::Print (Ptr<OutputStreamWrapper> stream) const
161 {
162     *stream->GetStream () << std::setiosflags (std::ios::fixed) << m_ipv4Route->GetDestination () << "
163     << m_iface.GetLocal () << "\n";
164 }
165
166 void
167 RoutingTable::Print (Ptr<OutputStreamWrapper> stream) const
168 {
169
170     *stream->GetStream () << "\nLOADng Routing table\n" << "DST\t\t\tDestination\t\tGateway\t\t\tInter
171     for (std::map<Ipv4Address, RoutingTableEntry>::const_iterator i = m_ipv4AddressEntry.begin (); i
172         != m_ipv4AddressEntry.end (); ++i)
173     {
174         i->first.Print (*stream->GetStream());
175         *stream->GetStream() << "\t\t";
176         i->second.Print (stream);
177     }
178     *stream->GetStream () << "\n";
179 }
180

```

```

181 bool
182 RoutingTable::AddIpv4Event (Ipv4Address address,
183                             EventId id)
184 {
185     std::pair<std::map<Ipv4Address, EventId>::iterator, bool> result = m_ipv4Events.insert (std::make_
186     return result.second;
187 }
188
189 bool
190 RoutingTable::AnyRunningEvent (Ipv4Address address)
191 {
192     EventId event;
193     std::map<Ipv4Address, EventId>::const_iterator i = m_ipv4Events.find (address);
194     if (m_ipv4Events.empty ())
195     {
196         return false;
197     }
198     if (i == m_ipv4Events.end ())
199     {
200         return false;
201     }
202     event = i->second;
203     if (event.IsRunning ())
204     {
205         return true;
206     }
207     else
208     {
209         return false;
210     }
211 }
212

```

```

213 bool
214 RoutingTable::ForceDeleteIpv4Event (Ipv4Address address)
215 {
216     EventId event;
217     std::map<Ipv4Address, EventId>::const_iterator i = m_ipv4Events.find (address);
218     if (m_ipv4Events.empty () || i == m_ipv4Events.end ())
219     {
220         return false;
221     }
222     event = i->second;
223     Simulator::Cancel (event);
224     m_ipv4Events.erase (address);
225     return true;
226 }
227
228 bool
229 RoutingTable::DeleteIpv4Event (Ipv4Address address)
230 {
231     EventId event;
232     std::map<Ipv4Address, EventId>::const_iterator i = m_ipv4Events.find (address);
233     if (m_ipv4Events.empty () || i == m_ipv4Events.end ())
234     {
235         return false;
236     }
237     event = i->second;
238     if (event.IsRunning ())
239     {
240         return false;

```

```

241     }
242     if (event.IsExpired ())
243     {
244         event.Cancel ();
245         m_ipv4Events.erase (address);
246         return true;
247     }
248     else
249     {
250         m_ipv4Events.erase (address);
251         return true;
252     }
253 }
254
255 EventId
256 RoutingTable::GetEventId (Ipv4Address address)
257 {
258     std::map <Ipv4Address, EventId>::const_iterator i = m_ipv4Events.find (address);
259     if (m_ipv4Events.empty () || i == m_ipv4Events.end ())
260     {
261         return EventId ();
262     }
263     else
264     {
265         return i->second;
266     }
267 }
268 }
269 }

```

## wsn-application

```

1  #include "ns3/log.h"
2  #include "ns3/address.h"
3  #include "ns3/inet-socket-address.h"
4  #include "ns3/inet6-socket-address.h"
5  #include "ns3/packet-socket-address.h"
6  #include "ns3/node.h"
7  #include "ns3/nstime.h"
8  #include "ns3/data-rate.h"
9  #include "ns3/random-variable-stream.h"
10 #include "ns3/socket.h"
11 #include "ns3/simulator.h"
12 #include "ns3/socket-factory.h"
13 #include "ns3/packet.h"
14 #include "ns3/uinteger.h"
15 #include "ns3/integer.h"
16 #include "ns3/double.h"
17 #include "ns3/trace-source-accessor.h"
18 #include "wsn-application.h"
19 #include "ns3/udp-socket-factory.h"
20 #include "ns3/string.h"
21 #include "ns3/pointer.h"
22 #include "ns3/LOADng-packet.h"
23
24 #include <cmath>
25
26 namespace ns3 {
27
28 NS_LOG_COMPONENT_DEFINE ("WsnApplication");
29
30 NS_OBJECT_ENSURE_REGISTERED (WsnApplication);
31
32
33 TypeId
34 WsnApplication::GetTypeId (void)
35 {
36     static TypeId tid = TypeId ("ns3::WsnApplication")
37         .SetParent<Application> ()
38         .SetGroupName ("Applications")
39         .AddConstructor <WsnApplication> ()
40         .AddAttribute ("DataRate", "The data rate in on state.",
41             DataRateValue (DataRate ("500kb/s")),
42             MakeDataRateAccessor (&WsnApplication::m_cbrRate),
43             MakeDataRateChecker ())
44         .AddAttribute ("PacketSize", "The size of packets sent in on state",
45             UIntegerValue (512),
46             MakeUIntegerAccessor (&WsnApplication::m_pktSize),
47             MakeUIntegerChecker<uint32_t> (1))
48         .AddAttribute ("PacketDeadlineLen", "The deadline range of packets",
49             IntegerValue (3),
50             MakeIntegerAccessor (&WsnApplication::m_pktDeadlineLen),
51             MakeIntegerChecker<int64_t> (1))
52         .AddAttribute ("PacketDeadlineMin", "The minimum deadline of packets",
53             IntegerValue (5),
54             MakeIntegerAccessor (&WsnApplication::m_pktDeadlineMin),
55             MakeIntegerChecker<int64_t> (1))
56         .AddAttribute ("Remote", "The address of the destination",
57             AddressValue (),
58             MakeAddressAccessor (&WsnApplication::m_peer),
59             MakeAddressChecker ())
60         .AddAttribute ("PktGenRate", "Packet generation rate",
61             DoubleValue (1.0),

```

```

61         MakeDoubleAccessor (&WsnApplication::m_pktGenRate),
62         MakeDoubleChecker <double>())
63     .AddAttribute ("PktGenPattern", "Packet generation distribution model",
64                 IntegerValue (0),
65                 MakeIntegerAccessor (&WsnApplication::m_pktGenPattern),
66                 MakeIntegerChecker <int>())
67     .AddAttribute ("MaxBytes",
68                 "The total number of bytes to send. Once these bytes are sent, "
69                 "no packet is sent again, even in on state. The value zero means "
70                 "that there is no limit.",
71                 UIntegerValue (0),
72                 MakeUIntegerAccessor (&WsnApplication::m_maxBytes),
73                 MakeUIntegerChecker<uint64_t> ())
74     .AddAttribute ("Protocol", "The type of protocol to use.",
75                 TypeIdValue (UdpSocketFactory::GetTypeId ()),
76                 MakeTypeIdAccessor (&WsnApplication::m_tid),
77                 MakeTypeIdChecker ())
78     .AddTraceSource ("Tx", "A new packet is created and is sent",
79                    MakeTraceSourceAccessor (&WsnApplication::m_txTrace),
80                    "ns3::Packet::TracedCallback")
81     .AddTraceSource ("PktCount", "Total packets count",
82                    MakeTraceSourceAccessor (&WsnApplication::m_pktCount),
83                    "ns3::TracedValueCallback::UInt32")
84 ;
85 return tid;
86 }
87
88
89 WsnApplication::WsnApplication ()
90 : m_socket (0),

```

```

91     m_connected (false),
92     m_residualBits (0),
93     m_lastStartTime (Seconds (0)),
94     m_totBytes (0),
95     m_pktCount (0)
96 {
97     NS_LOG_FUNCTION (this);
98 }
99
100 WsnApplication::~WsnApplication()
101 {
102     NS_LOG_FUNCTION (this);
103
104     // NS_LOG_UNCOND(m_pktCount);
105 }
106
107 void
108 WsnApplication::SetMaxBytes (uint64_t maxBytes)
109 {
110     NS_LOG_FUNCTION (this << maxBytes);
111     m_maxBytes = maxBytes;
112 }
113
114 Ptr<Socket>
115 WsnApplication::GetSocket (void) const
116 {
117     NS_LOG_FUNCTION (this);
118     return m_socket;
119 }
120

```

```

121 void
122 WsnApplication::DoDispose (void)
123 {
124     NS_LOG_FUNCTION (this);
125
126     m_socket = 0;
127     // chain up
128     Application::DoDispose ();
129 }
130
131 // Application Methods
132 void WsnApplication::StartApplication () // Called at time specified by Start
133 {
134     NS_LOG_FUNCTION (this);
135
136     // Create the socket if not already
137     if (!m_socket)
138     {
139         m_socket = Socket::CreateSocket (GetNode (), m_tid);
140         if (Inet6SocketAddress::IsMatchingType (m_peer))
141         {
142             m_socket->Bind6 ();
143         }
144         else if (InetSocketAddress::IsMatchingType (m_peer) ||
145                PacketSocketAddress::IsMatchingType (m_peer))
146         {
147             m_socket->Bind ();
148         }
149         m_socket->Connect (m_peer);
150         m_socket->SetAllowBroadcast (true);

```

```

151     m_socket->ShutdownRecv ();
152
153     m_socket->SetConnectCallback (
154         MakeCallback (&WsnApplication::ConnectionSucceeded, this),
155         MakeCallback (&WsnApplication::ConnectionFailed, this));
156     }
157     m_cbrRateFailSafe = m_cbrRate;
158
159     // Insure no pending event
160     CancelEvents ();
161
162     StartSending();
163 }
164
165 void WsnApplication::StopApplication () // Called at time specified by Stop
166 {
167     NS_LOG_FUNCTION (this);
168
169     CancelEvents ();
170     if(m_socket != 0)
171     {
172         m_socket->Close ();
173     }
174     else
175     {
176         NS_LOG_WARN ("WsnApplication found null socket to close in StopApplication");
177     }
178 }
179
180 void WsnApplication::CancelEvents ()

```

```

181 {
182     NS_LOG_FUNCTION (this);
183
184     if (m_sendEvent.IsRunning () && m_cbrRateFailSafe == m_cbrRate )
185     { // Cancel the pending send packet event
186         // Calculate residual bits since last packet sent
187         Time delta (Simulator::Now () - m_lastStartTime);
188         int64x64 t bits = delta.To (Time::S) * m_cbrRate.GetBitRate ();
189         m_residualBits += bits.GetHigh ();
190     }
191     m_cbrRateFailSafe = m_cbrRate;
192     Simulator::Cancel (m_sendEvent);
193     Simulator::Cancel (m_startStopEvent);
194 }
195
196 // Event handlers
197 void WsnApplication::StartSending ()
198 {
199     NS_LOG_FUNCTION (this);
200     m_lastStartTime = Simulator::Now ();
201     ScheduleNextTx (); // Schedule the send packet event
202 }
203
204 // Private helpers
205 void WsnApplication::ScheduleNextTx ()
206 {
207     NS_LOG_FUNCTION (this);
208
209     if (m_maxBytes == 0 || m_totBytes < m_maxBytes)
210     {

```

```

211         uint32_t bits = m_pktSize * 8 - m_residualBits;
212         NS_LOG_LOGIC ("bits = " << bits);
213         Time nextTime (Seconds (bits /
214             static_cast<double>(m_cbrRate.GetBitRate ()))); // Time till next pack
215
216         switch(m_pktGenPattern)
217         {
218             case 0:
219                 // suppose periodic model
220                 nextTime += Seconds(1.0/m_pktGenRate);
221                 break;
222             case 1:
223                 // suppose Poisson model
224                 {
225                     Ptr<UniformRandomVariable> m_uniformRandomVariable = CreateObject<UniformRandomVariabl
226                     double p = m_uniformRandomVariable->GetValue (0,1);
227                     double poisson, expo = exp(-m_pktGenRate);
228                     int k;
229
230                     poisson = expo;
231                     for(k=1; poisson < p; k++)
232                     {
233                         double temp = pow(m_pktGenRate, k);
234                         for(int i=1; i<=k; i++) temp /= i;
235                         poisson += temp*expo;
236                     }
237                     k--;
238                     if(k>0) nextTime += Seconds(1.0/k);
239                     else
240                     {

```



```

241         Simulator::Schedule (Seconds(1.0), &WsnApplication::ScheduleNextTx, this);
242         return;
243     }
244
245     break;
246 }
247 default:
248     break;
249 }
250 NS_LOG_LOGIC ("nextTime = " << nextTime);
251 m_sendEvent = Simulator::Schedule (nextTime,
252                                     &WsnApplication::SendPacket, this);
253 }
254 else
255 { // All done, cancel any pending events
256     StopApplication ();
257 }
258 }
259
260 void WsnApplication::SendPacket ()
261 {
262     NS_LOG_FUNCTION (this);
263
264     NS_ASSERT (m_sendEvent.IsExpired ());
265     LOADng::LOADngHeader hdr;
266     Ptr<Packet> packet = Create<Packet> (m_pktSize - sizeof(hdr));
267     Ptr<UniformRandomVariable> m_uniformRandomVariable = CreateObject<UniformRandomVariable> ();
268     int64_t temp = (m_uniformRandomVariable->GetInteger(0, m_pktDeadlineLen) + m_pktDeadlineMin) + Now
269
270     m_pktCount++;
271
272     hdr.SetDeadline(Time(temp));
273     NS_LOG_INFO(temp << ", " << hdr.GetDeadline());
274     packet->AddHeader(hdr);
275     m_txTrace (packet);
276     m_socket->Send (packet);
277     m_totBytes += m_pktSize;
278     if (InetSocketAddress::IsMatchingType (m_peer))
279     {
280         NS_LOG_INFO ("At time " << Simulator::Now ().GetSeconds ()
281                     << "s on-off application sent "
282                     << packet->GetSize () << " bytes to "
283                     << InetSocketAddress::ConvertFrom(m_peer).GetIpv4 ()
284                     << " port " << InetSocketAddress::ConvertFrom (m_peer).GetPort ()
285                     << " total Tx " << m_totBytes << " bytes");
286     }
287     else if (Inet6SocketAddress::IsMatchingType (m_peer))
288     {
289         NS_LOG_INFO ("At time " << Simulator::Now ().GetSeconds ()
290                     << "s on-off application sent "
291                     << packet->GetSize () << " bytes to "
292                     << Inet6SocketAddress::ConvertFrom(m_peer).GetIpv6 ()
293                     << " port " << Inet6SocketAddress::ConvertFrom (m_peer).GetPort ()
294                     << " total Tx " << m_totBytes << " bytes");
295     }
296     m_lastStartTime = Simulator::Now ();
297     m_residualBits = 0;
298     ScheduleNextTx ();
299 }
300 void WsnApplication::ConnectionSucceeded (Ptr<Socket> socket)
301 {
302     NS_LOG_FUNCTION (this << socket);
303     m_connected = true;
304 }
305
306 void WsnApplication::ConnectionFailed (Ptr<Socket> socket)
307 {
308     NS_LOG_FUNCTION (this << socket);
309 }
310
311 } // Namespace ns3
312

```



## Python codes that merges the above codes

### wscript

```
## -*- Mode: python; py-indent-offset: 4; indent-tabs-mode: nil; coding: utf-8; -*-
def build(bld):
    module = bld.create_ns3_module('LOADng', ['internet'])
    module.includes = '.'
    module.source = [
        'model/LOADng-rtable.cc',
        'model/LOADng-packet-queue.cc',
        'model/LOADng-packet.cc',
        'model/LOADng-routing-protocol.cc',
        'model/wsn-application.cc',
        'helper/LOADng-helper.cc',
        'helper/wsn-helper.cc',
    ]
    module_test = bld.create_ns3_module_test_library('LOADng')
    module_test.source = [
        'test/LOADng-testcase.cc',
    ]

    headers = bld(features='ns3header')
    headers.module = 'LOADng'
    headers.source = [
        'model/LOADng-rtable.h',
        'model/LOADng-packet-queue.h',
        'model/LOADng-packet.h',
        'model/LOADng-routing-protocol.h',
        'model/wsn-application.h',
        'helper/LOADng-helper.h',
        'helper/wsn-helper.h',
    ]
    if (bld.env['ENABLE_EXAMPLES']):
        bld.recurse('examples')

## bld.ns3_python_bindings()
```

### Scratch Folder

The main programs run on scratch folder

### ImprovedLOADng.cc

```
1  #include "ns3/netanim-module.h"
2  #include "ns3/core-module.h"
3  #include "ns3/network-module.h"
4  #include "ns3/internet-module.h"
5  #include "ns3/point-to-point-module.h"
6  #include "ns3/applications-module.h"
7  #include "ns3/animation-interface.h"
8  #include <ctype.h>
9  #include <math.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <time.h>
14
15 #include <fstream>
16 #include <iterator>
17 #include <vector>
18
19 #include "const.h"
20 #include "loadng.h"
21 #pragma GCC diagnostic ignored "-Wdeprecated-declarations"
22 #pragma GCC diagnostic ignored "-Wunused-variable"
23 #pragma GCC diagnostic ignored "-Wunused-value"
24 #pragma GCC diagnostic ignored "-Wwrite-strings"
25 #pragma GCC diagnostic ignored "-Wparentheses"
26 #include <iostream>
27
28 using namespace std;
29
30 int NUM NODES = 50;
```

```

31 int NETWORK_X = 100;
32 int NETWORK_Y = 100;
33 double B_POWER = 0.75;
34 double CLUSTER_PERCENT = 0.05;
35 int TOTAL_ROUNDS = 200;
36 double LOADng_RREQ_DISTANCE = 25;
37 int LOADng_RREQ_MESSAGE = 16;
38 double SCHEDULE_DISTANCE = 25;
39 void writefile(vector<int> vec);
40 int SCHEDULE_MESSAGE = 16;
41 int BASE_STATION_X_DEFAULT = 1000;
42 int BASE_STATION_Y_DEFAULT = 1000;
43 int DEAD_NODE = -2;
44 int MESSAGE_LENGTH = 8;
45 int TRIALS = 1;
46 struct sensor {
47     short xLoc;
48     short yLoc;
49     short lPeriods;
50     short ePeriods;

```

```

61 double bCurrent;
62 double bPower;
63 double pAverage;
64 int round;
65 int head;
66 int cluster_members;
67 int head_count;
68 };
69
70 struct sensor BASE_STATION;
71 double computeEnergyTransmit(float distance, int messageLength);
72 double computeEnergyReceive(int messageLength);
73 void initializeNetwork(struct sensor network[]);
74 float averageEnergy(struct sensor network[]);
75 struct sensor *loadConfiguration(char *filename);
76 int runLoadngSimulation(const struct sensor network[]);
77
78 int sensorTransmissionChoice(const struct sensor a);
79
80 int main(int argc, char *argv[]) {
81     struct sensor *network;

```

```

91
92 int i = 0;
93 // int j = 0;
94 int rounds_LOADng = 0;
95 int rounds_DIRECT = 0;
96 int modification = 0;
97 int found = FALSE;
98 // double average_comparison = 0.0;
99 char *filename = new char[10];
100
101 BASE_STATION.xLoc = BASE_STATION_X_DEFAULT;
102 BASE_STATION.yLoc = BASE_STATION_Y_DEFAULT;
103
104 cout << "proper file" << endl;
105
106 found = TRUE;
107
108 if (found == FALSE) {
109     strcpy(filename, ".config");
110     network = loadConfiguration(".config");
111 } else {
112     network = loadConfiguration(argv[found + 1]);
113 }
114
115 initializeNetwork(network);
116
117 for (i = 0; i < TRIALS; i++) {
118     rounds_LOADng += runLoadngSimulation(network);

```

```

121     initializeNetwork(network);
122 }
123
124 printf("The LOADng simulation was able to remain viable for %d rounds\n",
125        rounds_LOADng / TRIALS);
126
127 return 0;
128
129 } // end main function
130
131 int runLoadngSimulation(const struct sensor network[]) {
132     struct sensor *network_LOADng;
133     int i = 0; // indexing variables
134     int j = 0;
135     int k = 0;
136     int closest = 0;
137
138     int round = 0; // current round
139     int failed_transmit = 0; // round where a failed transmission occurred
140
141     int testing = 0; // testing variable, TO BE REMOVED
142     int bits_transmitted = 0; // count of bits transmitted
143     int power = FALSE;
144     int temp_cluster_members = 0;
145
146     double average_energy = 0.0;
147     double distance_X_old = 0.0;
148     double distance_Y_old = 0.0;
149
150
151     double distance_old = 0.0;
152     double distance_X_new = 0.0;
153     double distance_Y_new = 0.0;
154     double distance_new = 0.0;
155     int recent_round = 20;
156     double threshold = CLUSTER_PERCENT / (1 - (CLUSTER_PERCENT * (round % 20)));
157     double random_number;
158     int cluster_head_count = 0;
159     double percent_found = 0.0;
160     double mid_value = 0.0;
161     bool flag = 1;
162     vector<double> energy;
163     vector<double> profile;
164     vector<double> dir;
165     // old :: network_LOADng = (struct sensor *) malloc(NUM_NODES * sizeof(struct
166     // sensor));
167     network_LOADng = (struct sensor *) malloc(60 * sizeof(struct sensor));
168
169     for (i = 0; i < NUM_NODES; i++) {
170         network_LOADng[i].bPower = network[i].bPower;
171         network_LOADng[i].xLoc = network[i].xLoc;
172         network_LOADng[i].cluster_members = 0;
173         network_LOADng[i].yLoc = network[i].yLoc;
174         network_LOADng[i].ePeriods = network[i].ePeriods;
175         network_LOADng[i].bCurrent = network[i].bCurrent;
176         network_LOADng[i].bPower = network[i].bPower;
177         network_LOADng[i].pAverage = network[i].pAverage;
178     }
179
180
181     printf("\nRunning the LOADng Transmission Simulation \n");
182     // our iterating loop runs of total rounds, this is
183     // the expected lifetime of the network
184
185     while (averageEnergy(network_LOADng) > .10) {
186         // for(j = 0; j < TOTAL_ROUNDS; j++){
187
188         // here we recalculate all the variables
189         // which are round dependent
190         double check = (CLUSTER_PERCENT * float(round % 20));
191         cout << "check: " << check << endl;
192         double threshold = CLUSTER_PERCENT / (1 - (CLUSTER_PERCENT * (round % 20)));
193         cluster_head_count = 0;
194         cout << "round: " << round << endl;
195         cout << "threshold: " << threshold << endl;
196         // advertisement phase
197         // we determine which nodes will be cluster heads
198         for (i = 0; i < NUM_NODES; i++) {
199             if (network_LOADng[i].round < (j - recent_round) ||
200                 (j - recent_round == 0)) {
201                 // cout << "loop1 " << endl;
202                 if (network_LOADng[i].head != DEAD_NODE) {
203                     random_number = .00001 * (rand() % 100000);
204                     // cout << "random number " << random_number << endl;
205                     if (random_number <= threshold) {
206
207                         network_LOADng[i].head_count++;
208
209                         network_LOADng[i].round = j;
210                         network_LOADng[i].head = -1;

```

```

211         cluster_head_count++;
212     }
213 }
214 }
215 }
216 }
217
218 percent_found += (double)cluster_head_count / (double)NUM_NODES;
219
220
221 for (i = 0; i < NUM_NODES; i++) {
222     if (network_LOADng[i].head == -1) {
223         network_LOADng[i].bCurrent -=
224             computeEnergyTransmit(LOADng_RREQ_DISTANCE, LOADng_RREQ_MESSAGE);
225     } else {
226         network_LOADng[i].bCurrent -= computeEnergyReceive(LOADng_RREQ_MESSAGE);
227     }
228 }
229
230 // CLUSTER SET-UP PHASE
231
232
233 for (i = 0; i < NUM_NODES; i++) {
234     closest = -1;
235     if ((network_LOADng[i].head != -1) && network_LOADng[i].head != DEAD_NODE) {
236         // if the node's round is not equal to the
237         // current round, the node is not a cluster
238         // head and we must find a cluster head for
239         // the node to transmit to
240         for (k = 0; k < NUM_NODES; k++) {
241
242             if (network_LOADng[k].head == -1 && closest != -1) {
243                 distance_X_old =
244                     network_LOADng[i].xLoc - network_LOADng[closest].xLoc;
245                 distance_Y_old =
246                     network_LOADng[i].yLoc - network_LOADng[closest].yLoc;
247                 distance_old =
248                     sqrt(pow(distance_X_old, 2) + pow(distance_Y_old, 2));
249                 distance_X_new = network_LOADng[i].xLoc - network_LOADng[k].xLoc;
250                 distance_Y_new =
251                     network_LOADng[i].yLoc - network_LOADng[k].yLoc;
252                 distance_new =
253                     sqrt(pow(distance_X_new, 2) + pow(distance_Y_new, 2));
254                 if (distance_new < distance_old) closest = k;
255                 } else if (network_LOADng[k].head == -1 && closest == -1) {
256                     closest = k;
257                 }
258             }
259             network_LOADng[i].head = closest;
260             network_LOADng[closest].cluster_members++;
261         }
262     }
263
264     for (i = 0; i <= NUM_NODES; i++) {
265         if (network_LOADng[i].head == -1) {
266             // if the node is going to be a cluster head, it transmits
267             // the schedule to the other nodes
268             network_LOADng[i].bCurrent -=
269                 computeEnergyTransmit(SCHEDULE_DISTANCE, SCHEDULE_MESSAGE);
270         } else
271             network_LOADng[i].bCurrent -= computeEnergyReceive(SCHEDULE_MESSAGE);
272     }
273
274     // DATA TRANSMISSION
275     // non cluster heads send their data to the cluster heads who then
276     // broadcast the data to the base station
277     for (i = 0; i < NUM_NODES; i++) {
278         network_LOADng[i].lPeriods++;
279         if (network_LOADng[i].head != -1 && network_LOADng[i].head != DEAD_NODE) {
280             distance_X_new =
281                 network_LOADng[i].xLoc - network_LOADng[network_LOADng[i].head].xLoc;
282             distance_Y_new =
283                 network_LOADng[i].yLoc - network_LOADng[network_LOADng[i].head].yLoc;
284             distance_new = sqrt((pow(distance_X_new, 2) + pow(distance_Y_new, 2)));
285             network_LOADng[i].bCurrent -=
286                 computeEnergyTransmit(distance_new, MESSAGE_LENGTH);
287             network_LOADng[network_LOADng[i].head].bCurrent -=
288                 computeEnergyReceive(MESSAGE_LENGTH);
289         }
290         if (network_LOADng[i].bCurrent < 0.0 && network_LOADng[i].head != -1) {
291             network_LOADng[i].head = DEAD_NODE;
292         }
293     }
294 }
295
296 for (i = 0; i <= NUM_NODES; i++) {
297     if (network_LOADng[i].head == -1) {
298         distance_X_new = network_LOADng[i].xLoc - BASE_STATION.xLoc;
299         distance_Y_new = network_LOADng[i].yLoc - BASE_STATION.yLoc;
300         distance_new = sqrt(pow(distance_X_new, 2) + pow(distance_Y_new, 2));

```

```

301     double energy_enough =
302         network_LOADng[i].bCurrent -
303         computeEnergyTransmit(
304             distance_new,
305             (MESSAGE_LENGTH * (network_LOADng[i].cluster_members + 1)));
306
307
308     if (round < 2001) {
309         double required_energy =
310             computeEnergyTransmit(distance_new, MESSAGE_LENGTH);
311         network_LOADng[network_LOADng[i].head].bCurrent -=
312             computeEnergyReceive(MESSAGE_LENGTH);
313         energy.push_back(required_energy);
314     }
315     if (energy_enough > 0.0) {
316         network_LOADng[i].bCurrent -= computeEnergyTransmit(
317             distance_new,
318             (MESSAGE_LENGTH * (network_LOADng[i].cluster_members + 1)));
319         bits_transmitted +=
320             (MESSAGE_LENGTH * (network_LOADng[i].cluster_members + 1));
321     } else {
322         failed_transmit++;
323         network_LOADng[i].head = DEAD_NODE;
324         // if (flag) {
325             cout << "LOADng's first node dies: " << round << endl;
326             flag = 0;
327         }
328         temp_cluster_members = network_LOADng[i].cluster_members + 1;
329         network_LOADng[i].bCurrent += computeEnergyTransmit(
330             distance_new,
331             (MESSAGE_LENGTH * (network_LOADng[i].cluster_members + 1)));
332     while (power == FALSE) {
333         if ((network_LOADng[i].bCurrent -
334             computeEnergyTransmit(
335                 distance_new, (MESSAGE_LENGTH * temp_cluster_members))) >
336             0) {
337             bits_transmitted += (MESSAGE_LENGTH * temp_cluster_members);
338             power = TRUE;
339         } else
340             temp_cluster_members--;
341         if (temp_cluster_members == 0) power == TRUE;
342     }
343 }
344 }
345 }
346 double max = 0.0000;
347 for (unsigned int i = 0; i < energy.size(); i++) {
348     if ((energy[i]) > max) max = energy[i];
349 }
350 // cout << max << endl;
351 profile.push_back(max);
352
353 // round has completed, increment the round count
354 for (i = 0; i <= NUM_NODES; i++) {
355     network_LOADng[i].cluster_members = 0;
356     if (network_LOADng[i].bCurrent > 0.0) network_LOADng[i].head = 0;
357 }
358
359 cluster_head_count = 0;
360
361
362     dir.push_back(averageEnergy(network_LOADng));
363     if (round == 1000)
364         cout << "network_pwr: " << averageEnergy(network_LOADng) << endl;
365     round += 1;
366     j = round;
367 }
368 ofstream d_stats;
369 d_stats.open("LOADng_simulation.txt");
370 // d_stats << "loadng_simulation" << endl;
371 for (unsigned int i = 0; i < dir.size(); i++)
372     d_stats << dir[i] << endl; // d_stats << dir[i] << endl ;
373
374 return round;
375 } // end runLoadngSimulation function
376
377
378 double computeEnergyTransmit(float distance, int messageLength) {
379
380
381     float E_elec = 50 * pow(10, -9);
382     float epsilon_amp = 100 * pow(10, -12);
383     double EnergyUse = 0.00;
384
385     EnergyUse = (messageLength * E_elec) +
386                 (messageLength * epsilon_amp * pow(distance, 2));
387
388     return EnergyUse;
389 } // end computeEnergyTransmit function
390

```

```

391 double computeEnergyReceive(int messageLength) {
392     return (messageLength * (50 * pow(10, -9)));
393 } // end computeEnergyReceive function
394
395 void initializeNetwork(struct sensor network[]) {
396     int i = 0;
397     srand((unsigned int)time(0));
398     for (i = 0; i < NUM_NODES; i++) {
399         network[i].xLoc = rand() % NETWORK_X;
400         network[i].yLoc = rand() % NETWORK_Y;
401         network[i].lPeriods = 0;
402         network[i].ePeriods = TOTAL_ROUNDS;
403         network[i].bCurrent = B_POWER;
404         network[i].bPower = B_POWER;
405         network[i].pAverage = 0.00;
406         network[i].round = FALSE;
407         network[i].head = FALSE;
408     }
409 } // end initializeNetwork function
410
411 struct sensor *loadConfiguration(char *filename) {
412     // run for
413     int total_size = 50;
414     int i = 0;
415
416     int axis = 0;
417
418     char buf[80]; // buffer for reading lines from file
419     char copy[80]; // temporary memory for storing the
420     char cut[8]; // integer result of a line
421     char cut2[8];
422
423     FILE *fp;
424     // open file with settings as read only,
425     // if file cannot be opened, allocate the memory
426     // based on the default values and return
427     if ((fp = fopen(filename, "r")) == NULL) {
428         return (struct sensor *)malloc(60 * sizeof(struct sensor));
429     }
430
431     // read entire configuration file
432     while (fgets(buf, 80, fp) != NULL) {
433         // if the line is the NUM_NODES variable
434         strncpy(cut, buf, 7);
435         strncpy(cut2, cut, 7);
436         if (cut[0] == '#') {
437             i = 0;
438         } else if (cut[0] == 'N' && cut[1] == 'U') {
439             i = 9;
440             // advance pointer to = sign
441             while (buf[i] != '=') {
442                 i++;
443             }
444             // then advance pointer one more space
445             // past the equal sign
446
447             i++;
448             // if there is any whitespace, advance
449             // pointer past it
450             while (buf[i] == ' ') {
451                 i++;
452             }
453             // copy the contents of the line read from the file
454             // into the copy variable from the iterator i
455             strcpy(copy, buf + i);
456             // convert the copied characters to NUM_NODES
457             NUM_NODES = atoi(copy);
458         }
459
460         else if (cut[0] == 'N' && cut[1] == 'E') {
461             i = 6;
462             // advance pointer to = sign
463             while (buf[i] != '=') {
464                 if (buf[i] == 'X') axis = 1;
465                 if (buf[i] == 'Y') axis = 0;
466                 i++;
467             }
468             // then advance pointer one more space
469             // past the equal sign
470             i++;
471             // if there is any whitespace, advance
472             // pointer past it
473             while (buf[i] == ' ') {
474                 i++;
475             }
476             // copy the contents of the line read from the file
477
478
479
480

```



```

481     // into the copy variable from the iterator i
482     strcpy(copy, buf + i);
483     if (axis == 1) NETWORK_X = atoi(copy);
484     if (axis == 0) NETWORK_Y = atoi(copy);
485 } else if (cut[0] == 'R' && cut[2] == 'U' && cut[4] == 'D') {
486     i = 6;
487     // advance pointer
488     while (buf[i] != '=') {
489         i++;
490     }
491     // then advance pointer one more space
492     // past the equal sign
493     i++;
494     // if there is any whitespace, advance
495     // pointer past it
496     while (buf[i] == ' ') {
497         i++;
498     }
499     // copy the contents of the line read from the file
500     // into the copy variables from the iterator i
501     strcpy(copy, buf + i);
502     // convert the copied characters to an integer
503     // and store it in total rounds
504     TOTAL_ROUNDS = atoi(copy);
505 } else if (cut2[0] == 'B' && cut2[1] == '_' && cut2[2] == 'P') {
506     i = 7;
507     // advance pointer to = sign
508     while (buf[i] != '=') {
509         i++;
510     }
511
512     // then advance pointer one more space
513     // past the equal sign
514     i++;
515     // if there is any whitespace, advance
516     // pointer past it
517     while (buf[i] == ' ') {
518         i++;
519     }
520     // copy the contents of the line read from the file
521     // into the copy variable from the iterator i
522     strcpy(copy, buf + i);
523     // convert the copied characters to B_POWER
524     B_POWER = atof(copy);
525 }
526 }
527 if (fclose(fp) != 0) {
528     printf("Error closing config file.\n");
529     exit(1);
530 }
531
532 return (struct sensor *)malloc(60 * sizeof(struct sensor));
533 }
534 // end loadConfiguration function
535
536 float averageEnergy(struct sensor network[]) {
537     float average = 0.0;
538     float starting_power = 0.00;
539     float current_power = 0.00;
540     int i = 0;
541
542     for (i = 0; i <= NUM_NODES; i++) {
543         starting_power += network[i].bPower;
544         current_power += network[i].bCurrent;
545     }
546
547     return current_power / starting_power;
548 } // end averageEnergy function
549
550
551 int sensorTransmissionChoice(const struct sensor a) {
552     // Preconditions: a is an initialized sensor
553     // Postconditions: 1 is returned if a should transmit for
554     //                 the current round, 0 otherwise.
555     int remaining_periods = 0;
556
557     remaining_periods = a.ePeriods - a.lPeriods;
558     if ((remaining_periods * a.pAverage) > a.bCurrent)
559         return 0;
560     else
561         return 1;
562 }

```