San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

1-1-2010

WiSeNetor: A Scalable Wireless Sensor Network Simulator

Gauri Krishna Gokhale
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Recommended Citation
Gokhale, Gauri Krishna, "WiSeNetor: A Scalable Wireless Sensor Network Simulator" (2010). *Master's Projects.* Paper 29.

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact Library-scholarworks-group@sjsu.edu.

WiSeNetor: A Scalable Wireless Sensor Network Simulator

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Gauri Krishna Gokhale

May 2010

© 2010

Gauri Krishna Gokhale

ALL RIGHTS RESERVED

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Thesis Titled

WiSeNetor: A Scalable Wireless Sensor Network Simulator

by

Gauri Krishna Gokhale

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp, Department of Computer Science Date

Dr. Robert Chun, Department of Computer Science Date

Dr. Chaitanya Gharpure, Google Inc. Date

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research Date

ABSTRACT

WiSeNetor is a teaching and a research tool that simulates a scalable wireless sensor network on a single computer, based on the “Spamulator” (Aycock, J., Crawford, H., & deGraaf, R., 2008) which simulates the Internet on a single computer. Routing protocols and network discovery algorithms used in mesh networks and cluster tree networks can be demonstrated using this tool.

WiSeNetor contains a network creation module, simulated network devices and it simulates routing algorithms. The network creation module spawns a network according to user specified network type, where the type can be a cluster tree or mesh. In this process, neighbor tables are populated and the Spamulator is initiated. The underlying network module of the Spamulator has been reused in WiSeNetor to achieve better scalability.

Each simulated network device has an associated server program and a client program that process incoming requests and forward them to appropriate neighboring nodes, respectively. Network devices also log all of the service messages in individual log files that may be used to trace the routing or network discovery process.

WiSeNetor has achieved scalability up to 15,000 nodes in the network. Message latency and the average number of hops during simulation testing were comparable to the findings in (Eamsomboon,P., Keeratiwintakorn,P., & Mitrpant, C, 2008) which validates the WiSeNetor.

ACKNOWLEDGEMENT

Many thanks to Dr. Mark Stamp for his guidance and for a great learning experience. His advice enabled me to make right decisions and choose appropriate approach for the project. Great thanks to Dr. John Aycock, University of Calgary, for sharing his brilliant software, the Spamulator.

This project would not have been possible without the support of my husband, Sameer, and my little daughter, Tanvi. I am immensely grateful for the good wishes and blessings of the elders in my family.

TABLE OF CONTENTS

INTRODUCTION **9**

WHAT IS A WIRELESS SENSOR NETWORK? 9

Sensor Network application classes *9*

THE 802.15.4 STANDARD AND ZIGBEE **12**

STAR TOPOLOGY 12

CLUSTER NETWORK TOPOLOGY 13

CLUSTER TREE NETWORK TOPOLOGY 16

MESH NETWORK TOPO 17

Route discovery process *18*

SPAMULATOR: SIMULATING INTERNET ON A SINGLE COMPUTER **20**

ARCHITECTURE OF SPAMULATOR 20

DESIGN CRITERIA FOR WISENETOR **22**

MAIN DESIGN CONSIDERATION 22

OTHER DESIGN CONSIDERATIONS 22

ROUTING ALGORITHM 22

CHOOSING CORRECT INFRASTRUCTURE TO ACHIEVE SCALABILITY 22

ARCHITECTURE OF WISENETOR **24**

IMPLEMENTATION **25**

Operating System *25*

Programming language and scripting language *25*

SIMULATION RESULTS **42**

VALIDATION OF WISENETOR **45**

CONCLUSION **47**

FUTURE WORK **47**

REFERENCES **48**

APPENDIX A **50**

LIST OF FIGURES

FIGURE 1. TYPICAL MULTI-HOP WIRELESS SENSOR NETWORK 9
FIGURE 2. ENVIRONMENTAL DATA COLLECTION NETWORK DEPLOYED IN A FARM 10
FIGURE 3: CLUSTER NETWORK 14
FIGURE 4: CLUSTER TREE TOPOLOGY 16
FIGURE 5: TYPICAL MESH NETWORK 18
FIGURE 6: SPAMULATOR ARCHITECTURE (AYCOCK ET AL., 2008) 20
FIGURE 7: ARCHITECTURE OF WISENETOR 24
FIGURE 8: STATUS UPDATE PROTOCOL AT SERVER SIDE 28
FIGURE 9: STATUS UPDATE AT CLIENT SIDE 29
FIGURE 10: WORKING OF ROUTING PROTOCOL AT SERVER SIDE 31
FIGURE 11: WORKING OF ROUTING PROTOCOL AT CLIENT SIDE 32
FIGURE 12: RREQ MESSAGE FRAME IN WISENETOR 34
FIGURE 13: RREP MESSAGE FRAME IN WISENETOR 34
FIGURE 14: RTE MESSAGE FRAME IN WISENETOR 35
FIGURE 15: WORKING OF MESH SERVER PROGRAM 36
FIGURE 16: SUB PROCESS THAT HANDLES RREQ REQUEST 37
FIGURE 17: SUB PROCESS THAT HANDLES RREP REQUEST 38
FIGURE 18: SUB PROCESS THAT HANDLES RTE MESSAGE 39
FIGURE 19: HANDLING OF RREQ MESSAGE BY MESH CLIENT PROGRAM 40
FIGURE 20: HANDLING OF RREP MESSAGE BY MESH CLIENT PROGRAM 41
FIGURE 22: TIME ELAPSED IN SPAWNING MESH NETWORK VS NUMBER OF NODES 42
FIGURE 23: END-TO-END DELAY FOR INCREASING NUMBER OF NODES 45
FIGURE 24: NUMBER OF HOPS AS NUMBER OF NODES INCREASES 46

LIST OF TABLES

TABLE 1: ROUTING TABLE FOR PAN SHOWN IN FIGURE 3 15
TABLE 2: ROUTING TABLE FOR NODE 6 SHOWN IN FIGURE 3 15
TABLE 3. SCALABILITY OF SENSE 23

Introduction

What is a wireless sensor network?

A wireless sensor network (WSN) is a wireless network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations (“Wireless Sensor Network”, n.d.) .

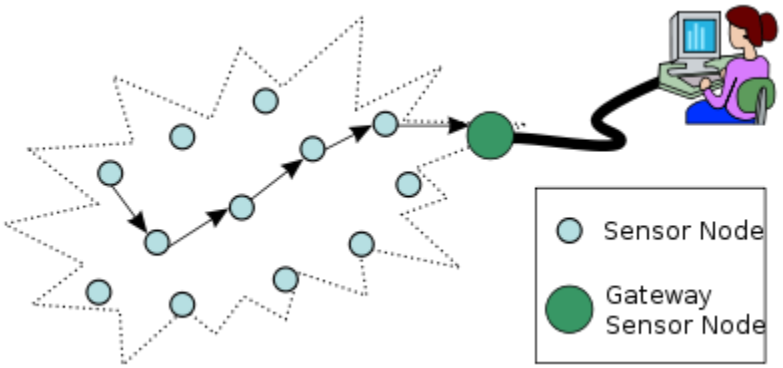


Figure 1. Typical multi-hop wireless sensor network

The main features of a WSN are low power consumption, dynamic detection of network topology, and low maintenance.

Sensor Network application classes

For WSNs, there are three main application classes:

1. Environmental data collection,
2. Security monitoring,
3. Sensor node tracking.

In environmental data collection, a scientist might want to gather environmental data pertaining to temperature, pressure, or humidity through a large number of sensor nodes deployed over a vast area. This will help the scientist to detect trends in the environment. The sensors will route the data to a base station. Environmental data collection applications typically use tree-based routing topologies where each routing tree is rooted at high-capability nodes that sink data. Data is periodically transmitted from child node to parent node, up the tree-structure until it reaches the sink. With tree-based data collection, each node is

responsible for forwarding the data of all its descendants (Hill, 2004). Figure 2 denotes an environmental data collection system ("Picture For Environmental Data Collection System", n.d.).

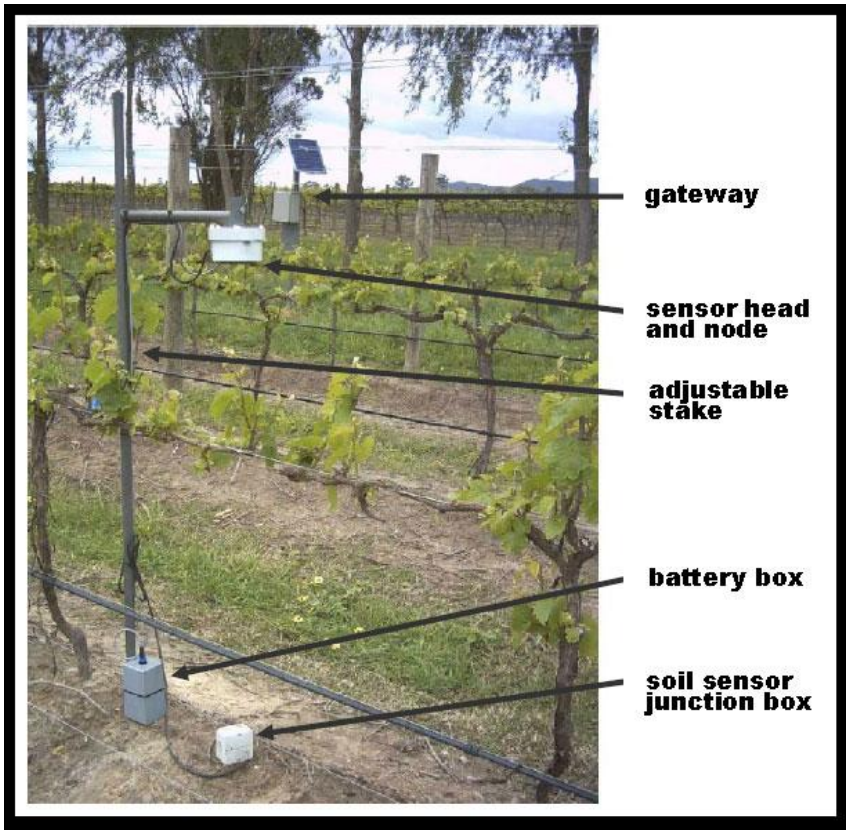


Figure 2. Environmental data collection network deployed in a farm

Typically, these applications include monitoring of humidity or temperature in a farm to protect the crop from damage due to unpredictable weather conditions.

Security monitoring applications: Sensors monitor any changes in surrounding conditions. They send a data report only when an anomaly occurs, unlike the environmental data collection applications which send data continuously to the desired base stations. Security monitoring applications are used in some military applications and to monitor fire-hazards (Hill, 2004).

Node tracking applications: These applications are used when one would like to track valuable assets or personnel. Objects to be tracked are tagged by sensor nodes. The location of these tagged objects is recorded as they move along the route (Hill, 2004).

The 802.15.4 standard and ZigBee

Although the IEEE 802.15.4 standard has been designed as a MAC layer specification for wireless networks, it does support certain network topologies. The IEEE 802.15.4 standard supports multiple network topologies like star networks and peer-to-peer networks like mesh, tree, cluster and cluster tree networks. It only suggests certain network routing algorithms for the above networks.

The star network is a single-hop network where the communication devices are in range of a single intermediate device. In multi-hop, peer-to-peer networks communication happens via multiple intermediate devices.

802.15.4 standard defines two types of devices called full functional device (FFD) and reduced functional device (RFD). A FFD is a network coordinator or a router that provides full set of services to the network and a RFD is an end device that provides limited or reduced set of services. In this discussion, we refer to a FFD as a PAN coordinator (PAN coordinator is responsible for functions like spawning a network and also maintaining the network) or a router (Router has capability to route a message) and RFD as an end device (It is not capable of routing a message). We may refer to a network device also as a sensor node. Every network has only one PAN coordinator. Selection of a PAN coordinator is application specific. In some applications, there may be a dedicated PAN coordinator where the consumer does not have any control over the network design. Some applications support event-determined PAN coordinator where a FFD could become a PAN coordinator given an external stimulus. Self-determined PAN coordinator is where all FFDs in a network start competing to become a PAN coordinator upon power up. Location determining network applications use the self-determined PAN coordinator. In WiSeNetor, we use only one dedicated PAN coordinator.

The IEEE 802.15.4 standard defines the physical (PHY) layer and the medium access control (MAC) sub-layer. The ZigBee Alliance builds on top of the IEEE 802.15.4 and defines the network (NWK) layer and the framework for the application layer for wireless sensor network. The Zigbee specification defines three kinds of network topologies, namely, star topology, cluster topology and mesh topology (Zigbee, 2008).

Star Topology

In a star network, an FFD can become a PAN coordinator and behave like a master device. All the other network devices, FFD or RFD, behave like slave devices. Star networks are typically used in applications that have limited coverage area. The PAN coordinator is in control of all the activities happening in

the network. It is aware of all the transactions happening among other network devices that are its child devices. Therefore, every device needs a single entry, that of PAN coordinator, in its ACL (Access Control List). Routing is very simple in star network and involves single message exchange between the PAN coordinator and the child devices. Thus, there is no need of a complicated routing algorithm. It is important to know that a star network can not be used in large scale sensor networks because of scalability issues.

There are 3 kinds of network topologies suited for large scalable sensor networks mentioned below. These are all peer-to-peer network topologies.

1. Cluster network topology
2. Cluster tree topology
3. Mesh network topology

Cluster Network Topology

A cluster network is based on the concept of parent-child relationship. First device to join a cluster network becomes the PAN coordinator. Thereafter, a new device can join the network as a child of the PAN coordinator. Subsequent devices can join the network as a child of PAN or a parent that is closest to the PAN. Any parent device can have multiple children or grandchildren but each child node can have only one parent.

Network Discovery: Each network device transmits beacons. A device that wants to join the network may hear these beacons and it chooses its parent that is closest to the PAN coordinator. The PAN coordinator is central to managing the structure of the network and the network's maintenance.

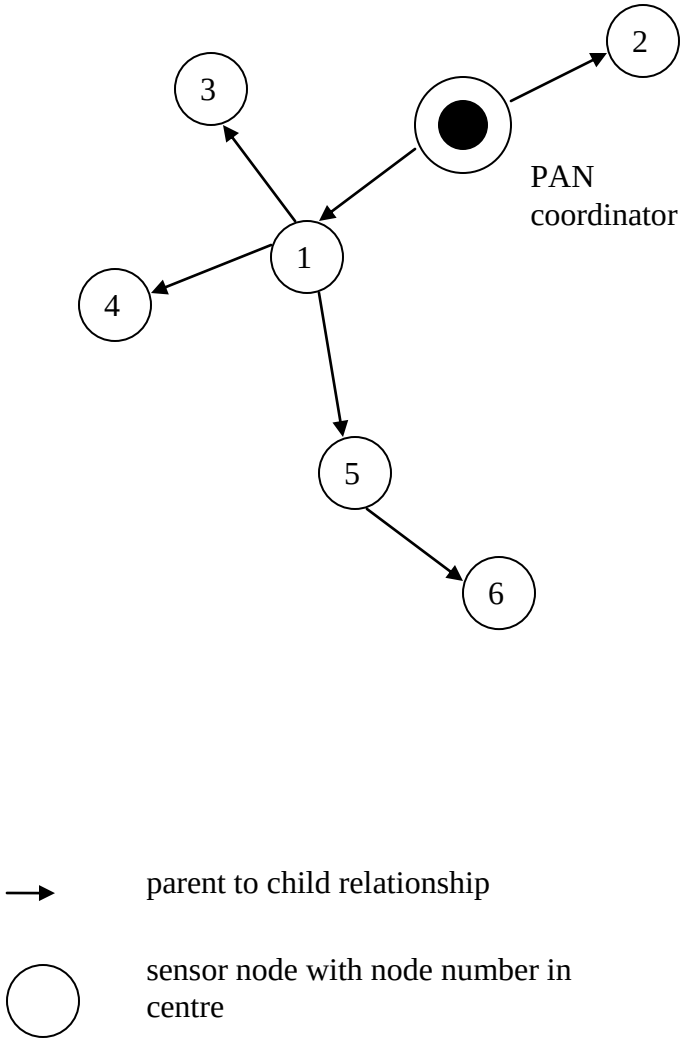


Figure 3: Cluster network

The PAN coordinator maintains information about the nodes in the network using the status update protocol. The PAN coordinator can find broken communication links in the network using status update protocol or even find newly added nodes in the network.

The PAN coordinator sends a status update message to all parents. All the parents relay this status update message to their child nodes. The message is forwarded until the leaf nodes are reached. Leaf nodes are the nodes which do not have any child nodes.
The leaf nodes send a status update response message, containing their address, to their parents. The parents eavesdrop on the message and update

this information in their own routing tables. Parent nodes also aggregate information contained in the status update response message from their child nodes. They send this newly updated status update response message to their parents. This process eavesdropping on the message and aggregating information continues until the PAN coordinator is reached. Now, the PAN coordinator has all the updated information about its network.

Routing

Routing Table: The routing table stores a list of network devices downstream and also the address of its parent. Table 1 depicts a typical routing table for the PAN in cluster network topology. First row contains the address of a node's parent. For PAN, there is no parent. From the second row onwards, first column contains addresses of all the nodes in the network and second column contains the address of the parent node that is directly child of PAN.

Table 1: Routing Table for PAN shown in figure 3

Parent	-
1	1
2	2
3	1
4	1
5	1
6	1

Table 2: Routing Table for node 6 shown in figure 3

Parent	5
6	6

Routing Algorithm:

In a cluster network routing algorithm, a network device checks first if the destination device or node has an entry in the routing table. If an entry is found in the routing table, the message is passed to the appropriate device downstream. If an entry is not found, the message is routed to the parent considering that the parent will have more information about the network. In worst case scenario, the message would be routed up to the PAN coordinator. In this type of network, the message is guaranteed delivery to its destination.

One advantage of a cluster network is smaller size of the routing table. The routing table size is smaller because table entry for every possible node is not

required. Smaller table size means lower memory requirement which, in turn, means lower product cost.

One major disadvantage of a cluster network is unequal distribution of message traffic through the network. The nodes logically closer to the PAN coordinator may be involved more, in the routing of messages, than the nodes logically farther than the PAN. This would lead to unequal battery life among the nodes. Unequal battery life would lead to partitioning of the network which means network failure.

Cluster Tree Network Topology

As the number of nodes in a cluster network increases, the routing table size becomes impractically large. The routing table size might increase because a node might have to include every child or grandchild in the routing table. To mitigate this problem, hierarchy could be introduced to the cluster network such that the network has multiple smaller clusters. The resulting network is called cluster-tree network topology.

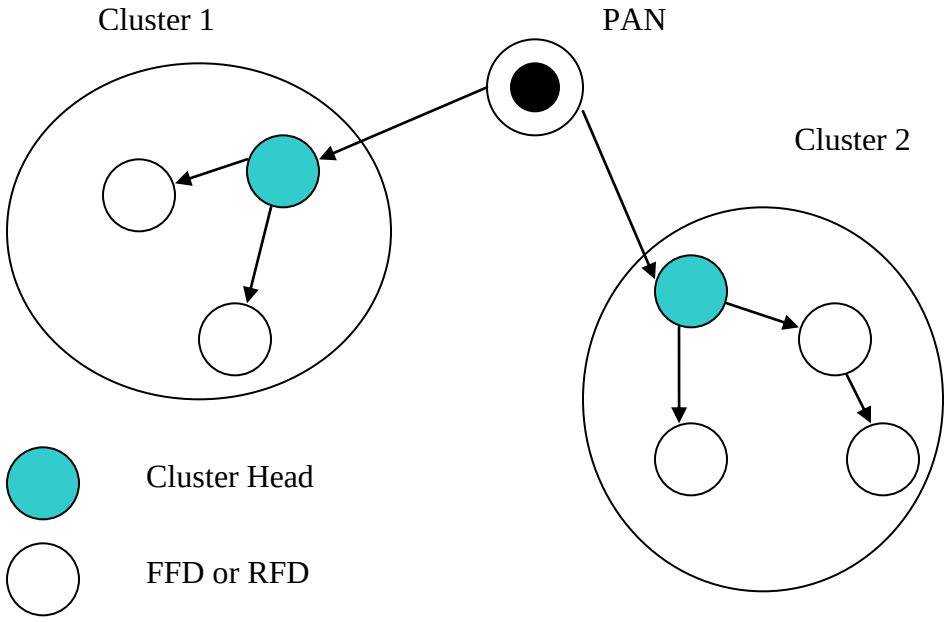


Figure 4: Cluster Tree Topology

Network address of every node consists of two parts; a cluster identifier and a network identifier. There is one PAN coordinator which manages the entire network and there are several cluster heads or cluster coordinators that manage different clusters.

Status update process is done in the similar way like that of the cluster network.

Routing:
In a cluster tree network, a device first checks the routing table to see if the cluster identifier of the destination device is present. If a routing table entry is not present, which means cluster is unknown; the message is routed to its parent device. If the cluster identifier is present, then the routing table is searched to check if network identifier is present. From this point, the routing is continued just like in the cluster network topology.

Mesh Network Topology

In a mesh network, routing is done in a decentralized way. It allows full peer-to-peer communication.

Routing: The PAN coordinator or router may store a routing table to be able to relay a message. A routing table consists of various fields like the destination address (could be coordinator, router or an end device that would be the final receiver of the message), next hop address (could be coordinator or router that would relay the message), status (possible values could be active, inactive, discovery underway, discovery failed, validation underway), no route cache(flag that indicates that destination does not store route address, route record required(flag indicating route record command frame needs to be sent to the destination prior to next data packet), group ID flag (indicates destination address is a Group ID).

A device is said to have routing capacity if it has the ability to route a message to the destination device using its routing table. A coordinator or a router has routing table capacity. A coordinator or a router also maintains a route discovery table during the route discovery process. A route discovery table consists of fields like route request ID, source address, sender address, forward cost, residual cost and expiration time.

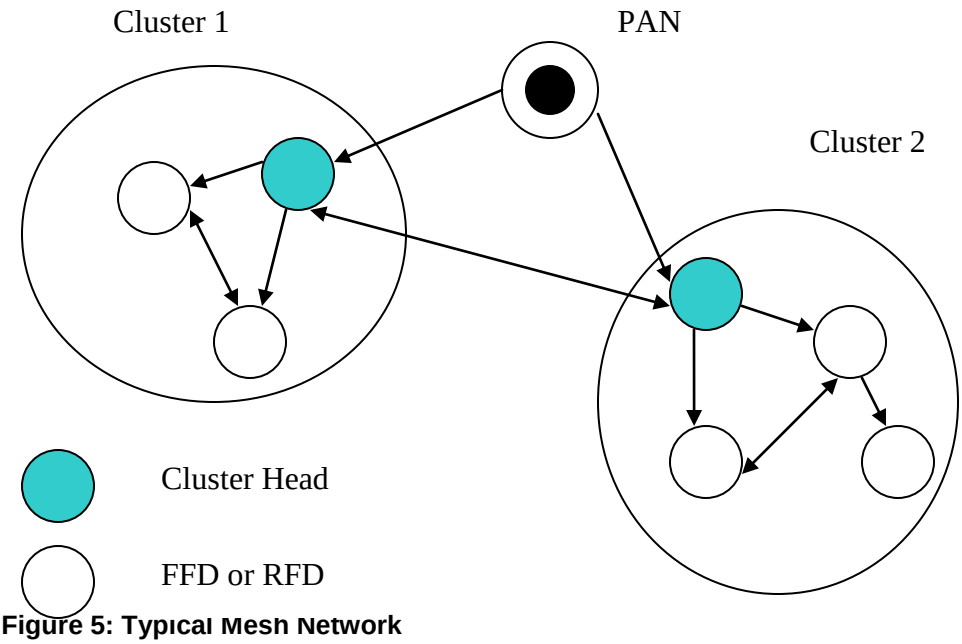


Figure 5: Typical Mesh Network

Route discovery process

In the route discovery process (Zigbee, 2008), the network devices together find and establish a route to destination device. The route discovery process begins if there doesn't exist a routing table entry with the same route request ID for the specified destination address.

Every device employs a route request ID counter. Each time a route request command is issued, the device increments the counter and is stored as route request ID field in the route discovery table. The route request command frame (RREQ) sets following fields:

1. Command Frame identifier is set to route request
2. Route request ID is set to the value stored in the route discovery table.
3. Source address
4. Sender address
5. Destination address

The device can now broadcast this frame to all its neighboring devices that have routing capacity. All the end devices shall discard the route request command frame. A device checks to see if a route discovery table exists and that an entry with the route request ID exists in the table. If not, the device creates a route discovery entry. If a device finds that it is indeed the destination device, it constructs the route request reply (RREP) message or command frame. This

RREP is sent to the sender of the RREQ. The RREP command frame consists of following fields:

1. Route request ID is set to the value stored in the route discovery table.
2. Originator's address, that is, the device that has created the RREP command frame and is forwarding it.
3. Destination address

After receiving the RREP message, a device creates a routing table entry for the specified destination device. It sets the next hop address to the sender of the RREP message and the destination address specified in the RREP message. This RREP message is relayed to the device that sent RREQ message, corresponding to the entry in the route discovery table. This process of relaying the RREP message to the sender of RREQ message and creating routing table entry completes when the source device finally gets the RREP message. The route discovery table entry is discarded after a certain expiration time.

It can be concluded that flat networks like the star network are not suitable for networks that are highly scalable. If communication is limited to a single base station, there will be communication overhead and management delay and the performance will also be limited. Clustering can be done to achieve better performance. Certain number of sensors can be part of a cluster which can be managed by a cluster head. Therefore, a cluster-tree network topology with mesh network in every cluster will be a good solution for a self-healing, self-organizing and highly scalable sensor network.

Spamulator: Simulating Internet on a single computer

The Spamulator has been developed by John Aycock, Heather Crawford and Rennie deGraaf of the University of Calgary, AB, Canada (Aycock, Crawford, & deGraaf, 2008). It simulates the Internet on a single computer.
The Spamulator simulates network services provided by a million domains. It is lightweight in resource usage and runs on a single computer. Interaction with the Spamulator can be done using any unmodified client software, such as web browsers. Users can write their own software to use the simulated Internet using any programming language or without any special libraries. The Spamulator can be used to simulate email harvesting and bulk mailing, among other features of Internet (Aycock et al., 2008).

Architecture of Spamulator

The architecture of Spamulator is kept simple. Network topology, latency or failures have not been simulated because they are not needed in the intended use of Spamulator.

Following are the four main building blocks of WiSeNetor.

1. NeRD or Network Rerouting Daemon. It is also called LNS or loopback network simulator
2. Client program,
3. Local DNS Server,
4. Simulated Sever.

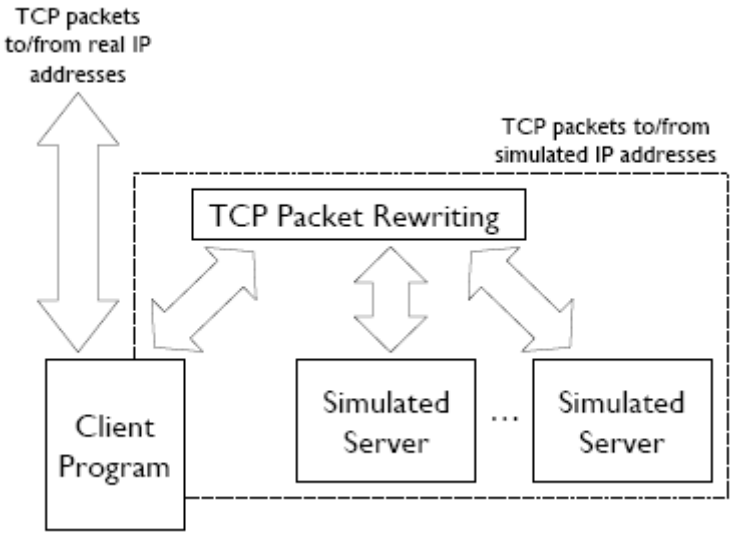


Figure 6: Spamulator Architecture (Aycock et al., 2008)

NeRD or network rerouting daemon is the core of Spamulator. The NeRD examines a packet coming from a client program and then decides how to transform it into a new packet and then reroutes it to a simulated server. If incoming packet needs a new connection, appropriate server program is launched. A server is a simple executable program written in C language and it uses the concept of pipes to communicate with the NeRD. Exchange of information between the server and NeRD include IP addresses, ports.
The local DNS Server is a standard unmodified and authoritative for all top-level domains that have been simulated.

Design Criteria for WiSeNetor

Main design consideration

WiSeNetor should be highly scalable which will enable the testing of a large network in a resource-constrained laboratory. It will be useful for teaching and could also prove to be a useful research tool.

Other design considerations

Self-organization: WSNs were designed to have low maintenance cost. Therefore, each sensor device should start participating in a network without any need for addressing, association, or any kind of configuration. This idea of self-organization comes from the ad-hoc networks. The network employs the proper message path from source to destination (Gutierrez, Callaway, & Barrett, 2003).

Routing Algorithm

We need to have a routing algorithm considering various topologies and the self-organizing feature of the WSN. For the cluster network and cluster tree topology, routing algorithm discussed under *Cluster Network Topology* is implemented. For the mesh network topology, Zigbee Routing Protocol is implemented.

Choosing correct infrastructure to achieve scalability

Already existing simulators were studied including ns2 and SENSE and their scalability was tested.

NS2 is a popular network simulator which simulates different types of protocols and networks. It does not scale well which is the major requirement of the project. It scales only up to 500 nodes. Also because of its object-oriented design there is interdependence between modules which makes addition of a new protocol even more difficult (Source Forge, Network simulator).

SENSE is a simulator completely dedicated to wireless sensor networks. SENSE is well designed and well-structured software. The scalability of SENSE was also checked. In order to do so, the simulation parameters in the sim_routing.cc file were modified and the output was printed to a text file (Lisee, Chen, G., Szymanski, B., & Rensselaer Polytechnic Institute, 2006).

Table 3. Scalability of SENSE

Number of nodes	Time taken to execute the program in minutes	Result
110 (default value)	5	Pass
500	35	Pass
1000	More than 180	Fail

Table 3 summarizes the findings of an experiment conducted to test the scalability of SENSE. SENSE could not scale more than 1000 nodes. From the above findings, it was concluded that even SENSE does not scale well.

Spamulator scales better than SENSE, up to a million nodes. Spamulator has a very simple design and hence, Spamulator is selected as basis for WiSeNetor.

Modules from Spamulator that are reused

1. The connection infrastructure can be reused in WiSeNetor.
2. The part implementing scalability can also be reused.

Modules that need to be implemented

1. Network topology detection module.
2. Routing Algorithm: Routing is not an issue in Spamulator because the Internet can be a collection of various networks with no specific topology. Therefore, we need to design a routing algorithm for WiSeNetor.
3. Gateway: An intelligent unit which processes data sent by sensor nodes and generates results.

Architecture of WiSeNetor

WiSeNetor has two important modules:

1. Network Creation Module: The network creation module spawns a network with user specified parameters like number of clusters, number of routers and depth of a network. This module also creates data structures like routing tables or neighbor tables.
   The network creation module runs a shell script to initialize and start the Spamulator.

2. Simulated Network Devices:
   The simulated network devices have two parts, the server side part which is part of Spamulator and the client side part which sits outside the Spamulator. The reason for splitting this module in two parts is that the Spamulator closes all the file descriptors and doesn't allow the server side to access the routing or neighbor table (Aycock, 2007). When a message comes from one network device to another network device, the Spamulator launches the server part of the simulated network device. The server side, in turn, launches the client part that processes this message. This module is also responsible for logging messages to keep track of message delivery.

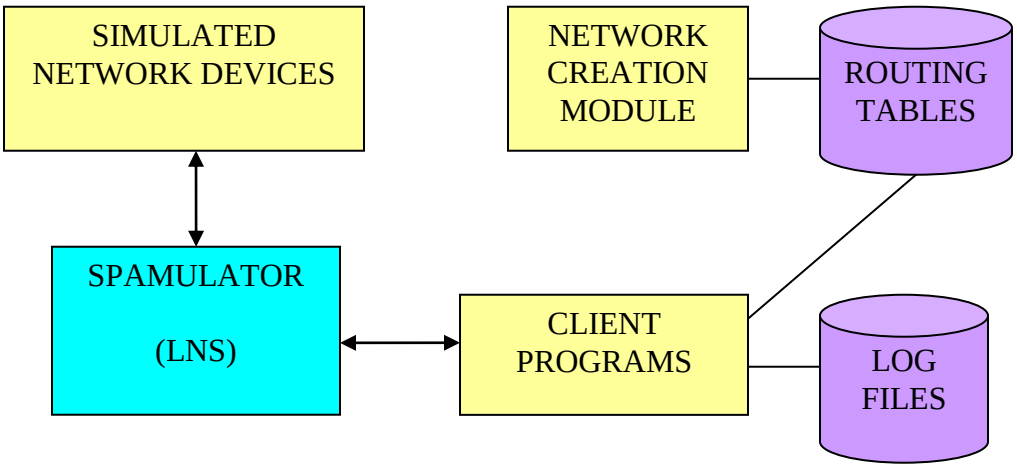


Figure 7: Architecture of WiSeNetor

IMPLEMENTATION

Operating System

Spamulator has been used as the foundation for WiSeNetor. Spamulator runs on Scientific Linux but scripts have been created which can be used to run the Spamulator on Ubuntu (A Linux based operating system) as well. Therefore, WiSeNetor uses Ubuntu (version 8.10) to be compatible with the Spamulator.

Programming language and scripting language

The server programs, client programs and the network creation modules have been implemented in C/C++ programming languages. Shell script has been used to initiate the Spamulator which includes creation of Ip tables, described below.

IP Tables

Iptables allow Linux administrators to shield the operating system from unwanted applications or clients. At the kernel level, the IP packets are inspected before they are forwarded to the destined application. In order to be able to make the decision to forward or drop packets, the kernel software needs to be appropriately configured. In Linux, the iptables configuration file located at `/etc/sysconfig/iptables` can be modified to get the desired behavior. Certain iptables commands can be used to configure the iptables to behave in a certain way. User needs to specify the type of packets (tcp or udp), source and the destination address of the packet and target that specifies what action can be taken on packets that match the specified criteria. Before the Loopback Network Simulator could run, there need to be rules in the OUTPUT chain of iptables' "mangle" table to redirect all packets destined to simulated subnets and all responses from them to the iptables QUEUE target. Also, the "ip_queue" module must be loaded. For more information on iptables, please refer (Boucher, M., Josefsson, M., Kadlecsik, J., McHardy, P., Morris, J., Welte, H., & Russell).

Modules implementing various network topologies

The WiSeNetor implements three kinds of networks, cluster, cluster tree and mesh networks.

1. **Cluster Network**

   Components:

   Components of cluster network include one PAN coordinator and multiple network devices including parent nodes and child nodes.

   Creation of cluster network:

The network creation module spawns a cluster network according to the user specified inputs. For cluster network, there is only one cluster in the network and the PAN coordinator, therefore, is the cluster head and also responsible for maintenance of the network. This module mangles the iptables for specified address range.

The network creation module also creates routing table for every network device in the network. Every routing table contains entries for a node's parent node and its children and grandchildren. If a node is a leaf node, the routing table contains only one entry for its parent node. Every entry consists of the IP address of a child node or grandchild node and the IP address of parent of the given child node or grandchild node.

Following algorithm depicts how the cluster network is created

1. Enter type of protocol to simulate. For status update enter '1' and for routing, enter '2'.

   option ← {1,2}
2. Create copies of server and client programs.
3. For the given depth (num_iter)
   a. Create a routing table for given network device (parent node or leaf node)
   b. First entry in routing table is for the IP address of network device's parent node

      Routing File ←"P" IP address of parent node

   c. If network device is a parent node,

      Routing Table ← IP address of its child or grandchild
      Routing Table ← IP address of parent of child or grandchild
   d. Increment counter for ip address and increment counter for port number.
4. Close all open files and stop.

Every network device in the cluster network has two parts, one part (server part) that handles incoming requests from other network devices and other part (client part) that processes these requests and creates response messages and sends them to other network devices.

When a server starts, all its file descriptors are closed except for the standard output, which is a pipe connecting to the Spamulator. The server must begin its operation by preparing to listen for a TCP connection, and sending the port number it will listen at to the Spamulator on standard output – write this as a two-byte number in network byte order. When a server exits, the Spamulator is notified by closing standard output [1].

The server or a node needs to record messages coming from its neighboring node. The LNS or loopback network simulator does not permit the server to write to a file. One solution to this problem is to use the `system()` function call which invokes a command processor to execute command. First step is to create a file with an appropriate name to avoid race conditions using the “touch filename” command. The string “touch filename” is passed as an argument to `system()` function call and after the command execution, the processor gives the control back to the program. Second step – use “echo” command to append messages to the file created in the first step. In this manner, messages can be recorded by a node.

In this module, the status update protocol and routing protocol have been implemented. The PAN coordinator initiates the status update protocol in order to keep track of newly added nodes or even missing broken links. The PAN coordinator sends a “SU” (status update) message to its immediate children. These child nodes, in turn, forward this message to their direct child nodes. This process continues until all the leaf nodes in the cluster network have been reached. Once the leaf nodes get the “SU” message from their parents, they send a status update response message, “SUR ip_address”, which also contains their ip address, to their parent nodes. The parent nodes forward this message until the PAN coordinator has been reached. In this way, the PAN gets a list of existing nodes in the network.

Following flowchart depicts the working of status update protocol (server part)

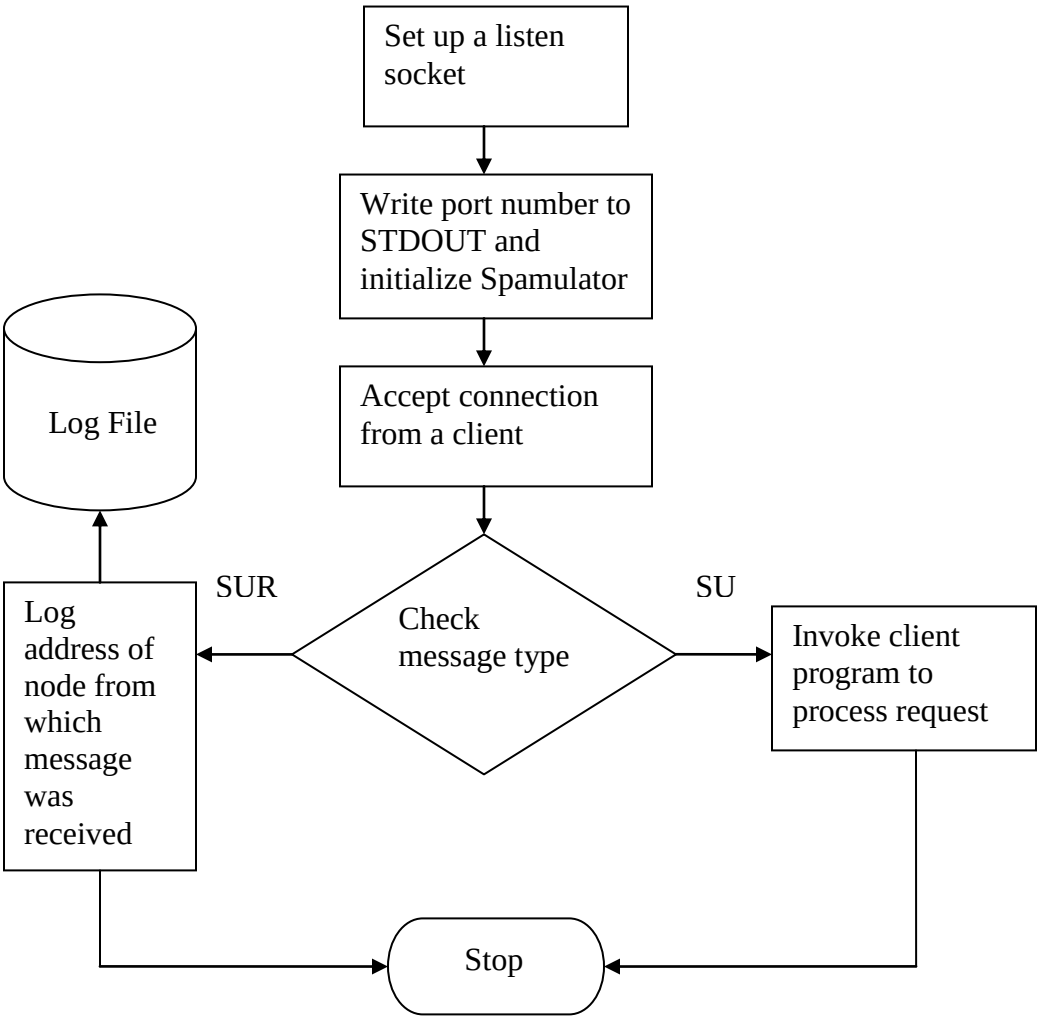


Figure 8: Status Update Protocol at server side

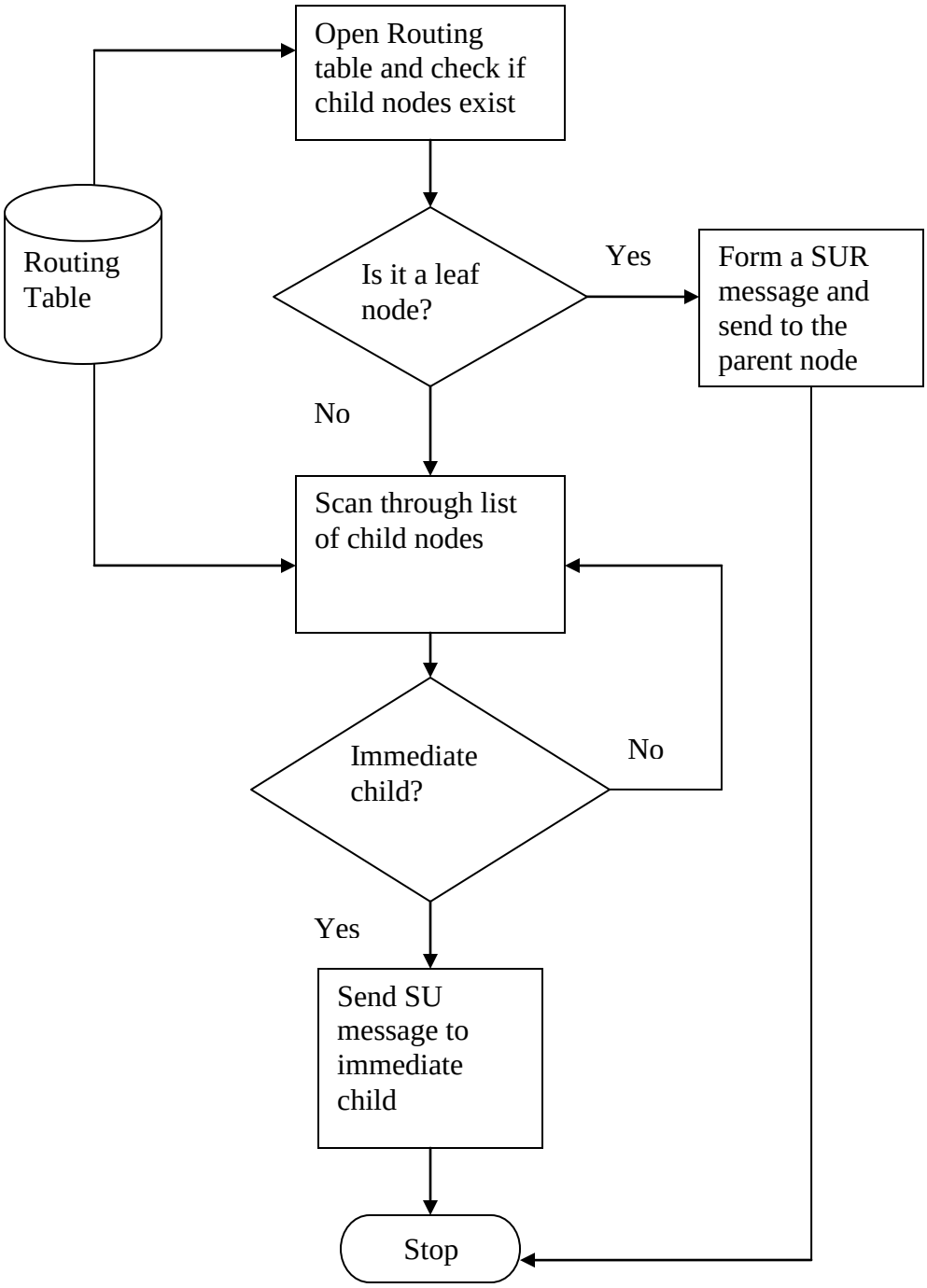


Figure 9: Status Update at client side

Routing in cluster network:

User can specify the source node and the destination node IP address in the route. The Spamulator invokes executable for the user specified source node. The source node checks for the destination IP address in its routing table and if it finds an entry in the routing table, it sends the message to the destination node. If an entry is not found in the routing table, it sends the request to its parent node. This process of searching for the destination node continues until it has been reached. The routing message starts with a “*” which means it is a routing request and it is followed by the ip address of the destination node. An example of a routing message would look like, “* 10.0.0.1 message”.

Following flowchart depicts working of module that implements routing protocol in cluster network

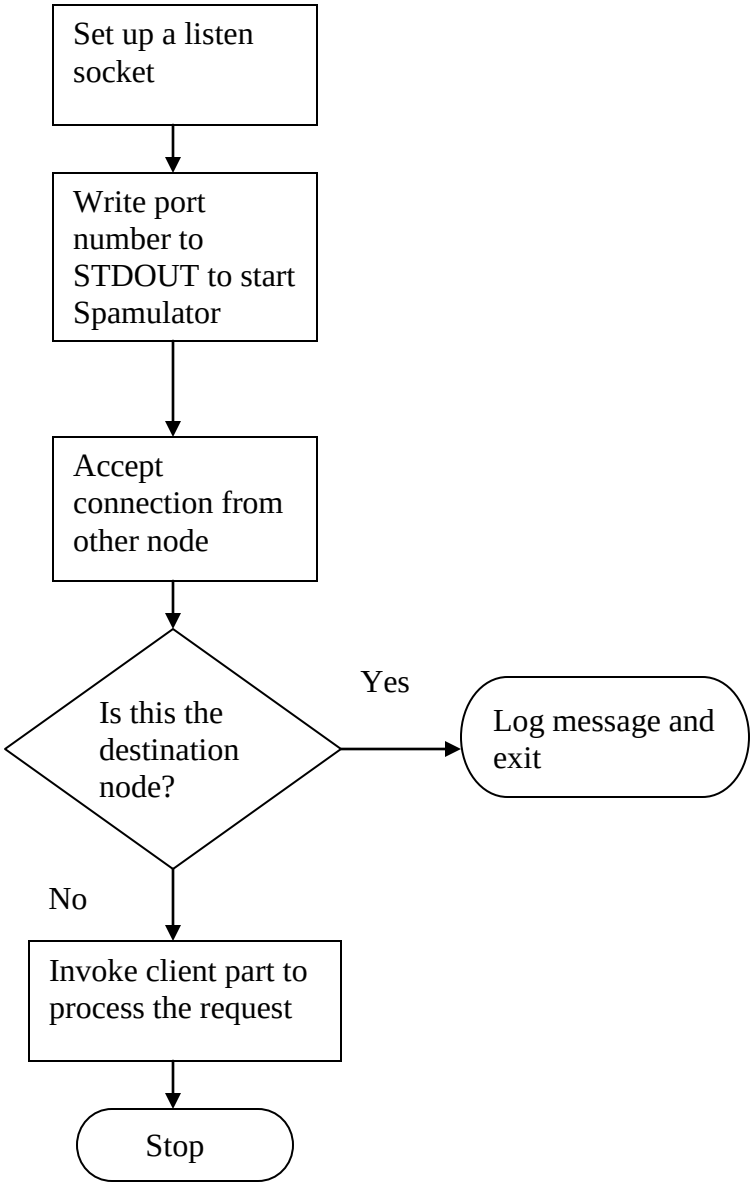


Figure 10: Working of routing protocol at server side

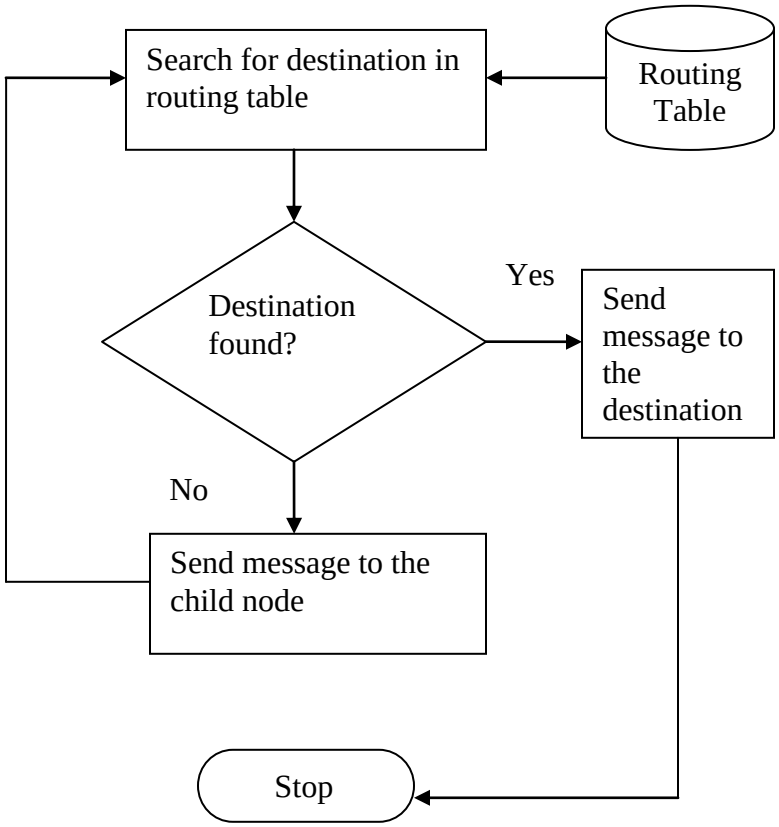


Figure 11: Working of routing protocol at client side

2. Cluster Tree Network

User can specify multiple clusters in order to spawn a cluster tree network. There is one PAN coordinator which acts as a parent to all the cluster heads. The first octet of the ip address denotes the cluster id. The PAN coordinator acts as a hub in order to send a routing message from one cluster to another.
The cluster-tree network reuses all the modules in cluster network topology. Status Update protocol and routing protocol in the cluster-tree network are similar in functionality as that of cluster network protocol.

3. Mesh network

Network creation module

The network creation module spawns a mesh network according to the user specified inputs. It also mangles the iptables for specified address range.

The network creation module also creates neighbor table for every network device in a cluster, in the network. As many clusters as desired can be simulated by simply adding a rule in the iptables. Every neighbor table contains entries for the devices or nodes it is connected to. Every entry consists of the type of device (FFD or router denoted as R and end device or RFD as E) and the ipaddress along with the port number (for example, 1.0.0.1:9154).

Following is the algorithm for spawning a mesh network

1. Enter number of clusters(n) and depth for each cluster(d)

2. n represents the cluster id for any given cluster
    a. 1st octet of every cluster represents cluster id

3. while n, for every cluster and for given depth (d),
    maximum number of devices in cluster is 765
    a. compute IP address and port number for network device
        if last octet of IP address gets count > 255
          start utilizing 3rd octet
        if 3rd octet of IP address gets count > 255
          start utilizing 2nd octet
        if 2nd octet of IP address gets count > 255
          stop adding any more devices to the cluster

    b. create a neighbor file for the network device
        if (network device is a router (R))

            Neighbor file ← IP address of parent node
            Neighbor file ← IP addresses of child nodes

        else if(network device is an end device (E))

            Neighbor file ← IP address of parent node

c. create an executable copy of client program and a server program for the network device
d. connect neighboring nodes
   Neighbor file ←IP address of left neighbor
   Neighbor file ← IP address of right neighbor
a. increment counter for port number and ip address

. 4. Close all the open files and exit

Every network device in the mesh network has two parts, one part (server part) that handles incoming requests from other network devices and other part (client part) that processes these requests and creates response messages and sends them to other network devices.

Routing in mesh topology is on the lines of Zigbee's mesh topology. A message can be routed from a source node (FFD or RFD) to any destination node (FFD or RFD).

If a device finds that there is no entry for the specified destination device, then it initiates a route discovery process. It sends a route request or RREQ message to all the routers or FFDs present in its neighbor file. The RREQ message frame consists of the message identifier, source node IP address, destination node IP address, sender's IP address. The frame looks like below:

RREQ	Msg ID	Src IP	Dest IP	Sender IP

Figure 12: RREQ message frame in WiSeNetor

The route discovery table consists of message Identifier, source node IP address and destination node IP address. The route discovery process continues through one or several hops until the destination is reached. The destination device sends an RREP or a route response message to the sender node recorded in the route discovery process. A RREP message consists of fields like destination node IP address and source node IP address.

RREP	Destination IP	Source IP

Figure 13: RREP message frame in WiSeNetor

Every intermediate node creates a routing table for the specified destination node. Every entry in the routing table consists of destination IP address and the next hop node IP address. This process of forwarding the RREP message continues until the source node has been reached.

If a device finds an entry in the routing table, it sends a RTE message to the next hop address in the routing table and does not initiate a route discovery process. The RTE message frame looks like below:

RTE	Message

Figure 14: RTE message frame in WiSeNetor

Following flowchart depicts working of the server program

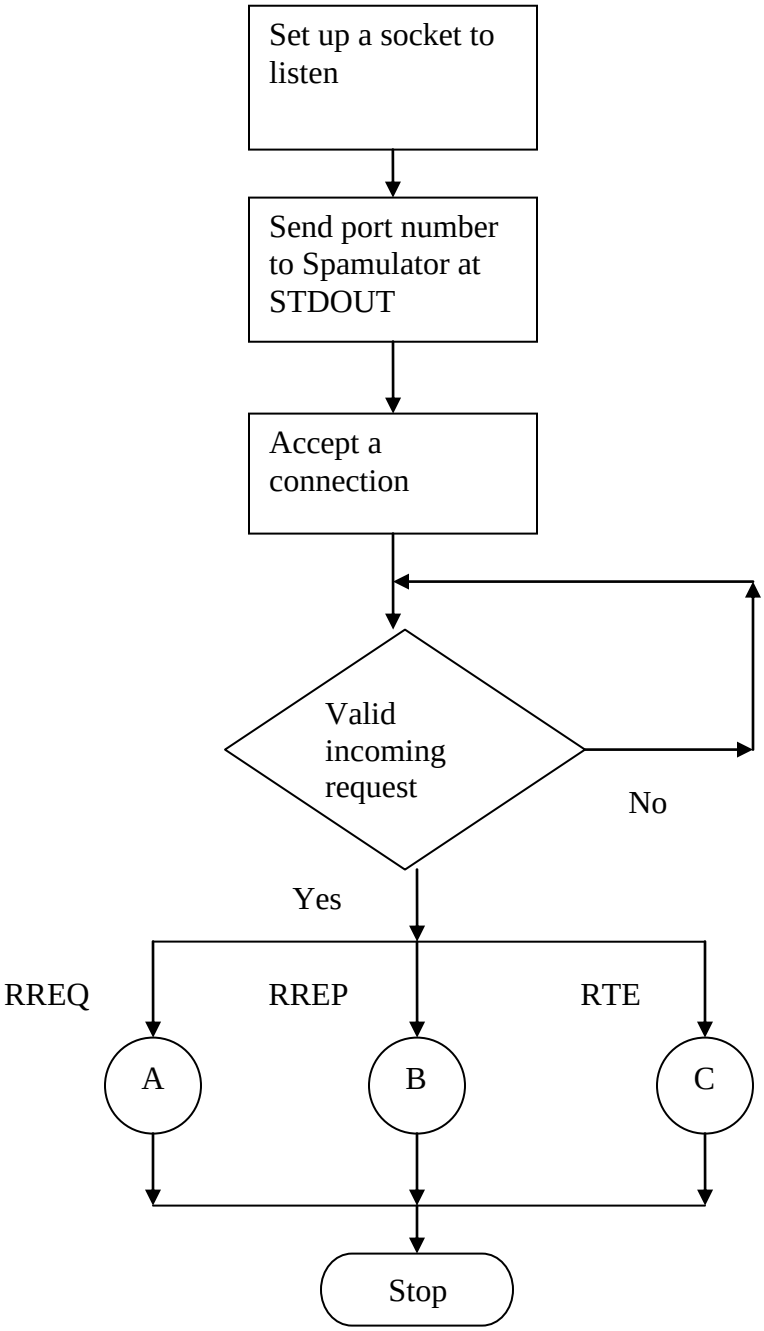


Figure 15: Working of Mesh Server Program

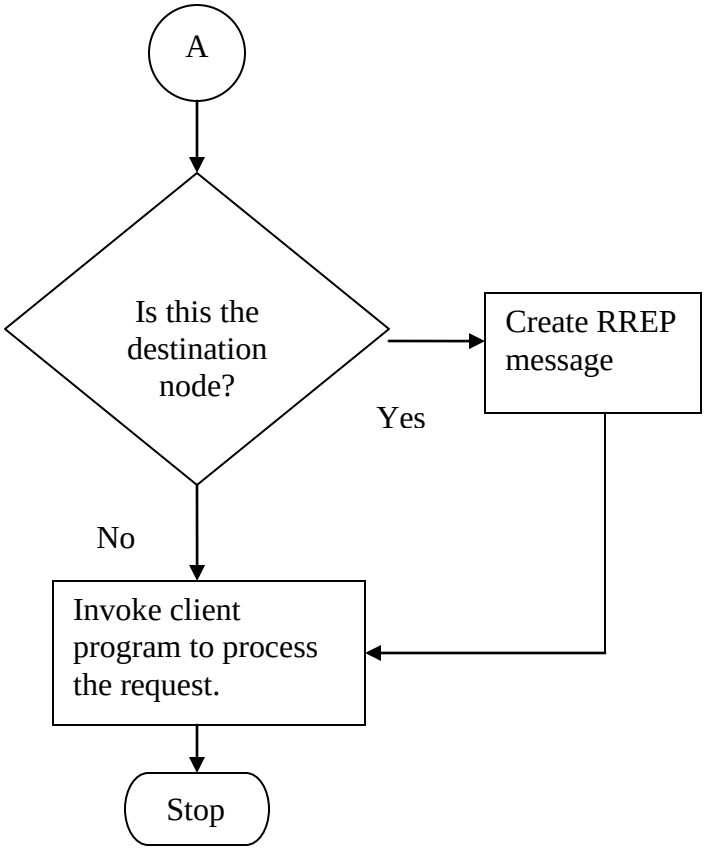


Figure 16: Sub process that handles RREQ request

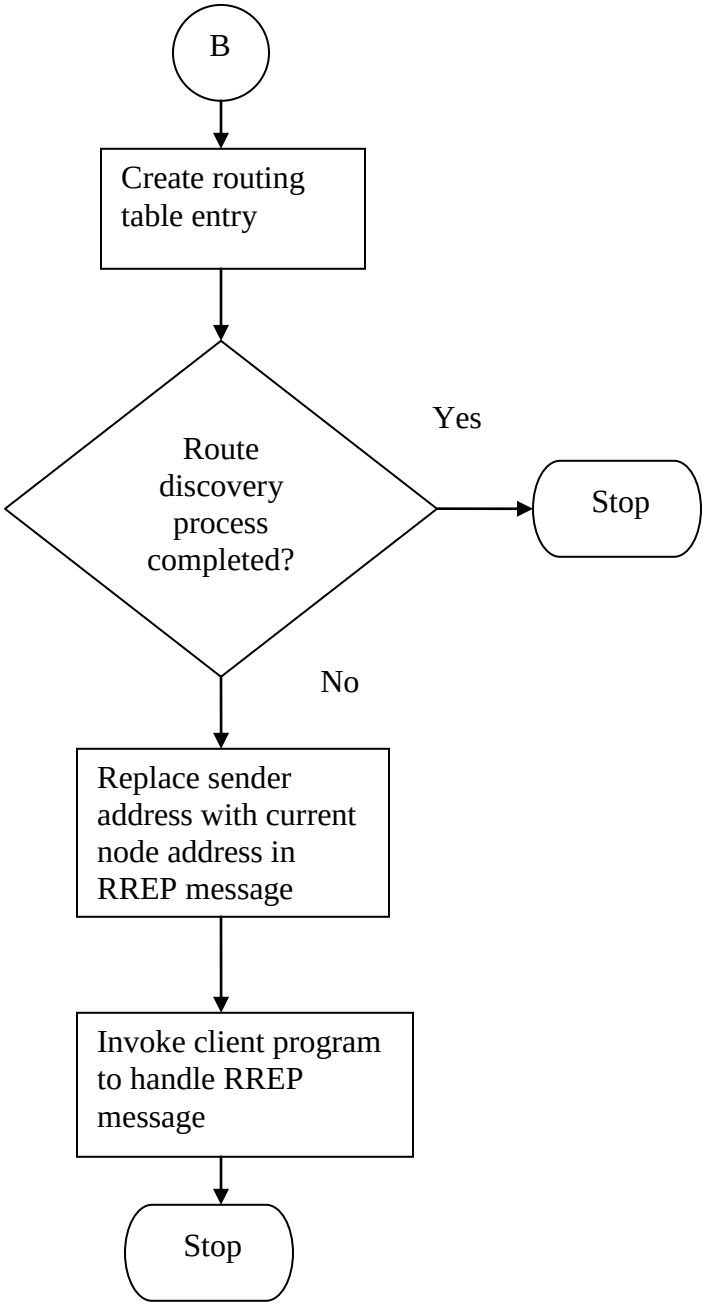


Figure 17: Sub process that handles RREP request

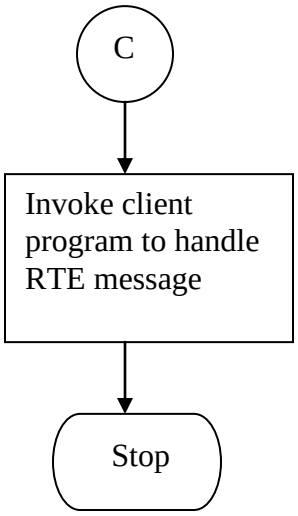


Figure 18: Sub process that handles RTE message

Following flowchart depicts how the mesh client handles various requests like RREQ, RREP and RTE.

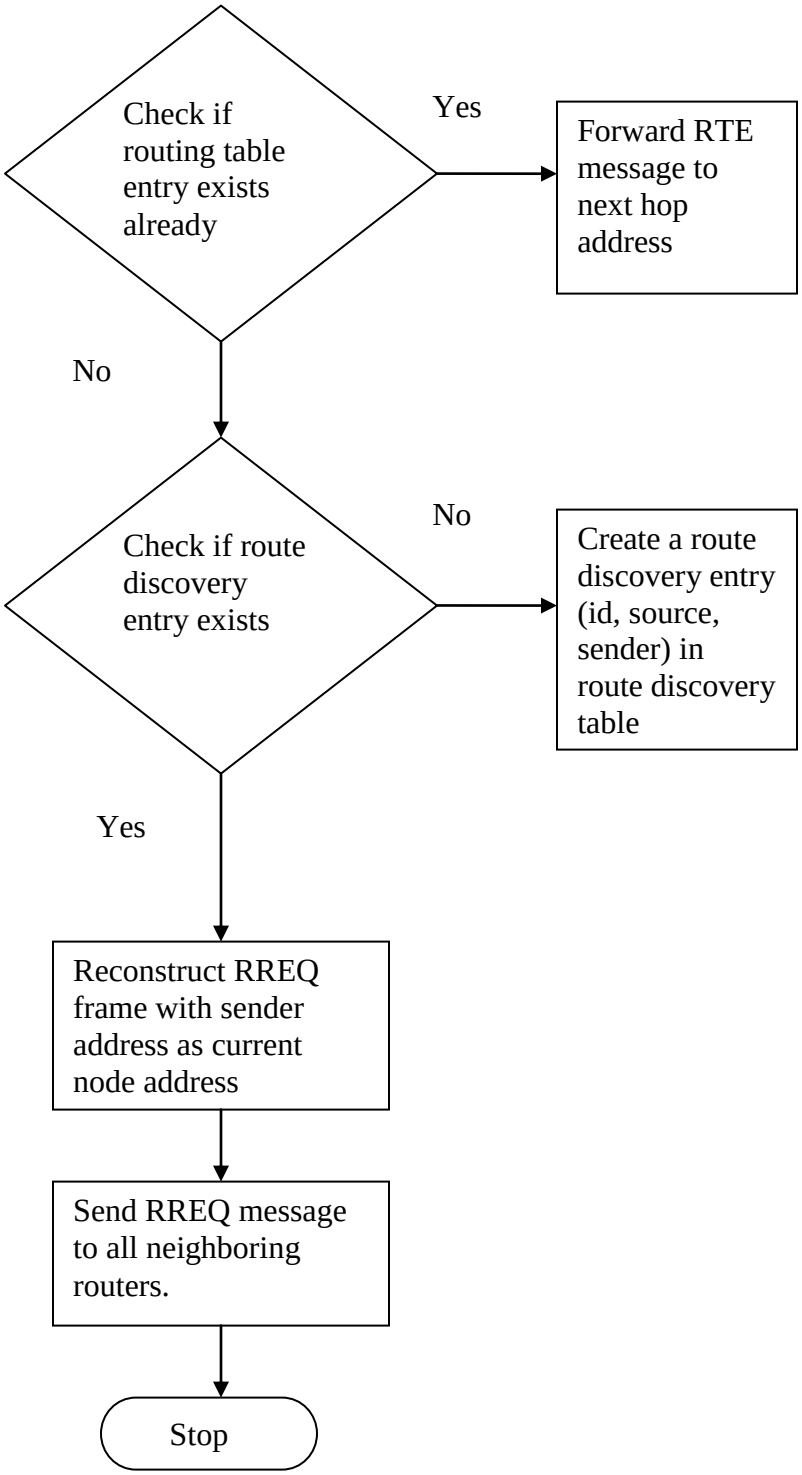


Figure 19: Handling of RREQ message by mesh client program

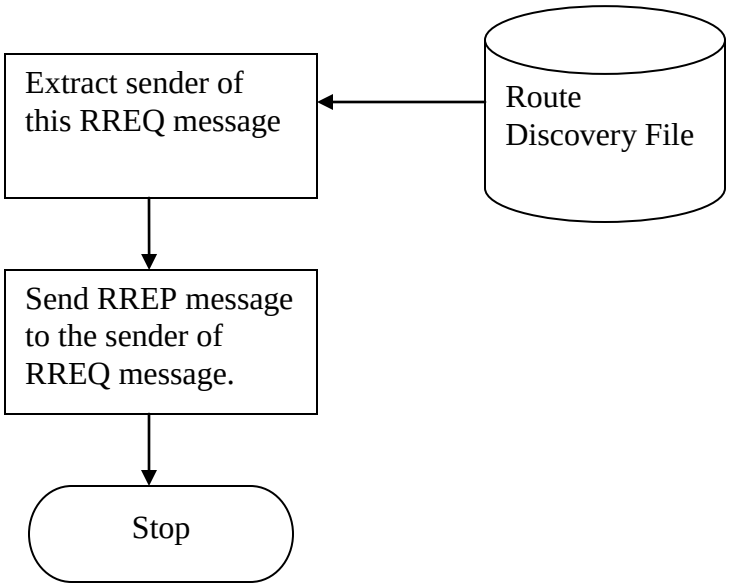
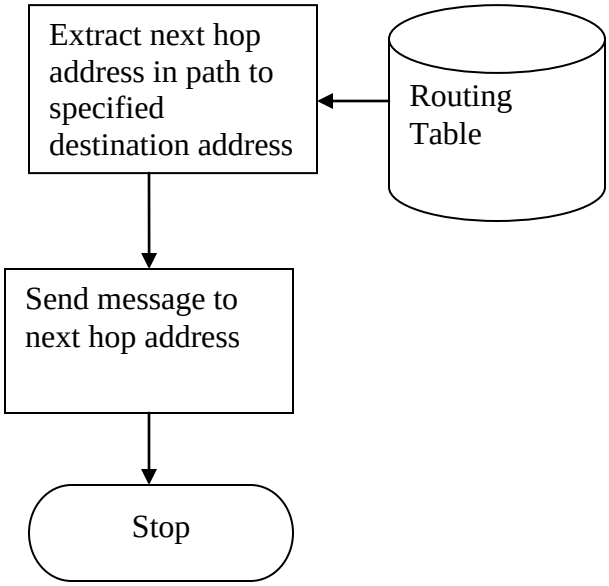


Figure 20: Handling of RREP message by mesh client program



Simulation Results

Mesh network was tested for scalability and correctness of routing for a given message. There is one PAN coordinator per network and its fixed IP address is 60.0.0.0. There can be multiple clusters and every cluster has a cluster head. These cluster heads are direct children of the PAN coordinator. There are maximum routers per node and end devices per node. Mesh network can support up to 15,000 nodes in a network.

Following graph shows time elapsed to spawn a mesh network versus number of nodes in the mesh network. Time elapsed in spawning a network is directly proportional to number of nodes in mesh network. Time required in spawning a mesh network increases as the number of nodes increases.

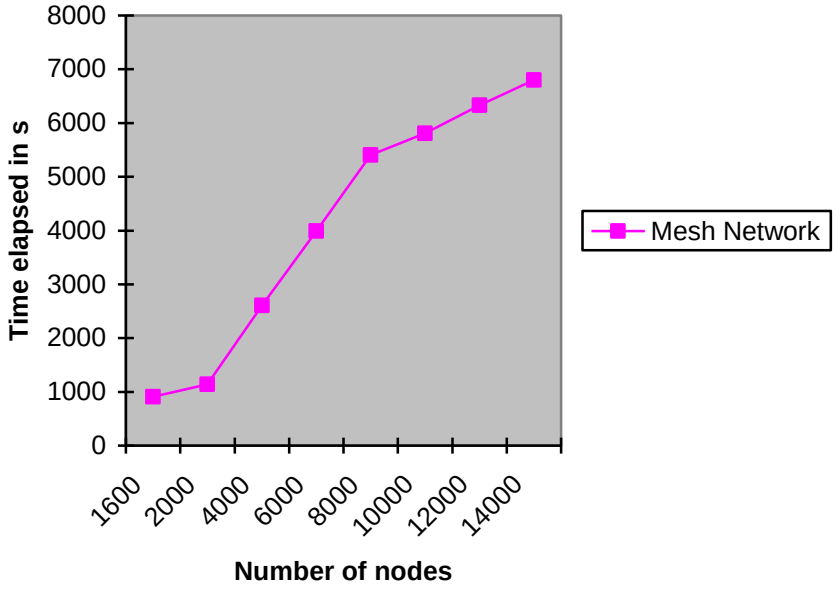


Figure 22: Time elapsed in spawning mesh network Vs number of nodes

Following are sample test results for input parameters entered by the user.

```
Number of clusters: 3
Cluster Head for cluster#1: 1.0.0.0
Cluster Head for cluster#2: 2.0.0.0
Cluster Head for cluster#3: 3.0.0.0
Depth of cluster#1: 20
Depth of cluster#2: 20
Depth of cluster#3: 20

Total number of nodes in network: 2296
```

At this point, the network is spawned and neighbor table for every node is created. Following table depicts neighbor table for cluster head 1.0.0.130. First column depicts device type of neighboring device and second column depicts IP address of neighboring device and its port number.

R	1.0.0.127:9279
E	1.0.0.131:9283
E	1.0.0.132:9284
R	1.0.0.133:9285
R	1.0.0.187:9339
R	1.0.0.73:9225

Send message from Node: 1.0.0.130 (Router)
Send message to Node: 3.0.0.102 (End device)

Log files are created for every node which helps the user in tracking the progress of routing in the network. Following is a sample log file for source node and destination file.

RREQ 1 3.0.0.102 1.0.0.130 1.0.0.130:9282
RREQ
Contents of log file created by source device 1.0.0.130

```
Invoking send rreq
I am in send rreq
Message: RREQ 13.0.0.102 1.0.0.130 1.0.0.130:9282
Invoking send rrep
I am in RREP
RREP 3.0.0.102 1.0.0.130 1.0.0.127:9279

1.0.0.130 9282 RREQ 1 3.0.0.102 1.0.0.130
1.0.0.130:9282
```

```
RTE filename: /home/gauri/meshfiles/1.0.0.130rte.txt
No routing table for this node
Sending message to 1.0.0.127:9279 sender1.0.0.130:9282
NEW cmd frame in RREQ RREQ 1 3.0.0.102 1.0.0.130
1.0.0.130:9282
Sending message to 1.0.0.131:9283 sender1.0.0.130:9282
NEW cmd frame in RREQ RREQ 1 3.0.0.102 1.0.0.130
1.0.0.130:9282
Sending message to 1.0.0.132:9284 sender1.0.0.130:9282
NEW cmd frame in RREQ RREQ 1 3.0.0.102 1.0.0.130
1.0.0.130:9282
Sending message to 1.0.0.133:9285 sender1.0.0.130:9282
NEW cmd frame in RREQ RREQ 1 3.0.0.102 1.0.0.130
1.0.0.130:9282
Sending message to 1.0.0.187:9339 sender1.0.0.130:9282
NEW cmd frame in RREQ RREQ 1 3.0.0.102 1.0.0.130
1.0.0.130:9282
Sending message to 1.0.0.73:9225 sender1.0.0.130:9282
NEW cmd frame in RREQ RREQ 1 3.0.0.102 1.0.0.130
1.0.0.130:9282
```

Also, route discovery table (if a routing table entry for destination does not exist) is created by every device that falls on the route to the destination.
Following is a route discovery table entry (message id, source, sender) created by the source device, 1.0.0.130.

```
1 1.0.0.130 1.0.0.130:9282
```

Following is the route discovery table entry created by the destination device.

```
1 1.0.0.130 3.0.0.100:9822
```

After the route discovery process is completed, routing table entry (destination, next hop) for the destination device is created by every device in the route.
Following is the routing table entry created by the source device 1.0.0.130.

```
3.0.0.102 1.0.0.127:9279
```

Contents of log file created by destination device, 3.0.0.102

```
RREQ 1 3.0.0.102 1.0.0.130 3.0.0.100:9822
Invoking send req
I am in send rreq
Message: RREQ13.0.0.1021.0.0.1303.0.0.100:9822
Destination reached 3.0.0.102
```

Validation of WiSeNetor

Bangkok, Thailand faces the problem of traffic jams. An inter vehicle communication system that provides real-time road information was proposed to mitigate this problem of traffic jams. The proposed system uses several base stations along the roads and treats cars as nodes in a wireless system. Performance of several wireless systems including Zigbee for inter vehicle communication has been studied in (Eamsomboon,P., Keeratiwintakorn,P., & Mitrpant, C , 2008). The experiment uses real map of busiest section of the city as the topology, which is mesh topology, and AODV as the routing protocol.

In this section, we compare the performance of WiSeNetor and Zigbee used in vehicles in Bangkok metropolitan area as we increase the number of nodes. Performance is in terms of end-to-end delay of a message and average number of hops.

end-to-end delay = received packet timestamp –
sent packet timestamp

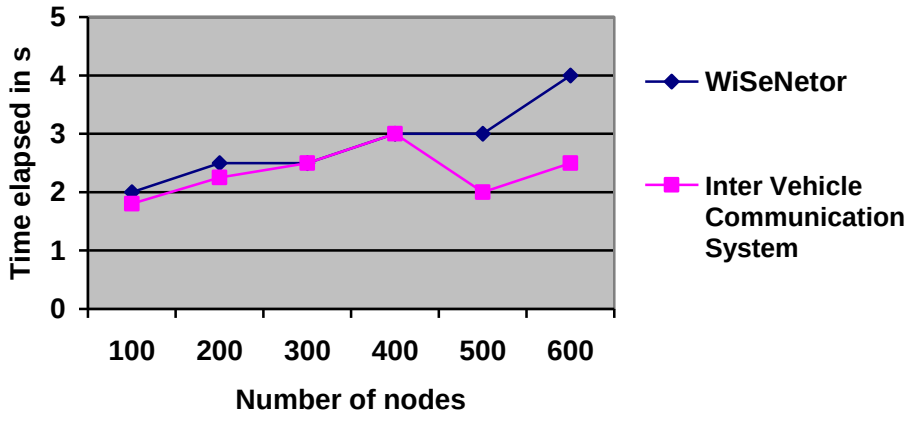


Figure 23: End-to-End delay for increasing number of nodes

Figure 23 depicts the end-to-end delay for a message in WiSeNetor and in the inter vehicle communication system. It can be seen that the end-to-end delay increases as the number of nodes in the network increases. The increase in the end-to-end delay is not drastic though. Each test case was run one hundred times and time elapsed was recorded. Figure 23 depicts average time elapsed in seconds for given test case.

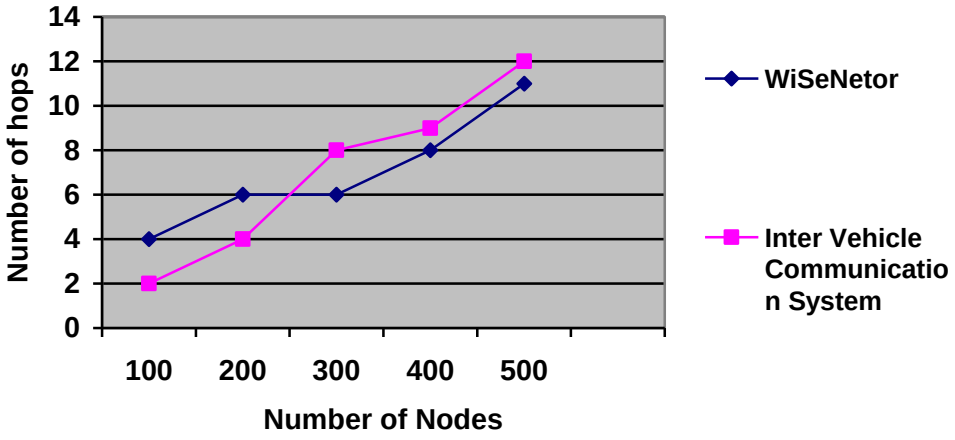


Figure 24: Number of hops as number of nodes increases

Figure 24 depicts the number of hops as number of nodes increases in the network. Each test case was run 100 times and average number of hops was recorded. Figure 24 depicts the average number of hops for given test case. The figure shows that the number of hops increases drastically as the number of nodes increases. This is also true in real-world Zigbee systems because of the short range communication in the network.

This experiment validates the WiSeNetor and the results obtained are in accordance with the results obtained in the inter vehicle communication system.

Conclusion

WiSeNetor will prove to be a good teaching tool or a research tool. WiSeNetor simulates a scalable wireless sensor network with cluster tree and mesh topologies successfully. The simulator was able to support up to 15,000 nodes in the wireless sensor network. Routing of messages in cluster tree topology was done successfully in accordance with IEEE 802.15.4 specification and in mesh network the Zigbee routing protocol was used. It was found that end to end delay for any message through a cluster tree network was more than that of the mesh network. Therefore, it can be concluded that the cluster tree topology may be used as a backup mechanism in case the mesh network fails due to some reason. Simulation results of WiSeNetor matched the simulation results specified in (Eamsomboon,P, 2008) which validates the WiSeNetor.

Wireless sensor network is an upcoming technology and many companies have successfully deployed these networks in various domains. Meshnetics with Zigbee have deployed a wireless sensor network to reduce the energy consumption by 37% in the building automation domain (Zigbee Alliance, Success Stories). EpiSensor with Zigbee have deployed a Zigbee network in the Antarctic to facilitate climate change research.

Future Work

In this project, a scalable wireless sensor network with cluster tree topology and mesh topology were successfully implemented. The mesh network routing protocol was implemented according to the Zigbee specification. Zigbee specification also describes about the security architecture for wireless mesh networks. Therefore, WiSeNetor may be used as a foundation for building the security infrastructure. Various attacks on the security of a wireless mesh network may be simulated and counter measures to avoid those attacks may also be simulated.

References

Eamsomboon,P., Keeratiwintakorn,P., & Mitrpant, C (October 24, 2008) . The Performance of Wi-Fi and Zigbee Networks for Inter-Vehicle Communication in Bangkok Metropolian area*.* *ITS Telecommunications, 2008.* doi: 10.1109/ITST.2008.4740296

Wikimedia Foundation, Inc. (October 17, 2004). Wireless Sensor Networks. http://en.wikipedia.org/wiki/Wireless_Sensor_Networks

Hill, J.L. (2004), System Architecture for wireless sensor network. In *Proceedings of the 2004 Americas Conference on Information Systems*. http://www.eecs.harvard.edu/~mdw/course/cs263/papers/jhill-thesis.pdf

Picture for Environmental data collection system (2007). Retrieved from http://blog.xbow.com/photos/uncategorized/2007/10/18/nodedeployment_2.jpg

Aycock, J., Crawford, H., & deGraaf, R. (2008), Spamulator: The Internet on a Laptop*. Annual Joint Conference Integrating Technology into Computer Science Education,* 142-147. doi: http://doi.acm.org/10.1145/1384271.1384311

Gutierrez, J.A., Callaway, E., & Barrett, R. (2003), *IEEE 802.15.4 Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensor Networks.* Institute of Electrical & Electronics Engineering

Zigbee Alliance (January 17, 2008) *Zigbee Specification*

Lisee, M., Chen, G., Szymanski, B., & Rensselaer Polytechnic Institute. (2006) SENSE: Sensor Network Simulator and Emulator http://www.ita.cs.rpi.edu/sense/index.html

Source Forge, *Network Simulator* http://sourceforge.net/projects/nsnam/

Aycock, J., University of Calgary. (October 9, 2007),

Writing Spamulator Extensions
Unpublished documentation

Boucher, M., Josefsson, M., Kadlecsik, J., McHardy, P., Morris, J., Welte, H., & Russell.
Iptables(8) Linux man page
http://linux.die.net/man/8/iptables

Zigbee Alliance, Success Stories
http://www.zigbee.org/Markets/SuccessStories/tabid/228/Default.aspx

http://www.sensorsmag.com/articles/0203/38/main.shtml

http://www.sensorsmag.com/sensors/Feature+Articles/What-a-Mesh-Part-2mdashNetworking-Architectures-an/ArticleStandard/Article/detail/575800?contextCategoryId=34388

Appendix A

Appendix A describes the main code snippets from the WiSeNetor software. This appendix includes code snippet from the server and client programs for both, the mesh network and the cluster tree network.

1. Code snippet for server in a mesh network: This code snippet depicts the overall working of the server program. It sets up a connection to communicate with the client, parses the command frame from the incoming request and then invokes appropriate function to handle the incoming request. Incoming requests can be RREP, RREQ and RTE.

```
// Set up socket to listen at

  sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
  int on = 1;
  int ret = setsockopt( sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on) );

  listen(sock, 5);

  // Send port number to Spamulator

  socklen_t len = sizeof(sin);
  getsockname(sock, (sockaddr*)&sin, &len);
  port = sin.sin_port;
  write(STDOUT_FILENO, &port, sizeof(port));


  // Finally, accept the connection
  clilen = sizeof(cli_addr);
  conn = accept(sock, NULL,NULL);
  if(conn < 0)
  {
        exit(0);
  }

  // create a log file to dump messages from the client
  string addr = argv[1];
  string filename= "/home/gauri/logmessages/"+addr+"out.txt";
  string cmd = "touch "+filename;
  system(cmd.c_str());

  time(&seconds);
  srand((unsigned int) seconds);


  // Now talking to client....
  while(1)
  {

        // write data recieved from client
         fflush(stdout);
         bzero(recv_data,1024);
         read(conn,recv_data,1023);

        if(recv_data[0]=='R' && recv_data[1]=='R' && recv_data[2]=='E')
        {
                rdata = recv_data;
                cmd = "echo " + rdata + ">>" +filename;
                system(cmd.c_str());



                size_t loc;
```

```
                loc=rdata.find_first_of(" ");
                string msg = rdata.substr(0,loc) ;

                cmd = "echo " + msg + ">>" +filename;
                system(cmd.c_str());

                if(msg.compare("RREQ") == 0 )
                {
                        cmd = "echo Invoking send req >>" +filename;
                        system(cmd.c_str());
                        send_rreq(recv_data,argv[1],argv[2]);
                        sent = 1;
                }
                if(msg.compare("RREP") == 0)
                {
                        cmd = "echo Invoking send rrep >>" +filename;
                        system(cmd.c_str());
                        send_rrep(recv_data,argv[1],argv[2]);
                        sent = 1;

                }
        }
        if(recv_data[0]=='R' && recv_data[1]=='T' && recv_data[2]=='E')
        {
                rdata = recv_data;
                /* extract destination address */
                size_t loc;
                loc=rdata.find_last_of(" ");
                string cframe = rdata.substr(0,loc) ;
                string dest = rdata.substr(loc+1) ;

                if(dest.compare(addr)==0)
                {
                        cmd = "echo Destination Reached >>" +filename;
                        system(cmd.c_str());
                        exit(1);
                }

                cmd = "echo Invoking send RTE >>" +filename;
                system(cmd.c_str());
                send_rte(recv_data,argv[1],argv[2]);
                sent = 1;

        }

        if(sent == 1  )
                break;

    }//end of outer while

    close(conn);
    shutdown(conn, 2);
```

2. Code snippet for client program

   The server program invokes the client program to handle every incoming request. The client program runs as a separate process. If an entry is not found in the routing table for given destination, a route discovery entry is made and the message is forwarded to the neighboring nodes. Following is the code snippet from the main function which depicts the overall functioning of the client.

```
        /* check command type : RREQ or RREP */
        int ret = chech_cmd_type(argv3);
        if(ret == 1)// RREQ
```

```
{
        sent = 0;
        string destination = find_destination(argv3);

        int retval = find_routingtable_entry(destination,argv[1],argv1);

        if(retval == 1)
        {
                return 0;
        }

        /* create route discovery entry if not present */
        int is_present = create_discovery_entry(argv1,argv3);
        if(is_present == 1)//if already present do not
                                        broadcast message again
        {
                string log = "echo create discovery returned 1
                        >>" + logfile;
                system(log.c_str());
                return 0;
        }

        /* if not found, send RREQ(id,dest,src) to neighbors */


        /* open file to check for neighbors */
        ifstream myfile (rbuffer);
        if (myfile.is_open())
        {
                while (getline (myfile,line) != NULL )
                {
                        . . . .

                        if ( (devtype.compare("R") == 0) ||
                        (destination.compare(addrpart) == 0))
                        {

                                . . . .

                        // log message and send message to child node

                        cmd = "echo Sending message to "+addr+">>"+logfile;
                        system(cmd.c_str());

                        len = addrpart.length();
                        memset( srvr, '\0', BUFFER_MAX );
                        addrpart.copy( srvr, len );


                        //reconstruct command frame with new
                        sender address
                        string newframe = cmdfr + " " + argv1 +
                                                ":" + argv2;

                        cmd = "echo NEW cmd frame in RREQ
                                "+newframe+">>"+logfile;
                        system(cmd.c_str());
                        len = newframe.length();
                        memset( cmdframe, '\0', 100);
                        newframe.copy( cmdframe, len );

                        /* call sendmessage routine to broadcast the route
discovery message to neighbors */

                        send_message(srvr,port,cmdframe);

                        }
                }
        }//end of while

      myfile.close();
```

```
   }//end of outermost if

}//end of if ret == 1 i.e. RREQ

else if(ret == 2)//RREP
{
        string cmd = "echo Sending RREP message >>"+logfile;
                system(cmd.c_str());

        send_rrep(argv1,argv3,argv[3],logfile);

}
else if(ret == 3)//RTE
{
        string cmd = "echo Sending RTE message >>"+logfile;
                system(cmd.c_str());

        string destination = find_destination(argv3);


        /* if not found, send RREQ(id,dest,src) to neighbors */
int present =
        find_routingtable_entry(destination,argv[1],argv1);
        if(present == 1)
        {
                cmd = "echo RTE message realyed >>"+logfile;
                system(cmd.c_str());
        }

}
```

3. The following code snippet is used to spawn a mesh network. This program creates nodes in the network by assigning an IP address, a port number and a device type to them. This program also creates a neighbor table for every node , which is a list of other nodes that a node is connected to.

```
pan_file.open(pan,ios::app);
 if(pan_file.is_open())
 {
        for (k=0;(k<MAX_ROUTERS) && (count<= 765);k++) // number of immi. children
                                                          to PAN (here 10)
        {
                //neighbor[k][0] = cnum + ".0.0.0:" + pport;

              for (i=1;(i<depth[d]) && (count <= 765);i++)//number of levels in
                                                          the cluster

                {

                        if((count == 256) || (count == 511))
                        {
                                //restart counter
                                cnt = 0;
                        }

                        if(count == 765)
                        {
                                cout<<"*******Network cannot add any more
                                        nodes"<<endl;
                        }

                        sprintf(ct,"%d",cnt) ;
                        sct = ct;
```

```
port_num++;
sprintf(pport,"%d",port_num);


if(count <= 255)
{
        parents[i] = cnum + ".0.0." + sct + ":" + pport;
        neighbor[k][i] = cnum + ".0.0." + sct + ":" + pport;
        client = cnum + ".0.0." + sct;

}

if((count > 255) && (count <=510))
{
        parents[i] = cnum + ".0." + sct + ".255:" + pport;
        neighbor[k][i] = cnum + ".0." + sct + ".255:" +
                                pport;
        client = cnum + ".0." + sct + ".255";
}

if((count > 510) && (count <= 765))
{
        parents[i] = cnum + "." + sct + ".255.255:" + pport;
        neighbor[k][i] = cnum + "." + sct + ".255.255:" +
                                pport;
        client = cnum + "." + sct + ".255.255";
}

//convert client from string type to char array
string::size_type len = client.length();
memset( clientarray,'\0',20 );
client.copy( clientarray,len );

bzero(pfilename,50);
strcpy(pfilename,"meshrouting/");
strcat(pfilename,clientarray);
strcat(pfilename,".txt");


compileprograms(parents[i],client);
count++;
cnt++;

myfile[i].open (pfilename,ios::app);
if(myfile[i].is_open())
{
        myfile[i]<<"R "<<parents[i-1]<<endl;
        myfile[i-1]<<"R "<<parents[i]<<endl;
        if(i == 1)
                pan_file<<"R " <<parents[1]<<endl;

        for(j=0;(j<2) && (count <= 765);j++)//number of
                                children each parent has
        {

                if((count == 256) || (count == 511))
                {
                        //restart counter
                        cnt = 0;
                }

                if(count == 765)
                {
                        cout<<"*******Network cannot add any
                                more nodes"<<endl;
                        //return 0;
                }

                sprintf(ct,"%d",cnt) ;
                sct = ct;
```

```
                                        port_num++;
                                        sprintf(pport,"%d",port_num);


                                        if(count <= 255)
                                        {
                                                child_ip  =  cnum + ".0.0." + sct +
                                                        ":" + pport;
                                                client = cnum + ".0.0." + sct;
                                        }

                                        if((count > 255) && (count <=510))
                                        {
                                                child_ip = cnum + ".0." + sct +
                                                ".255:" + pport;
                                                client = cnum + ".0." + sct + ".255";
                                        }

                                        if((count > 510) && (count <= 765))
                                        {
                                                child_ip = cnum + "." + sct +
                                                                ".255.255:" + pport;
                                                client = cnum + "." + sct +
                                                                ".255.255";
                                        }

                                        //cskip = calculate_cskip(i+1);
                                        //convert client from string type to char
                                        array
                                        string::size_type len = client.length();
                                        memset( clientarray,'\0',20 );
                                        client.copy( clientarray,len );

                                        bzero(filename,50);
                                        strcpy(filename,"meshrouting/");
                                        strcat(filename,clientarray);
                                        strcat(filename,".txt");

                                        ofstream childfile(filename,ios::app);
                                        if (childfile.is_open())
                                        {
                                                childfile << "R "<<parents[i]<<endl;

                                                compileprograms(child_ip,client);

                                                myfile[i] << "E "<<child_ip<<endl;

                                        }

                                        childfile.close();
                                        count++;
                                        cnt++;
                                }

                        }
                        else
                        {
                                cout<< "can't open file"<<endl;
                                exit(1);
                        }

                n=0;
                }//end of for(i=0 ...
                for(i=0;i<depth[d];i++)
                                myfile[i].close();

          }//end of for(k=0 ...

        /* Connect neighbors */
        connect_neighbors(neighbor,depth[d]);
        }//end of pan file open
```

```
pan_file.close();

}//end of for (d=0 ...

connect_clusters(clusters,number);
```

4. The following code snippet is from the server program that handles incoming routing request in a cluster tree network. A valid routing request starts with a delimiter '*'. Upon receipt of a valid incoming routing request, a client program is invoked to handle it.

```
// Set up socket to listen at

  sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
  int on = 1;
  int ret = setsockopt( sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on) );

  listen(sock, 5);

  // Send port number to Spamulator

  socklen_t len = sizeof(sin);
  getsockname(sock, (sockaddr*)&sin, &len);
  port = sin.sin_port;
  write(STDOUT_FILENO, &port, sizeof(port));


  // Finally, accept the connection
  conn = accept(sock, NULL, NULL);

  . . . .

  write(conn,recv_data,sizeof(recv_data));

  time(&seconds);
  srand((unsigned int) seconds);


  // Now talking to client....
  while(1)
  {
        sent = 0;
        // write data recieved from client
         read(conn,recv_data,40);
        write(conn,recv_data,sizeof(recv_data));
        //check the  delim - "*"
        if(recv_data[0] == '*')
        {
                char * ptr;

                . . . .

                //check if destination has been reached
                if(strcmp(argvs[1],argv[1]) == 0)
                {
                        cmd = "echo Destination Reached:  "+ addr + ">>" +
                                                filename;
                        system(cmd.c_str());
                        sent = 1;
                        break;
                }


                char *execargs[] = { (char*)filename2,
(char*)argv[1],(char*)argv[2], (char*)argvs[1], (char*)argvs[2],NULL};
```

```
pid_t  pid;
int    status;

if ((pid = fork()) < 0)
 {     /* fork a child process           */
         exit(1);
}
else if (pid == 0)
{           /* for the child process:         */
        if (execvp((const char *)filename2,execargs) < 0)
         {     /* execute the command  */
                exit(1);
         }
}
else
{                                   /* for the parent:      */
        while (wait(&status) != pid)       /* wait for completion
                                              */
 ;

}
//int ret = execvp((const char *)filename2,execargs);
```

5. This code snippet is from the client program invoked by the server handling routing requests in a cluster tree network. If the node is present in the routing table, the message is sent down the tree, otherwise the message is sent to the parent node. This continues until the destination node is reached.

```
// myfile points to routing table for this node
if (myfile.is_open())
{

        while (! myfile.eof() )
        {

                getline (myfile,line);

                // extract IP address and port number of child and parent
                nodes
                . . . .


                //store address of parent node ... will need in case dest
                is not present
                if(addr_child.compare("P")== 0)
                {
                        parentnodeaddr = addrpart;
                        parentnodeport = port;
                }

                string totaladdr = argv3 + ":" + argv4;
                if( (addr_child.compare("P")!=0) &&
                (addr_child.compare("N")!=0))
                {
                        //check if destination is present in the routing
                        table .... if present, send message
                        if(totaladdr.compare(addr_child) == 0)
                        {
                                cmd1 = "echo Destination found in routing
                                        table >>"+logfile;
                                system(cmd1.c_str());

                                // log message and send message to child
                                        node

                                string cmd = "echo Sending message to
                                        "+addr_child+">>"+logfile;
```

```
                                system(cmd.c_str());

                                len = addrpart.length();
                                memset( srvr, '\0', BUFFER_MAX );
                                addrpart.copy( srvr, len );
                                send_message(srvr,port,argv[3],argv[4]);
                                sent = 1;


                }//end of outer if

            }//end of while



        if(sent !=1)
        {
                //send message to parent node
                len = parentnodeaddr.length();
                memset( srvr, '\0', BUFFER_MAX );
                parentnodeaddr.copy( srvr, len );
                send_message(srvr,parentnodeport,argv[3],argv[4]);
                string cmd = "echo Sending message to parent:
                                "+parentnodeaddr+">>"+logfile;
                system(cmd.c_str());
        }


    myfile.close();

}//end of outermost if
```