IMPROVING GEOGRAPHIC ROUTING

WITH NEIGHBOR SECTORING

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

_____

Jingren Jin

Certificate of Approval:

_____	_____
Kai H. Chang Professor Computer Science and Software Engineering	Alvin S. Lim, Chair Associate Professor Computer Science and Software Engineering
_____	_____
Min-Te Sun Assistant Professor Computer Science and Software Engineering	George T. Flowers Interim Dean Graduate School

IMPROVING GEOGRAPHIC ROUTING

WITH NEIGHBOR SECTORING

Jingren Jin

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
December 17, 2007

IMPROVING GEOGRAPHIC ROUTING

WITH NEIGHBOR SECTORING

Jingren Jin

_____
Signature of Author

_____
Date of Graduation

THESIS ABSTRACT

IMPROVING GEOGRAPHIC ROUTING WITH NEIGHBOR SECTORING

Jingren Jin

Master of Science, December 17, 2007
(B.S., Yanbian University of Science & Technology, 2004)

74 Typed Pages

Directed by Alvin S. Lim

An ad hoc network consists of many mobile devices which forms a network automatically. Among many ad hoc network routing algorithms, geographic routing algorithm is known as an efficient and scalable routing protocol. The most popular method for geographic routing is Greedy Forwarding. Although Greedy Forwarding is effective in many cases, packets may get routed to dead-end nodes.

We present a new geographic routing algorithm, Geographic Routing with Neighbor Sectoring (GRNS), which has ability to reduce the probability of forwarding packets to dead-end nodes. The GRNS algorithm, like any other geographic routing algorithms, uses location information for packet delivery in multi-hop ad hoc networks. In GRNS, each

node in the network divides its neighbors into 16 sectors and informs its neighboring nodes of its identification, position information and its own sectoring information. A node forwards packets according to its neighboring nodes information (e.g. position, sectoring information) stored in the routing table. The simulation result shows the path length of GRNS is slightly longer than or similar to Greedy Forwarding in networks without dead-end nodes, and less than GPSR in networks with dead-end nodes. The performance of GRNS is very closer to Greedy Forwarding but reduces the probability of forwarding packets to a dead-end node.

ACKNOWLEGEMENTS

First and foremost, I would like to thank God who is my life and strength and with whom I can do all things. I would like to thank my advisor Dr. Alvin Lim for the guidance he has provided throughout my study at Auburn. I would also thank the advisory committee members, Dr. Kai Chang and Dr. Min-Te Sun, my outside reader, Dr. Rob Martin, and my friend Kyu Han Koh. Finally, I would like to thank my wife for her support. Without them, I would have never been able to complete this thesis.

Style manual or journal used: IEEE style guide

Computer Software used: Microsoft word 2003

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Nowadays, wireless network technology has become more and more popular because of the increasing number of mobile wireless devices. In the near future, many more electronic devices will be introduced with the wireless technology.

An ad hoc network is a type of wireless network. It consists of many mobile devices which forms a network automatically. It is different from traditional wireless networks. Ad hoc networks are self-configurable networks and do not require any infrastructures such as base stations or access points. Because of these characteristics of ad hoc networks, the traditional routing protocols were not feasible for the ad hoc networks, and new routing protocols were needed.

Many routing protocols were introduced for the ad hoc network, e.g. DSR [1], DSDV [2], AODV [3], TORA [4], and Geographic Routing [5] [6] [7] [8] [9]. Routing information (e.g. routes to other nodes) is needed for the protocols such as DSR, DSDV, AODV and TORA. Also the routing information is always consistent and up-to-date. So in these protocols, each node has to flood the entire network in order to get the necessary information to maintain the consistency. So it is not feasible for large-scale and mobile ad hoc networks.

Among those routing protocols mentioned above, geographic routing protocol is known as an efficient and scalable routing protocol. Geographic routing uses location information for packet delivery in multi-hop wireless networks. The nodes locally exchange information obtained through GPS (Global Positioning System) or other location determination techniques. Since the nodes locally select next hop nodes based on the neighboring nodes' information and the destination node's location, the routing establishment is not required. The geographic routing is more feasible for large-scale ad hoc networks because of its stateless nature and low maintenance overhead.

The most popular method for geographic routing is simply forwarding data packets to the neighbor which is closest to the destination. Although this greedy method is effective in many cases, packets may get routed to where no neighbor is closer to the destination than the current node.

In this work, a new routing algorithm GRNS (Geographic Routing with Neighbor Sectoring) is presented for the geographic routing. GRNS has ability to reduce the probability of routing the packets to a dead-end node that has no neighbor closer to the destination. In this algorithm, neighbors of a node are divided into 16 sectors. The packet will be forwarded to the node which has more neighbors closer to the destination. In this way, it increases the probability that can avoid routing packets to the dead-end node.

The rest of the thesis is organized as follows. In Chapter 2, we discuss the background information for this research and related research that motivates this research. We summarize general concepts of ad hoc network first, and then discuss several well-known ad hoc protocols. In Chapter 3, first we introduce GRNS. Then we present the algorithm about beaconing, neighbor sectoring and packet forwarding. Finally we present

the implementation of the algorithm. In Chapter 4, we introduce simulation environment. Then we examine the performance of GRNS and compare it with Greedy Forwarding and GPSR. Finally in Chapter 5, we conclude our work and describe some future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Ad Hoc Network

Since mobile devices (laptops, cell phones, and etc) and 802.11/Wi-Fi wireless networking became widespread in the mid to late 1990s, the ad hoc network became a popular research subject [11].

An ad hoc network is a kind of wireless network that consists of wireless nodes. It does not need the support of any infrastructures such as base stations or wireless access points. The nodes in the ad hoc network allow the information to be exchanged between them, unlike traditional wireless network in which the wireless devices communicate with each other through a certain infrastructure such as a base station or access point. In the ad hoc network each node performs as a router which forwards packets for other nodes. Main characteristics [10] of ad hoc networks are listed below:

- Lack of pre-configuration, meaning network configuration and management must be automatic and dynamic.
- Node mobility, resulting in constantly changing network topologies.
- Multi-hop routing.

- Resource limited devices, e.g. laptops, PDAs and mobile phones have power and CPU processing constraints.
- Resource limited wireless communications, e.g. reduced to 10's of kilobits per second by the fact that many nodes must share the radio medium.
- Potentially large networks, e.g. a network of sensors may comprise thousands or even tens of thousands of mobile nodes.

Because of its dynamic and self configuring nature, the ad hoc network is particularly useful in many situations where rapid network deployments are required or it is expensive to deploy and manage network infrastructure. Here are some applications [10] where ad hoc networks can be used:

- Users in a same building sharing documents and other information via their laptops and handheld computer;
- Armed forces creating a tactical network in unfamiliar territory for communications and distribution of situational awareness information;
- Small sensor devices located in animals and other strategic locations that collectively, monitor habitats and environmental conditions;
- Emergency services communicating in a disaster area and sharing video updates of specific locations among workers in the field, and back to headquarters.

Ad hoc networks are extremely useful in many areas, but they also involve many communication problems such as routing. We will study several well-known ad hoc network routing algorithms in the next section.

2.2 Ad Hoc Routing Algorithms

Most research in ad hoc networks focuses on finding efficient methods for forwarding packets through the network. A number of different algorithms for routing in ad hoc networks have been proposed and evaluated in various works. Several well-known routing protocols will be briefly overviewed in the following sections.

2.2.1 Dynamic Source Routing (DSR)

The Dynamic Source Routing protocol (DSR) [1] is a simple and efficient routing protocol for the multi-hop wireless ad hoc network. The network can be completely self-organizing and self-configuring with DSR, which means that there is no need for an existing network infrastructure or administration. DSR is a reactive routing protocol which is able to manage an ad hoc network without using periodic table-update messages like table-driven routing protocols do.

DSR works in two phases: route discovery and route maintenance. When a node has packets to send, it will check the route cache to determine whether it has a route to the destination. If the route exists, the node will use this route to send the packets. Otherwise the route discovery process is executed by broadcasting a route request packet. The route request packet contains the source address, the destination address and a unique identification number. Each node which received the route request packet checks its cache whether a route to the destination exists. If the route does not exist, the node will add its own address and broadcast the packet. Figure 2.1 shows the propagation of the route request packet. When the route request packet reaches the destination or a node that has the route to the destination, a route reply packet is generated and sent to the source

according to the path recorded in the route request packet. Figure 2.2 shows the propagation of the route reply packet.

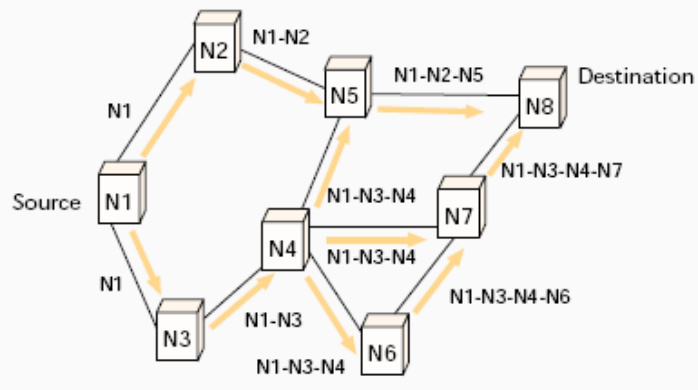

Figure 2.1: The propagation of the route request packet

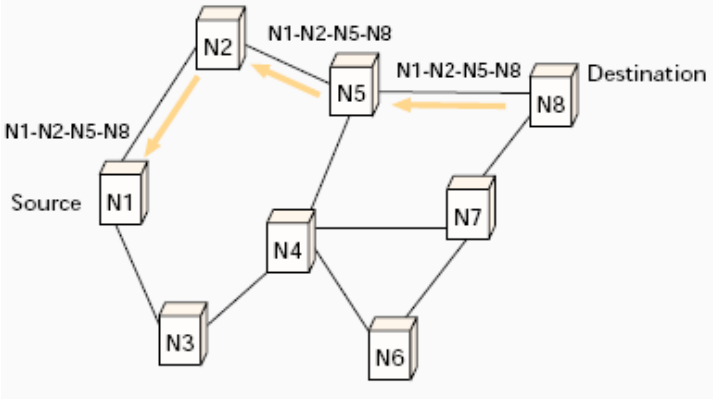

Figure 2.2: The propagation of the route reply packet

In DSR, each node is responsible for confirming that the next hop node in the route receives the packet. Also a node only forwards each packet once. If a node does not receive the packet, the packet is retransmitted up to certain number of times until a confirmation is received from the next hop node. If the node fails to receive the confirmation, a route error packet is generated and sent to the source node. So the source

node can check its route cache for another route to the destination. If no route exists in the cache, the source route will generate a route request packet and broadcast in order to find a new route to the destination.

DSR is based on the Link-State-Algorithms. This means each node saves the best way to a destination. Also if a change occurs in the network topology, the whole network will get this information by flooding.

2.2.2 Destination-Sequenced Distance-Vector (DSDV) Routing

Destination-Sequenced Distance-Vector Routing (DSDV) [2] is a table-driven routing algorithm for the ad hoc network based on the Bellman-Ford algorithm. It was developed by C. Perkins in 1994 [2].

In DSDV, each node maintains a routing table which includes routes to every node in the network and the number of hops. Each entry in the routing table contains a sequence number generated by the destination node. A node distinguishes stale routes from new ones with the sequence number. The sequence number is also used to avoid routing loops.

Routing information is transmitted by broadcasting. Updates have to be transmitted periodically or immediately when there are significant topology changes in the network. There are two possibilities to update routing table:

- Full dump: all information from the transmitting node (whole routing table).
- Incremental dump: all information that has changed since the last full dump.

If a node received new routing information, it will use the route with the most recent sequence number. If the new route has equal sequence number but better metric, this route will be used.

Table 2.1 shows an example of a routing table. Naturally it contains all in the existing network destinations. The Next Hop informs what node is the first node for reaching the destination. The metric a supplement to the costs for routes, is an additional path description. There are even sequence numbers assigned by the destination for sending the next update.

Destination	**Next Hop**	**Hops**	**Seq. No**	**Install Time**
A	A	0	A-846	001000
B	B	1	B-470	001200
C	B	3	C-920	001500
D	B	4	D-502	001200

Table 2.1: Example of a routing table in DSDV

2.2.3 Ad hoc On-Demand Distance Vector (AODV) Routing

The Ad hoc On-Demand Distance Vector (AODV) routing algorithm [3] is a reactive routing protocol designed for ad hoc networks. AODV is based on DSDV which is described in section 2.2.2. AODV significantly improved DSDV.

AODV uses a destination sequence number for each routing table entry. The sequence number is created by the destination node. The sequence number which includes in a route request or route reply is sent to requesting nodes. Loop freedom and simplicity is ensured because of the use of the sequence numbers. Nodes in the network determine the freshness of routing information using the sequence number. If a node has two routes to the same destination, it will choose the route with the greatest sequence number.

Three kinds of packets are defined by AODV: Route Requests (RREQs), Route Replies (RREPs) and Route Errors (RERRs). When a source node needs to make a connection with a destination node and no route to the destination exists in the routing table, the source node broadcasts a RREQ packet to the network. The RREQ packet is forwarded until it reaches the destination node or an intermediate node that has the route to the destination node. When the RREQ packet reaches the destination node or an intermediate node with the route to the destination node, the node sends a RREP packet to the source node. The RREP packet travels back along the reverse path until it reaches the source node. So AODV can only support bidirectional links. When a link fails, a RERR packet is passed back to a transmitting node, and the route discovery process is repeated.

In AODV, there is no extra traffic for communication along existing links. Also, distance vector routing is simple, and doesn't require much memory or calculation.

2.2.4 Temporally Ordered Routing Algorithm (TORA)

The Temporally Ordered Routing Algorithm (TORA) [4] presented by Park and Corson is a loop-free and highly adaptive distributed routing algorithm based on the link reversal routing algorithms.

TORA is source-initiated routing algorithm and has multipath routing capability. TORA tries to localize control messages to a very small set of nodes where there is a topological change. So nodes must maintain routing information about their one-hop neighbors. TORA has three main phases: route creation, route maintenance and route erasure.

Each node maintains a separate directed acyclic graph (DAG) to other nodes. When a source node has packets to send to a destination node, it broadcasts a QUERY packet with the destination address. When the QUERY packet reaches the destination node or an intermediate node that contains the route to the destination node, the destination or the intermediate node generates an UPDATE packet which contains its own height with respect to the destination node and sends it back. Each node that received the UPDATE packet sets its height to a value greater than that of its neighbor from which it received the UPDATE packet. When a node discovers a network partition, it will generate a CLEAR packet and broadcasts it. The CLEAR packet resets the routing state and removes the routes that do not exist any more from the network.

Nodes use a height metric to establish a DAG rooted at the destination node in the route creation and maintenance phases. The height metric is dependent on the logical time of a link failure, so time becomes an important factor for TORA. In TORA, it assumes that all nodes are synchronized with each other. This can be accomplished via an external time source such as the Global Positioning System.

2.2.5 Geographic Routing

Geographic routing also known as position based routing uses location information for packet transmission in ad hoc networks [5] [6] [7] [8] [9]. Nodes in the network locally exchange location information which is obtained through GPS (Global Positioning System) or other location determination techniques [12].

Greedy Forwarding is the most popular routing method in geographic routing. It is simply forwarding packets to the neighbor node which is closest to the destination node

[5] [6] [7] [13] [14] [15]. Although the Greedy Forwarding method is effective in most cases, packets can get to the node where there is no neighbor closer to the destination node. Therefore Greedy Forwarding method alone does not guarantee the delivery of packets because of the dead-end nodes. Many recovery methods have been introduced in order to route around such dead-end nodes and guaranteed packet delivery if a route to the destination node exists [5] [6] [7] [16]. The Greedy Perimeter Stateless Routing (GPSR) is one of the routing algorithms that can avoid the dead-end nodes.

2.2.5.1 Greedy Perimeter Stateless Routing (GPSR)

Greedy Perimeter Stateless Routing (GPSR) [6] is a routing algorithm that combines two different routing methods. The first method is the Greedy Forwarding method. The Greedy Forwarding method will be used as long as possible if the packets do not reach at dead-end node. But when the packet arrives on a node and the node does not have any neighbor nodes that nearer to the destination node, the node can not use Greedy Forwarding method to transmit the packet. Then it will use the second method, the perimeter forwarding.

In GPSR, every node in the network has a local table, in which all neighbors of the node is listed by name and position. A proactive Broadcast (beacon) refreshes this table of each node in a regular time interval. The source node gives the packet the destination node's address. This address will not be changed by any other node who forwards the packet. Table 2.2 shows the header structure of the GPSR packets.

The basis of the Greedy Forwarding Method is that the source node knows the geographic position of the destination node. The position of the destination node will be

integrated into the packet. The node that received the packet looks in its local table where the positions of all neighboring nodes are listed. The node forwards the packet to a neighboring node which is the nearest to the destination and so on until the packet arrived on the destination node. This is illustrated in figure 2.3 below.

Field	Function
D	Destination Location
Lp	Location Packet Entered Perimeter Mode
Lf	Point on xV Packet Entered Current Face
e0	First Edge Traversed on Current Face
M	Packet Mode: Greedy or Perimeter

Table 2.2: The header structure of the GPSR packet

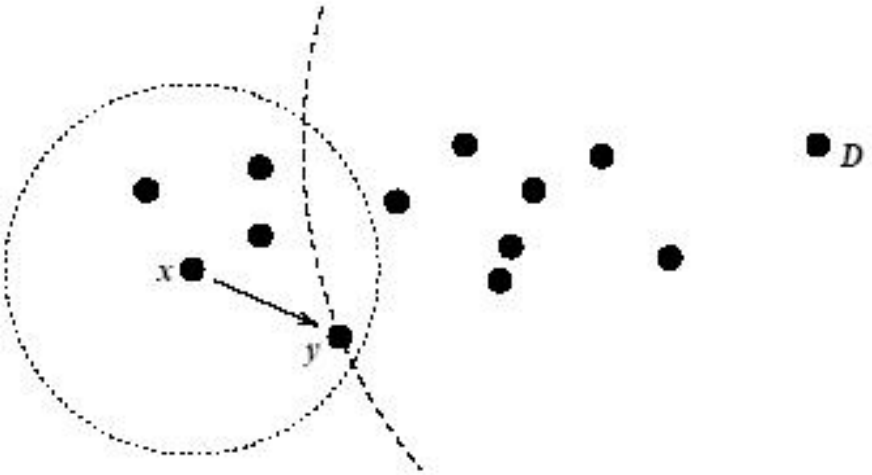

Figure 2.3: Greedy Forwarding example. Y is x's closest neighbor to D

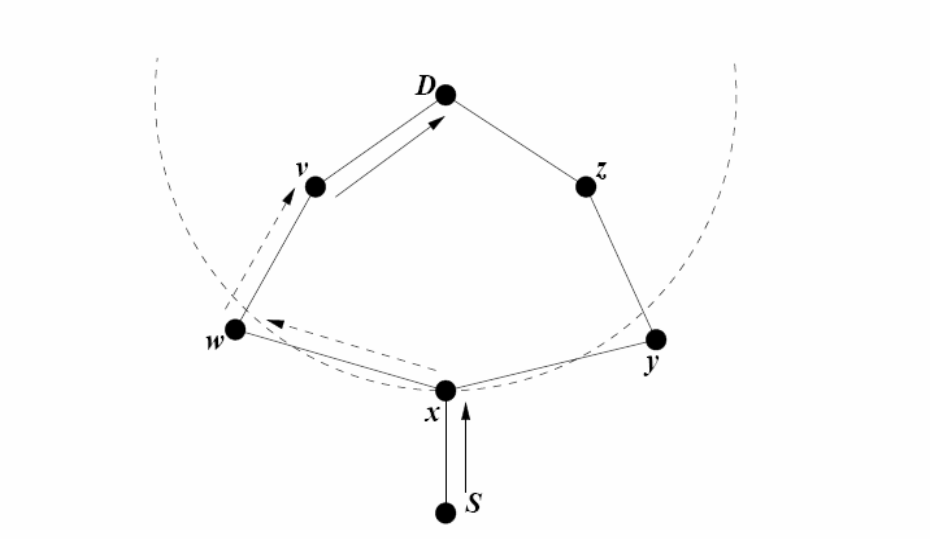

Figure 2.4: Perimeter mode example

Now, when a packet arrives on a dead-end node which does not have any neighboring node nearer to the destination, the node can not use Greedy Forwarding method any more. So the packet mode will be changed into perimeter mode and the perimeter method will be active, as shown in figure 2.4. Node S has a packet to send to the node D. When the packet arrives on the node x which does not have nearer node to the destination, the node x has to use the perimeter method. The node x changes the packet mode to the perimeter mode, records its own position in the packet, and forwards the packet to the node w which is the first hop on the perimeter. The node w is not closer to the destination node D than x, so w forwards the perimeter mode packet to the node v using righthand-rule. Because the node v is closer to the destination node D than the node x, the node v changes the packet mode into the Greedy Forwarding mode, and forwards it greedily to the destination node D.

2.3 Analysis of Ad Hoc Routing Algorithms

In this section, we analyze ad hoc routing algorithms mentioned in the previous sections.

DSDV is quite suitable for creating ad hoc networks with small number of nodes. In DSDV, a regular update of the routing tables is required, thus it reduces the bandwidth efficiency. When there is a network topology change, it generates high traffic by broadcasting in order to maintain the routing tables. So DSDV is not an effective algorithm for a network with large number of nodes and for a network with frequent topology change.

AODV is a loop free routing algorithm. AODV creates no extra traffic for communication along existing route, therefore it reduces control overhead. Also distance vector routing is simple, and does not require much memory space or calculation. However, in AODV, more time is required for establishing a connection, and the initial communication to establish a route is heavier than some other algorithms.

DSR is a reactive routing protocol and does not need to periodically flood the network for updating the routing tables like DSDV. In DSR, the nodes in the network are able to make use of the route cache information efficiently in order to reduce the control overhead. The source node only tries to find a route if there is no route in its cache. DSR algorithm is only efficient for ad hoc networks with less than 200 nodes. In route discovery process, flooding the network can cause collisions between the packets. Also a small time delay always exists at the beginning of a new connection because the source node has to find the route to the destination node first.

TORA is a highly adaptive, efficient and scalable routing algorithm [18]. It is a source-initiated on-demand protocol and it finds multiple routes between the source and the destination. TORA is a fairly complicated protocol but its main feature is that when a link fails the control messages are only propagates around the point of failure. While other protocols need to re-initiate a route discovery when a link fails, TORA would be able to patch itself up around the point of failure. This feature allows TORA to scale up to larger networks but has higher overhead for smaller networks.

Geographic routing incurs low route discovery overhead relative to flooding-based approaches, and hence conserves energy and bandwidth. It is stateless in the sense that nodes need not maintain per-destination information, and only neighbor location

information is needed to route packets. In mobile networks with frequent topology changes, geographic routing can find new routes quickly by using only local topology information. For these reasons, geographic routing is expected to become the protocol of choice for many applications in wireless networks.

CHAPTER 3

GRNS ALGORITHM AND IMPLEMENTATION

3.1 Overview of GRNS

GRNS (Geographic Routing with Neighbor Sectoring) algorithm like any other geographic routing algorithms uses location information for packet delivery in multi-hop ad hoc networks. The nodes in the network locally exchange information obtained through GPS (Global Positioning System) or other location determination techniques. Since the nodes locally select next hop nodes based on the neighboring nodes' information and the destination node's location, the routing establishment is not required.

As mentioned in previous chapters, the most popular method for geographic routing is simply forwarding data packets to the neighbor which is closest to the destination. Although this greedy method is effective in many cases, packets may get routed to where no neighbor is closer to the destination than the current node. In this case, the most well-known recovery method, GPSR, tries to route around the dead-end nodes and deliver the packet to the destination node if it exists. The detour found by the perimeter mode is generally lengthy, so it results in more number of hops. This can waste energy and make excessive delay. [17]

In GRNS, each node in the network divides its neighbors into 16 sectors. Each node will inform its neighbor with its position and its sectoring information which includes the

number of neighboring nodes in each sector. When a node has a packet to send to a certain destination node, it will use the position information of its neighboring nodes to calculate the distance from each neighboring node to the destination node. For each neighboring node that is closer to the destination, the routing metric F is calculated from Equation (1) shown in following section. The node which has a packet to send chooses the neighboring node that has the biggest routing metric value F as the next hop node. The specific algorithm used to decide the next hop node will be discussed in following section.

GRNS algorithm is a kind of geographic routing algorithm that can lower the probability of forwarding packets to the dead-end nodes. In following section we will discuss the specific algorithm and the implementation.

3.2 Algorithm

GRNS is a simple geographic routing algorithm. The uniqueness of GRNS is dead-end node prevention. A node that has packets to transmit determines next hop node according to the position information and the sectoring information of its neighboring nodes, then forwards packets to the appropriate node.

3.2.1 Beaconing

In GRNS, each node knows its neighbors' status by using a simple beaconing protocol. Each node in the network periodically broadcast beacon packets. The beacon packets contain the node's identifier (e.g. ip address), location information, velocity information and sectoring information. We encode each node's location as two four-byte

floating-point quantities, for x and y coordinate values. This beaconing mechanism is similar to the pro-active routing protocol which is avoided by DSR and AODV.

Table 3.1 below is the beacon packet header structure. The field id is the identifier of the sender. Nodes in the network have different identifiers. The field x_axis and y_axis are the (x, y) coordinates of the sender. The field velocity is the current velocity information of the sender. The field sector[16] is contains the sectoring information of the sender.

Field	**Type**
id	char [10]
x_axis	float
y_axis	float
velocity	float
sector[16]	int

Table 3.1: The header structure of the beacon packet

3.2.2 Sectoring

In GRNS, the number of sectors can be 3, 4, 5, 6,… any number, but in this work each node divides its neighbors into 16 sectors according to its neighbors' position information.

Once a node received a beacon packet, first it decides whether the node that sent the beacon packet is in its transmission range. If the beacon packet sender is in the node's range, it will calculate which sector the sender is in and records the sender's information including the sender's sectoring information in its routing table. Table 3.2 shows the

algorithm for Neighbor Sectoring. $x_{current}$ and $y_{current}$ are current node's (x, y) coordinates, $x_{neighbor}$ and $y_{neighbor}$ are neighboring node's (x, y) coordinates, distance is the distance from the current node to the neighboring node, and ratio is the absolute ratio value of $\left(y_{neighbor} - y_{current}\right)$ and the distance between the current node and the neighboring node.

$distance = \sqrt{(x_{neighbor} - x_{current})^2 + (y_{neighbor} - y_{current})^2}$; $ratio = \left| \frac{(y_{neighbor} - y_{current})}{distance} \right|$

if ($x_{neighbor} \geq x_{current}$ && $y_{neighbor} \geq y_{current}$)
{
    if ($0 \leq ratio < \sin(\pi/8)$)
    {
        the neighbor node belongs to sector 0;
    }
    else if ($\sin(\pi/8) \leq ratio < \sin(\pi/4)$)
    {
        the neighbor node belongs to sector 1;
    }
    else if ($\sin(\pi/4) \leq ratio < \sin(3\pi/8)$)
    {
        the neighbor node belongs to sector 2;
    }
    else
    {
        the neighbor node belongs to sector 3;
    }
}
else if ($x_{neighbor} < x_{current}$ && $y_{neighbor} \geq y_{current}$)
{
    if ($\sin(3\pi/8) \leq ratio < \sin(\pi/2)$)
    {
        the neighbor node belongs to sector 4;
    }
    else if ($\sin(\pi/4) \leq ratio < \sin(3\pi/8)$)
    {
        the neighbor node belongs to sector 5;
    }
    else if ($\sin(\pi/8) \leq ratio < \sin(\pi/4)$)
    {
        the neighbor node belongs to sector 6;
    }
    else
    {
        the neighbor node belongs to sector 7;
    }

```
}
else if ( x_neighbor ≤ x_current && y_neighbor ≤ y_current )
{
        if ( 0 ≤ ratio < sin(π/8) )
        {
                the neighbor node belongs to sector 8;
        }
        else if ( sin(π/8) ≤ ratio < sin(π/4) )
        {
                the neighbor node belongs to sector 9;
        }
        else if ( sin(π/4) ≤ ratio < sin(3π/8) )
        {
                the neighbor node belongs to sector 10;
        }
        else
        {
                the neighbor node belongs to sector 11;
        }
}
else
{
        if ( sin(3π/8) ≤ ratio < sin(π/2) )
        {
                the neighbor node belongs to sector 12;
        }
        else if ( sin(π/4) ≤ ratio < sin(3π/8))
        {
                the neighbor node belongs to sector 13;
        }
        else if ( sin(π/8) ≤ ratio < sin(π/4))
        {
                the neighbor node belongs to sector 14;
        }
        else
        {
                the neighbor node belongs to sector 15;
        }
}
```

Table 3.2: Algorithm for Neighbor Sectoring

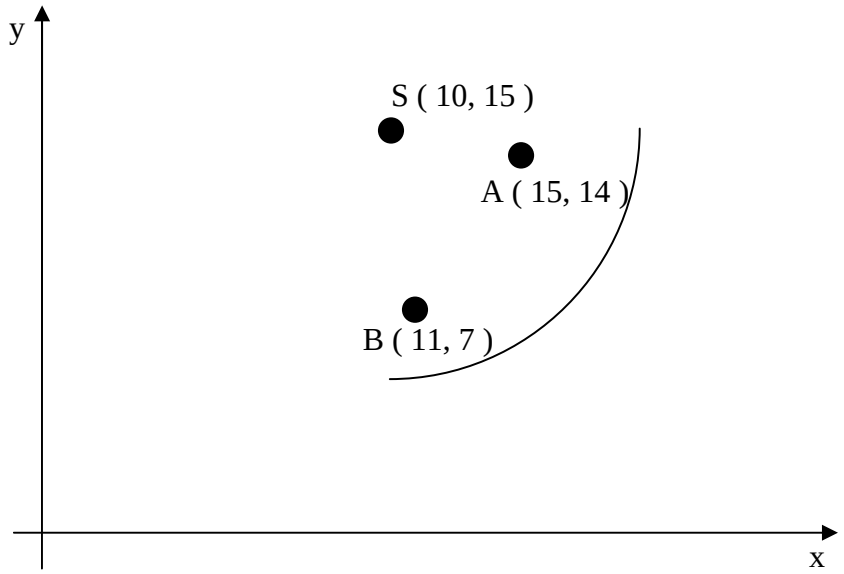

Figure 3.1: Example of sector decision

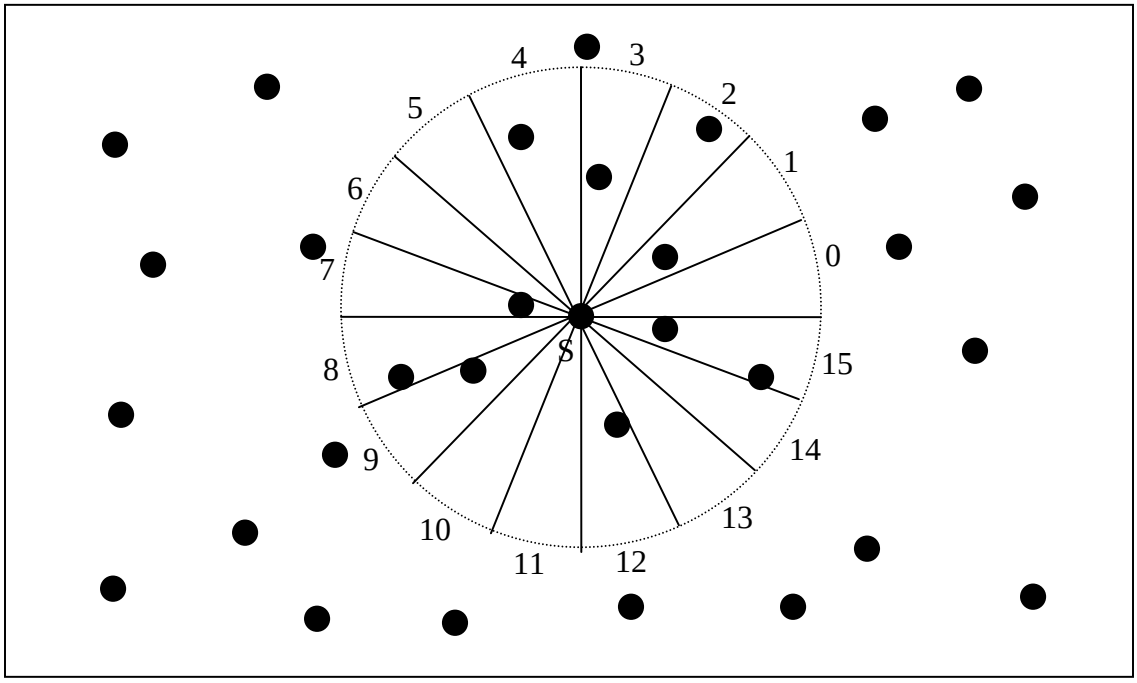

Figure 3.2: Sectoring example

Figure 3.1 shows an example how a node decides which sector a neighboring node belongs to. The current node S has two neighboring nodes, A and B. The (x, y) coordinates of S, A and B are (10, 15), (11, 7), (15, 14). From these (x, y) coordinates, we can decide A belongs to S's sector 12 and B belongs to S's sector 15 according to the algorithm shown in Table 3.2.

Figure 3.2 illustrates an example of sectoring for node S. S has ten neighboring nodes, and S classifies these neighboring nodes into appropriate sectors according to the neighboring nodes' location information and calculates the number of nodes in each sector. This sectoring information is included in the beacon packet.

3.2.3 Packet Forwarding

Once a node has a packet to send or a node receives a packet to forward, the node has to decide to which neighboring node it should forward the packet. If the destination node is a neighboring node or in the range of a neighboring node, it will directly forward the packet to the node. Otherwise, Equation (1) is used by the node to decide the next hop node.

$$F = \frac{n * D_1}{D_2} \qquad (1)$$

In Equation (1) F is the routing metric, n is the number of the neighboring node's neighbors that are closer to the destination. The n value can be calculated from the sectoring information of the neighboring node in the current node's routing table. D_1 is the distance from the current node to the destination node, D_2 is the distance from the neighboring node to the destination node, and D_2 is smaller than D_1. A node will forward

data packets to the neighboring node with the largest routing metric F. This lowers the probability of forwarding a data packet to a dead-end node. A dead-end node has no neighboring node closer to the destination node, and therefore its value of n is 0 and consequently F is 0, so the data packet will not be forwarded to the dead-end node. Table 3.3 shows how n is calculated. If the destination node is in the range of a neighboring node, the packet will be forwarded to that neighboring node no matter what the value of F is.

Direction of the destination relative to neighboring node is in the following sector	**n**
Sector 0 direction	Total number of nodes in sector 0, 1, 2, 3, 13, 14 and 15
Sector 1 direction	Total number of nodes in sector 0, 1, 2, 3, 4, 14 and 15
Sector 2 direction	Total number of nodes in sector 0, 1, 2, 3, 4, 5 and 15
Sector 3 direction	Total number of nodes in sector 0, 1, 2, 3, 4, 5 and 6
……	……
Sector 13 direction	Total number of nodes in sector 0, 10, 11, 12, 13, 14 and 15
Sector 14 direction	Total number of nodes in sector 0, 1, 11, 12, 13, 14 and 15
Sector 15 direction	Total number of nodes in sector 0, 1, 2, 12, 13, 14 and 15

Table 3.3: Calculation of n value

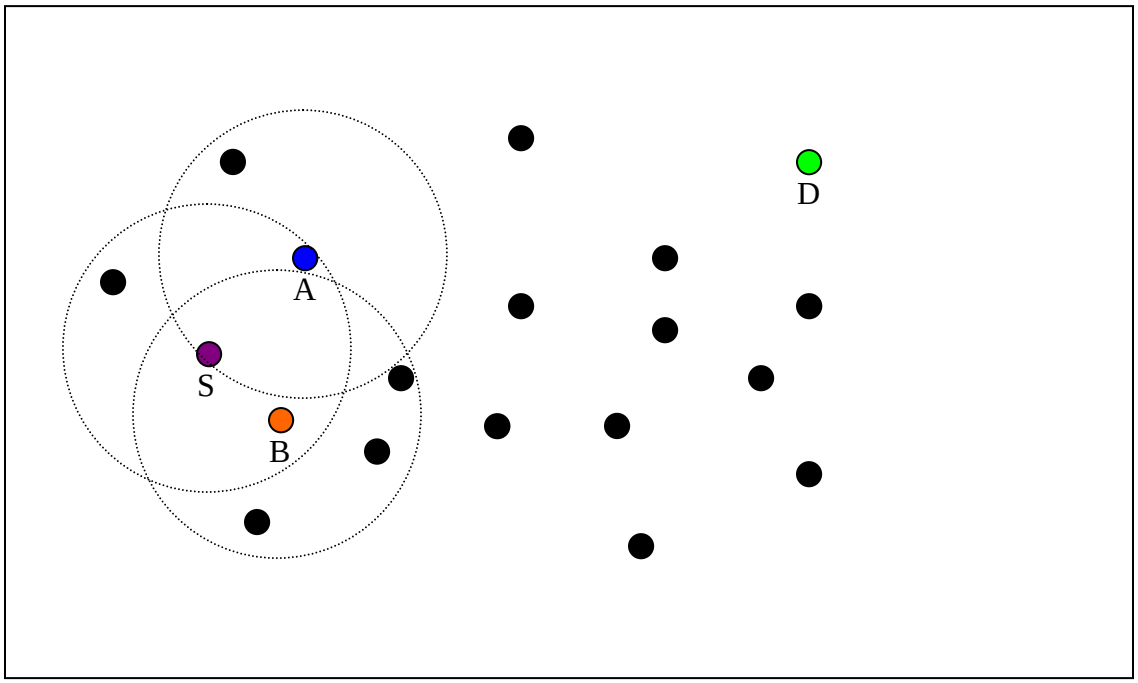

Figure 3.3: Packet forwarding example

Figure 3.3 shows a packet forwarding example. In Figure 3.3, a node S has a packet to send to the destination node D, and S has two neighboring nodes A and B that are closer to the destination node D. A is closer to D than B, so Greedy Forwarding will forward the packet to A rather than B, but in GRNS it is different. In GRNS, S will calculate the routing metrics of A and B according to the sectoring information of A and B in its routing table. Since A has no neighboring nodes closer to D, A is dead-end node. So the routing metric value of F_A is 0. For the node B, it has two neighboring nodes that are closer to D, and the routing metric value of F_B is greater than 0. So S forwards the packet to B rather than A because F_B is greater than F_A .

3.3 Implementation of GRNS

The implementation of GRNS consists of a router and a network configuration file, config.txt. The router represents a node in the network that has the following three functionalities.

- The router can be a source node that generates packets into the network.
- The router can be an intermediate node that forwards packets to a next-hop node.
- The router can be a destination node that accepts packets.

The network configuration file-config.txt contains identifier and location information of all nodes in the network.

In this work, we assume all the nodes in the network are static. We also assume no beaconing is required. Once simulation begins, each node knows its neighboring nodes' information by reading the configuration file which contains all necessary information of all nodes in the network.

3.3.1 Packet Header

Table 3.4 below is the packet header structure. The field seq contains the sequence number of each GRNS packet. Different packets have different sequence numbers in order to identify each packet. The seq field is an integer type data. The field s_id and d_id are the identifiers of the source node and the destination node. Nodes in the network have different identifiers. The field s_x_axis and s_y_axis are the (x, y) coordinates of the source node. Similarly the field d_x_axis and d_y_axis are the (x, y) coordinates of the destination node. The field num_hops is the hop counts of the packets from the source node to the current node. The filed num_hops is implemented in order to compare the

path length of GRNS with Greedy Forwarding and GPSR algorithms. In the real systems, this field will not be implemented.

Field	Type
seq	int
s_id	char [10]
d_id	char [10]
s_x_axis	int
s_y_axis	int
d_x_axis	int
d_y_axis	int
num_hops	int

Table 3.4: The header structure of the GRNS packet

3.3.2 Implementation of the Router

The router is implemented in grns.c. As mentioned before, each node maintains a routing table that contains neighboring nodes' identifier, (x, y) coordinates, ip address and port number. Table 3.5 shows the structure of the routing table. In grns.c, the routing table which contains neighboring nodes' information is implemented as a linked list. Table 3.6 is the routing table structure that is implemented in grns.c.

Field	Function
id	The neighboring node's identifier
x_axis	The neighboring node's x coordinate
y_axis	The neighboring node's y coordinate
ip	The neighboring node's ip address
port	The neighboring node's port number
sector	The neighboring node's sectoring information

Table 3.5: The structure of the routing table

```
struct routingnode
{
        char name[10];              // name of the neighboring node
        char x_axis[5];             // neighboring node's x coordinate
        char y_axis[5];             // neighboring node's y coordinate
        char ip[15];                // neighboring node's ip
        char port[5];               // neighboring node's port#
        int sector[16];             // sector[0]: number of nodes in sector 0 for the neighboring node,
                                    // sector[1]: number of nodes in sector 1 for the neighboring node,
                                    //and so on
        struct routingnode * next;  // next neighboring node's structure
};
```

Table 3.6: Implementation of the routing table in grns.c

Three main functions, read_file(), sectoring() and search_next(), are implemented in grns.c.

- read_file(FILE *config_file)
  - The input for this function is a FILE pointer which is "config.txt" file. This text file contains all nodes' information (e.g. identifier, location information, ip address, and port number).

  - This function reads the network configuration file and decides which nodes are the neighboring nodes. Also this function calls the sectoring() function to decide neighboring node's sectoring information.
  - It returns the header pointer of the linked list which implements the routing table.

- sectoring(double ratio, int cur_x, int cur_y, int neighbor_x, int neighbor_y)
  - The inputs for this function are one double variable which is the absolute ratio value of $\left(y_{neighbor} - y_{current}\right)$ and distance between the current node and the neighboring node, and four integer variables which are the (x, y) coordinates of the current node and the neighboring node.
  - This function is called in the read_file() function. After reading the whole network configuration file, a node establishes a linked list that works as a routing table containing its neighboring nodes' information except the sectoring information. Then each neighboring node calculates its sectoring information with location information by reading the network configuration file. This is because we do not implement beaconing in this work.
  - This function returns the sector number that the node belongs to.

- search_next(struct routingnode *neighbor_list, int x, int y)
  - The input for this function is the header pointer of the linked list and the (x, y) coordinates of the destination node.
  - This function is called when an intermediate node has a packet to send out. Since an intermediate node needs to find the next hop node, it traverse the linked list which contains all neighboring nodes' information, and then it

calculates an F value for each neighboring node using location information and sectoring information for each node. The intermediate node chooses the neighboring node with the largest F value as the next hop node.

- This function returns the pointer of the neighboring node that is the next hop node.

When the router works as a source node, it generates packets which include destination's identifier and (x, y) coordinates for the destination node. The destination identifier is read from the command line when the node is initialized. The source node knows the coordinates of the destination node when it executes the read_file() function and reads the network configuration file which contains all nodes' information in the network. Then it is looking for the next hop node by executing the search_next() function, and forwards the packet to the appropriate node. Figure 3.4 shows the flow chart for the router that works as a source node.

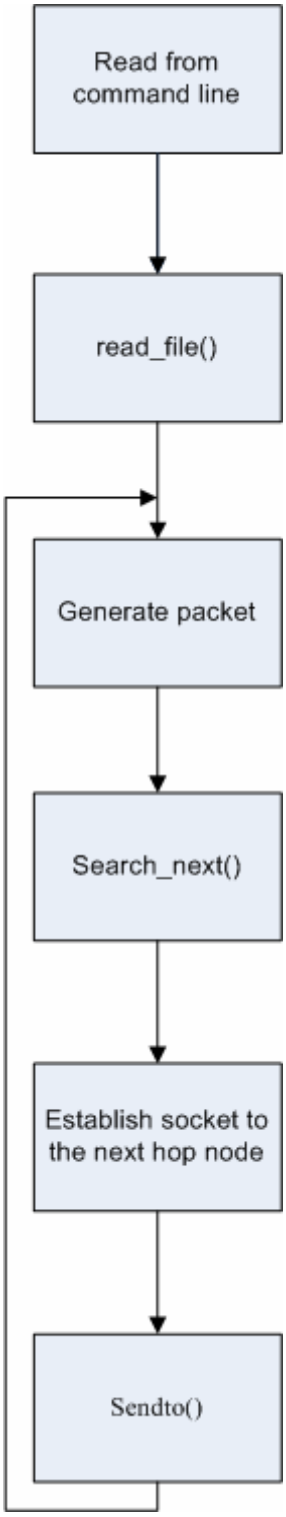

Figure 3.4: Flow chart of a node working as a source node

When the router works as an intermediate node or a destination node, it first establishes an UDP socket and listens to the network. When it received a packet, the node compares the destination information (e.g. identifier and location) with its own information (e.g. identifier and location). If it is the destination node, it will send back acknowledgement. If the current node is not the destination node, it will look for the next hop node by executing the search_next() function, and then forward the packet to the appropriate node. Figure 3.5 shows the flow chart for the router that works as an intermediate node or a destination node.

In the real systems, each node in the network knows the neighboring nodes' information by the periodic beaconing. Once a node receives a beacon packet, it decides whether the sender is in its range. If the sender is in the range, it will add the sender information to its own routing table if the sender is a new neighboring node, or update the sender's information in the routing table if the sender is an already known neighboring node. If it does not hear from the known neighboring nodes for a certain period, it removes those nodes from its routing table. Once a node receives a data packet, it calculates the routing metric F for each neighboring node that closer to the destination node, and it selects the neighboring node that has the biggest routing metric value F as the next hop node.

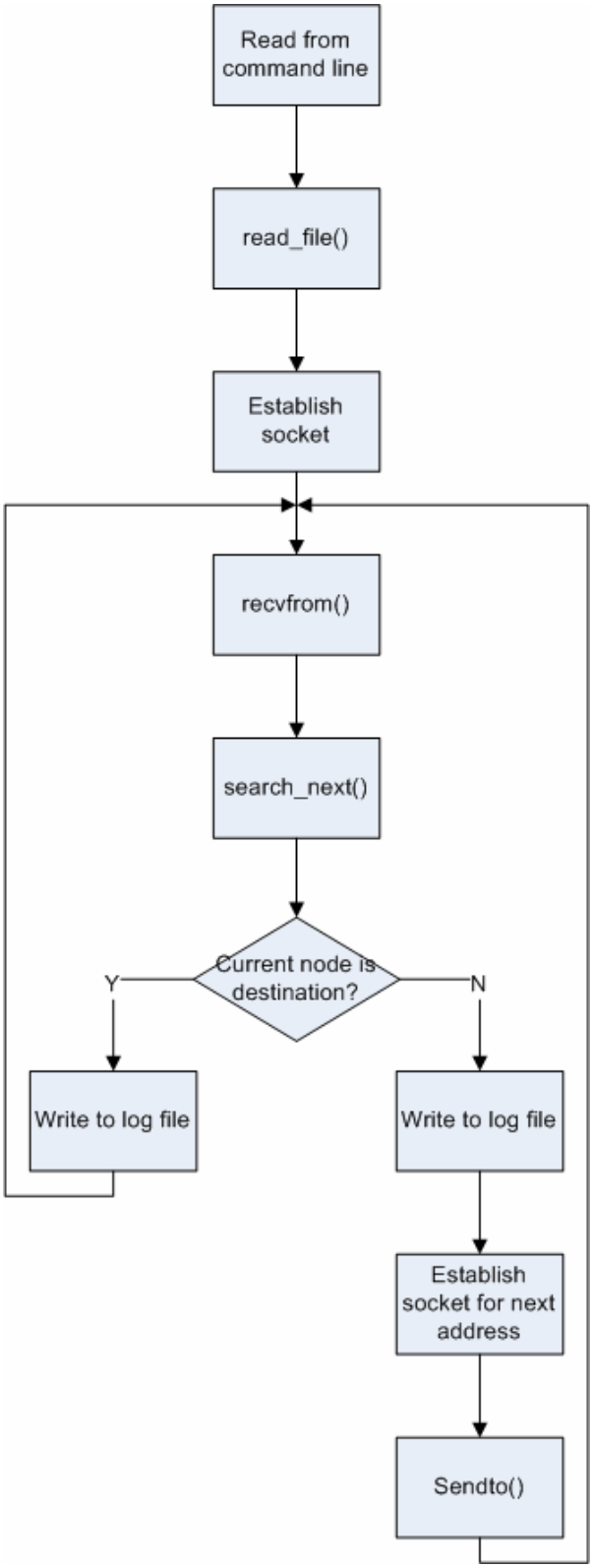

Figure 3.5: Flow chart of a node working as an intermediate or a destination node

3.3.3 Implementation of the Network Configuration file

The forwarding of the packet is based on the current network topology, indicated by the network configuration file. This file is generated by a random generator named node_generator.c. This network configuration file will be taken as a command line argument to initiate a router. Every node knows its own GPS position and that of all its neighbors. This information is found in the network configuration file and formatted as shown below. The first column is the node's identifier, the second and the third columns are the (x, y) coordinates for the corresponding node, the forth and fifth columns represent the IP address and the port number of the corresponding node.

config.txt
node1 0 0 131.204.14.29 5000
node2 133 27 131.204.14.29 5001
node3 72 16 131.204.14.29 5002
… …
node29 180 19 131.204.14.30 5002
node30 174 12 131.204.14.29 5002
……

Table 3.7: Structure of the network configuration file

3.4 Development Environment

All the software is implemented using C programming language and compiled and run on a Linux system. All our simulations are done on two Linux-based machines. The machines used are faster than the actual nodes in the real world. Each is 1000MHZ CPU with 1.86 gigabytes of ram and has 100Mbps wired connections. The operating system is openSUSE 10.2 (i586). We used GNU C compiler 4.1.2 to compile our simulation program.

CHAPTER 4

PERFORMANCE EVALUATION

In this chapter, we discuss the results from our performance evaluation. We compare the performance of GRNS against the Greedy Forwarding and GPSR algorithms.

4.1 Evaluation

In this work, nodes in the network do not have mobility which means the network has static topology. We simulated GRNS in different network topology. Also we simulated with different network density. Then if there are no dead-end nodes on the path, we compare the path length with Greedy Forwarding. If there are dead-end nodes on the path, we compare the path length with GPSR. In this work the path length is defined as the number of hops and the transmission range for each node is 200m.

4.1.1 Performance Evaluation with Greedy Forwarding

In order to compare the performance of GRNS with Greedy Forwarding, we ran four sets of simulations, with different network density. In each simulation set, we performed ten individual simulations. We compare the average path length of GRNS and Greedy Forwarding.

We randomly deployed 50, 70, 90 and 110 nodes in 1000 meters by 1000 meters area in each simulation set. The nodes in the network do not have mobility. We randomly select a source node and a destination node, where the distance between the source node and the destination node is greater than 800 meters. Figure 4.1 shows the average path length (number of hops) of GRNS and Greedy Forwarding in each simulation set.

From the Figure 4.1 we can see the path length of GRNS is greater than Greedy Forwarding in a lower density network, but very closer to the Greedy Forwarding in a higher density network. This is because there is a higher probability that there are more neighboring nodes closer to the destination node in higher density networks.

In order to see how GRNS algorithm reduces the probability of forwarding packets to the dead-end nodes, we ran three sets of simulation with different network density, 50, 70, 90 and 110 nodes in 1000 meters by 1000 meters area. Each set contains twenty individual simulations. In each simulation, a source node and a destination node are randomly selected, and there are dead-end nodes between the source node and the destination node. Figure 4.2 shows the packet delivery rate of GRNS and Greedy Forwarding with different network density. From Figure 4.2 we can see GRNS reduces the probability of forwarding packets to the dead-end nodes.

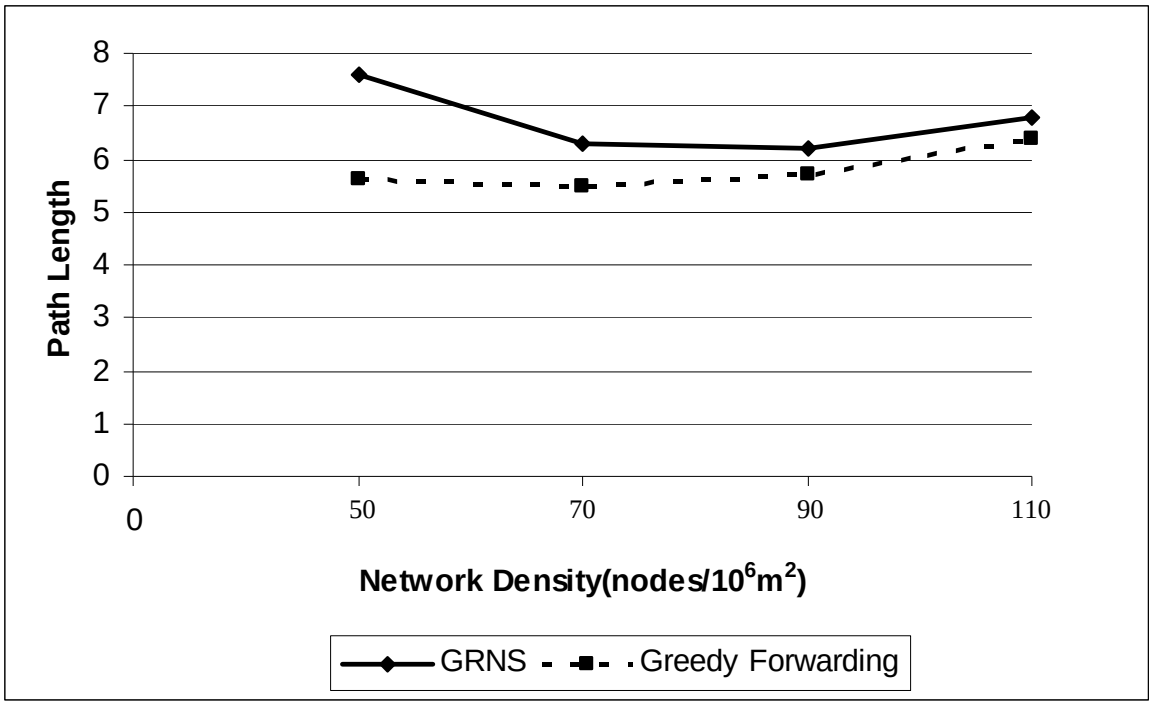

Figure 4.1: Average path length with different network density

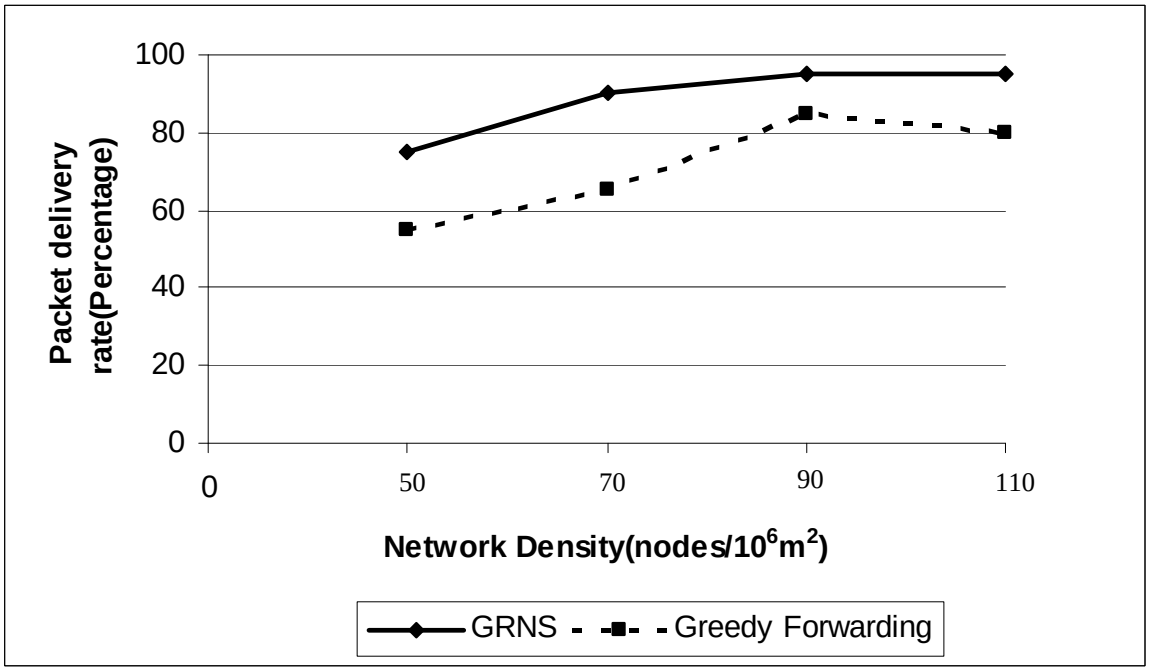

Figure 4.2: Packet delivery rate with different network density

4.1.2 Performance Evaluation with GPSR

In order to compare the performance of GRNS with GPSR, we ran three sets of simulations, with different network density, 50, 70, 90 and 110 nodes in 1000 meters by 1000 meters area. We performed ten individual simulations in each set. We randomly select a source node and a destination node where there is at least one dead-end node, and the distance between the source node and the destination node is greater than 800 meters. Figure 4.3 shows the average path length (number of hops) of GRNS and GPSR in each simulation set. From this result, we can see GRNS gives shorter path length than GPSR. This is because the detour found by the perimeter mode is generally long, and it results in more number of hops [17]. In GRNS, instead of detours it prevents forwarding to dead-end node.

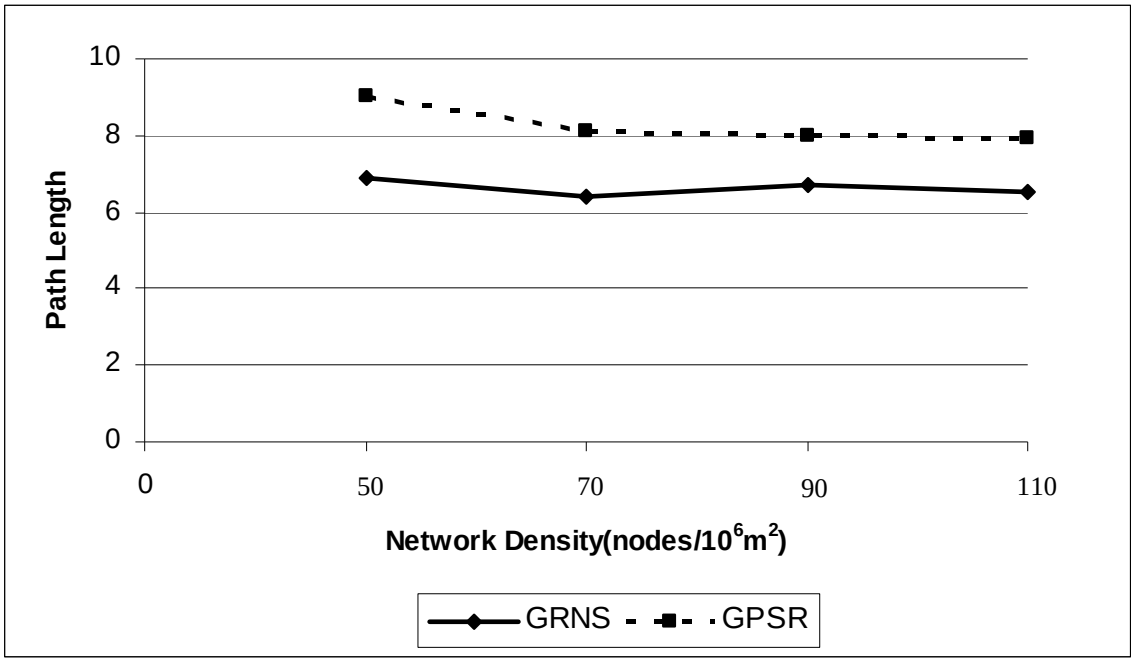

Figure 4.3: Average path length comparison with GPSR

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we have proposed, simulated and evaluated a new geographic routing algorithm, GRNS. We concluded that GRNS improves reliability of geographic routing protocol. GRNS decides next hop node using the position and sectoring information of the neighboring nodes, and it reduces the probability of forwarding packets to the dead-end nodes. From the simulation result, we notice that,

- The path length of GRNS is closer to the Greedy Forwarding method in high density networks.
- The path length of GRNS is slightly greater than that of the Greedy Forwarding method in low density networks.
- For networks with dead-end nodes, GRNS reduces the probability of forwarding packets to the dead-end nodes.
- For networks with dead-end nodes, the path length of GRNS is shorter than the GPSR algorithm.

5.2 Future Work

Since we simulated with a static network topology, in the future, we will implement GRNS with beaconing and simulate it in a network which has mobile nodes. Also we will implement GRNS with awareness to energy consumption, velocity of mobile nodes and various transmission ranges in mobile nodes.

BIBLIOGRAPHY

[1] D. Johnson and D. Maltz, “Dynamic Source Routing in Ad Hoc Wireless Networks,” *Mobile Computing*, Chapter 5, Kluwer Academic, pp. 153-181, 1996

[2] C. E. Perkins and P. Bhagwat, “Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers,” *Computer Communications Review*, pp. 234-244, October 1994

[3] C. E. Perkins and E.M. Belding-Royer, “Ad hoc on-demand distance vector(AODV) routing,” *in IEEE Workshop on Mobile Computing Systems and Applications*, Feb. 1999

[4] V. D. Park and M. S. Corson, “A Highly Adaptive Distributed Routing Algorithm for Mobile and Wireless Networks,” *Proceeding of IEEE INFOCOM’97*, pp. 103-112, April 1997

[5] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia, “Routing with guaranteed delivery in ad hoc wireless networks,” *in Proceedings of the 3rd International Workshop on Discrete algorithms and methods for mobile computing and communications*. 1999

[6] B. Karp and H. T. Kung, “GPSR: greedy perimeter stateless routing for wireless networks,” *in Proceedings of the 6th ACM/IEEE MobiCom,* pp. 243–254, 2000

[7] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger, “Geometric ad-hoc routing: of theory and practice,” *in Proceedings of the 22nd annual symposium on Principles of distributed computing*, pp. 63–72, 2003

[8] D. Niculescu and B. Nath, “Trajectory based forwarding and its applications,” *in Proceedings of the 9th ACM/IEEE MobiCom,* pp. 260–272, 2003

[9] I. Stojmenovic, “Position-based routing in ad hoc networks,” *IEEE Communications Magazine*, pp. 128–134, July 2002.

[10] ARC Communications Research Network. website: http://www.acorn.net.au/report/adhocnetworks/adhocnetworks.cfm

[11] Wikipedia. website: http://www.wikipedia.org

[12] J. Hightower and G. Borriello, “Location systems for ubiquitous computing,” *IEEE Computer*, vol. 34, no. 8, pp. 57–66, 2001.

[13] G. G. Finn, “Routing and Addressing Problems in Large Metropolitan-scale Internetworks,” *Technical Report ISI/RR-87-180*, USC/ISI, March 1987.

[14] T. C. Hou and V.O.K. Li, “Transmission Range Control in Multihop Packet Radio Networks,” *IEEE Transactions on Communications*, Vol. 34, no. 1, pp. 38–44, 1986.

[15] H. Takagi and L. Kleinrock, “Optimal Transmission Ranges for Randomly Distributed Packet Radio Terminals,” *IEEE Transactions on Communications*, Vol. 32, no. 3, pp. 246–257, 1984.

[16] L. Blazevic, S. Giordano, and J. Y. Le Boudec, “Self organized terminode routing,” *Journal of Cluster Computing*, vol. 5, no. 2, April 2002.

[17] M. Sun, X. Ma, J. Liu and X. Liu, “A Greedy Smart Path Pruning Strategy for Geographical Routing in Wireless Networks,” *in Proceedings of IEEE MILCOM 2005*, October 2005

[18] E. Royer and C.K. Toh, “A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks,” *IEEE Personal Communications*, VOL. 7, no. 4, pp. 46-55, April 1999

APPENDICES

APPENDIX A

grns.c

```
/************************************/
/* grns.c                           */
/* behaves as a node in the network */
/************************************/

#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <strings.h>
#include <arpa/inet.h>
#include <math.h>


#define NODENUMBER 50
#define RANGE 200
#define PI 3.14159265

struct grns_packet
{
        int seq;                        // sequence number of each GRP packet
        char s_id[10];                  // source node
        char d_id[10];                  // destination node
        int s_x_axis;                   // source node's x position
        int s_y_axis;                   // source node's y position
        int d_x_axis;                   // destination node's x position
        int d_y_axis;                   // destination node's y position
        int num_hops;                   // number of hops
        char data[100];                         // carrying data
};
```

```
struct routingnode
{
	char id[10];				// id of the routing node
	char x_axis[5];				// routing node's x position
	char y_axis[5];				// routing node's y position
	char ip[15];				// routing node's ip
	char port[5];				// routing node's port#
	int sector[16];
	struct routingnode * next;	// next routingnode structure
};

struct allnodes
{
	char id[10];				// id of the routing node
	char x_axis[5];				// routing node's x position
	char y_axis[5];				// routing node's y position
	char ip[15];				// routing node's ip
	char port[5];				// routing node's port#
};

struct allnodes nodes[NODENUMBER];
int myX, myY, myPort;
char myIP[15];
char myName[10];

int sectoring( double ratio, int cur_x, int cur_y, int neighbor_x, int neighbor_y );
struct routingnode * read_file( FILE * config_file );
struct routingnode * search_next( struct routingnode *h, struct grns_packet *g );

// return the sectoring information
int sectoring( double ratio, int cur_x, int cur_y, int neighbor_x, int neighbor_y )
{
	int sector_number;
	sector_number = 0;

	if ( neighbor_x >= cur_x && neighbor_y >= cur_y )
	{
		if ( ( ratio >= sin(0) ) && ( ratio < sin( PI / 8.0 ) ) )
		{
			sector_number = 0;
		}
		else if ( ( ratio >= sin( PI / 8.0 ) ) && ( ratio < sin( PI / 4.0 ) ) )
		{
			sector_number = 1;
		}
```

```
        else if ( ( ratio >= sin( PI / 4.0 ) ) && ( ratio < sin( PI * 3.0 / 8.0 ) ) )
        {
                sector_number = 2;
        }
        else
        {
                sector_number = 3;
        }
}

else if ( neighbor_x < cur_x && neighbor_y >= cur_y )
{
        if ( ( ratio >= sin( PI * 3.0 / 8.0 ) ) && ( ratio < sin( PI / 2.0 ) ) )
        {
                sector_number = 4;
        }
        else if ( ( ratio >= sin( PI / 4.0 ) ) && ( ratio < sin( PI * 3.0 / 8.0 ) ) )
        {
                sector_number = 5;
        }
        else if ( ( ratio >= sin( PI / 8.0 ) ) && ( ratio < sin( PI / 4.0 ) ) )
        {
                sector_number = 6;
        }
        else
        {
                sector_number = 7;
        }
}

else if ( neighbor_x < cur_x && neighbor_y < cur_y )
{
        if ( ( ratio >= sin(0) ) && ( ratio < sin( PI / 8.0 ) ) )
        {
                sector_number = 8;
        }
        else if ( ( ratio >= sin( PI / 8.0 ) ) && ( ratio < sin( PI / 4.0 ) ) )
        {
                sector_number = 9;
        }
        else if ( ( ratio >= sin( PI / 4.0 ) ) && ( ratio < sin( PI * 3 / 8.0 ) ) )
        {
                sector_number = 10;
        }
        else
```

```
                {
                        sector_number = 11;
                }
        }

        else
        {
                if ( ( ratio >= sin( PI * 3.0 / 8.0 ) ) && ( ratio < sin( PI / 2.0 ) ) )
                {
                        sector_number = 12;
                }
                else if ( ( ratio >= sin( PI / 4.0 ) ) && ( ratio < sin( PI * 3.0 / 8.0 ) ) )
                {
                        sector_number = 13;
                }
                else if ( ( ratio >= sin( PI / 8.0 ) ) && ( ratio < sin( PI / 4.0 ) ) )
                {
                        sector_number = 14;
                }
                else
                {
                        sector_number = 15;
                }
        }

        return sector_number;
}

struct routingnode * read_file( FILE * config_file )
{
        /* in order to establish the linked list of routingnodes,
        establish first the pointers to the linked list */

        struct routingnode *p, *h, *s;
        int i, j;
        int x1, y1, x2, y2;
        int sector_number = 0;
        double distance = 0;
        double distance1 = 0;
        double distance2 = 0;
        double ratio = 0;

        if ( ( h = ( struct routingnode * )malloc( sizeof( struct routingnode ) ) ) ==
NULL )
        {
```

```
            printf( "memery allocation failure\n" );
            exit( 0 );
        }
        h->id[0] = '\0';
        h->next = NULL;

        /* h is the poniter to the header of the linked list. */
        /* establish the linked list to store routing information*/

        p = h;
        for ( i = 0; i < NODENUMBER; i++ )
        {
            if ( !feof(config_file) )
            {
                fscanf( config_file, "%s", nodes[i].id );
                fscanf( config_file, "%s", nodes[i].x_axis );
                fscanf( config_file, "%s", nodes[i].y_axis);
                fscanf( config_file, "%s", nodes[i].ip );
                fscanf( config_file, "%s", nodes[i].port );
                if ( strcmp( myName, nodes[i].id ) == 0 )
                {
                    myX = atoi( nodes[i].x_axis );
                    myY = atoi( nodes[i].y_axis );
                    myPort = atoi( nodes[i].port );
                    strcpy( myIP, nodes[i].ip );
                }
            }
            else
            {
                nodes[i].id[0] = '\0';
                nodes[i].x_axis[0] = '\0';
                nodes[i].y_axis[0] = '\0';
                nodes[i].ip[0] = '\0';
                nodes[i].port[0] = '\0';
            }
        }

        for ( i = 0; i < NODENUMBER; i++ )
        {
            if ( ( s = ( struct routingnode * )malloc( sizeof( struct routingnode ) ) ) ==
NULL )
            {
                printf( "memery allocation failure\n" );
                exit(0);
            }
```

```
                if ( strcmp( myName, nodes[i].id ) != 0 )
                {
                        x1 = atoi( nodes[i].x_axis );
                        y1 = atoi( nodes[i].y_axis );
                        distance = sqrt( ( double ) ( ( x1 - myX ) * ( x1 - myX ) + ( y1 -
myY ) * ( y1 - myY ) ) );

                        if ( distance <= RANGE )
                        {
                                p->next = s;
                                strcpy( s->id, nodes[i].id );
                                strcpy( s->x_axis, nodes[i].x_axis );
                                strcpy( s->y_axis, nodes[i].y_axis );
                                strcpy( s->ip, nodes[i].ip );
                                strcpy( s->port, nodes[i].port );

                                for ( j = 0; j < 16; j++ )
                                {
                                        s->sector[j] = 0;
                                }

                                for ( j = 0; j < NODENUMBER; j++ )
                                {
                                        if ( strcmp( s->id, nodes[j].id ) != 0 )
                                        {
                                                x2 = atoi( nodes[j].x_axis );
                                                y2 = atoi( nodes[j].y_axis );
                                                distance1 = sqrt( ( double ) ( ( x2 - x1 ) *
( x2 - x1 ) + ( y2 - y1 ) * ( y2 - y1 ) ) );
                                                if ( distance1 <= RANGE )
                                                {
                                                        ratio = fabs( ( double )( y2 - y1 ) /
distance1 );
                                                        sector_number = sectoring( ratio, x1,
y1, x2, y2 );
                                                        s->sector[sector_number]++;
                                                }
                                        }
                                }
                                s->next = NULL;
                                p = s;
                        }
                }
        }
```

```
	return h;
}

/* return the node information pointer for the next hop  */
struct routingnode * search_next( struct routingnode *h, struct grns_packet *g )
{
	struct routingnode *p, *k;
	/* p is the pointer to the current node to be compared while
	k is the pointer to the node nearest to the destination node */

	int num_neighbor = 0;
	int flag = 0;
	int direction = 0;
	double distance = 0.0;
	double distance1 = 0.0;
	double temp = 0.0;
	double factor = 0.0;
	double ratio = 0.0;
	char * nexthop = "dead end";

	/*temp1 is the shorest distance so far and temp2 is the
	 distance between current node and destination node */
	p = h->next;
	distance = sqrt( ( double ) ( ( g->d_x_axis - myX ) * ( g->d_x_axis - myX ) ) +
( g->d_y_axis - myY ) * ( g->d_y_axis - myY ) );

	k = NULL;
	while ( p != NULL )
	{
		if(strcmp(g->d_id, p->id) == 0)
		{
			printf("next is destination");
			nexthop = p->id;
			k = p;
			flag = 1;
			break;
		}

		p = p->next;
	}

	p = h->next;

	while ( p != NULL && flag == 0)
	{
```

```
		num_neighbor = 0;
		distance1 = sqrt( ( double ) ( ( g->d_x_axis - atoi( p->x_axis ) ) * ( g->d_x_axis - atoi( p->x_axis ) ) ) + ( g->d_y_axis - atoi( p->y_axis ) ) * ( g->d_y_axis - atoi( p->y_axis ) ) );

		if(distance1 <= RANGE)
		{
			printf("last - 1 node");
			nexthop = p->id;
			k = p;
			break;
		}
		else if(distance1 < distance && distance1 > RANGE)
		{
			ratio = fabs( ( double )( g->d_y_axis - atoi( p->y_axis ) ) / distance1 );
			direction = sectoring( ratio, atoi( p->x_axis ), atoi( p->y_axis ), g->d_x_axis, g->d_y_axis );

			switch ( direction )
			{
			case 0:
				num_neighbor = p->sector[0] + p->sector[1] + p->sector[2] + p->sector[3]
							+ p->sector[13] + p->sector[14] + p->sector[15];
				break;
			case 1:
				num_neighbor = p->sector[0] + p->sector[1] + p->sector[2] + p->sector[3]
							+ p->sector[4] + p->sector[14] + p->sector[15];
				break;
			case 2:
				num_neighbor = p->sector[0] + p->sector[1] + p->sector[2] + p->sector[3]
							+ p->sector[4] + p->sector[5] + p->sector[15];
				break;
			case 3:
				num_neighbor = p->sector[0] + p->sector[1] + p->sector[2] + p->sector[3]
							+ p->sector[4] + p->sector[5] + p->sector[6];
				break;
```

```
                        case 4:
                                num_neighbor = p->sector[7] + p->sector[1] + p->sector[2]
+ p->sector[3]
                                                + p->sector[4] + p->sector[5]
+ p->sector[6];
                                break;
                        case 5:
                                num_neighbor = p->sector[7] + p->sector[8] + p->sector[2]
+ p->sector[3]
                                                + p->sector[4] + p->sector[5]
+ p->sector[6];
                                break;
                        case 6:
                                num_neighbor = p->sector[7] + p->sector[8] + p->sector[9]
+ p->sector[3]
                                                + p->sector[4] + p->sector[5]
+ p->sector[6];
                                break;
                        case 7:
                                num_neighbor = p->sector[7] + p->sector[8] + p->sector[9]
+ p->sector[10]
                                                + p->sector[4] + p->sector[5]
+ p->sector[6];
                                break;
                        case 8:
                                num_neighbor = p->sector[7] + p->sector[8] + p->sector[9]
+ p->sector[10]
                                                + p->sector[11] + p-
>sector[5] + p->sector[6];
                                break;
                        case 9:
                                num_neighbor = p->sector[7] + p->sector[8] + p->sector[9]
+ p->sector[10]
                                                + p->sector[11] + p-
>sector[12] + p->sector[6];
                                break;
                        case 10:
                                num_neighbor = p->sector[7] + p->sector[8] + p->sector[9]
+ p->sector[10]
                                                + p->sector[11] + p-
>sector[12] + p->sector[13];
                                break;
                        case 11:
                                num_neighbor = p->sector[14] + p->sector[8] + p-
>sector[9] + p->sector[10]
```

```
                                                        + p->sector[11] + p-
>sector[12] + p->sector[13];
                                        break;
                                case 12:
                                        num_neighbor = p->sector[14] + p->sector[15] + p-
>sector[9] + p->sector[10]
                                                        + p->sector[11] + p-
>sector[12] + p->sector[13];
                                        break;
                                case 13:
                                        num_neighbor = p->sector[14] + p->sector[15] + p-
>sector[0] + p->sector[10]
                                                        + p->sector[11] + p-
>sector[12] + p->sector[13];
                                        break;
                                case 14:
                                        num_neighbor = p->sector[14] + p->sector[15] + p-
>sector[0] + p->sector[1]
                                                        + p->sector[11] + p-
>sector[12] + p->sector[13];
                                        break;
                                case 15:
                                        num_neighbor = p->sector[14] + p->sector[15] + p-
>sector[0] + p->sector[1]
                                                        + p->sector[2] + p-
>sector[12] + p->sector[13];
                                        break;
                                default:
                                        num_neighbor = 0;
                                        break;
                                }

                                factor = ( double )num_neighbor / ( distance1 / distance );

                                if ( temp < factor )
                                {
                                        nexthop = p->id;
                                        temp = factor;
                                        k = p;
                                }
                        }

                        p = p->next;
                }
                /* nexthop is the next node to send packet*/
```

```
        printf( "the next node is %s\n", nexthop );

        return k;
}

FILE * file_pointer;

main( int argc, char *argv[] )
{
        char *id = argv[1];
        strcpy( myName, id );
        char * sNode = argv[2];
        char * dNode = argv[3];
        int i;

        struct routingnode * head, * cur;
        file_pointer = fopen( argv[4], "r" );
        head = read_file( file_pointer );
        cur = head;

        if ( strcmp( myName, sNode ) == 0 )
        {
                struct grns_packet *grp;
                grp = ( struct grns_packet * )malloc( sizeof( struct grns_packet ) );
                grp->seq = 0;
                strcpy( grp->s_id, sNode );
                strcpy( grp->d_id, dNode );
                for ( i = 0; i < NODENUMBER; i++ )
                {
                        if ( strcmp( dNode, nodes[i].id ) == 0 )
                        {
                                break;
                        }
                }
                grp->s_x_axis = myX;
                grp->s_y_axis = myY;
                grp->d_x_axis = atoi( nodes[i].x_axis );
                grp->d_y_axis = atoi( nodes[i].y_axis );
                grp->num_hops = 0;

                strcpy( grp->data, "grns data" );


                /* establish socket*/
                int fd;
```

```
                int address_len;
                struct sockaddr_in address;

                fd = socket( AF_INET, SOCK_DGRAM, 0 );
                bzero( &address, sizeof( address ) );
                address.sin_family = AF_INET;
                address.sin_addr.s_addr = inet_addr( myIP );
                address.sin_port = htons( ( unsigned short ) myPort );
                address_len = sizeof( address );
                bind( fd, ( struct sockaddr * )&address, address_len );

                struct sockaddr_in next_address;
                socklen_t len = sizeof( struct sockaddr_in );
                int n;

                struct routingnode *s;

                /* resolve the grns packet and get the rounting node for the next hop */
                FILE *fp;

                if ((fp = fopen("route_grpn.txt", "wb+"))==NULL)
                {
                        fprintf(stderr, "Error opening file.\n");
                        exit(1);
                }
                fprintf(fp, "source: %s, Destination: %s\n", sNode, dNode);

                fclose( fp );

                s = search_next( head, grp );

                if ( s != NULL )
                {
                        /* send the grns packet to the next hop node */
                        bzero( &next_address, sizeof( next_address ) );
                        next_address.sin_family = AF_INET;
                        next_address.sin_addr.s_addr = inet_addr( s->ip );
                        next_address.sin_port = htons( ( unsigned short )atoi( s->port ) );

                        sendto( fd, grp, sizeof( struct grns_packet ), 0, ( struct sockaddr
* )&next_address, len );

                        printf( "sending packet using grpn\n" );
                }
        }
```

```
	else
	{
		/* establish socket*/

		int fd;
		int address_len;
		struct sockaddr_in address;

		fd = socket( AF_INET, SOCK_DGRAM, 0 );
		bzero( &address, sizeof( address ) );
		address.sin_family = AF_INET;
		address.sin_addr.s_addr = inet_addr( myIP );
		address.sin_port = htons( ( unsigned short ) myPort );
		address_len = sizeof( address );
		bind( fd, ( struct sockaddr * )&address, address_len );

		while (1)
		{
			struct sockaddr_in last_address;
			struct sockaddr_in next_address;
			socklen_t len = sizeof( struct sockaddr_in );
			char buf[256];
			struct grns_packet *p;
			struct routingnode *s;
			p = ( struct grns_packet * )buf;

			/* receive UDP packet*/
			int n;
			n = recvfrom( fd, p, 256, 0, ( struct sockaddr * )&last_address,
&len );

			/* resolve the received packet as grp packet and get the rounting
node for the next hop */

			FILE *fp;
			if ((fp = fopen("route_grpn.txt", "a+"))==NULL)
			{
				fprintf(stderr, "Error opening file.\n");
				exit(1);
			}

			p->num_hops++;

			fprintf(fp, "%s - ", myName);
```

```
                        fclose( fp );

                        s = search_next( head, p );

                        if ( s == NULL )
                        {
                                continue;
                        }

                        if ( strcmp( p->d_id, myName ) == 0 )
                        {
                                /* the current node is the destination node */
                                printf(" Received grpn packet from %s\n", p->s_id);
                        }
                        else /* the current node is not the destination node */
                        {
                                /* send the grns packet to the next hop node */
                                bzero( &next_address, sizeof( next_address ) );
                                next_address.sin_family = AF_INET;
                                next_address.sin_addr.s_addr = inet_addr( s->ip );
                                next_address.sin_port = htons( ( unsigned short )atoi( s-
>port ) );

                                sendto(fd, p, n, 0, ( struct sockaddr * )&next_address, len);
                        }
                }
        }
}
```

APPENDIX B

nodegenerator.c

```
/*********************************/
/* nodegenerator.c                */
/* generate nodes in the network  */
/*********************************/

#include <stdio.h>
#include <stdlib.h>

#define XSIZE 1000
#define YSIZE 1000

int main(int argc, char *argv[])
{
        int node = 0;
        int numNode = 70;
        int port = 5000;
        int i, j;
        double r = 0;
        long int x, y;
        char *ip = "131.204.14.184";
        srand(time(NULL));

        FILE *fp1;

        if ((fp1=fopen("node.txt", "wb+"))==NULL)
        {
                fprintf(stderr, "Error opening file.\n");
                exit(1);
        }

for ( i = 1; i <= numNode; i++ )
        {
                r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
                x = (int)(r * (double)XSIZE);
```

```
            r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)));
            y = (int)(r * (double)YSIZE);
            fprintf(fp1, "node%d %ld %ld %s %d\n", i, x, y, ip, port + 1 + i);
        }

        fclose(fp1);

        return 0;
}
```