1-1-2007

Cluster-based route discovery protocol

Shashirekha Yellenki
University of Nevada, Las Vegas

CLUSTER-BASED ROUTE DISCOVERY PROTOCOL

by

Shashirekha Yellenki

A thesis submitted in partial fulfillment
Of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howards R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2007

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

JANUARY 31ST , 2007

The Thesis prepared by

SHASHIREKHA YELLENKI

Entitled

CLUSTER - BASED ROUTE DISCOVERY PROTOCOL

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

Cluster-Based Route Discovery Protocol

by

Shashirekha Yellenki

Dr. Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

An ad hoc network is a collection of wireless mobile hosts forming a network without the aid of any established infrastructure or centralized administration. In such an environment, it may be necessary for one mobile host to enlist the aid of other hosts in forwarding a packet to its destination due to the limited range of each mobile host's wireless transmissions. Many protocols have been proposed to route packets between the hosts in such a network.

The on-demand routing protocol is a well-known method. It establishes the routes and uses them only when a need arises. For wireless communication channels, the problem is further complicated by the mobility of the nodes, which induces structural changes in the routing. So, the mobility management of mobile nodes is important in mobile ad hoc networks.

Clustering is a scheme to build a network control structure that increases network availability, reduces the delay in responding to changes in network state, and improves data security. It promotes more efficient use of resources in controlling large dynamic

networks. Clustering is crucial for scalability as the performance can be improved by simply adding more nodes to the cluster.

This thesis presents a protocol for routing in ad hoc networks that uses ad-hoc on-demand routing and also takes care of the mobility management. The protocol adapts quickly to frequent host movement, yet requires little or no overhead during periods in which hosts move less frequently. Moreover, the protocol routes packets through a dynamically established and nearly optimal path between two wireless nodes. We propose a self-organizing clustering protocol to store the routing data in multiple nodes and to distribute the routing load. It also achieves higher reliability --- if a node in a cluster fails, the data is still accessible via other cluster nodes.

ACKNOWLEDGEMENTS

It is a pleasure to thank the many people who made this thesis possible. It is difficult to express my gratitude towards my thesis advisor, Dr. Ajoy K. Datta. It is only his enthusiasm, inspiration, and explaining things clearly and simply, that made working on this thesis fun and interesting for me. I would have been lost without him. I would like to than Dr. Doina Bein for her tremendous help, support, encouragement, and patience in working with and guiding me throughout this project. I would also like to thank Dr. Yoohwan Kim, Dr. John Minor, and Dr. Venkatesan Muthukumar for agreeing to be members of my committee, for spending their valuable time in reviewing my thesis.

Lastly, and most importantly, I wish to thank my parents. They raised me, supported me, taught me, loved me, and stood behind me all the time. This thesis is dedicated to them.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

A mobile ad-hoc network (MANET) is a self-configuring network of mobile hosts connected by wireless links, the union of which forms an arbitrary topology. The routers are free to move randomly and organize themselves arbitrarily. Thus, the network's wireless topology may change rapidly and unpredictably. This network transmits from computer to computer without the use of a central base station (access point). Such a network may operate in a stand-alone fashion, or may be connected to the larger Internet.

Minimal configuration and quick deployment make ad hoc networks suitable for emergency situations like natural or human-induced disasters, military conflicts, emergency medical situations, etc. The earliest MANETs were called "packet radio" networks, and were sponsored by DARPA in the early 1970s. It is interesting to note that these early packet radio systems predated the Internet, and indeed were part of the motivation of the original Internet Protocol suite.

In spite of the various applications served by the ad-hoc networks, they still have to overcome the defects such as the limited wireless transmission range, interference caused due to its broadcast nature, route changes and packet losses induced due to the node mobility, battery constraints, and potentially frequent network partitions. A major challenge faced in MANET's is locating the devices for communication, especially with high node mobility and sparse node density. Present solutions provided by the ad hoc

routing protocols range from flooding [10] the entire network with route requests, to deploying a separate location management scheme [13] to maintain a device location database. Nodes make use of the real life concept of making acquaintances and keeping in touch with them regarding each other's current locations.

1.1 Contributions

In this thesis, we design a mobility management based Cluster routing leader election algorithm for MANET. Every node starts as a clusterhead. Eventually, a set of nodes is chosen as the clusterheads. These special nodes maintain the routing tables with shortest paths for intra-cluster and inter-cluster routing [5, 16]. We use the concept of mobility management and on demand routing scheme [6, 11] to design a link–cluster routing protocol. Routing tables can be used to locate the *destination* while communicating in ad hoc networks. Such protocols limit the search for a route to only when the need arises, thus reducing the overhead of unnecessary data storage. We follow an alternate clusterhead gateway path to quickly find a route.

1.2 Outline of the Thesis

In Chapter 2, we present an overview of the ad-hoc routing algorithms, clustering schemes for routing efficiency, various location-management schemes and end it with a brief description of the link-cluster architecture. Chapter 3 includes the data structures used by the proposed algorithm. The main three components of the algorithm along with their proof of correctness are presented in three subsequent chapters. Finally, the thesis ends with the concluding remarks and suggestions for future research in Chapter 7.

CHAPTER 2

AD HOC NETWORK ROUTING PROTOCOLS

In a MANET, hosts keep moving, causing frequent network topology changes. Therefore, the task of finding and maintaining routes is nontrivial. Routing protocols for ad hoc networks are divided into two classes:

Proactive: Continuously updates reachability information in the network so that when a route is needed, it is immediately available [15]. *Examples:* DSDV and OLSR.

Reactive: Route discovery is initiated only when needed, and route maintenance is needed to provide information about invalid routes [12, 15]. *Examples:* DSR and AODV.

The conventional routing protocols are insufficient for ad hoc networks, since the amount of routing related traffic may waste a large portion of the wireless bandwidth. A few demand-driven route-establishing protocols like DSR and AODV have been proposed. Some zone routing protocols like ZRP and Safari have been proposed that initiate the route discovery phase on demand, but limit the scope of proactive procedure only to the initiator's local neighborhood or the receiver's neighborhood. The Location aided routing protocols [13] use location information (obtained using the GPS) to reduce the search space, resulting in fewer route discovery messages for a desired route.

In our algorithm, we consider a network with link-cluster architecture and discover an optimal route for the nodes to communicate with each other [5]. We use the concept of proactive protocols to route the packets within the cluster and the concept of reactive

protocols to route the packets between the clusters. Such combination of proactive and reactive protocols used for routing the packets is called a hybrid protocol [15]. We also use the concept of location management when a node leaves a cluster to update the routing tables [16]. We now give a brief description of all those concepts used in our algorithm.

2.1 Clustering

The events that affect the structure of the network as well as the controls applied in response to such events cause changes in the network state. The task of controllers is to detect and respond to such changes by sensing and collecting the local state information and distributing it to other controllers in the network. The changes in the network state are more frequent in the *mobile networks*, where the node movements affect both node interconnectivity and link quality and the *wireless networks*, where the links are limited and highly volatile. Moreover, small changes in the environment may result in large changes in radio signal propagation, causing them to experience path loss, fading, loss of wireless transmissions, and interference, thus constraining the available capacity of the wireless links.

Controllers consume storage, transmission, and processing resources whenever they perform certain tasks. They need not respond to all the changes taking place in the network which may be trivial. In a highly dynamic network, the response delay of the controllers may be greater than the time between the state changes taking place. Hence, the sensitivity for a network controller depends on the particular control function to be

performed, the resources available, the volatility of network state, and the anticipated magnitude and extent of the consequences of a state change.

The *cluster-based control structures* [5, 8] can significantly reduce the overhead costs imposed by routing without unduly sacrificing the quality of the routes produced. In ad hoc networks, cluster-based control structures contribute to improved efficiency of resource use by managing wireless transmissions among multiple nodes to reduce channel contention, forming routing backbones to reduce network diameter, and abstracting network state information to reduce its quantity and variability.

2.2 Link-Cluster Architecture

Link-cluster architecture [1, 2, 7] is a network control structure in which nodes are partitioned into clusters that are interconnected. The union of the members of all the clusters covers all the nodes in the network. In every cluster, nodes are classified in three ways: clusterhead, gateway, and ordinary node. A clusterhead schedules the transmissions and allocates resources within clusters. *Gateways* connect adjacent clusters. An *ordinary node* belongs to a single cluster (has a unique clusterhead).

Clusters are of two types: overlapping and disjoint. *Overlapping:* If a gateway node is a member of both clusters, then such clusters are termed as the overlapping clusters. *Disjoint:* If a gateway node is a member of exactly one cluster and forms a link to a member of another cluster, then such clusters are termed as the disjoint clusters. In this research, we will consider only the disjoint clusters. In the following sections, we will describe the clusterheads and gateway nodes in more detail. We will also briefly present the node mobility and routing ideas.

2.2.1 Clusterheads

Each cluster has exactly one clusterhead. A clusterhead schedules the transmissions and allocates resources within clusters. Discussed below are two clusterhead election algorithms.

Identifier-based Clustering Algorithm: The identifier-based clustering algorithm [6] makes use of the concept of a *unique identifier* that differentiates every single node in the network from the other. The node with the highest or lowest identifier becomes the clusterhead [4]. *Connectivity-based Clustering Algorithm:* The connectivity-based clustering algorithm makes use of the number of neighbors a node has. The node with the highest connectivity is chosen as the clusterhead. If two nodes have the same connectivity, the identifiers can be used to resolve the conflict.

2.2.2 Gateway Nodes

Gateways connect adjacent clusters. Conferring gateway status to all the members ensures connectivity between individual gateways. Two types of clusters are formed based on whether a single gateway or a gateway pair connects the two clusters. They are overlapping clusters and disjoint clusters. *Overlapping clusters:* If a node has two clusterheads at one hop distance, then that node becomes the gateway and is said to connect two overlapping clusters. Here, the gateway is a node with the highest or lowest identifier. Thus overlapping clusters have a single gateway connecting them. *Disjoint clusters:* If a clusterhead in one cluster is a neighbor of a node and can reach the other clusterhead in any other cluster in two *hops*, then it is a candidate gateway linked to candidate gateway in another cluster. The two gateways selected are linked pair in which

one member has one highest or lowest identifier among all candidates connecting two clusters. Thus, disjoint clusters are formed with a gateway-pair connecting them.

2.2.3 Node Mobility

In the presence of mobile nodes, a clusterhead needs to update the cluster membership, and clusterhead and gateway information. A node's clusterhead is likely to change more frequently with connectivity-based clustering than with identifier-based clustering since the connectivity gets affected.

The identifier-based clustering algorithm reduces the number of changes in clusterhead status required after node movement. The change in clusterhead occurs only if two clusterheads move within the range of each other, where one of them relinquishes its role, or if an ordinary node moves out of range of all other nodes, in which case it becomes the clusterhead of its own cluster. Cluster maintenance schemes are designed to minimize the number of changes in the set of existing clusters. They do not re-cluster after every movement, but instead make small adjustment to cluster membership as necessary, as in only when the most highly connected node in a cluster moves.

2.3 Routing within a Cluster

The algorithm uses a simple link-state routing protocol that uses distance or hop count as its primary metric for determining the best forwarding path within a cluster. The clusterhead makes a list of nodes it can reach, and the number of *hops* it will cost. This table is called a *routing table* [16]. The nodes within the cluster routinely send the clusterhead messages to enquire if their clusterhead still is active or not. The

clusterheads regularly send messages to the nodes within its two hop neighborhood to enquire if they still belong to their cluster and to keep the routing tables up-to-date.

Bad routing paths are purged from the routing table. A routing path becomes bad when the route no longer exists or when the nodes move. If two identical paths to the same network exist, only the one with the smallest hop-count is kept. Thus, the updated table is always used for forwarding the messages.

This protocol uses Dijkstra's Shortest Path First algorithm to construct a list of nodes describing the network that represents the minimum delay paths. This list is used in creating the routing directory consisting of information about *destination* node and the next hop node. This directory is in turn used for forwarding the packets. In short, this protocol responds quickly and correctly to changes in network topologies, is capable of detecting and routing packets, routes traffic on minimum hop paths, and loops do not exist in the network [12].

2.4 Routing between the Clusters

Our route discovery algorithm makes use of a protocol that creates routes on an on-demand basis while routing between the clusters. Such protocols are called reactive protocols. Traditional *proactive* protocols find routes between all source-destination pairs regardless of the use or need for such routes. The key motivation behind the design of on-demand protocols is the reduction of the routing load.

Our algorithm mainly uses the AODV protocol for inter cluster routing. AODV uses a table-driven routing framework and *destination* sequence numbers. To maintain routing information, AODV uses traditional routing tables, one per *destination* and relies

on these tables for routing rather than on source routing. All routing packets carry the sequence numbers to maintain freshness of routing information and to prevent routing loops [12]. A routing table entry is expired if not used recently.

AODV uses an expanding ring search initially to discover routes to an unknown *destination.* If the route to a previously known *destination* is needed, the hop-wise distance is used for the search. Route discovery in AODV is based on query and reply cycles. AODV relies on route discovery flood more often [10], which may carry significant, network overhead. The *destination* replies only once to the request arriving first and the routing table maintains at most one entry per *destination.* In AODV always fresher routes are considered and the unused route entries are deleted after an expiry time.

2.5 Location Management

As wireless devices become more capable, location will play a key role in the services offered to the nodes that want to communicate with each other. Location management [13] forms an essential entity in protocols that use geographic routing. The nodes periodically select nodes that take on the role of a location server of their current location. All the gateway nodes and the clusterhead node which are present in the cluster region C_u of the clusterhead node u act as location servers for all the nodes in the cluster region C_u. When a node moves across two clusterhead regions, the node updates its home region C_u of the movement by a location update or by sending a leave message.

Discovery of a node's location: A source node x from outside the cluster that wishes to communicate with a node y in the cluster region C can now use the clusterhead and gateway tables to identify the location of the node y and send a location query packet

towards region C to obtain the current location of y. The first location server to receive the query for u responds with the current location of y to which data packets are routed.

CHAPTER 3

CLUSTER-BASED ROUTE DISCOVERY ALGORITHM

The proposed protocol consists of three main steps: Clusterhead Election, Gateway Election, and Route discovery that are implemented in three different modules. We will use asynchronous message passing systems. The algorithm uses cluster-based network and the concept of location management [13] to implement an efficient routing mechanism. In this chapter, we describe the data structures and assumptions used in our algorithm.

3.1 Model

We use a conventional message passing model of communication. Assume that some node x wants to send a message to node y. The message follows a route that is a sequence of communication links in the network (abstracted as a simple path). A routing algorithm specifies the route by directing each intermediate node on the route which outgoing edge the message should be sent depending on the destination. We assume that the network has an error correcting protocol in place that takes care of necessary re-transmissions in case of message losses or corruptions.

3.2 Data Structures

In a cluster-based network, the network is divided into clusters. In every cluster, nodes are divided into three categories: clusterhead, gateway and ordinary node.

Definition 3.1 Routing Table: This table is maintained in every clusterhead and gateway node. It keeps track of routes (and in some cases, metrics associated with those routes) to different destinations.

Definition 3.2 Clusterhead (CH): A clusterhead schedules the transmissions and allocates resources within clusters.

Definition 3.3 Gateway: Any node with links to more than one cluster is a candidate for a gateway node connecting these clusters [5]. We will describe the conditions to be satisfied by these candidates to become gateway nodes.

A gateway node that belongs to the inter-cluster routing table of the clusterhead is called a bordering gateway node.

3.2.1 Variables

The algorithm uses a variable N_i^1 representing the one-hop neighborhood set of node i and a variable N_i^2 representing the two-hop neighborhood set of node i. These two sets are maintained by an underlying local topology maintenance protocol that adjusts its value in case of topological changes in the network due to failures of nodes or links. The variable nb is used to identify the neighbor of the current node from which it received a message. The variable HighestIndex always points to the last row of the routing tables. Node i has a unique ID, ID.i. The variables path and newpath represent a list of links that is traversed by messages. For Example, if a path has a list of nodes A, B, and C, there are links from A to B, and B to C that have been traversed by a message. Similarly, when a

node ID is added to the path or newpath variable, there is a link from the path to the added node ID. Taking the previous example, if path = path + D is written, it means that there are links from A to B, B to C, and C to D.

Every node has four variables (c.i, d.i, n.i, and g.i) to maintain the status. The variable c.i has the ID of the clusterhead of node i, d.i holds an integer value representing the distance from node i to its clusterhead, n.i has the ID of the neighbor of the node i along the shortest path towards its clusterhead, and g.i is a boolean value that is T (true) if node i is a gateway node or F (false) otherwise.

For a clusterhead, c.i = ID.i, d.i = 0, n.i = nil, and g.i = T or F depending on whether it is a gateway or not.

For a gateway node, c.i = Single ID / array of IDs of its clusterhead, d.i = Distance / array of distances from its clusterhead, n.i = Next hop / array of next hop neighbors on shortest path to its clusterhead and g.i = T.

3.2.2 Tables

Every node in a network has a sequence table that keeps track of the messages already received by the node and makes the routing messages loop-free [3, 12]. Only gateways and clusterheads maintain the tables used for routing [5]. The clusterhead routing table contains entries for the nodes in its cluster (or clusterhood). A clusterhead has another table that is used to route messages outside the cluster. This table has entries of all the *destination* and boundary gateway pairs. The gateway tables contain all the entries of the *destination-clusterhead* pairs of all the clusters they connect to. The routing table is updated whenever a new clusterhead is elected or some changes occur related to paths in the routing table. The ordinary nodes have no routing tables. The only

routing information they have is a variable indicating the neighbor on the shortest path towards their clusterhead.

The following is the detailed description of the tables held by different nodes:

A Clusterhead has 3 tables:

1. *Routing table.*
2. *CG_TABLE.*
3. *SEQ_TABLE.*

ROUTING TABLE

Dest	CH	Path from CH to dest	Next-hop	# hops	g.i

Routing table contains information for routing within the cluster. It has the following six columns:

Dest: The ID of the node within its own cluster.

CH: The node's own ID.

Path from CH to dest: The entire path from the Clusterhead (itself) to the node in the *Dest* field.

Next-hop: The next hop neighbor from the clusterhead to reach the *Dest* node.

#hops: The distance (in number of *hops*) from the Clusterhead to the node in the *Dest* field.

g.i: T if the *Dest* node is a gateway; F otherwise.

CG_TABLE

Index	GW	Node	Next-hop

CG_TABLE or the Clusterhead's gateway table contains the routing information for inter-cluster routing along with the bordering gateway nodes' information. It has the following four columns:

Index: A counter to keep track of the number of rows in the table.

GW: ID of the bordering gateway node that acts as the temporary *destination* in order to reach the actual *destination* in the *Dest* field.

node: ID of the node whose route has to be found and can be reached through the gateway node in that row i.e., the *GW* field in the same row.

Next-hop: Next hop neighbor from the gateway node to reach the *Dest* node.

SEQ_TABLE

Sender	Seq

SEQ_TABLE or sequence table keeps track of the messages already received and makes the routing messages loop-free. It has the following two columns:

Sender: ID of the node that initiated the message.

Seq: Sequence number of the message sent.

A gateway has two tables:

1. *GC_TABLE.*
2. *SEQ_TABLE.*

GC_TABLE

Index	CH	Node	Next-hop

GC_TABLE or Gateway's clusterhead table contains route information for inter cluster routing with the bordering clusterheads' information. It has the following three different columns when compared to the *CG_TABLE*:

CH: ID of the bordering clusterhead node that acts as the temporary *destination* in order to reach the actual *destination* in the *Dest* field.

node: ID of the node whose route has to be found and can be reached through the clusterhead node in that row i.e., the *CH* field in the same row.

Next-hop: Next hop neighbor from the clusterhead node to reach the *Dest* node.

SEQ_TABLE

Sender	Seq

SEQ_TABLE or sequence table keeps track of the messages already received and makes the routing messages loop-free. It is similar to the *SEQ_TABLE* of the clusterhead node. It has the same two columns.

An ordinary node maintains only one table.

1. SEQ_TABLE.

SEQ_TABLE

Sender	Seq

SEQ_TABLE or sequence table keeps track of the messages already received and makes the routing messages loop-free. It is similar to the *SEQ_TABLE* of the clusterhead node as well as the gateway node. It has the same two columns.

3.3 Assumptions

The following assumptions have been made to design the proposed algorithm:

Assumption 3.1: A node knows and can distinguish its immediate neighbors.

Assumption 3.2: Every node knows its next hop neighbor on the shortest path towards its clusterhead. (Every node knows n.i).

Assumption 3.3: Initially, every node is a clusterhead of itself, i.e., ID.i = c.i for all nodes.

Assumption 3.4: Every link is bidirectional.

Assumption 3.5: Every node has a sequence table, SEQ_TABLE that makes the routing messages loop-free [3, 12].

Assumption 3.6: A node can be both a clusterhead and a gateway.

CHAPTER 4

CLUSTERHEAD ELECTION ALGORITHM

Clusterhead Election Algorithm contains the actions related to selection of clusterheads among the nodes in the cluster, and creating and/or updating entries in the routing tables in each clusterhead and regarding intra-cluster routing.

Section 4.1 explains the predicates used in the algorithm. In section 4.2, we give a brief description of the messages used for electing a clusterhead. Section 4.3 includes the detailed description of the actions performed on receiving the clusterhead election messages followed by the complete code for the proposed algorithm. The chapter ends with some proofs to support the module in section 4.5.

4.1 Predicates

Predicate *is_CH*(i) $\equiv$ ($c.i == ID.i \wedge n.i ==$ nil $\wedge\ d.i == 0$) is true if i is announced a clusterhead, the *c.i* variable has its own ID with the distance from its clusterhead (which is itself) to itself is equal to zero, and the next hop neighbor on the shortest *path* to its clusterhead is equal to nil.

Predicate *is_EG*(i) $\equiv$ ($\exists$ j $\in N_i^{\,1} \wedge c.j \neq c.i$) is true if i has at least one neighbor that belongs to a different cluster. If this predicate is true, then i is an eligible gateway node.

Predicate *is_G*(i) ≡ ((*is_EG*(i) ∧ *g.i* = = T) ∧ ¬(∃ *is_G*(j) ∈ N_i^1 ∧ *c.j* = = *c.i* ∧ *s.j* = = *s.i*)) is true if i is an eligible gateway node and has no neighboring gateways from its own cluster that connects at least the same clusters it connects. If this predicate is true, then i is a candidate for a gateway node.

Predicate *is_BG*(i) ≡ (*is_G*(j) ∧ *j* ∈ CG_TABLE(*i*)) is true if j is a gateway node and is a member of i's intra-cluster table.

Predicate *is_faulty*(i) ≡ (*c.i* = = nil ∨ *n.i* ∉ N_i^1 ∨ *d.i* < 0 ∨ *d.i* >2) returns true if there exist no clusterheads within two hop distance from i, or it has no immediate neighbors that are on the shortest *path* towards its clusterhead.

4.2 Messages

The clusterhead selection protocol must satisfy three conditions: each non-gateway node belongs to a single cluster, each non-clusterhead is within two *hops* from its clusterhead, and there are no adjacent clusterheads [9].

Messages *CL_ANN*, and *CL_REQ* contain the following fields: *sender* (*sender* ID), *dest* (*destination* ID), *path* (*path* from the *sender* to the current node) and *hops* (either the number of *hops* the message went or the number of *hops* the message went – 1).

Messages *CL_REJ*, *CL_CHG* and *leave* contain the following fields: *sender*, *dest* and *hops*.

Message *CL_ACCEPT* has the following fields: *sender*, *dest*, *path* (*path* from the *sender* to the current node), *hops*, *count* (distance in hop *count* from the *sender* to the current node), *g.i* (true or false based on whether the *dest* node is a gateway node or not).

Message *ctable_copy* contains the following fields: *dest*, *sender*, *path* (*path* from the *sender* to the *dest* node), nexthop (the next hop neighbor to reach the *dest* node), *hops*, *count* (distance in hop *count* from the *sender* to the current node), *g.i* (true or false based on whether the *dest* node is a gateway node or not).

4.3 Algorithm

A clusterhead will have $ID.i = c.i$ and $d.i = 0$ and $n.i$ = nil. If any of the variables specified have a different value, the node is not a clusterhead. A node can act as a clusterhead as well as a gateway at the same time. A clusterhead will periodically do the following: checks the consistency of each variable, Broadcasts *CL_ANN* messages to all its neighbors within its two hop distance, checks if any other clusterhead is in its range and if it finds one whose ID is bigger than itself then it gives up its clusterhead status by broadcasting *CL_REJ* messages and erases the unused rows from the CG_TABLE periodically.

An ordinary node belongs to a single cluster, i.e., has a unique clusterhead. An ordinary node periodically checks its clusterhead (alive or not) by sending a *CL_REQ* message to *n.i*. In case it has no clusterhead within its two hop distance, it sets its variables accordingly and waits for a *CL_ANN* message from a node within its two *hops* distance [9]. It becomes a clusterhead if there is no clusterhead within two *hops*.

A *CL_REQ* message travels at most two *hops* from the *sender*. Once the *CL_REQ* message reaches the right *destination* but finds that its clusterhead moved from that location, the node in that particular location or the node which was supposed to be the one hop neighbor on the shortest *path* from the *sender* to the supposed-to-be

clusterhead's location sends a *CL_CHG* message indicating that the previous clusterhead no longer exists in that location.

Action E.01 is responsible for periodically checking the clusterhead of node i. The value of time-period is dependent on the time unit of the network, and has to be at least four time units for a message to make a round-trip of two *hops*. When a node finds itself a clusterhead, it sets *n.i* to nil, *d.i* to 0, and broadcasts a *CL_ANN* message to all the nodes within two *hops*.

Whenever a node receives a message, it first checks if its own ID matches with that of the *destination* node in the message it receives. If it matches, it acts accordingly and if it does not match, forwards the message if required.

Upon receiving CL_ANN message (Action E.02): A Clusterhead drops it. A Gateway drops it. An ordinary node does the following: If the *sender* is its own Clusterhead, then it updates its variables *d.i* and *n.i*, and forwards the message. If the *sender* is not its own Clusterhead and it does not have a Clusterhead, it selects the *sender* as its own Clusterhead and sets its *c.i*, *d.i,* and *n.i* appropriately and forwards the message. If the *sender* is not his own Clusterhead and it has a Clusterhead, then it drops the message.

Upon receiving a CL_REJ message (Action E.03): A clusterhead drops it. A gateway or an ordinary node does the following: If the *sender* is its own clusterhead, then it sets *c.i* to nil, *d.i* to $+\infty$ and *n.i* to nil, and forwards the message to its immediate neighbors except the one from whom the message was received. If the *sender* is not its own Clusterhead, then it drops the message.

Upon receiving a CL_REQ message (Action E.04): A Clusterhead does the following: If the message is addressed to it, it sends a CL_ANN message to the *sender*. Otherwise,

it replies to the *sender* with a *CL_CHG* message to inform the *sender* that its clusterhead either has moved or is dead. A gateway or an ordinary node does the following: If the message is addressed to it, it drops the message. Otherwise, a gateway does the following: If the addressee is a direct neighbor, then it forwards to the neighbor. If the addressee is not a direct neighbor, then it sends the *CL_CHG* message since the distance between the *sender* and addressee is more than 2.

Upon receiving a CL_CHG message (Action E.05): Any node does the following: If the node gets the message from *n.i* and is the *destination*, then it updates its variables *c.i*, *d.i,* and *n.i,* and then it forwards it to its neighbors if the hop *count* is still valid and the addressee is a direct neighbor. If the addressee is not a direct neighbor, then it drops it.

Upon receiving a CL_ACCEPT message (Action E.06): If a clusterhead receives it and is the *destination*, then it updates its routing table and sends the updated message to the bordering gateway nodes. If a node that is not a *destination* receives it, it forwards the message to all its neighbors if the hop *count* is still valid, but drops the message if the hop *count* is invalid.

Upon receiving a leave message (Action E.07): If the clusterhead that is the *destination* receives the message, it updates the routing table and sends the updated message to all its bordering gateways. If the receiving node is not a *destination* node, then it forwards the message to all its neighbors if the hop *count* is still valid, but drops the message if the hop *count* is invalid.

Upon receiving a ctable_copy message (Action E.08): If the receiving clusterhead node is the *destination*, then the row is copied into the routing table if it meets the constraint that the *destination* node is within two hop distance.

Upon receiving a ctable_updated message (Action E.09): A gateway node checks if the message is from a clusterhead whose cluster member is one of its neighbors. If it is, it updates its GC_TABLE, else the message is ignored.

Predicates:

$is\_CH(i) \equiv (c.i = ID.i \ \wedge \ n.i = \text{nil} \wedge d.i = 0)$
$is\_EG(i) \equiv (\exists\ j \in N_i^{\ 1} \wedge c.j \neq c.i)$
$is\_G(i) \equiv ((is\_EG(i) \wedge g.i == T) \wedge \neg(\exists\ is\_G(j) \in N_i^{\ 1} \wedge c.j == c.i \wedge s.j == s.i))$
$is\_BG(i) \equiv (\exists\ j \in CG\_TABLE(i) \wedge is\_G(j))$
$is\_faulty(i) \equiv (c.i = \text{nil} \vee n.i \notin N_i^{\ 1} \vee d.i < 0 \vee d.i > 2)$

E.O1 *Timeout* →

```
if (c.i == ID.i) then
   if (n.i ≠ nil) then n.i =nil
   if (d.i ≠ 0) then d.i =0
   send CL_ANN (ID.i, j, path, 0) ∀ j ∈ Ni 1
   if (no_REQ4Long FromSender) then
         remove row from Ctable and send CL_REJ (sender,dest,0) ∀ j ∈ Ni 1
else
   if (is_faulty(i)) then
      if (no_ANN4Long) then
         c.i = ID.i
         if (n.i ≠ nil) then n.i =nil
         if (d.i ≠ 0) then d.i =0
         send CL_ANN (ID.i, j, path, 0) ∀ j ∈ Ni 1
   else
      if (c.i ≠ nil) then
         send CL_REQ (ID.i ,c.i ,0) to n.i
```

E.O2 ***Receive*** *CL_ANN* **(***sender, dest, path, hops***) from nb** →

```
if (hops < 2 ∧ dest == ID.i ∧ c.nb == sender) then
   if (sender == c.i) then
      if (n.i ∉ Ni 1) then
         n.i = nb
      if (hops == 0) then
         d.i = 1
         if (is_G(i)) then
            send CH_ACCEPT (ID.i, sender, rpath, 1, 0, T) to nb
         else // Ordinary node
            send CH_ACCEPT (ID.i, sender, rpath, 1, 0, F) to nb
         send CL_ANN (sender, j, path, 1) ∀ j ∈ Ni 1
      else
   if (hops == 1) then
         d.i = 2
         if (is_G(i)) then
            send CH_ACCEPT (ID.i, sender, rpath, 2, 0, T) to nb
```

```
            else
                send CH_ACCEPT (ID.i, sender, rpath, 2, 0, F) to nb
    else
        if (is_faulty(i)) then
            c.i =sender
            n.i =nb
            if (hops = = 0) then
                d.i = 1
                send CL_ANN (sender, j, path, 1)  ∀ j ∈ Ni 1 / nb
                if (is_G(i)) then
                    send CH_ACCEPT (ID.i, sender, rpath, 1, 0, T) to nb
                else
                    send CH_ACCEPT (ID.i, sender, rpath, 1, 0, F) to nb
            else
            if (hops = = 1) then
                d.i = 2
                if (is_G(i)) then
                    send CH_ACCEPT (ID.i, sender, rpath, 2, 0, T) to nb
                else
                    send CH_ACCEPT (ID.i, sender, rpath, 2, 0, F) to nb
    else
        if (is_CH(i)) then
            if (ID.i < sender) then
                c.i = sender
                n.i = nb
                newpath = ID.i + path
                send ctable_copy (dest, sender, newpath, ID.i ,hops+1, 0, g.i) to nb
                if (hops = = 0) then
                    d.i = 1
                    send  CL_REJ (ID.i , dest, 0) ∀ j ∈ Ni 1
                    send CL_ANN (sender, dest, path, 1)  ∀ j ∈ Ni 1 / nb
                    if (is_G(i)) then
                        send CL_ACCEPT (ID.i, sender, rpath, 1, 0, T) to nb
                    else
                        send CL_ACCEPT (ID.i, sender, rpath, 1, 0, F) to nb
                else
                if (hops = = 1) then
                    d.i = 2
                    send  CL_REJ (ID.i , dest, 0) ∀ j ∈ Ni 1
                    if (is_G(i)) then
                        send CH_ACCEPT (ID.i, sender, rpath, 2, 0, T) to nb
                    else // Ordinary node
                        send CH_ACCEPT (ID.i, sender, rpath, 2, 0, F) to nb
    else
        if (is_G(i)) then
            c.i = sender
            n.i = nb
            if (hops = = 0) then
                d.i = 1
                send CH_ANN (sender, j, path, 1) ∀ j ∈ Ni 1 / nb
                send CH_ACCEPT (ID.i, sender, rpath, 1, 0, T) to nb
            else
            if (hops = = 1) then
                d.i = 2
                send CH_ACCEPT (ID.i, sender, rpath, 2, 0, T) to nb
```

```
else   // Ordinary node
      if (c.i < sender) then
          send leave (ID.i, c.i, 0) to n.i
          c.i = sender
          n.i = nb
          if (hops = = 0) then
              d.i = 1
              send CH_ANN (sender, j, path, 1) ∀ j ∈ Ni 1 / nb
              send CH_ACCEPT (ID.i, sender, rpath, 1, 0, F) to nb
          else
          if (hops = = 1) then
              d.i = 2
              send CH_ACCEPT (ID.i, sender, rpath, 2, 0, F) to nb
else  // if hop count is greater than or equal to 2
      drop the message
```

E.O3 ***Receive*** **CL_REJ** (*sender, dest, hops*) **from nb** →

```
if (hops<2) then
      if (sender = = c.i  ∧ dest = = ID.i) then
          c.i = ID.i
          n.i = nil
          d.i = 0
          if (hops = = 0)
              send CL_REJ (sender, dest, 1) ∀ j ∈ Ni 1 / nb
else  // if hops != 0 or hops != 1
      drop the message
```

E.O4 ***Receive*** CL_REQ (*sender, dest, path, hops*) **from nb** →

```
if (hops < 2) then
   if (ID.i = = dest) then
      if (¬ is_CH(i)) then
         send CL_REQ (ID.i , sender, 0)  to nb
      else  // if it is a ClusterHead, then
         send CL_ANN (ID.i , sender, path, 0) to nb
         if (sender ∉ routingtable_i) then
            if (hops = = 0) then
                update (sender, ID.i, rpath, nb, 1, g.i)
            else
                if (hops = = 1) then
                   update (sender, ID.i, rpath, nb, 2, g.i)
            send ctable_updated (sender, j, ID.i)  ∀ j ∈ is_BG(i)
    else   // if (ID.i ≠ dest)
       if (hops = = 0) then
          if (dest ∈ Ni 1) then
           send  CL_REQ (sender, dest,  path, 1)  ∀ j ∈ Ni 1  / nb
else
      if (hops≥ 2) ∧ (ID.i = = n.sender)  then
     send CL_CHG (ID.i, d.ist, 0) to nb
else  drop the message
```

E.O5 ***Receive*** **CL_CHG** (*sender, dest, hops*) **from nb** →

```
if (hops < 2) then
    hops ++
    if ((ID.i = = dest) ∧ (nb = = n.i)) then
        c.i = nil
        n.i = ∞
        d.i = nil
    send CL_CHG (sender, dest, hops) ∀ j ∈ Ni 1
else drop the message
```

E.O6 ***Receive*** *CL_ACCEPT* **(***sender, dest, path, hops, count, g.i***) from nb →**

```
if (hops < 2) then
  if (dest = = ID.i) then
    if (is_CH(i)) then
      if (sender∉ routingtable_i) then
        if (hops = = 0) then
          update (sender, ID.i, rpath, nb, 1, g.i)
        else
          update (sender, ID.i, rpath, nb, 2, g.i)
        send ctable_updated (sender, j, ID.i)  ∀ j ∈ is_BG(i)
    else // if ¬ CH(i)
      send CL_REJ (ID.i ,sender, 0) to nb
  else
  if (dest ≠ ID.i ∧ dest ∈ Ni 1)
    if (hops = = 0) then
      send CH_ACCEPT (sender, dest, path, 1, 0, g.i) ∀ j ∈ Ni 1
    else
      send CH_ACCEPT (sender, dest, path, 2, 0, g.i) ∀ j ∈ Ni 1
else // if hops>2
  drop the message
```

E.O7 ***Receive*** **leave** **(***sender, dest, hops***) from nb →**

```
if (hops < 2) then
    if (ID.i = = c.i ) then
        if (dest = = c.i) then
            remove row from routingtable_i where sender ∈ routingtable(dest)_i
            send ctable_updated (sender, j, ID.i)  ∀ j ∈ is_BG(i)
    else // if the current node is not a clusterhead
      send leave (sender, dest, hops)  ∀ j ∈ Ni 1 / nb
  hops++
else
  drop the message
```

E.O8 ***Receive*** *ctable_copy* **(***dest ,sender, path, nexthop, hops, count, g.i***) from nb →**

```
if (count < 2) then
  if (dest = = ID.i) then
    if (is_CH) then
      if (hops+count <= 2) then
        newpath = path + path_from_oldCH_to_currentNode
        copy the row (dest, ID.i, newpath, nb, hops+count, g.i )
```

```
            else  // if (hops+ count > 2)
                drop the row
          else  // if the dest is not a CH any more
              drop the message
    else  // if the current node is not the destination
        if (count = = 1) then
              send ctable_copy (dest, sender, path, nexthop, hops, count, g.i_j) ∀ j ∈ N_i^1 / nb
    count++
else // if count > = 2
    drop the message
```

E.09 ***Receive*** *ctable_updated* (*node, dest, CH*) **from nb** →

```
if (is_CH(i)) then
    drop the message
else // if the current node is not a clusterhead
if (is_G(i)) then
    if (CH ∈ GC_TABLE(CH, index) ∧ node ∈ GC_TABLE(node, index) ∧ index >=0
                                    ∧ dest = = ID.i ∧ index < = HighestIndex) then
        remove row from GC_TABLE
        HighestIndex ← HighestIndex - 1
    if (CH ∈ GC_TABLE(CH, index) ∧ node ∉ GC_TABLE(node, index) ∧ index >=0
                                    ∧ dest = = ID.i ∧ index < = HighestIndex) then
        HighestIndex ← HighestIndex + 1
        GC_TABLE_update (HigestIndex, sender, dest, nb)
    if (nb = = CH) then ∀ j ∈ N_i^1 / nb
        send ctable_updated (node, dest, CH)
else // Ord.inary node
    if (nb = = CH) then
        send ctable_updated (node, dest, CH) ∀ j ∈ N_i^1 / nb
else // in any other case
    drop the message
```

4.4 Proof of Correctness

Lemma 4.1 The maximum number of hops between a clusterhead and a member of its own cluster is two.

Proof: In clusterhead election module, *Actions E.02* and *E.06* ensure that any clusterhead announcement (*CL_ANN*) message or the clusterhead accept (*CL_ACCEPT*) message can travel at most a distance of two *hops*. For a node to be a member of a cluster it has to receive the clusterhead announcement message from a clusterhead and send the clusterhead accept message back to the clusterhead, which is possible only if the node is at a two-hop distance from its clusterhead. □

Lemma 4.2 No two clusterheads can be neighbors of each other.

Proof: We prove this lemma by contradiction. Suppose there are two clusterheads that are neighbors. *Action E.02* ensures that the clusterhead announcement (*CL_ANN*) message of one clusterhead reaches the other that is at one or two-hop distance from it (Lemma 4.1). When a clusterhead receives a clusterhead announcement message, it compares its own ID with the *sender*'s ID. If its ID is less than that of the *sender*'s ID, it relinquishes its role as a clusterhead and sends the clusterhead reject (*CL_REJ*) message to all its two-hop neighbors. *Action E.03* ensures that the clusterhead reject message reaches all the two-hop neighbors. So, it no longer remains a clusterhead which contradicts our assumption that there can be two clusterheads that can be neighbors. □

Lemma 4.3 The minimum number of hops between two clusterheads is three.

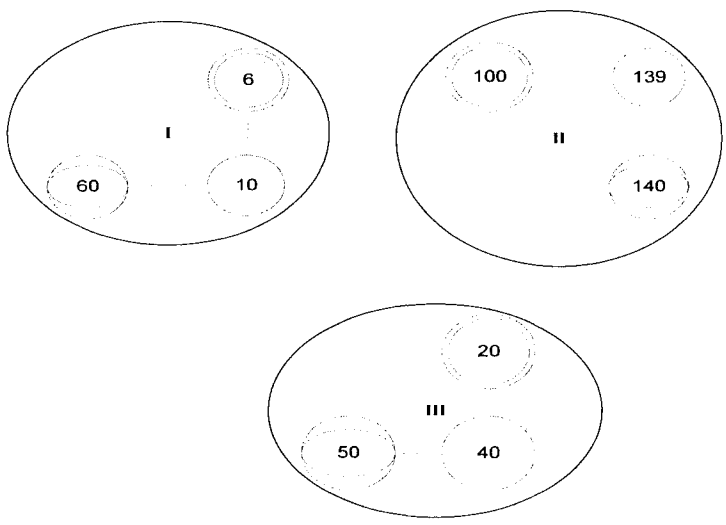

Figure 4.1. A network with three clusters before the nodes move.

Proof: From Lemma 4.2, no two clusterheads can be neighbors of each other. Assume that the distance between two clusterheads is two *hops*. But that cancels one of the two clusterheads by comparing the IDs because the node between them becomes a gateway that acts as a common node for both clusters.

Consider a network with nine nodes as shown in Figure 4.1. The nodes with ID 60 (clusterhead of cluster I), 140 (clusterhead of cluster II), and 50 (clusterhead of cluster III) are clusterheads. After some nodes move, the network looks like the one in Figure 4.2. The distance between node 60 and node 50 is 2 *hops*. Now, the intermediate node with ID 100 that connects the two nodes acts as a gateway that belongs to all the three clusters and allows the clusterhead announcement message from cluster I to reach cluster

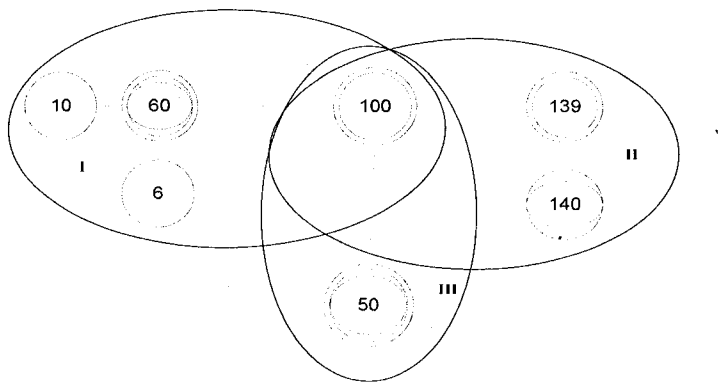

Figure 4.2. The Network of Figure 4.1 after the nodes move.

III through it. In our clusterhead election module, *Action E.02* makes sure that the two clusterheads' announcement messages reach each other and the one with the lower ID relinquishes its role as a clusterhead. Thus, node 50 relinquishes its role as a

clusterhead and the network now looks as shown in Figure 4.3. Therefore, there cannot be a clusterhead at a distance of two *hops* from another clusterhead. □

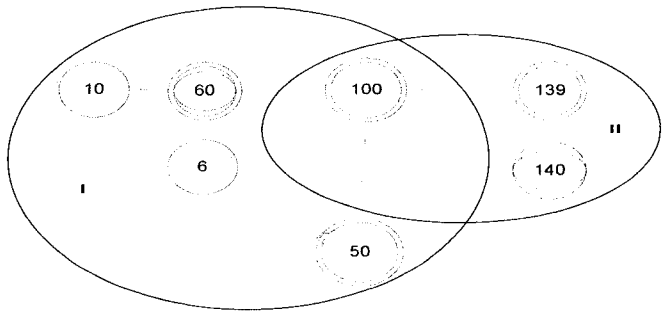

Figure 4.3. The Final Network of Figure 4.1 after clustering.

Lemma 4.4 The maximum number of hops between the clusterheads of two neighboring clusters is five.

Proof: We need to prove the following two results:

Case I: Two clusterheads can be at a distance of five hops from each other.

Proof: In this network, exactly one clusterhead announcement message reaches every node. So, the cluster structure does not change any more.

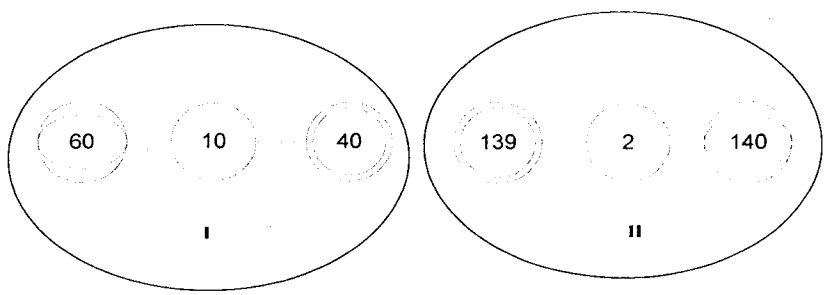

Figure 4.4. Two Clusterheads at a distance of five hops from each other.

Consider Figure 4.4. There are two clusterheads, 60 and 140 which are at distance of five *hops* from each other. All the nodes are clustered according to the rules of link-cluster architecture. Thus, the clusterheads do not change. □

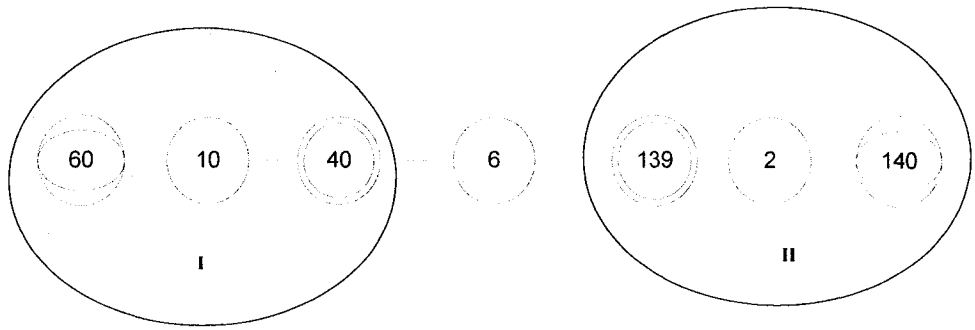

Figure 4.5. Two Clusterheads at a distance of six hops from each other.

Case II: Two clusterheads can never be at a distance of more than five hops from each other.

Proof: We can prove this case by contradiction. Let us assume that the maximum distance between the two clusterheads is six. According to our module, *Action E.02* makes sure that the clusterhead announcement message travels at most a distance of two *hops.* Then, there is at least one node that does not receive any clusterhead announcement message. This node waits for a timeout period (*Action E.01*) and at timeout, sets itself a clusterhead forming its own cluster.

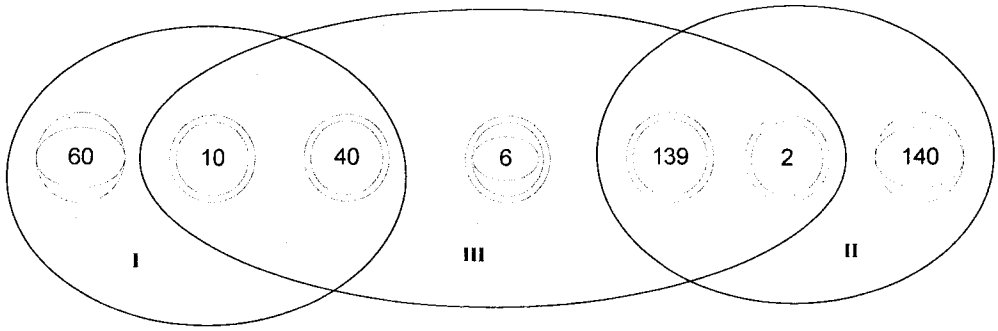

Figure 4.6. Final Network of Figure 4.5 after Clustering.

Consider the network as shown in Figure 4.5. The clusterheads 60 and 140 are at a distance of six *hops* from each other. The clusterhead election messages travel at most two *hops*. So, no clusterhead announcement messages reach node 6. Node 6 waits for a timeout interval and elects itself as a clusterhead. Now, the network has three clusters as shown in Figure 4.6. Therefore, there cannot be two clusterheads at a distance of six *hops* from. □

Lemma 4.5 All clusterhead election messages follow a loop-free *path*.

Proof: As per *Assumption 3.4*, every link is bi-directional. In the clusterhead election module, it was made sure that the clusterhead election messages traverse at most one hop before being discarded (a non-clusterhead node can be at distance of at most two *hops* from its clusterhead). So, there is a fair chance that a message generated by a node reaches itself in at most two *hops* forming a loop. For example, consider a network of four nodes as shown in Figure 4.7.

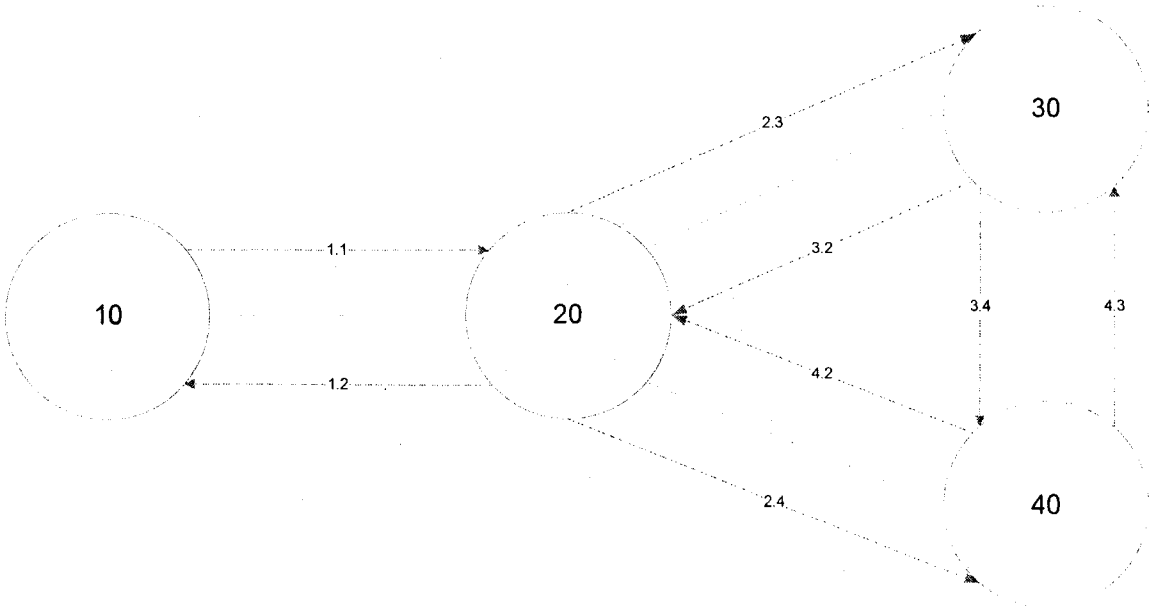

Figure 4.7. A four-node Network.

If the loop length is two, then the message would bounce between the two nodes, e.g., following the *path* 1.1 and 1.2. But another constraint, strictly implemented in every action, states that the message is not sent back to the neighbor that has delivered it. Thus, the message does not use the *path* 1.2 after it reaches node 20 from node 10. If the loop length is three, then the message would cycle among three nodes, e.g., following the *path* 2.3, 3.4, and 4.2. But since a message can traverse at most one hop before being discarded, the node 40 it will not send the message further. So, the *path* 4.2 will not be used. □

CHAPTER 5

GATEWAY ELECTION ALGORITHM

A gateway node must be connected to more than one cluster. This is implemented by checking if the node has at least two neighbors that belong to different clusters. A gateway updates its routing table according to the changes made in the bordering clusterhead tables and when it receives a message from a node whose entry does not exist in its table. If a gateway has to play the role of a clusterhead, it can do so without making any changes in its table entries [14]; it will be updated later when new nodes join or *leave* [16], but the number of tables it now holds is changed.

Gateway Election Algorithm contains the actions related to selection of gateways among the nodes in the cluster, and creating or updating entries in the gateways' tables used for inter-cluster routing. Section 5.1 explains the additional predicates used in the algorithm (that were not used in clusterhead election). In Section 5.2, we give a brief description of the messages used to elect a gateway. Section 5.3 includes the detailed description of the actions performed upon receiving the gateway election messages followed by the complete code of the proposed algorithm. We provide proof of correctness of this module in Section 5.4.

5.1 Predicates

Predicate *is_BC*(i) $\equiv$ (*is_CH*(j) $\wedge$ $j \in$ GC_TABLE(*i*)) is true if j is a clusterhead and is a member of i's intra-cluster table. If this predicate is true, then j is a clusterhead of a cluster connected to the gateway i.

5.2 Messages

Message *GW_ANN* contains the following fields: *sender*, *dest*, path (path from the *sender* to the current node), and *hops*.

Message *GW_REJ* contains the following fields: *sender*, *dest*, and *hops*.

5.3 Algorithm

A gateway node periodically does the following: It checks if there exists another gateway node in its two hop distance that at least connects the clusters connected by itself. If one exists, it relinquishes its role as a gateway by updating its *g.i* variable and sending a *GW_REJ* message. It checks if there exists another gateway in two hop distance that connects the same clusters. If it finds one, it compares its own ID with it. If it has a smaller ID, then it relinquishes its role as a gateway by updating its *g.i* variable and sending a *GW_REJ* message. In our module, *Action G.01* takes care of it.

Upon receiving GW_ANN message (Action G.02): If the node is a clusterhead as well as the *destination* node, it updates its inter-cluster table. If the node is a gateway, it checks if there exists another gateway node in its two hop distance that at least connects the clusters connected by it. If it finds one, it relinquishes its role as a gateway by updating *g.i* and sends a *GW_REJ* message. If there exist another gateway in two hop

distance that connects the same clusters, it compares its own ID with it, and if it has a lesser ID value, it relinquishes its role as a gateway by updating *g.i* and sends a *GW_REJ* message. If the distance from the *sender* to its neighbors is within two *hops*, a node updates the hop count and forwards the message to all its neighbors.

Upon receiving GW_REJ message (Action G.03): If the node is the *destination* clusterhead and contains the *sender*'s ID as its bordering gateway node, it removes all such rows containing the *sender*'s ID in the *GW* field of its tables. If the distance from the *sender* to its neighbors is within two *hops*, a node updates the hop count and forwards the message to all its neighbors.

Predicates:

is_BC(i) ≡ (*is_CH*(j) ∧ *j* ∈ GC_TABLE(*i*))

G.01 *Timeout* →

if (*is_G*(i)) ***then***
  if ($\exists$ j ∈ N_i^1 ∧ $\exists$ j ∈ N_i^2 ∧ *is_G*(i)) ***then***
    if (GC_TABLE(i) ⊂ GC_TABLE(j)) ***then***
      g.i = F
      send *GW_REJ* (*ID*$_i$, *k*, *0*) ∀ *k* ∈ *is_BC*(CH)
    else
    if (GC_TABLE(i) = = GC_TABLE(j)) ***then***
      if (ID.i < ID.j) ***then***
        g.i = F
        send *GW_REJ* (*ID*$_i$, *k*, *0*) ∀ k ∈ *is_BC*(CH)
else
  do nothing

G.02 ***Receive*** *GW_ANN* (*sender, dest, path, hops*) **from** *nb* →

if (*hops* <2) ***then***
    if (*dest* = = ID.i ∧ *is_CH*(i)) ***then***
      update CG_TABLE (*sender, sender, nb*)
    else *// if the current node is not the destination and a clusterhead*
    if (*is_G*(i)) ***then***
      if (GC_TABLE(i) = = GC_TABLE(*sender*)) ***then***
        if (ID.i < *sender*) ***then***
          g.i = F
          send *GW_REJ* (*ID.i* , *j* ,*0*) ∀ j ∈ *is_BC*(i)

```
    if (hops = = 0) then
        send GW_ANN (sender, j, path, 1) ∀ j ∈ N_i^1 / nb
    else
        if (hops = = 1) then
            send GW_ANN (sender, j, path, 2) ∀ j ∈ N_i^1 / nb
    hops ++
else // if hops !=0 or hops !=1
    ignore the message
```

G.03 ***Receive*** GW_REJ (*sender, dest, hops*) **from** *nb* →

```
if (hops <2) then
  if (dest = = ID.i ∧ is_CH(i) ∧ sender ∈ is_BC(i)) then
     remove rows from CG_TABLE
  if (hops = = 0) then
     send GW_REJ (sender, j, 1) ∀ j ∈ N_i^1 / nb
  else
  if (hops = = 1) then
     send GW_REJ (sender, j, 2) ∀ j ∈ N_i^1 / nb
   hops ++
else // if hops !=0 or hops !=1
    ignore the message
```

5.4 Proof of Correctness

Lemma 5.1 A node with at least one neighbor that belongs to a different cluster becomes an eligible gateway node.

Proof: We prove this lemma by contradiction. Suppose that there exists a gateway node that has all the neighbors in the same cluster. Then it has connections with the members of only one cluster. By definition of a gateway node, it is clear that a gateway must connect at least two clusters. If a node connects two clusters then, it has at least one neighbor that does not belong to its own cluster.

In Figure 5.1, nodes 10, 20, 30, 40, 50, and 60 are eligible gateway nodes because all the nodes have at least one neighbor that does not belong to its own clusterhead. □

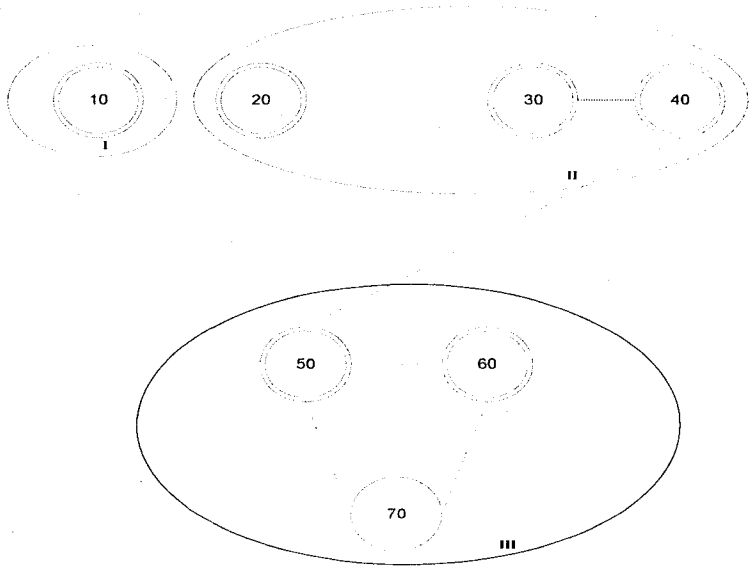

Figure 5.1. Eligible gateway nodes.

Lemma 5.2 If there exist only one link connecting two neighboring clusters then the eligible gateway nodes on both ends of the link will be selected as gateway nodes.

Proof: We prove this lemma by contradiction. Suppose the nodes connecting the clusters are not gateway nodes. But, by the definition of a gateway and Lemma 5.1, both the nodes are eligible gateway nodes because both of them have at least one neighbor that does not belong to its own cluster. In our module, we eliminate the eligible gateway nodes becoming the gateway nodes only if they belong to the same cluster. So, both the nodes become the gateway nodes that contradict the assumption that they are not the gateway nodes. □

Lemma 5.3 If two eligible gateway nodes in a cluster connect the same set of clusters, then the node with the higher UID becomes a gateway node.

Proof: An eligible gateway node is allowed to remain a gateway node if and only if it satisfies the condition that it has no other gateway node in its own cluster connecting the same clusters it is connecting. This is implemented to avoid the storage of redundant data.

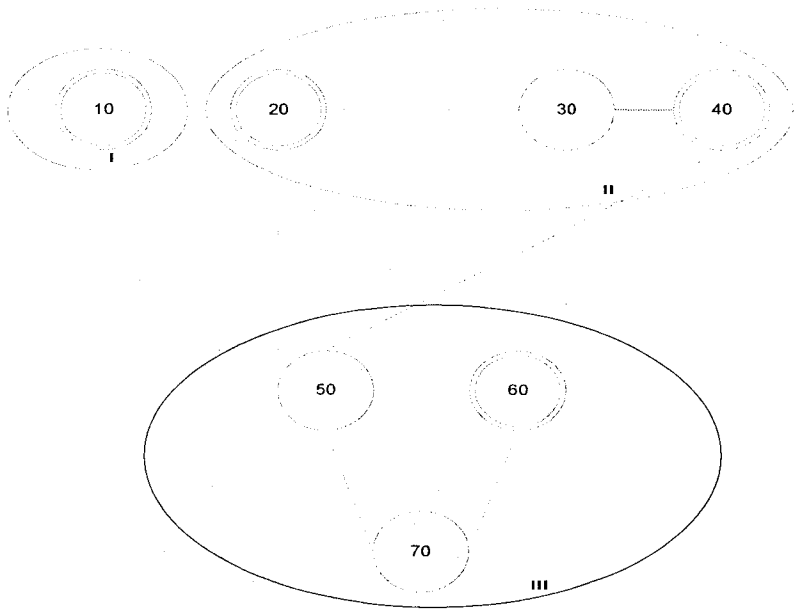

Figure 5.2. Eligible gateway nodes after eliminating the gateways connecting the same clusters.

Considering the network in Figure 5.1, nodes 30 and 40 belong to the same cluster II and connect the same two clusters: cluster II and cluster III. In this case, when the gateway announcement (*GW_ANN*) message of node 40 reaches node 30, it compares its own ID with that of node 40 and finds that node 40 has a larger ID and also connects the same clusters. Then node 30 relinquishes its role as a gateway node and sends the gateway reject (*GW_REJ*) message to all the clusterheads of clusters it connects. *Actions*

G.01, *G.02*, and *G.03* ensure this. Similarly, the eligible gateway nodes 50 and 60 belong to cluster III and connect the same clusters: cluster II and cluster III. Following the same procedure used in cluster II, node 50 relinquishes its role as a gateway node and sends the gateway reject (*GW_REJ*) message to all the clusterheads of clusters it connects. *Actions G.01, G.02*, and *G.03* ensure this. Thus, the final gateway nodes are reduced to nodes 10, 20, 40, and 60 as shown in Figure 5.2. □

Lemma 5.4 Consider two nodes i and j in a cluster c. Assume that Nodes i and j connect the cluster sets S_i and S_j, respectively. If $S_i \supset S_j$, then i becomes a gateway node.

Proof: An eligible gateway node is allowed to remain a gateway node if and only if it satisfies the condition that it has no other gateway node in its own cluster connecting at least the same clusters it is connecting. This is implemented to avoid the redundant data storage.

For example, in Figure 5.2, the nodes 20 and 40 belong to the same cluster II. The cluster set that the node 20 connects are: I, II and III where as the cluster set that the node 40 connects are: II and III. In this case, when the gateway announcement (*GW_ANN*) message of 20 reaches node 40, it finds that node 20 connects more number of clusters including the same clusters it connects. Then node 40 relinquishes its role as a gateway node and sends the gateway reject (*GW_REJ*) message to all the clusterheads of clusters it connects. *Action G.01*, *Action G.02* and *Action G.03* take care of this. Thus finally, nodes 10, 20, and 60 become the gateway nodes as shown in Figure 5.3. □

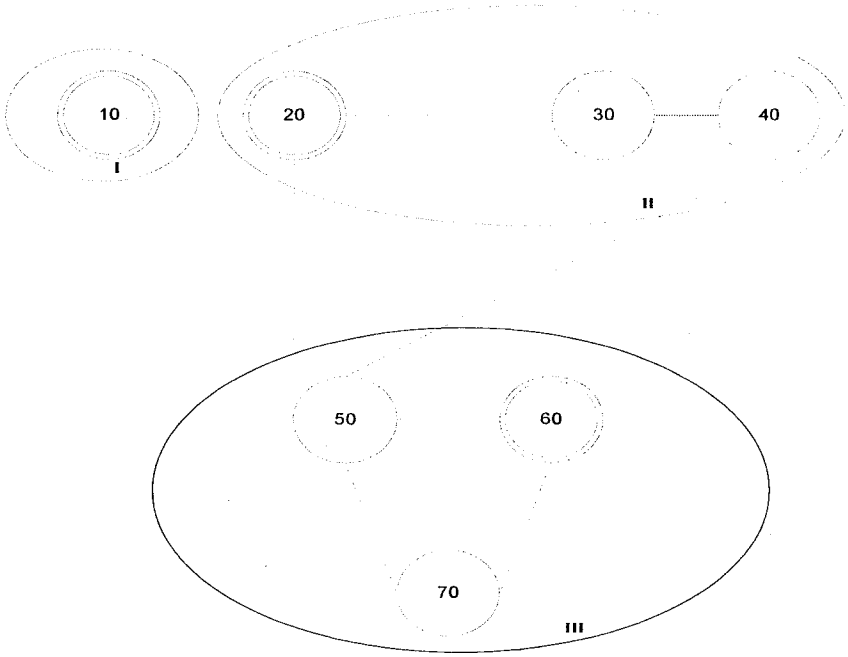

Figure 5.3. Final gateway nodes.

Lemma 5.5 All the gateway election messages follow a loop-free path.

Proof: The proof is very similar to that of *lemma 4.5.* As per *Assumption 3.4*, every link is bi-directional. In the gateway election module, it was made sure that the gateway election messages traverse at most one hop before being discarded (a gateway node can be at distance of at most two *hops* from its clusterhead). So, there is a fair chance that a message generated by a node reaches itself in at most two *hops* forming a loop. □

CHAPTER 6

ROUTE DISCOVERY ALGORITHM

This algorithm is responsible for creating and/or updating entries in the routing tables in each clusterhead and also the gateways for the inter-cluster routing. Section 6.1 includes an overview of the algorithm. In section 6.2, we give a brief description of the messages used for discovering a route. Section 6.3 includes the detailed description of the actions performed on receiving the route discovery messages, followed by the complete code for the proposed algorithm. The chapter ends with some proofs to support the module in Section 6.4.

6.1 Overview

Two types of routing techniques, proactive and reactive, are used to route the packets within the clusters and between the clusters, respectively.

For routing within the cluster, each clusterhead keeps information in its routing table about the nodes that belong to its own cluster. This information is collected in the Module Clusterhead Election (Algorithm) using *CL_REQ* messages. These messages are periodically sent by a non-clusterhead node to check the status of its own clusterhead and the *path* towards it.

For routing between the clusters, the clusterheads as well as the gateway nodes keep information of the gateway-destination and clusterhead-destination pairs, respectively to reach the temporary destination, which is a milestone in reaching the actual *destination.* This data is collected only when there is a need to communicate with the node and stored in the inter-cluster tables. These tables purge the routes that are unused for a long time and keep the entries updated. The following subsection explains a step by step flow of the algorithm. The step by step flow of the Algorithm is as follows:

1.*Sender* checks with its clusterhead if its routing table has an entry for the *destination* node that it wants to communicate with. If the clusterhead has an entry, the *sender* gets the *path* from the clusterhead and uses it to communicate.

2.If the clusterhead's routing table does not have an entry, it checks with the clusterhead's gateway table. If it finds an entry, then it uses that route to communicate.

3.If the clusterhead's gateway table does not have an entry, then it checks with the gateway's cluster tables of all the bordering gateways for the route. If it finds the route, it uses that to communicate.

4.The steps 2, 3, and 4 are repeated until the route is found.

6.2 Messages

Message *Routedisc* has the following fields: *sender*, *dest*, *tempdest* (clusterhead or a gateway node that might be a milestone in reaching the *destination* node), *path* (path from the *sender* the message has traveled so far), and *seq* (sequence number of the message that is initiated by the *sender*).

Message me_dest contains the following fields: *sender*, *dest*, *ch* (clusterhead of the *sender*), *path* (*path* from the *sender* to the *destination*), *path2ch* (path from the *destination* to the clusterhead of *destination* the message has traveled so far), and *chd* (clusterhead of the *destination*).

Message *shortestpath* has the following fields: *sender*, *dest* , *ch,* and *route* (path from the *sender* to the *destination*).

Message *ack* has the following fields: *sender*, *dest*, *ch,* and *path* (path from the *sender* to the *destination*).

Message *Ctable_updated* has the following fields: *node* (ID of the node that can be communicated with), *dest* (ID of the *destination* gateway node the message is sent to), and *CH* (the ID of the message initiating clusterhead).

Message *Gtable_updated* has the following fields: *node* (ID of the node that can be communicated with), *dest* (ID of the *destination* clusterhead node the message is sent to), and *GW* (the ID of the message initiating gateway node).

6.3 Algorithm

An ordinary node broadcasts a *routedisc* message to all its neighbors in its cluster. Upon receiving this message, a clusterhead looks in its routing table to see if the entry for that *destination* already exists. If it finds one, it immediately acknowledges the *sender* with a *shortestpath* message instead of waiting for the *destination* node to respond. If the *routedisc* message reaches the *destination* node, the *destination* node sends an *ack* message to the *sender*. Once the *shortestpath* message or the *ack* message reaches the *sender*, it can now start sending data packets following that *path*. In this module, we

followed the clusterhead, gateway, clusterhead *path* to find the *route*. We will now discuss all the actions in detail.

Upon receiving Routedisc message (Action A.01): If the node got the same message previously, it ignores the message. Else, it updates its sequence table. A clusterhead does the following: If it is the *destination*, it sends the *ack* message back to the *sender*. If it is the temporary *destination* and the *destination* node belongs to its cluster, it sends the *shortestpath* message back to the *sender*. If the *destination* as well as the *sender* does not belong to its cluster, it updates its inter-cluster table (clusterhead's gateway table) and sends the updated message to the bordering gateway nodes. If the *destination* belongs to the inter-cluster table, it forwards the message to the corresponding bordering gateway. If the *destination* does not belong to the inter-cluster table, it forwards the message to all the bordering gateways.

A gateway node does the following: If it is the *destination*, it sends the *ack* message to the *sender* and sends a message indicating itself as the *destination* to its clusterhead. If it is the temporary *destination*, it does the following: If the *destination* belongs to its inter-cluster table (gateway's clusterhead table), then it forwards the message to that particular clusterhead. If the *destination* is not found in its inter-cluster table, it forwards the message to all the clusterheads in its inter-cluster table. If the *sender* is not found in its inter-cluster table, it updates its table and sends the updated message to all the bordering clusterheads.

An ordinary node does the following: If it is the *destination*, it sends the *ack* message to the *sender* and sends a message indicating itself as the *destination* to its clusterhead. If it is not the *destination*, then it forwards the message to all its neighbors.

Upon receiving me_dest message (Action A.02): A clusterhead does the following: If it is the clusterhead of the *destination* and the *sender* does not belong to its inter-cluster routing table, it updates the table and sends the updated message to all its bordering clusterheads. A gateway node does the following: If the clusterhead of the *destination* is at one hop distance, it forwards the message. If the *sender* does not belong to the inter-cluster routing table, it updates the table and sends the updated message to all its bordering clusterheads. An ordinary node does the following: If the clusterhead of the *destination* is at one hop distance, it forwards the message.

Upon receiving shortestpath message (Action A.03): A clusterhead does the following: If the *sender* does not belong to its inter-cluster routing table, it updates its table and sends the updated message to all its bordering gateway nodes. If it is not the *destination,* then it forwards the message to all the neighboring nodes in the *route.* A gateway does the following: If the *sender* does not belong to its inter-cluster routing table, then it updates its table and sends the updated message to all its bordering clusterheads. If it is not the *destination*, then it forwards the message to all the neighboring nodes in the *route.* An ordinary node does the following: If it is not the *destination*, then it forwards the message to all the neighboring nodes in the *route.*

Upon receiving ack message (Action A.04): A clusterhead does the following: It updates its table and sends the updated message to the bordering gateways, and if required, forwards the message to all its neighbors. A gateway does the following: It updates its table and sends the updated message to the bordering clusterheads, and if required, forwards the message to all its neighbors. An Ordinary node does the following:

If the clusterhead of the destination is at one hop distance, it forwards the message to that particular neighbor.

Upon receiving Ctable_updated message (Action A.05): A clusterhead drops the message. A gateway does the following: It checks if the message is from a clusterhead whose cluster member is a neighbor. If yes, it updates its inter-cluster routing table. If it received the message from the *sender* (if the message's initiator is its neighbor), then it forwards the message to all its neighbors. An ordinary node does the following: If it got the message from the *sender* (if the message's initiator is its neighbor), then it forwards the message to all its neighbors. In all other cases, the message is ignored.

Upon receiving Gtable_updated message (Action A.06): A clusterhead does the following: It checks if the message is from a gateway node that is present in its inter-cluster routing table. If yes, it updates its inter-cluster routing table. If it received the message from the *sender* (if the message's initiator is its neighbor), then it forwards the message to all its neighbors. An ordinary node or gateway does the following: If it received the message from the *sender* (if the message's initiator is its neighbor), then it forwards the message to all its neighbors. In all other cases, the message is ignored.

A.01 ***Receive*** *Routedisc* (*sender, dest, tempdest, path, seq*) **from** *nb* →

```
if ((sender, seq) ∈ SEQ_TABLE_i) then
   drop the message
else
   update SEQ_TABLE_i (sender, seq)
if (is_CH(i)) then
   if (ID.i ∉ path) then
      path = path + ID.i
   if (dest = = ID.i) then
      send ack (dest, sender, ch, rpath) to nb
  else // if the clusterhead is not the destination
      if (ID.i = = tempdest) then
         if (dest ∈ routingtable_i) then
            send shortestpath (dest, sender, ch, route) to nb
```

```
            else // if destination does not belong to the routing table
            if (sender ∉ CG_TABLE(dest)) then
                update_CGtable (latest_GW_in_path, sender, nb)
                send Ctable_updated (sender, j, ID.i) ∀ j ∈ is_BG(i) / nb
            else
            if (dest ∈ CG_TABLE) then
                send Routedisc (sender, dest, GW(dest), path, seq) to nexthop(dest)
            else // if destination does not belong to the Clusterhead's gateway table
                send Routedisc (sender, dest, j, path, seq) ∀ j ∈ is_BG(i)
else // if the current node is not a Clusterhead
if (is_G(i)) then
   if (ID.i ∉ path) then
      path = path + ID.i
   if (dest = = ID.i) then
      send me_dest (sender, dest, ch, path, path2ch, chd) to n.i
      send ack (dest, sender, ch, rpath) to nb
   else //if the current node is not the destination
   if (ID.i = = tempdest) then
      if (dest ∈ GC_TABLE_i) then
         send Routedisc (sender, dest, CH(dest), path, seq) to nexthop(dest)
      else // if destination does not belong to the Gateway's clusterhead table
         send Routedisc (sender, dest, j, path, seq) ∀ j ∈ is_BC(i)

         if (sender ∉ GC_TABLE_i) then
            update_GCtable (latest_CH_in_path, sender, nb)
            send Gtable_updated (sender, j, ID.i) ∀ j ∈ is_BC(i) / nb
else // Ordinary node
   if (ID.i ∉ path) then
      path = path + ID.i
   if (dest = = ID.i) then
      send me_dest (sender, dest, ch, path, path2ch, chd) to n.i
      send ack (dest, sender, ch, rpath) to nb
   else
      send Routedisc (sender, dest, j, path, seq) ∀ j ∈ N_i^1 / nb
```

A.02 *Receive* me_dest (*sender, dest, ch, path, path2ch, chd*) from neighbor *nb* →

```
if (ID.i = = c.i) then // if the current node is a clusterhead
  if (chd = = ID.i) then // if the current node is the clusterhead of the destination
     if (sender ∈ routingtable_i) then
        do nothing
     else // if sender doesnot belong to its own cluster
      if (sender ∉ CG_TABLE(dest)) then
         update_CGtable (latest_GW_in_path, sender, nb)
         send Ctable_updated (sender, j, ID.i) ∀ j ∈ is_BG(i) / nb
  else // if the current node is not a clusterhead
  if (is_G(i)) then // if the current node is a gateway
    if (ID.i ∉ path2ch) then
       path2ch = path2ch + ID.i
    if (chd ∈ N_i^1) then // if the clusterhead of the destination is at 1 hop distance from
                                                                the current node
       send me_dest (sender, dest, ch, path, path2ch,chd) to n.i
       if (sender ∉ GC_TABLE_i) then
          update_GCtable (latest_CH_in_path, sender, nb)
```

send Gtable_updated (sender, j, ID.i) $\forall$ j $\in$ is_BC(i) / nb
else // *if the current node is neither a clusterhead nor a gateway*
if (ID.i $\notin$ path2ch) *then*
path2ch = path2ch + ID.i
if (chd $\in N_i^{1)}$ *then* // *if the clusterhead of dest is within one hop from the current node*
send me_dest (sender, dest, ch, path, path2ch,chd) $\forall$ j $\in N_i^1$ / nb

A.03 ***Receive*** *shortestpath* **(***sender, dest, ch, route***) from** *nb* $\rightarrow$

if (ID.i = = c.i) *then* // *if the current node is a clusterhead*
if (sender $\notin$ CG_TABLE) *then*
update_CGtable (latest_GW_in_path, sender, nb)
send Ctable_updated (sender, j, ID.i) $\forall$ j $\in$ is_BC(i)
if (dest = = ID.i) *then* // *if the message is addressed to the current node*
do nothing
else // *if the message is not addressed to the current node*
send shortestpath (sender, dest, ch, route) $\forall$ j $\in (N_i^1 \wedge$ route) / nb
else // *if the current node is not a clusterhead*
if (is_G(i)) *then* // *if the current node is a gateway*
if (sender $\notin$ GC_TABLE) *then*
update_GCtable (latest_CH_in_path, sender, nb)
send Gtable_updated (sender, j, ID.i) $\forall$ j $\in$ is_BC(i) / nb
if (dest = = ID.i) *then* // *if the message is addressed to the current node*
do nothing
else // *if the message is not addressed to the current node*
send shortestpath (sender, dest, ch, route) $\forall$ j $\in (N_i^1 \wedge$ route) / nb
else // *if the current node is neither a clusterhead nor a gateway*
if (ID.i $\neq$ dest) *then* // *if the current node is not the destination*
send shortestpath (sender, dest, ch, route) $\forall$ j $\in (N_i^1 \wedge$ route) / nb

A.04 ***Receive*** *ack* **(***sender, dest, ch, path***) from** *nb* $\rightarrow$

if (ID.i = = c.i) *then* // *if the current node is a clusterhead*
if (dest = = ID.i) *then* // *if the message is addressed to the current node*
update_CGtable (latest_GW_in_path, sender, nb)
send Ctable_updated (sender, j, ID.i) $\forall$ j $\in$ is_BG(i)
else // *if the message is not addressed to the current node*
update_CGtable (latest_GW_in_path, sender, nb)
send Ctable_updated (sender, j, ID.i) $\forall$ j $\in$ is_BG(i)
send ack (sender, dest, ch, path) $\forall$ j $\in (N_i^1 \wedge$ route) / nb
else // *if the current node is not a clusterhead*
if (is_G(i)) *then* // *if the current node is a gateway*
if (dest = = ID.i) *then* // *if the message is addressed to the current node*
update_GCtable (latest_CH_in_path, sender, nb)
send Gtable_updated (sender, j, ID.i) $\forall$ j $\in$ is_BC(i) / nb
else // *if the message is not addressed to the current node*
update_GCtable (latest_CH_in_path, sender, nb)
send Gtable_updated (sender, j, ID.i) $\forall$ j $\in$ is_BC(i) / nb
send ack (sender, dest, ch, path) $\forall$ j $\in (N_i^1 \wedge$ path) / nb
else // *if the current node is neither a clusterhead nor a gateway*
if (ID.i $\neq$ dest) *then* // *if the current node is not the destination*
send ack (sender, dest, ch, path) $\forall$ j $\in (N_i^1 \wedge$ path) / nb

A.05 ***Receive*** *Ctable_updated* **(***node, dest, CH***) from** *nb* $\rightarrow$

if (*is_CH*(i)) ***then***
  drop the message
else *// if the current node is not a clusterhead*
if (*is_G*(i)) ***then***
  if (CH ∈ GC_TABLE(*CH, index*) ∧ *node* ∈ GC_TABLE(*node, index*) ∧ *index* >=0
    ∧ *dest* = = *ID.i* ∧ *index* < = *HighestIndex*) ***then***
    remove row from GC_TABLE
    HighestIndex ← *HighestIndex* - 1
  if (CH ∈ GC_TABLE(*CH, index*) ∧ *node* ∉ GC_TABLE(*node, index*) ∧ *index* >=0
    ∧ *dest* = = *ID.i* ∧ *index* < = *HighestIndex*) ***then***
    HighestIndex ← *HighestIndex* + 1
    GC_TABLE_update (*HigestIndex, sender, dest, nb*)
  if (*nb* = = *CH*) ***then***
    send *Ctable_updated* (*node, dest, CH*) ∀ j ∈ N_i^1 / *nb*
else *// Ordinary node*
  if (*nb* = = *CH*) ***then***
    send *Ctable_updated* (*node, dest, CH*) ∀ j ∈ N_i^1 / *nb*
else *// in any other case*
  drop the message

A.06 ***Receive*** *Gtable_updated* (*node, dest, GW*) **from** *nb* →

if (*is_CH*(i)) ***then***
  if (*GW* ∈ CG_TABLE(*GW, index*) ∧ (*node* ∈ CG_TABLE(*node, index*) ∨ *node* ∈
    routingtable_i)) ***then***
    remove row from CG_TABLE
    HighestIndex ← *HighestIndex* - 1
  if (*GW* ∈ CG_TABLE(*GW, index*) ∧ *node* ∉ CG_TABLE(*node, index*) ∧ *node* ∉
    routingtable_i) ***then***
    HighestIndex ← *HighestIndex* + 1
    CG_TABLE_update (*HigestIndex, sender, dest, nb*)
  if (*nb* = = *GW*) ***then***
    send *Gtable_updated* (*node, dest, GW*) ∀ j ∈ N_i^1 / *nb*
else *// Current node is a gateway or an Ordinary node*
  if (*nb* = = *GW*) ***then***
    send *Gtable_updated* (*node, dest, GW*) ∀ j ∈ N_i^1 / *nb*
else *// in any other case*
  drop the message

6.3 Proof of Correctness

Lemma 6.1 All messages in the route discovery module follow a loop-free path.

Proof: We will consider the messages individually.

Case 1: *routedisc* message.

Every node has a sequence table that has entries for the *sender* ID (*sender*) and message ID (*seq*). When a route discovery message arrives, a node checks in its sequence table for an entry of the (*sender, seq*) pair. If it does not find an entry, it copies the (*sender, seq*) pair into its sequence table and forwards the message to all its neighbors except to the neighbor from which it got the message. If it finds an entry, it means the same message has already been sent to it. So, it discards the message making its traversal loop-free.

Case 2: *me_dest* message.

This message is sent to a node's own clusterhead and is always sent through the node that is the next-hop neighbor on the shortest path towards the clusterhead and is always forwarded to the neighbors from whom it did not get the message from. The message travels a distance of at most two *hops*. So, it can never form a loop.

Case 3: *shortestpath* and *ack* messages.

These messages always follow the reverse of the *routedisc* message which is loop-free as proved in Case 1 above. The reverse of a loop-free path is always a loop-free path.

Case 4: Table *updation* messages.

These messages are always sent to the bordering gateway nodes or clusterhead nodes and are always sent through the node that is the next-hop neighbor on the shortest path towards them. These messages are always forwarded to the neighbors from whom it did not get the message from. The messages travel at most a distance of two *hops*. So, they can never form a loop. □

Lemma 6.2 If both the sender and destination are in the same cluster, a route discovery message is always acknowledged.

Proof: When a node generates the route discovery (*routedisc*) message, it first sends it to its own clusterhead. Route discovery within a cluster means that the *sender* and *destination* belong to the same cluster. We need to prove the following two results:

Case I: The message reaches the *destination* before reaching the clusterhead.

Proof: It means that the *destination* is on the way to the clusterhead from the *sender*. In this case, the *destination* node directly sends the acknowledgement (*ack*) message to the *sender* following the reverse path followed by the route discovery message.

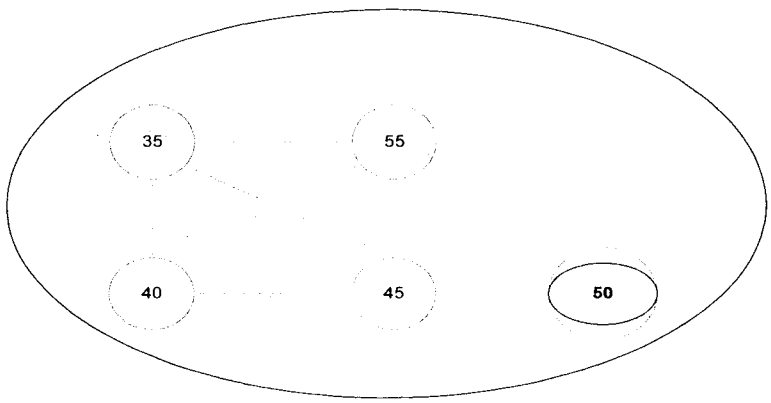

Figure 6.1. Message reaches the destination node before reaching the clusterhead.

Consider the following example in Figure 6.1. Suppose node 35 is the *sender* and node 45 is the *destination*. Then, when node 35 issues a *routedisc* message to find the route to node 45, the message hits the *destination* node 45 on its way to node 50 which is the clusterhead. Then node 45 sends the acknowledgement (*ack*) message to node 35. □

Case II: The message reaches the clusterhead before the destination.

Proof: All the clusterheads have entries for all the nodes in their intra-cluster table (*routing table* as named in our module) that belong to its own cluster. Once the clusterhead receives the message, it looks in its routing table, attaches the route from itself to the *destination* to the path followed by the route discovery message, and sends an acknowledgement message to the *sender* using a *shortestpath* message on the reverse path followed by the route discovery message.

Consider the following example in Figure 6.2. Suppose node 35 is the *sender* and node 45 is the *destination*. Then, when node 35 issues a *routedisc* message to find the route to node 45, the message reaches the node 50 which is the clusterhead. Then node 50 finds an entry in its routing table for node 45 that belongs to its own cluster. It then sends back the *shortestpath* message that acts as an acknowledgement message to the *sender* node 35 with the complete path from node 35 to node 45. □

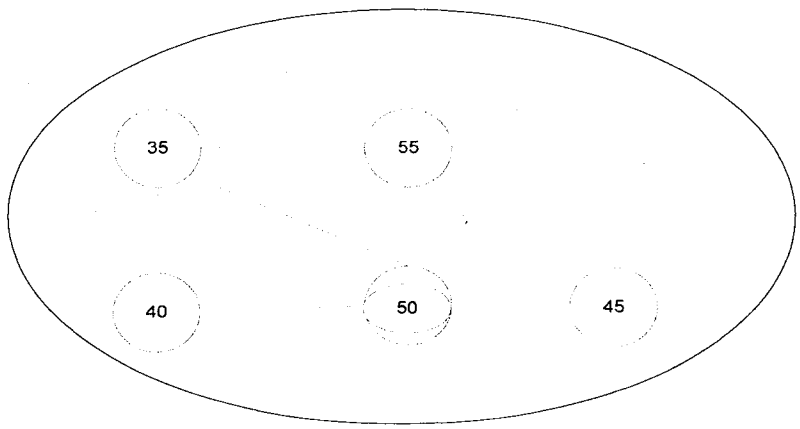

Figure 6.2. Message reaches the clusterhead before reaching the destination node.

Lemma 6.3 For any source and destination (regardless of their locations), a route discovery message is always acknowledged.

Proof: The lemma has the following two cases to be proved.

Case I: When the *sender* and *destination* belong to the same cluster.

Proof: Proof follows from Lemma 6.2. □

Case II: When the *sender* and *destination* belong to two different clusters.

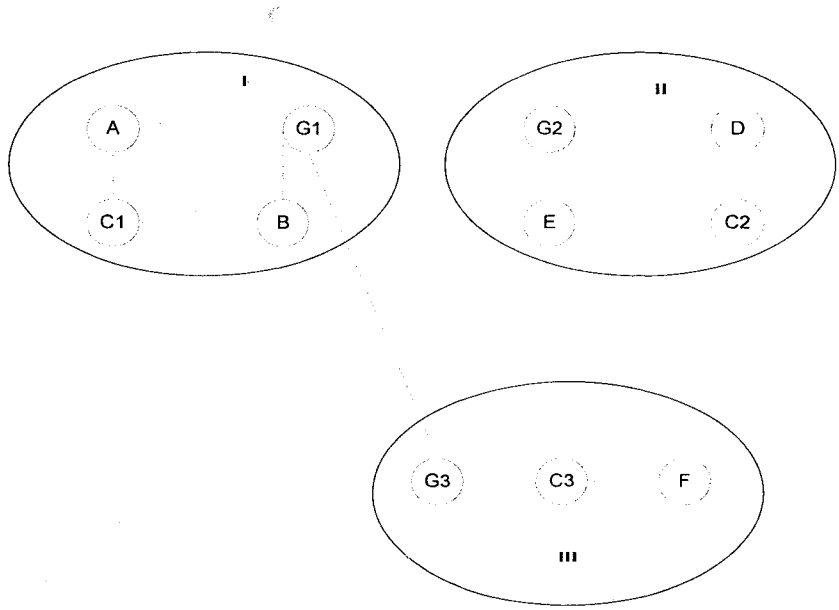

Figure 6.3. Sender and destination in neighboring clusters.

Proof: When a node generates the route discovery (*routedisc*) message, it first sends it to its own clusterhead. If the *sender* and *destination* do not belong to the same cluster, the routing information to the *destination* is not found in the intra-cluster (routing table) of the *sender*'s clusterhead. Then the clusterhead checks for the *destination's* entry in the inter-cluster table (*CG_TABLE* according to our module). The following two sub-cases must be proved:

Case IIa: *Sender* and *destination* belong to two neighboring clusters.

Proof: In Figure 6.3, suppose A is the *sender* and D is the *destination* node. If the clusterhead C1, finds the *destination's* entry in its inter-cluster table, it forwards the message to the corresponding gateway G1 that in turn forwards the message to the clusterhead C2 looking at the entry in its inter-cluster table (*GC_TABLE* according to our algorithm). C2 then acknowledges the *routedisc* message by looking at the entry of *destination* in its routing table with the *shortestpath* message. □

Case IIb: *Sender* and *destination* do not belong to neighboring clusters.

Proof: In Figure 6.4, suppose A is the *sender* and D is the *destination* node. If the clusterhead C1, does not find the *destination's* entry in its inter-cluster table, it forwards the message to all the bordering gateway nodes: G1 and G3. These nodes look into the entry in their inter-cluster table (*GC_TABLE* according to our algorithm) and if no entry is found, forward the message to all the clusterheads whose clusters are connected by these gateway nodes. Only C3 receives the message from both the gateway nodes in our example. The redundant messages are eliminated by Lemma 6.1. C3 in turn checks in its routing table if the entry for the *destination* exists and finds no entry. It then checks in its inter-cluster table and if it does not find an entry, forwards the message to all the bordering gateway nodes except the one from which it got the message. Thus the message reaches the gateway node G2 and then finally reaches the *destination's* clusterhead node C2 following the same procedure. C2 then finds an entry for the *destination* node D in its intra-cluster table. C2 then acknowledges the *sender* of the *routedisc* message with the *shortestpath* message. □

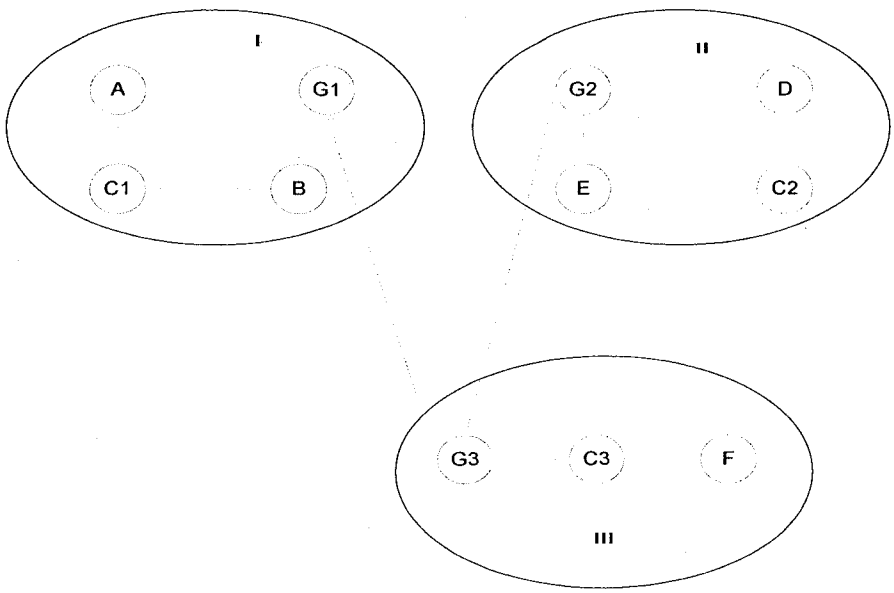

Figure 6.4. Sender and destination in non-neighboring clusters.

Lemma 6.4 If a node moves to another cluster, the route discovery algorithm will be able to find the node in finite time upon a request.

Proof: When a node is in a cluster, it periodically acknowledges a clusterhead that it is still in the cluster. When the node moves out of the cluster, the clusterhead waits for a timeout interval, then removes all the rows with this node as destination from its intra- and inter-cluster routing tables, and updates the same to its boundary gateway nodes so that they can remove the rows from their inter-cluster routing tables. Once the node moves out of a cluster, the following two cases arise:

Case I: The node joins another cluster.

Proof: It acknowledges the new clusterhead's CL_ANN message with a CH_ACCEPT message that it joined its cluster, and the new clusterhead updates its entry in its intra-cluster routing table.

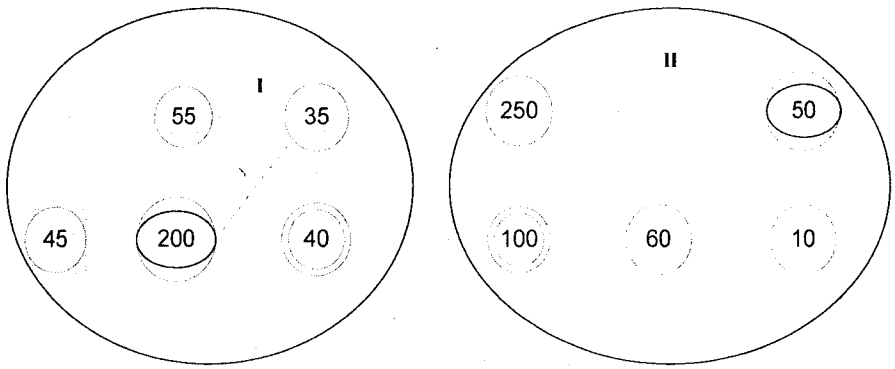

Figure 6.5. A two-cluster network before node 100 moves.

Consider Figure 6.5 that shows the nodes in two clusters before the node mobility occurs. Suppose node 100 moves to cluster I. Assume that the network changes to the one shown in Figure 6.6 due to this movement. Now, 100's routing information is erased from the intra-cluster routing table of node 50 and node 200 enters a new row in its intra-cluster routing table. Suppose node 250 wants to communicate with node 100. It sends the *routedisc* message to its clusterhead node 50. Node 50 does not find an entry in its intra-cluster table for node 100. It then checks its intra-cluster table and finds no entry for node 100. It then broadcasts the message to all its bordering gateways (in this case only node 60). Node 60 does not find an entry in its inter-cluster routing table.

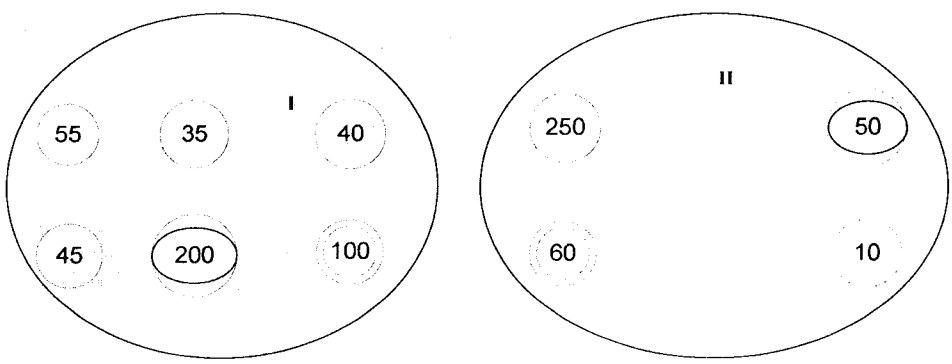

Figure 6.6. The network of Figure 6.5 after node 100 moves.

It would have purged the rows having node 60 in them after receiving the *update* message from the previous clusterhead 50. Node 60 broadcasts the message to all the clusterheads whose clusters it connects (in this case, only node 200). The message reaches the destination node 100 on its way to node 200. Then the node 100 sends an acknowledgement message to node 250 by following the reverse path (similar to Case 2 of *Lemma 6.3).* The route is thus discovered. □

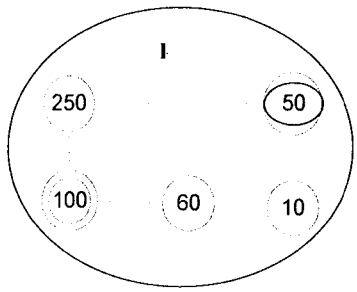

Figure 6.7. A single cluster network before node 100 moves.

Case II: The node itself becomes the clusterhead because it is not in two-hop distance from any clusterhead.

Proof: *Action E.01* makes sure that the node becomes a clusterhead of its own. Figure 6.7 shows the nodes in the network before any node moves. Suppose the network looks like Figure 6.8 after the node moves. Assume that node 250 wants to communicate with node 100. It sends a routedisc message to its clusterhead node 50. Node 50 does not find an entry in its intra-cluster table for node 100. It then checks its intra-cluster table and finds no entry for node 100. It then broadcasts the message to all its bordering gateways (in this case, only node 60). Node 60 does not find an entry in its inter-cluster routing table. It would have purged the rows having node 60 in them after receiving the

update message from the previous clusterhead 50. Node 60 broadcasts the message to all the clusterheads whose clusters it connects (in this case, only node 100). Then the node 100 sends an acknowledgement message to node 250 by following the reverse path (similar to Case II of *Lemma 6.3).* The route is thus discovered. □

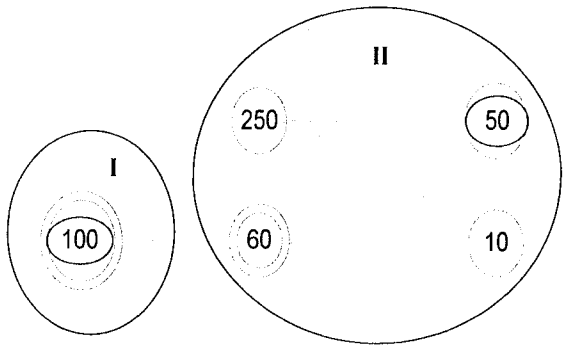

Figure 6.8. The network of Figure 6.7 after node 100 moves.

CHAPTER 7

CONCLUSION

We have presented a route discovery algorithm for MANETs based on link-cluster architecture. The algorithm selects the clusterheads and gateway nodes, and then builds routing tables for nodes both inside and outside the cluster. The proposed protocol guarantees that in a finite number of steps, the network is divided into clusters. The algorithm attempts to minimize the number of clusterheads and gateway nodes to avoid storing redundant data. For intra-cluster routing, the shortest paths are maintained. For inter-cluster routing, we implement routing on-demand (the shortest paths are maintained only for the nodes that need to send packets). The proposed algorithm adapts to arbitrary movement of nodes, and joining and/or leaving of existent nodes.

There are ample opportunities to explore several issues related to the topic of this thesis. This work includes the discovery of a route, forward path set-up, and path maintenance. One can study the next few steps of the complete routing that include reverse path set up and the actual data transmission. This thesis is implemented considering a single-layered cluster network. Performance can be improved by using a hierarchical structure.

BIBLIOGRAPHY

[1] D. J. Baker and A. Ephremides, A Distributed Algorithm for Organizing Mobile Radio Telecommunication Networks. *Proceedings of the* Second International Conference *on Distributed Computer Systems,* April 1981, 476–483.

[2] D. J. Baker and A. Ephremides. The Architectural Organization of a Mobile Radio Network via a Distributed Algorithm. IEEE *Transactions on Communications,* COM–29(11):1694–1701, November 1981.

[3] G. G. Chen, J. W. Branch, B. K. Szymanski. Self-selective routing for wireless ad hoc networks. *Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005),* IEEE *International Conference.* 22-24 August 2005.

[4] C. C. Chiang. Routing in Clustered Multihop, Mobile Wireless Networks. *Proceedings of the* ICOIN, 11, 1996.

[5] C. C. Chiang, H-K Wu, Winston Liu, and Mario Gerla. Routing in Clustered Multihop, Mobile Wireless Networks. *The* IEEE *Singapore International Conference on Networks,* pp.197-211, 1997.

[6] S. R. Das, C. E. Perkins, and E. M. Royer. Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks. *Proceedings of* INFOCOM *2000,* March 2000.

[7] A. Ephremides, J. E. Wieselthier, and D. J. Baker. A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling. *Proceedings of the* IEEE, 75(1):56–73, January 1987.

[8] M. Gerla and J. T. C. Tsai. Multicluster, Mobile, Multimedia Radio Network. *Wireless Networks,* 1(3):255–265, October 1995.

[9] T. Johansson, L. Carr-Motyckova, Bandwidth-constrained Clustering in Ad Hoc Networks, *Proceedings of The Third Annual Mediterranean Ad Hoc Networking Workshop*, Bodrum, Turkey, 27-30 June 2004, 379-385.

[10] Y.B. Ko and N.H. Vaidya, Location-aided routing in mobile ad hoc networks, *Technical report* 98-012, Texas A&M University (1998).

[11] S. J. Lee, C. K. Toh, and M. Gerla. Performance Evaluation of Table-Driven and On-Demand ad hoc routing protocols. *Proceedings of* IEEE *PIMRC'99*, pp. 297–301, September 1999.

[12] C. E. Perkins and E. M. Royer. Ad-Hoc On-Demand Distance Vector Routing. *Proceedings of the Second Annual* IEEE *Workshop on Mobile Computing Systems and Applications,* February 1999, 99–100.

[13] S. J. Philip, J. Ghosh, S. Khedekar, and Chunming Qiao. Scalability analysis of location management protocols for mobile ad hoc networks, *Wireless Communications and Networking Conference, 2004. WCNC. 2004* IEEE, *Department of Computer science and Engineering., State University of New York, Amherst, NY, USA.* 21-25 March 2004

[14] R. A. Ponce, Ashok Kumar, J. L. T. Xihuitl, and M. Bayoumi. Autonomous Decentralized Systems Based Approach to Object Detection in Sensor Clusters*. IEICE/IEEE *Joint Special Section on Autonomous Decentralized Systems.* IEICE *Transactions on Communications.* E88-B(12):4462-4469, 2005.

[15] E. M. Royer and C. K. Toh. A Review of Current Routing Protocols for Ad hoc Mobile Networks. IEEE *Personal Communications,* 6(2):46–55, April 1999.

[16] Zheng Kai, Wang neng, and Liu Ai-fang. Wireless Communications, Networking, and MobileComputing. *Proceedings 2005 International Conference.* Department of Computer Science, East China Normal University, Shanghai, China. 23-26 September 2005.

VITA

Graduate College
University of Nevada, Las Vegas

Shashirekha Yellenki

Home Address:
939 E.Flamingo Rd Apt #35
Las Vegas, NV 89119

Degrees:
Bachelor of Technology, Computer Science, 2004
Jawaharlal Nehru Technological University, India.

Thesis Title: Cluster-Based Route Discovery Protocol

Thesis Examination Committee:
Chairperson, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. Yoohwan Kim, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.