ON DEMAND MULTIPATH ROUTING PROTOCOL WITH LOAD BALANCING FOR ADHOC NETWORKS

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

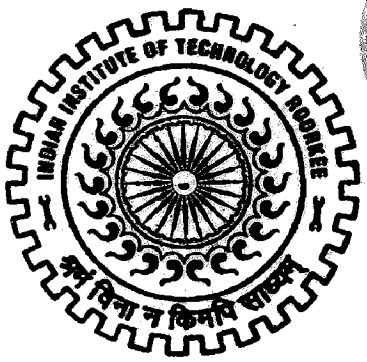
MASTER OF TECHNOLOGY

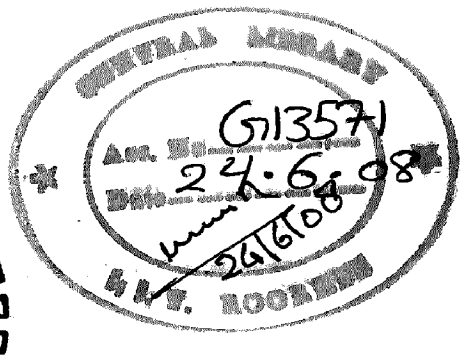
in

INFORMATION TECHNOLOGY

By

NARASIMHA REDDY JAKKAM





DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE - 247 667 (INDIA)

JUNE, 2007

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled **"ON DEMAND MULTIPATH ROUTING PROTOCOL WITH LOAD BALANCING FOR ADHOC NETWORKS"** towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology** in **Information Technology** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India) is an authentic record of my own work carried out during the period from July 2006 to June 2007, under the guidance of **Dr. Manoj Misra, Associate Professor, Department of Electronics and Computer Engineering, IIT Roorkee.**

I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.

Date: 29, JUNE 2007
Place: Roorkee

J. Narasimha Reddy
(Narasimha Reddy Jakkam)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 29-06-07
Place: Roorkee

Manoj Misra
(Dr. Manoj Misra)
Associate Professor
Department of Electronics and Computer Engineering
IIT Roorkee – 247 667

ACKNOWLEDGEMENTS

I would like to extend my heartfelt gratitude to my guide **Dr. Manoj Misra**, Associate Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for his able guidance, regular source of encouragement and assistance throughout this dissertation work. It is his vision and insight that inspired me to carry out my dissertation in the upcoming field of Localization using Image Processing. I would state that the dissertation work would not have been in the present shape without his umpteen guidance and I consider myself fortunate to have done my dissertation under him.

I also extend my sincere thanks to **Dr. D. K. Mehra**, Professor and Head of the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee for providing facilities for the work.

I also wish to thank all my friends for their valuable suggestions and timely help.

Finally, I would like to say that I am indebted to my parents for everything that they have given to me. I thank them for the sacrifices they made so that I could grow up in a learning environment. They have always stood by me in everything I have done, providing constant support, encouragement and love.

J. Narasimha Reddy

Narasimha Reddy Jakkam

ABSTRACT

A Mobile AdHoc Network (MANET) is a collection of mobile nodes that can communicate with each other using multihop wireless links without utilizing any fixed based-station infrastructure and centralized management. Each mobile node in the network acts as both a host generating flows or being destination of flows and a router forwarding flows directed to other nodes.

Multipath routing allows the establishment of multiple paths between a single source and single destination node. It is beneficial to avoid traffic congestion and frequent link breaks in communication because of the mobility of nodes. It results in an increased delivery ratio, smaller end-to-end delays for data packets. This work proposes an On-Demand Multipath Routing protocol with load balancing (ODMRLB) to find multiple node-disjoint paths and to distribute the traffic efficiently among the available routes. Simulation results show that the proposed protocol achieves higher packet delivery ratio and smaller end-to-end delay than NDMR and DSR.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction and Problem Statement

1.1 Introduction

A mobile ad hoc network (MANET) is an infrastructure less network consisting of a set of mobile nodes that are able to communicate with each other in a multi hop manner without the support of any base station or access point. A node in a MANET is not only a node but also a router that is responsible of relaying packets for other nodes. A MANET has the merit that it is quickly deployable. Applications of MANETs include communications in battlefields, disaster rescue operations, and outdoor activities.

1.2 Problem Statement

Since bandwidth may be limited in a wireless network, routing along a single path may not provide enough bandwidth for a connection. However, if multiple paths are used simultaneously to route data, the aggregate bandwidth of the paths may satisfy the bandwidth requirement of the application. Also, since there is more bandwidth available, a smaller end-to-end delay may be achieved

On-demand routing protocols in particular, are widely studied because they consume less bandwidth than proactive protocols. Ad Hoc On-demand Distance Vector (AODV) [1] and Dynamic Source Routing (DSR) [2] are the two most widely studied on-demand ad hoc routing protocols. Previous work [3, 4, 5] has shown limitations of the two protocols. The main reason is that both of them build and rely on a unipath route for each data session. Whenever there is a link break on the active route, both of the two routing protocols have to invoke a route discovery process. On-demand multipath routing protocols can alleviate these problems by establishing multiple paths between a source and a destination in a single route discovery. A new route discovery is invoked only when all of its routing paths fail or when there only remains a single path available.

Node-Disjoint Multipath Routing Protocol (NDMR) reduces routing overhead dramatically and achieves multiple node-disjoint routing paths. NDMR [6] also uses only one path at a time.

The aim of this dissertation work is

- To study the various routing protocols existing for MANETs,
- Analyze some of the proposed multipath routing techniques, and
- To provide an efficient multipath routing mechanism for utilizing the available bandwidth effectively and balancing the load .

This consists of simulating the protocol for performance analysis.

1.3 Organization of Report

Chapter 2 presents overview of routing in adhoc networks i.e discusses existing unipath and multipath routing protocols.

Chapter 3 describes the proposed multipath routing algorithm for finding multiple node-disjoint paths and traffic distribution among the routes.

Chapter 4 discusses about simulation of the protocol in the Global Mobile Information System Simulator (GloMoSim)[7].

Chapter 5 discusses about the simulation results obtained under varying the maximum speed of the nodes and number of sources.

Chapter 6 summarizes the contributions of dissertation followed by the scope for the future work.

This chapter provides background and describes related research efforts and existing problems in ad hoc routing protocols. Section 2.1 gives a general introduction about ad hoc networks. Section 2.2 explains several important concepts, including proactive versus reactive routing approaches and hierarchical routing. Section 2.3 describes some typical ad hoc proactive routing protocols. Section 2.4 presents several typical ad hoc reactive routing protocols. Section 2.5 provides a review of current on-demand multipath routing protocols in wireless ad hoc networks.

2.1 Ad hoc Networks

There are two architectures that allow two wireless stations to communicate with each other. The first one relies on a third fixed party i.e a base station that will hand over the offered traffic from a station to another, as illustrated in Figure 2.1. This same entity will regulate the allocation of radio resources. When a source node wishes to communicate with a destination node, the former notifies the base station, which eventually establishes the communication with the destination node. At this point, the communicating nodes do not need to know about the route from one to the other. All that matters is that both source and destination nodes are within the transmission range of the base station; if one of them loses this condition, the communication will abort.

The second approach, called ad-hoc, does not rely on any stationary infrastructure. All nodes in ad hoc networks are mobile and can be connected dynamically in an arbitrary manner. Each node in such networks behaves as a router and takes part in discovery and maintenance of routes to other nodes.

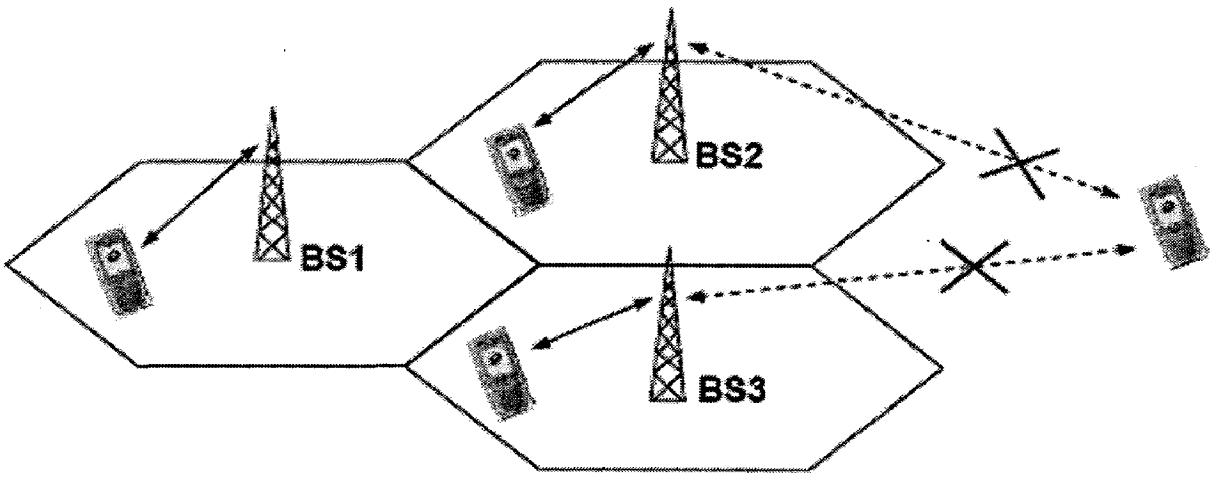


Figure 2.1 Illustration of the infrastructure network model

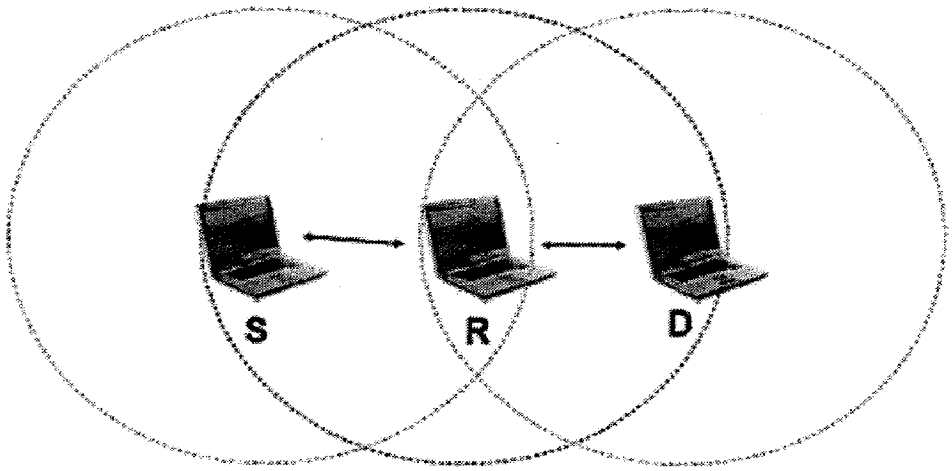


Figure 2.2 Illustration of the infrastructure-less networks

Figure 2.2 illustrates a simple 3-node ad-hoc network. In this figure, a source node S wants to communicate with a destination node D. S and D are not within transmission range of each other. Therefore, they both use the relay node R to forward packets from one to another. R functions as a host and a router at the same time. By definition, a router is an entity that determines the path to be used in order to forward a packet towards

its final destination. The router chooses the next node to which a packet should be forwarded according to its current understanding of the state of the network.

Wireless ad hoc networks can be broadly divided into two categories: quasi-static and mobile. In a quasi-static ad hoc network, nodes are static or portable. However, due to power controls and link failures, the resulting network topology may be dynamic. A typical sensor network [8] is an example of a quasi-static ad hoc network. In mobile ad hoc networks (MANETs), the entire network may be mobile, and nodes may move quickly relative to each other. A major technical challenge in a MANET is the design of efficient routing protocols to cope with the rapid topology changes.

2.2 Routing Classification in Ad Hoc Networks

Routing in wireless ad hoc networks is clearly different from routing found in traditional infrastructure networks. Routing in ad hoc networks needs to take into account many factors including topology, selection of routing path and routing overhead, and it must find a path quickly and efficiently. Ad hoc networks generally have lower available resources compared with infrastructure networks and hence there is a need for optimal routing. Also, the highly dynamic nature of these networks means that routing protocols have to be specifically designed for them, thus motivating the study of protocols that aim at achieving routing stability.

Designing a routing protocol for ad hoc networks is challenging because of the need to take into account two contradictory factors:

- a node needs to know at least the "reachability" information to its neighbours for determining a packet route; and
- the network topology can change quite often.

Furthermore, as the number of network nodes can be large, finding a route to the destinations also requires large and frequent exchange of routing control information among the nodes. Thus, the amount of update traffic can be quite high, and it is even higher when the network includes high mobility nodes, which can impact the route

overhead of routing protocols in such a way that there might be no bandwidth leftover for the transmission of data packets.

In wireless ad hoc networks, the communication range of a node is often limited and not all nodes can directly communicate with one another. Nodes are required to relay packets on behalf of other nodes to allow communication across the network. Since there is no pre-determined topology or configuration of fixed routes, an ad hoc routing protocol is used to dynamically discover and maintain up-to-date routes between communicating nodes.

2.2.1 Proactive versus Reactive Approaches

Ad hoc routing protocols may generally be categorized as being either proactive or on-demand (reactive) according to their routing strategy. Proactive protocols require that nodes in a wireless ad hoc network should keep track of routes to all possible destinations so that when a packet needs to be forwarded, the route is already known and can be used immediately. Any changes in topology are propagated through the network, so that all nodes know of those changes in topology. Examples include "destination-sequenced distance-vector" (DSDV) routing [9], "wireless routing protocol" (WRP) [10].

On-demand protocols only attempt to build routes when desired by the source node so that the network topology is detected as needed (on-demand). When a node wants to send packets to some destination but has no routes to the destination, it initiates a route discovery process within the network. Once a route is established, it is maintained by a route maintenance procedure until the destination becomes inaccessible or until the route is no longer needed. Examples include "ad hoc on-demand distance vector routing" (AODV) [1], "dynamic source routing" (DSR) [2]. Proactive protocols have the advantage that new communications with arbitrary destinations experience minimal delay, but suffer the disadvantage of the additional control overhead to update routing information at all nodes. To cope with this shortcoming, reactive protocols adopt the inverse approach by finding a route to a destination only when needed. Reactive protocols often consume much less bandwidth than proactive protocols, but they will

typically experience a long delay for discovering a route to a destination prior to the actual communication. However, because reactive routing protocols need to broadcast route requests, they may also generate excessive traffic if route discovery is required frequently.

2.2.2 Clustering and Hierarchical Routing

Scalability is one of the important problems in ad hoc networking. Scalability in ad hoc networks can be broadly defined as the network's ability to provide an acceptable level of service to packets even in the presence of a large number of nodes in the network. In proactive routing protocols, when the number of nodes in the network increase, the number of topology control messages increases nonlinearly and they may consume a large portion of the available bandwidth. In reactive routing protocols, large numbers of route requests to the entire network may eventually become packet broadcast storms. Typically, when the network size increases beyond certain thresholds, the computation and storage requirements become infeasible. When mobility is considered, the frequency of routing information updates may be significantly increased, thus worsening the Scalability issues.

One way to address these problems and to produce scalable and efficient solutions is hierarchical routing. Wireless hierarchical routing is based on the idea of organizing nodes in groups and then assigning nodes different functionalities inside and outside a group. Both the routing table size and update packet size are reduced by including in them only part of the network. For reactive protocols, limiting the scope of route request broadcasts also helps to enhance efficiency. The most popular way of building hierarchy is to group nodes geographically close to each other into clusters. Each cluster has a leading node (cluster head) to communicate with other nodes on behalf of these clusters. Example of hierarchical ad hoc routing protocol is "zone routing protocol" (ZRP).

2.3 Review of Ad hoc Proactive Routing Protocols

This section presents brief description for the proactive routing protocol Destination-Sequenced Distance-Vector (DSDV) Routing Algorithm.

2.3.1 Destination-Sequenced Distance-Vector Routing

The Destination-Sequenced Distance-Vector (DSDV) Routing Algorithm [9] is a proactive hop-by-hop distance vector routing protocol, which is based on the idea of the classical Bellman-Ford Routing Algorithm with certain improvements. Every mobile station maintains a routing table that lists all available destinations, the number of hops to reach the destination and the sequence number assigned by the destination node. The sequence number is used to distinguish stale routes from new ones to avoid the formation of loops. The stations periodically transmit their routing tables to their immediate neighbours. A station also transmits its routing table if a significant change has occurred in its table from the last update sent. The update is both time-driven and event-driven.

The routing table updates can be sent in two ways:

- a "full dump" where the full routing table is sent to the neighbours (which could span many packets); or
- an incremental update where only those entries from the routing table that have had a metric change since the last update are sent (and these must fit in a single packet). If there is space in the incremental update packet, then those entries whose sequence number has changed may be included. When the network is relatively stable, incremental updates are sent to avoid extra traffic and full dumps are relatively infrequent. In a fast-changing network, incremental packets can grow large so full dumps will be more frequent.

Each route update packet, in addition to the routing table information, also contains a unique sequence number assigned by the transmitter. The route labelled with the highest (i.e. most recent) sequence number is used. If two routes have the same sequence number then the route with the best metric (i.e. shortest route) is used. Based on past history, the stations estimate the settling time of routes. The stations delay the transmission of a

routing update by settling time so as to eliminate those updates that would occur if a better route were found very soon.

2.4 Review of Ad hoc Reactive Routing Protocols

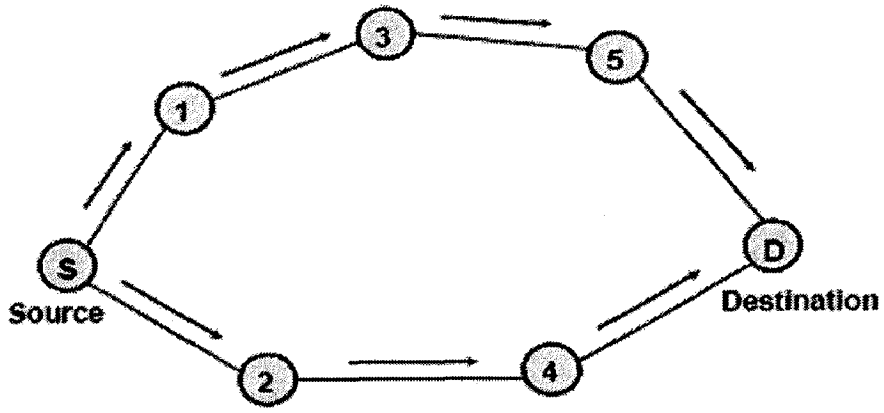
Reactive protocols take a lazy approach to routing. In contrast to proactive routing protocols, all up-to-date routes are not maintained at every node, but instead the routes are created as and when required. When a source wants to send to a destination, it invokes the route discovery mechanisms to find the path to the destination. In this section several typical reactive (on-demand) routing protocols are introduced.

2.4.1 Ad Hoc On-demand Distance Vector Routing (AODV)

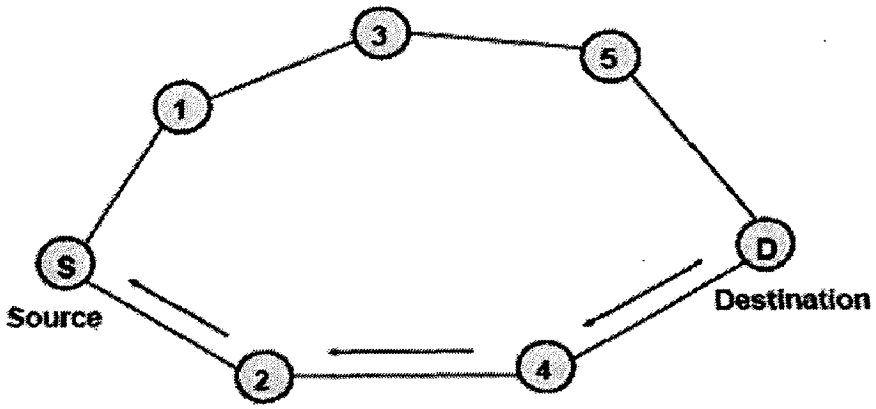
Ad hoc on-demand distance vector (AODV) routing [1] adopts both a modified on-demand broadcast route discovery approach used in DSR [2] and the concept of destination sequence number adopted from destination-sequenced distance-vector routing (DSDV)[9]. When a source node wants to send a packet to some destination and does not have a valid route to that destination, it initiates a path discovery process and broadcasts a route request (RREQ) message to its neighbours. The neighbours in turn forward the request to their neighbours until the RREQ message reaches the destination or an intermediate node that has an up-to-date route to the destination. Figure 2.3(a) illustrates the propagation of the broadcast RREQs in an ad hoc network.

In AODV, each node maintains its own sequence number and a broadcast ID. Each RREQ message contains the sequence numbers of the source and destination nodes and is uniquely identified by the source node's address and a broadcast ID. AODV utilizes destination sequence numbers to ensure loop-free routing and use of up-to-date route information. Intermediate nodes can reply to the RREQ message only if they have a route to the destination whose destination sequence number is greater or equal to that contained in the RREQ message. So that a reverse path can be set up, each intermediate node records the address of the neighbour from which it received the first copy of the RREQ message and additional copies of the same RREQ message are discarded.

Once the RREQ message reaches the destination (or an intermediate node with a fresh route) the destination (or the intermediate node) responds by sending a route reply packet



(a) RREQ messages propagation



(b) RREP message sent back to source

Figure 2.3 Route discovery in AODV

back to the neighbour from which it first received the RREQ message. As the RREP message is routed back along the reverse path, nodes along this path set up forward path entries in their routing tables (Figure 2.3(b)).

When a node detects a link failure or a change in neighbourhood, a route maintenance procedure is invoked: If a source node moves, it can restart the route discovery procedure to find a new route to the destination. If a node along the route moves so that it is no longer contactable, its upstream neighbour sends a link failure notification message to each of its active upstream neighbours. These nodes in turn forward the link failure

notification to their upstream neighbours until the link failure notification reaches the source node.

2.4.2 Dynamic Source Routing (DSR)

Dynamic source routing (DSR) [2] is an on-demand routing protocol for wireless ad hoc networks. DSR is based on the concept of source routing, in which a source node indicates the sequence of intermediate routes in the header of a data packet. Like other on-demand routing protocols, the operation of DSR can be divided into two procedures: route discovery and route maintenance.

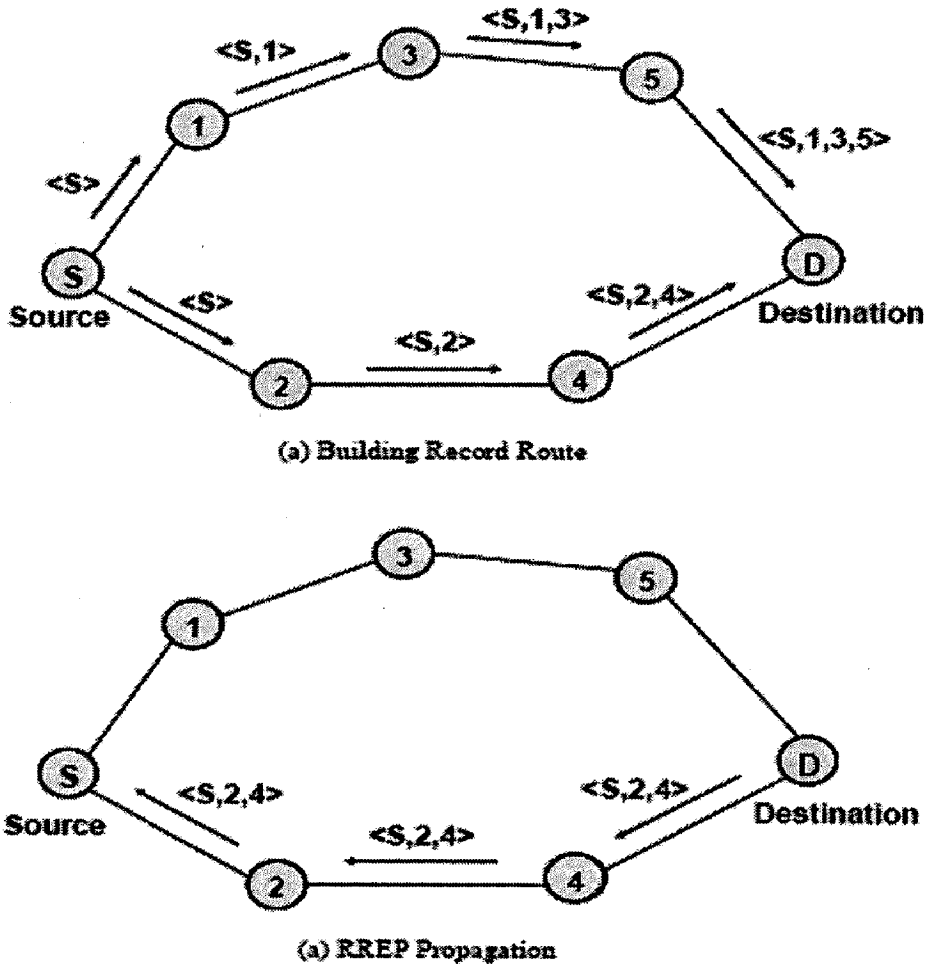


Figure 2.4 Route discovery in DSR

Each node in the network keeps a cache of the source routes that it has learned. When a node needs to send a packet to some destination, it first checks its route cache to determine whether it already has an up-to-date route to the destination. If no route is found, the node initiates the route discovery procedure by broadcasting a route request message to neighbouring nodes. This route request message contains the address of the source and destination nodes, a unique identification number generated by the source node, and a route record to keep track of the sequence of hops taken by the route request message as it is propagated through the network. When an intermediate node receives a route discovery request, it checks whether its own address is already listed in the route record of the route request message. If not, it appends its address to the route record and forwards the route request to its neighbours. Figure 2.4(a) illustrates the formation of the route record as the route request propagates through the network.

When the destination node receives the route request, it appends its address to the route record and returns it to the source node within a new route reply message. If the destination already has a route to the source, it can use that route to send the reply; otherwise, it can use the route in the route request message to send the reply. The first case is for situations where a network might be using unidirectional links and so it might not be possible to send the reply using the same route taken by the route request message. If symmetric links are not supported, the destination node may initiate its own route discovery message to the source node and piggyback the route reply on the new route request message. Figure 2.4(b) shows the transmission of route record back to the source node.

Route maintenance uses route error messages and acknowledgement messages. If a node detects a link failure when forwarding data packets, it creates a route error message and sends it to the source of the data packets. The route error message contains the address of the node that generates the error and the next hop that is unreachable. When the source node receives the route error message, it removes all routes from its route cache that have the address of the node in error. It may initiate a route discovery for a new route if needed. In addition to route error message, acknowledgements are used to verify the

correct operation of links. To reduce the route search overhead, an important optimization is allowing an intermediate node to send a route reply to the source node if it already has an upto- date route to the destination.

2.5 Ad Hoc On-demand Multipath Routing Protocols

Standard on-demand routing protocols in ad hoc wireless networks, such as AODV and DSR, are mainly intended to discover a single route between a source and destination node. When the route disconnects, nodes of the broken route simply drop data packets because no alternate path to the destination is available until a new route is established. Multipath routing is useful for finding multiple paths between a source and destination in a single discovery. These multiple paths between source and destination can be used to compensate for the dynamic and unpredictable topology change in ad hoc networks. Recently, several different multipath routing mechanisms have been proposed. This section introduces some main characteristics of these multipath protocols. AOMDV [12] and AODVM [11] routing protocols are based on the AODV [1] routing protocol, whereas SMR [13] and MSR [14] are based on DSR [2].

2.5.1 Ad hoc On-demand Multipath Distance Vector (AOMDV)

Ad hoc On-demand Multipath Distance Vector (AOMDV) [12] is an extension to the AODV protocol for computing multiple loop-free and link-disjoint paths. The protocol computes multiple loop-free and link-disjoint paths. Loop-freedom is guaranteed by using a notion of "advertised hop count". Link-disjointness of multiple paths is achieved by using a particular property of flooding.

To keep track of multiple routes, the routing entries for each destination contain a list of the next-hops together with the corresponding hop counts. All the next hops have the same sequence number. For each destination, a node maintains the advertised hop count, which is defined as the maximum hop count for all the paths. This is the hop count used for sending route advertisements of the destination. Each duplicate route advertisement received by a node defines an alternative path to the destination. To ensure loop freedom, a node only accepts an alternative path to the destination if it has a lower hop count than

the advertised hop count for that destination. Because the maximum hop count is used, the advertised hop count therefore does not change for the same sequence number. When a route advertisement is received for a destination with a greater sequence number, the next-hop list and advertised hop count are reinitialized.

AOMDV can be used to find link-disjoint routes. To find disjoint routes, each node does not immediately reject duplicate RREQs. Each RREQ carries an additional field called firsthop to indicate the first hop (neighbour of the source) taken by it. Also, each node maintains a first hop list for each RREQ to keep track of the list of neighbours of the source through which a copy of the RREQ has been received. In an attempt to get multiple link-disjoint routes, the destination replies to duplicate RREQs regardless of their first hop. To ensure link-disjointness in the first hop of the RREP, the destination only replies to RREQs arriving via unique neighbours. The trajectories of each RREP may intersect at an intermediate node, but each takes a different reverse path to the source to ensure link-disjointness.

2.5.2 Split Multipath Routing (SMR)

Split Multipath Routing (SMR) proposed in [13] is an on-demand multipath source routing protocol that builds multiple routes using a request/reply cycle. SMR can find an alternative route that is maximally disjoint from the source to the destination. When the source needs a route to the destination but no route information is known, it floods the Route Request (RREQs) message to the entire network in order to find maximally disjoint paths, so the approach has a disadvantage of transmitting more RREQ packets. Because this packet is flooded, several duplicates that traversed through different routes reach the destination. The destination node selects multiple maximally disjoint routes and sends Route Reply (RREP) packets back to the source via the chosen routes. In order to choose proper maximally disjoint route paths, the destination must know the entire path of all available routes. Therefore, SMR uses the source routing approach where the information of the nodes that comprise the route is included in the RREQ packet.

SMR is similar to DSR, and is used to construct maximally disjoint paths. Unlike DSR, intermediate nodes do not keep a route cache, and therefore, do not reply to RREQs. This is to allow the destination to receive all the routes so that it can select the maximally disjoint paths. Maximally disjoint paths have as few links or nodes in common as possible. Duplicate RREQs are not necessarily discarded. The algorithm only selects two routes. In the algorithm, the destination sends a RREP for the first RREQ it receives, which represents the shortest delay path. The destination then waits to receive more RREQs. From the received RREQs, the path that is maximally disjoint from the shortest delay path is selected. If more than one maximally disjoint path exists, the shortest hop path is selected. If more than one shortest hop path exists, the path whose RREQ was received first is selected. The destination then sends an RREP for the selected RREQ.

2.5.3 Multipath Source Routing (MSR)

Multipath Source Routing (MSR) [14, 15] is an extension of the on-demand DSR [2] protocol. It consists of a scheme to distribute traffic among multiple routes in a network. MSR uses the same route discovery process as DSR with the exception that multiple paths can be returned, instead of only one.

When a source requires a route to a destination but no route is known (in the cache), it will initiate a route discovery process by flooding a RREQ packet throughout the network. A route record in the header of each RREQ records the sequence of hops that the packet passes. An intermediate node contributes to the route discovery by appending its own address to the route record. Once the RREQ reaches the destination, a RREP will reverse the route in the route record of the RREQ and traverse back through this route.

Each route is given a unique index and stored in the cache, so it is easy to pick multiple paths from there. Independence between paths is very important in multipath routing, therefore disjoint paths are preferred in MSR. As MSR uses the same route discovery process as DSR, where the complete routes are in the packet headers, looping will not occur. When a loop is detected, it will be immediately eliminated.

Since source routing is used in MSR, intermediate nodes do nothing but forward the packet according to the route in the packet-header. The routes are all calculated at the source. A multiple-path table is used for the information of each different route to a destination. This table contains for each route to the destination: the index of the path in the route cache, the destination ID, the delay and the calculated load distribution weight of a route. The traffic to a destination is distributed among multiple routes. The weight of a route simply represents the number of packets sent consecutively on that path.

2.5.4 Ad hoc On-demand Distance Vector Multipath Routing

Ad hoc On-demand Distance Vector Multipath Routing (AODVM) [11] is an extension to AODV for finding multiple node disjoint paths. Instead of discarding the duplicate RREQ packets, intermediate nodes are required to record the information contained in these packets in the RREQ table. For each received copy of an RREQ message, the receiving intermediate node records the source that generated the RREQ, the destination for which the RREQ is intended, the neighbour that transmitted the RREQ, and some additional information in the RREQ table. Furthermore, intermediate relay nodes are precluded from sending an RREP message directly to the source.

When the destination receives the first RREQ packet from one of its neighbours, it updates its sequence number and generates an RREP packet. The RREP packet contains an additional field called "*last hop ID*" to indicate the neighbour from which the particular copy of RREQ packet was received. This RREP packet is sent back to the source via the path traversed by the RREQ. When the destination receives duplicate copies of the RREQ packet from other neighbours, it updates its sequence number and generates RREP packets for each of them. Like the first RREP packet, these RREP packets also contain their respective last hop nodes' IDs.

When an intermediate nod e receives an RREP packet from one of its neighbours, it deletes the entry corresponding to this neighbour from its RREQ table and adds a routing entry to its routing table to indicate the discovered route to the originator of the RREP packet (the destination). The node, then, identifies the neighbour in the RREQ table via

which, the path to the source is the shortest, and forwards the RREP message to that neighbour. The entry corresponding to this neighbour is then deleted from the RREQ table. In order to ensure that a node does not participate in multiple paths, when nodes overhear any node broadcasting an RREP message, they delete the entry corresponding to the transmitting node from their RREQ tables.

Intermediate nodes make decisions on where to forward the RREP messages (unlike in source routing) and the destination, which is in fact the originator of these messages, is unaware as to how many of these RREP messages that it generated actually made it back to the source. Thus, it is necessary for the source to confirm each received RREP message by means of a Route Confirmation message (RRCM). The RRCM message can, in fact, be added to the first data packet sent on the corresponding route and will also contain information with regards to the hop count of the route, and the first and last hop relays on that route.

Chapter 3 OnDemand Multipath Routing with Load Balancing

ODMRLB is proposed to find multiple nose-disjoint paths and to distribute the traffic efficiently among the available paths. It follows NDMR [6] approach to find multiple node-disjoint paths with minimum broadcast overhead.

When a source node wants to communicate with a destination node, it checks its route table to confirm whether it has a valid route to the destination. If so, it sends the packet to the appropriate next hop towards the destination. However, if the node does not have a valid route to the destination, it must initiate a route discovery process. To begin such a process, the source creates a RREQ (Route Request) packet. This packet contains message type, source address, current sequence number of source, destination address, the broadcast ID and route path. The broadcast ID is incremented every time when the source node initiates a RREQ. In this way, the broadcast ID and the address of the source node form a unique identifier for the RREQ.

Finding node-disjoint multiple paths with low broadcast overhead is not an easy task when the network topology is unknown and changing dynamically. This section briefly describes the mechanism of ODMRLB that enables path accumulation during a multipath route discovery cycle and records the shortest routing hops to minimize its routing overhead and achieve multiple node-disjoint routing paths. ODMRLB routing computation has three key components to avoid introducing a broadcast flood in MANETs:

- Path accumulation;
- Decreasing multipath broadcast routing packets;
- Selecting node-disjoint paths.

3.1 Path Accumulation

The main goal of ODMRLB is to build multiple node-disjoint paths and distributing the load efficiently among the routes. To achieve this goal, the destination must know the entire routing path list of all available routes so that it can select the right node-disjoint route paths from the candidate paths. When the RREQ packets are generated or forwarded by the nodes in the network, each node appends its own address to the routing request packets. When a RREQ packet arrives at its destination, the destination is responsible for judging whether or not the routing path is a node-disjoint path. After confirming a node-disjoint path, the destination generates a Route Reply (RREP) packet that contains the node list of the whole route path and unicasts it back towards the source that originated the RREQ message along the reverse route path. When an intermediate node receives a RREP, it updates its routing table entry and its reverse routing table entry by using the nodes list of the whole route path contained in the RREP.

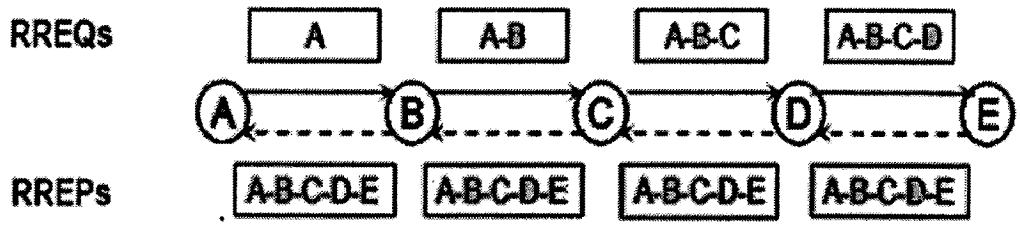


Figure 3.1 Path Accumulation in ODMRLB

As an example, consider five nodes A, B, C, D and E as shown in Figure 3.1 Node A wants to send data to node E. Since A does not have a route for E in its routing table, it broadcasts a route request. Node B receives the route request, appends its own address to the request, and forwards the request since it also has no route to E. Similarly, when node C and node D receive the RREQ, they append their address to the request and forward it. When the request reaches destination E, node E checks the path accumulation list (A-B-C-D) from the RREQ and judges whether or not the routing path is a node-disjoint path. If it is, node E generates a RREP packet that contains the path accumulation list of the

whole route path and unicasts it back to the source that originated the RREQ message along the reverse route path. If not, node E discards the received RREQ.

3.2 Decreasing Broadcast Routing Overhead

In DSR and AODV, if a source node does not know a route to a destination, it will initiate a route discovery by flooding a Route Request (RREQ) message. The RREQ message carries the source ID and the RREQ sequence number. When an intermediate node receives a RREQ, if it is the first time that the node receives this RREQ message, then the node will broadcast the RREQ message again. Otherwise, the node will drop the RREQ packet.

In ODMRLB, using this method of broadcasting RREQ, the possibility of finding node-disjoint multiple paths is almost zero so a novel method is used_. The reason is that later duplicate RREQ packets, which may come from a different path, are dropped. However, if all of the duplicate RREQ packets are re-broadcast, this will lead to a routing packet broadcast storm and decrease dramatically the performance of the ad hoc networks. In order to avoid this problem, a novel approach recording the Shortest Routing Hops of Loop-free Paths is implemented to decrease routing broadcast overhead.

When a node receives a RREQ packet for the first time, it checks the path accumulation list from the packet and calculates the number of hops from the source to itself and records the number as the shortest number of hops in its reverse route table entry. If the node receives the RREQ duplicate again, it computes the number of hops from the source to itself and compares it to the number of the shortest hops recorded in its reverse route table entry. If the number of hops is larger than the shortest number of hops in its reverse route table entry, the node drops the RREQ packet. Otherwise (less than or equal to), the node appends its own address to the route path list of the RREQ packet and broadcasts the RREQ packet to its neighbouring nodes.

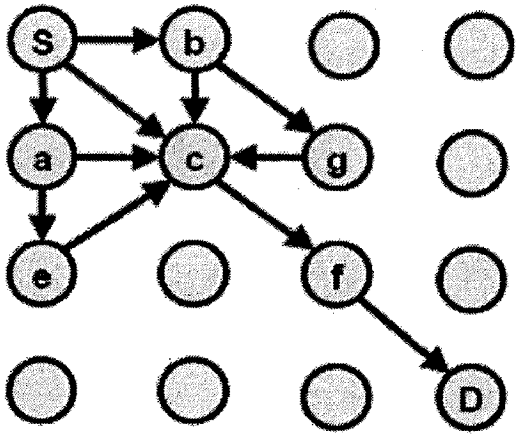


Figure 3.2 Shortest Routing Hops of Loop-free Paths

For example, in Figure 3.2, from source node S to node c there are five route paths: S-c, S-b-c, S-a-c, S-b-g-c, S-a-e-c. The numbers of hops are 1, 2, 2, 3 and 3 respectively. When node c receives the RREQ packet at the first time from path S-c, it records 1 as the shortest number of hops in its reverse route table entry. When the node c receives the RREQ duplicates from the other four route paths, it calculates the number of hops and compares it to the shortest number of hops in its reverse route table entry. Because the numbers of hops of route list of the four route paths are all greater than 1, the four RREQ duplicate packets are dropped.

From the example it can be seen that "recording the shortest routing hops" approach results in most of the RREQ packets being discarded in the process of discovering multiple node-disjoint paths. Furthermore, the approach can also avoid forming loop paths. This is a novel and practical approach to guarantee loop-free paths as well as to dramatically decrease the routing overhead.

Figure 3.3 illustrates the route request process with low overhead in the entire network. Source S broadcasts a route request packet. Each intermediate node uses the approach with low routing overhead to propagate and discard packets. Therefore, only seven packets (S-c-f-D, S-a-i-g-D, S-b-e-h-D, S-c-i-g-D, S-c-e-h- D, S-c-f-g-D, S-c-f-h-D) can reach the destination D. Most of packets are discarded. However, not all of paths packets that arrive in destination are node-disjoint. In next section how to choose node-disjoint paths will be discussed.

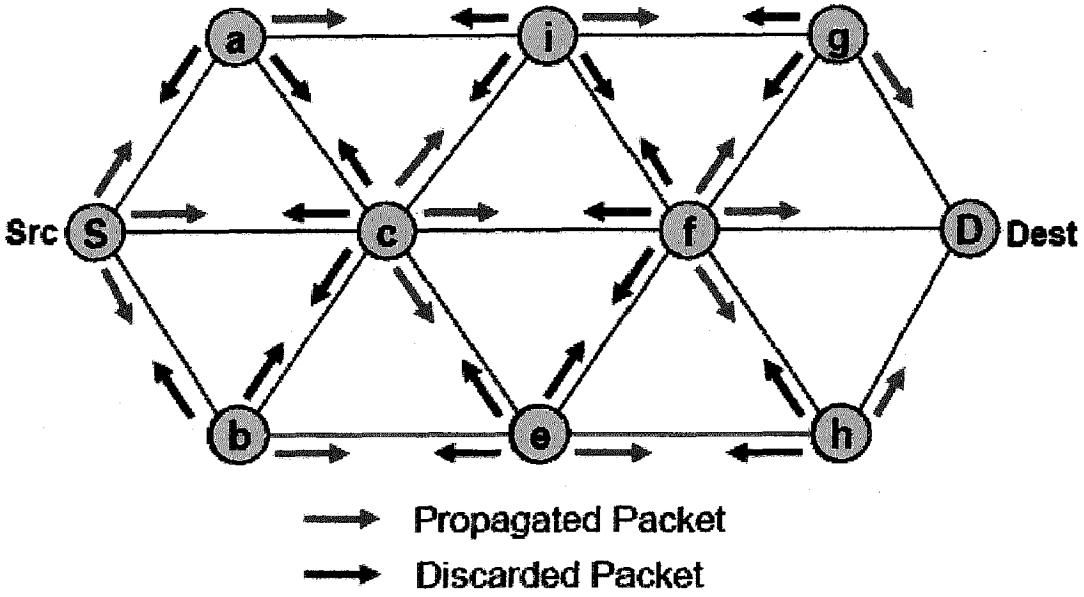


Figure 3.3 Route Request Process with Low Overhead

3.3 Selecting Node-Disjoint Paths

In the algorithm of selecting node-disjoint paths, the destination is responsible for selecting and recording multiple node-disjoint route paths. In order to decrease the overhead of the route table in each node, the number of node-disjoint routing paths has been limited to three although more than three node-disjoint routes can be searched. In Figure 3.4, its three node-disjoint route paths are: S-a-i-g-D, S-c-f-D, S-b-e-h-D. When receiving the first RREQ packet (the shortest route path: S-c-f-D), the destination records the list of node IDs for the entire route path in its reverse route table and sends a RREP that includes the route path towards the source along the reverse route. When the destination receives a duplicate RREQ, it will compare the whole route path in the RREQ to all of the existing node-disjoint route paths in its route table entry. If there is not a common node (except source and destination) between the route path from the current received RREQ and any node-disjoint route path recorded in the destination's reverse route table entry, the route path of the current RREQ (such as S-a-i-g-D or S-b-e-h-D) satisfies the requirement of node-disjointness and is recorded in the reverse route table of the destination. Otherwise, the route path (such as paths: S-c-i-g-D, S-c-e-h-D, S-c-f-g-D, S-c-f-h-D) and the current received RREQ are discarded.

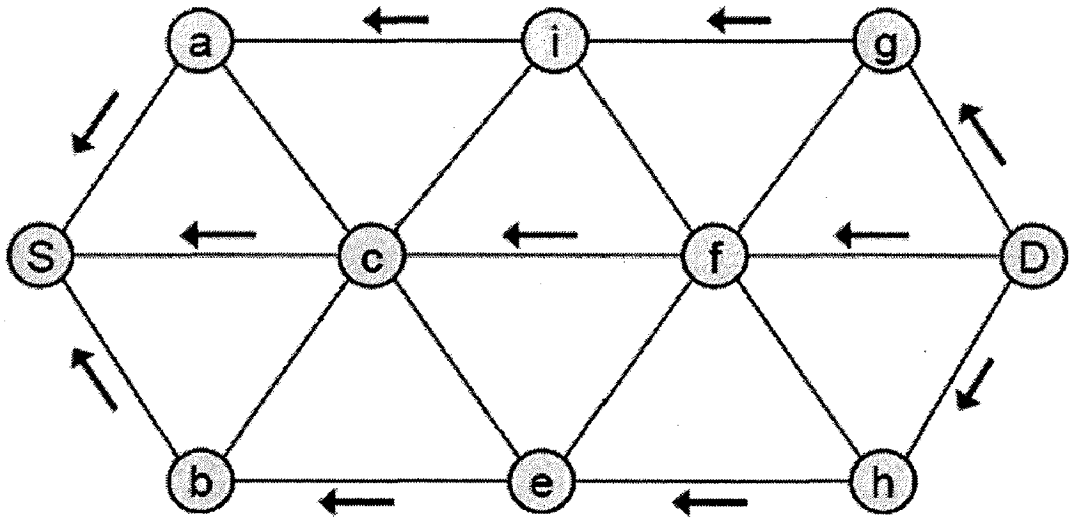


Figure 3.4 Node-Disjoint Paths

Because the node IDs of the entire path are included in the RREP, each intermediate node receiving a RREP can record some necessary information from the path to its route table before forwarding the RREP. At first, the intermediate node sets up a forward path entry to the destination in its route table and a reverse path entry to the source in its reverse route table. According to the information in path IDs list, the forward path entry records the IP address of the destination and the IP address of the neighbour from which the RREP arrived. The reverse path entry records the IP address of the source and the IP address of the next hop to the source. Finally the intermediate node forwards the RREP towards the source node along the reverse route path. When the RREP arrives at the source node, it does not need to be forwarded. The source node records the next hop to destination into its multiple route forward path entry. After the first RREP arrives at the source, the newly established route can now be used to send the data packets.

3.4 Load Balancing

Upon receiving the RREPs, source node routes data packets to destination node through the available paths. The source node can get a maximum of RMAX reply packets. Data packets are routed over the routes in such a way that the total number of congested packets in each route is maintained equally. The source node records the total number of packets sent to each route. When the host wants to choose a route for packet transmission, it checks these numbers. In addition, it has information about the size route of each route. Therefore, it chooses a route based on the product of number of packets transmitted and the size of the route. The route for which the calculated product is less is chosen for that packet transmission. By using this algorithm, traffic is guaranteed to be shared equally over multiple paths.

This section describes the data structures and main functions used to implement ODMRLB.

4.1 Data Structures

The following are the data structures used in implementing ODMRLB protocol.

An enumerated data structure called 'Packet Type' is used to identify type of a packet i.e., ROUTE_REQUEST packet, ROUTE_REPLY packet or ROUTE_ERROR packet.

```
typedef enum {
        ROUTE_REQUEST,
        ROUTE_REPLY,
        ROUTE_ERROR
} PacketType;
```

The following structure 'RouteRequest' represents the contents of a route request packet. *pktType* is to be set to ROUTE_REQUEST to identify the packet as Route Request packet . <*srcAddr* > represents the originator of the route request . <*targetAddr*> is the destination for which the root has to be discovered. < *seqNumber*> and <*srcAddr*> combinely used to uniquely identify a packet. Each intermediate node adds its ID to the <*path*> variable. So the <*path*> variable will be having a sequence of nodes through which the packet is propagated to reach the destination.

```
typedef struct{
   PacketType pktType;
   NODE_ADDR srcAddr;
   NODE_ADDR targetAddr;
   int seqNumber;
   int hopCount;
   NODE_ADDR path [MAX_SR_LEN];
   int broadcastid;
} RouteRequest;
```

The structure 'RouteReply', given next represents the contents of a route reply packet. *pktType* is to be set to ROUTE_REPLY to identify the packet as RouteReply packet. <*srcAddr* > represents the originator of the route reply packet. <*targetAddr*> is the source for which the root has discovered . The <*path*> variable will have a sequence of nodes through which the data packets can be routed from <*targetAddr*> to <*srcAddr*>. <*segLeft*> represents the number of nodes remaining through which the reply has to be propagated to reach the destination.

```
typedef struct{
   PacketType pktType;
   NODE_ADDR targetAddr;          /* Source of the route */
   NODE_ADDR srcAddr;             /* Destination of the route */
   int hopCount;
   int segLeft;
   NODE_ADDR path [MAX_SR_LEN];
} RouteReply;
```

The structure 'RouteError', given next represents the contents of a route error packet. *pktType* is to be set to ROUTE_ERROR to identify the packet as RouteError packet. <*srcAddr* > represents the originator of the route error packet . <*destAddr* > is the source of the broken root. <*unreachableAddr*> is the immediate downstream of the broken link The <*path*> variable will have a sequence of nodes through which the error packets has to be routed to reach the source of the broken route. When an intermediate node forwarding a packet detects through Route Maintenance that the next hop along the route for that packet is broken, if the node has another route to the packet's destination in its Route Cache, the node salvages (redirects) the packet rather than discarding it. To salvage a packet, the node replaces the original source route on the packet with the route from its Route Cache. The node then forwards the packet to the next node indicated

along this source route. If this packet is salvaged in this way, the *salvaged* bit has to be set to TRUE otherwise, it will be set to FALSE.

```
typedef struct
{
   PacketType pktType;
   NODE_ADDR srcAddr;              /* Originator of the Route Error */
   NODE_ADDR destAddr;             /* Source of the broken route */
   NODE_ADDR unreachableAddr;      /* Immediate downstream of broken link */
   int hopCount;
   BOOL salvaged;
   NODE_ADDR path [MAX_SR_LEN];
}RouteError;
```

Finally, the structure ODMRLB_Stats defines the total statistics at each node in the network.

```
typedef struct
{
   int numRequestSent;        /* Total no of route request pkts transmitted*/
   int numReplySent;          /* Total number of route reply packets transmitted */
   int numErrorSent;          / * Total number of route error packets transmitted */
   int numDataSent;           /* Total no of data pkts sent at the source */
   int numDataTxed;           /* Total number of data packets transmitted */
   int numDataReceived;       /* Total no. of data pkts received at destination*/
   int numRoutes;             /* Total number of routes discovered*/
   int numHops;               /* number of hops*/
   int numLinkBreaks;         /* Total number of link breaks discovered*/
   int numSalvagedPackets;    /*Total number of packets salvaged*/
   int numDroppedPackets;     /*Total number of packets dropped*/
  } ODMRLB_Stats;
```

4.2 Functions

- *RoutingOdmrlbHandleProtocolPacket*: This function is called when a packet is received from MAC layer. It checks whether the packet corresponds to ROUTE_REQUEST packet, ROUTE_REPLY packet, or ROUTE_ERROR packet and calls the appropriate function.

- *RoutingOdmrlbHandleRequest:* This function is called when the *RoutingOdmrlbHandleProtocolPacket* function receives a packet of type ROUTE_REQUEST. It checks whether the node receiving the packet is the destination for the packet. If it is not for that node it just broadcasts the packet by using the function *RoutingOdmrlbRelayRREQ*. If it is for that node, it stores the route in its reply cache table and initiates route reply by using the function *RoutingOdmrlbInitiateRREP* if it is the first route request packet received or if the path is non-disjoint with the other paths available for the source.

- *RoutingOdmrlbHandleReply:* This function is called when the *RoutingOdmrlbHandleProtocolPacket* function receives a packet of type ROUTE_REPLY. It checks whether this node is the destination of the packet .If this is the destination of the packet, this route is inserted into the route cache .If this node is not the destination of the packet it forwards the packet by using the function *RoutingOdmrlbRelayRREP* .

- *RoutingOdmrlbHandleError* :This function is called when the *RoutingOdmrlbHandleProtocolPacket* function receives a packet of type ROUTE_ERROR. It deletes routes in cache that use the broken link. If this node is the intermediate node of the broken route, then forwards the packet by using the function *RoutingOdmrlbRelayRERR* . If this node is the source of the broken route, it discards the packet.

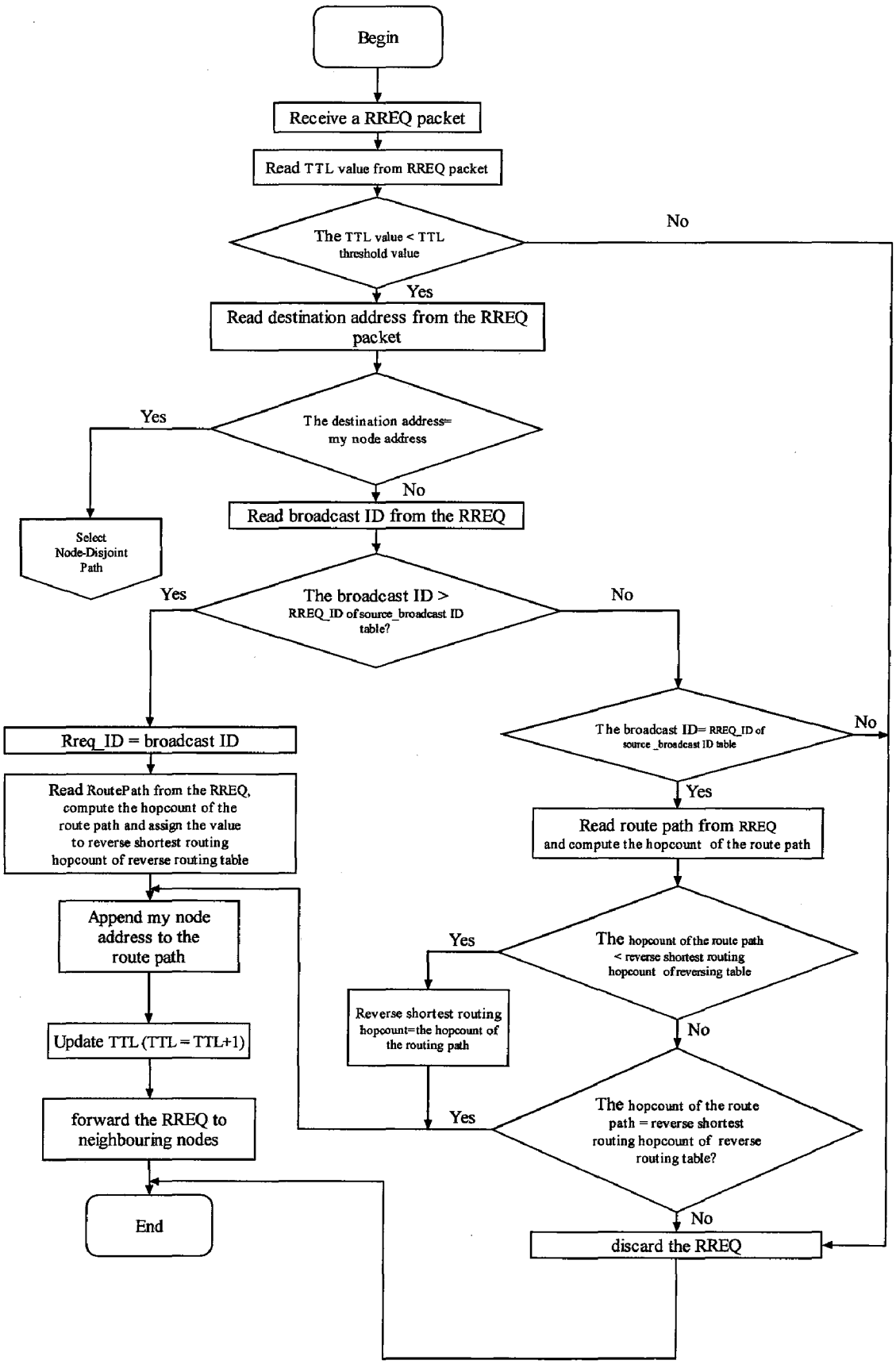


Figure 4.1 Flowchart of reducing broadcast routing overhead

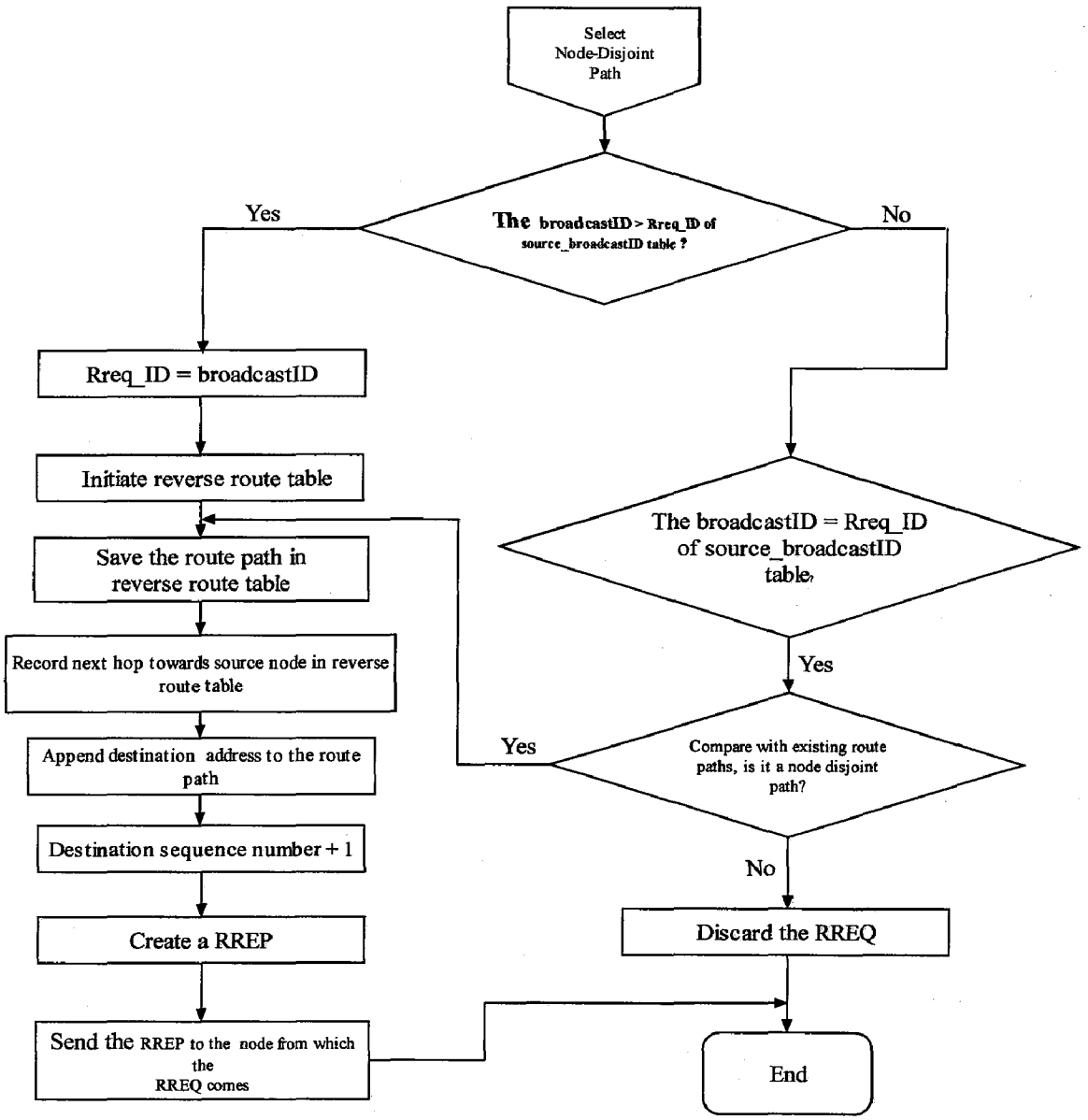


Figure 4.2 Flowchart of selecting node-disjoint paths

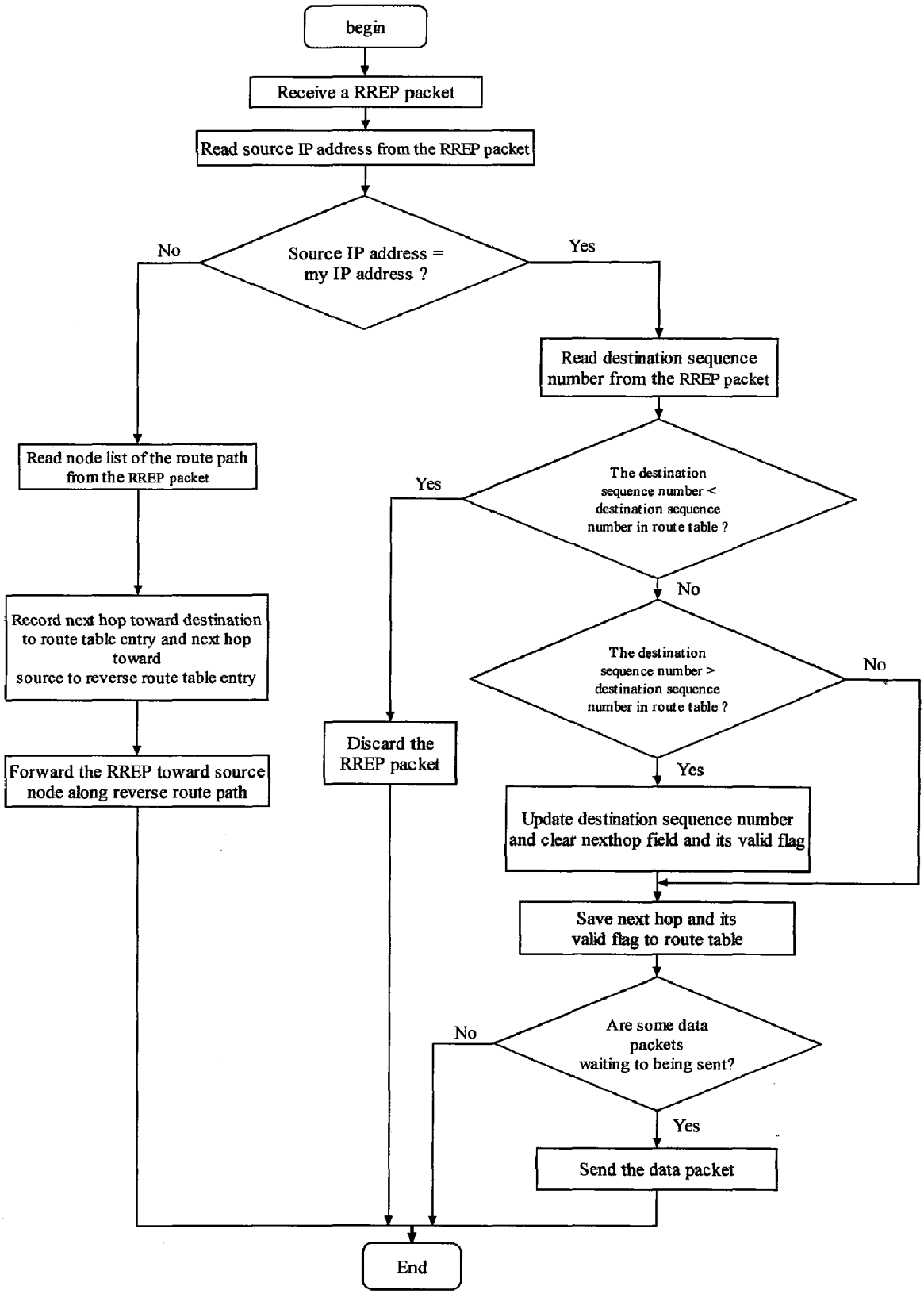


Figure 4.3 Flowchart of processing an incoming RREP packet

- *RoutingOdmrlbHandleProtocolEvent* : Whenever a timer expires for an event Routing*Odmrlb*HandleProtocolEvent function is called . It checks for the type of events such as NETWORK_FlushTables, NETWORK_CheckReplied .
  If the type of the event is NETWORK_CheckRequest, then the source node distributes the packets in to multiple paths in the inverse ratio of their hopcount using the function *RoutingOdmrlbStartTransmission.*

- *RoutingOdmrlbInitiateRREQ*: Initiate a Route Request packet when no route to destination is known.
- *RoutingOdmrlbRetryRREQ*: Send RREQ again after not receiving any RREP.
- *RoutingOdmrlbRelayRREQ*: Forward (re-broadcast) the Route Request.
- *RoutingOdmrlbInitiateRREP*: Destination of the route sends Route Reply in reaction to Route Request.
- *RoutingOdmrlbRelayRREP* : Forward the Route Reply.
- *RoutingOdmrlbInitiateRERR* : The node that detects the link break sends a Route Error back to the source.
- *RoutingOdmrlbRelayRERR* : Forward the Route Error packet.

The flow chart of reducing routing overhead is illustrated in Figure 4.1.The flow chart of selecting node-disjoint paths is shown in Figure 4.2. A flowchart of processing an incoming RREP packet is illustrated in Figure 4.3.

4.3 Input parameters

The input parameters that need to be specified in *config.in* file before starting the simulation are given in Table 1. After specifying the input configuration parameters, data sessions are specified in the application configuration (*app.conf*) file using the Constant Bit Rate (CBR) traffic generator. Table 2 shows the parameters in the *app.conf* file.

SIMULATION-TIME	300S
TERRAIN-DIMENSIONS	(1000, 1000)
NUMBER-OF-NODES	50
MOBILITY	RANDOM-WAYPOINT
MOBILITY-WP-PAUSE	50S
MOBILITY-WP-MIN-SPEED	0
MOBILITY-WP-MAX-SPEED	10
NETWORK-PROTOCOL	IP
ROUTING-PROTOCOL	ODMRLB
APP-CONFIG-FILE	./app.conf
APPLICATION-STATISTICS	YES
ROUTING-STATISTICS	YES
GUI-OPTION	YES
NODE-PLACEMENT	RANDOM

Table 4.1 input parameters for the simulation procedure

Traffic Generator	Source node	Destination node	items to send	Item size	Interval	Start time	End time
CBR	1	5	100	512	1.0S	0S	300S
CBR	2	6	100	512	1.5S	20S	200S
CBR	4	9	100	512	0.1S	200S	300S
CBR	3	8	100	512	0.1S	100 S	200S

Table 4.2 application specification parameters

In order to compare and evaluate performances of the three protocols (ODMRLB,NDMR and DSR) , two parameters are varied in the simulations:

- Maximum velocity of the nodes
- Number of sources

At first, simulations are carried out by keeping the number of sources constant and varying the velocity. The number of nodes and sources are 50 and 20 respectively.
Then, the number of sources is varied from 5 to 25 in intervals of 5 for 50 nodes . When varying the number of sources, velocity is kept at a uniform rate of 0-20m/s.

The following metrics are used in varying scenarios to evaluate the three different protocols:

- *Packet delivery ratio:* The ratio of the data packets delivered to the destinations to those generated by the CBR sources.
- *Average delay of data packets:* This includes all possible delays from the moment the packet is generated to the moment it is received by the destination node.

5.1 Varying Velocity

The first set of experiments varies the velocity for 20 sources of 50 nodes network. The mobility was varied to see how it affects the different metrics that are measured. The packet sending rate is fixed at 10 packets / sec. The results are collected at constant speeds of 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 m/s.

5.1.1 Packet Delivery Ratio

Packet delivery ratio is defined as ratio of the data packets delivered to the destination to those generated by the CBR sources. Packet delivery ratio is a very important metric since it shows the loss rate, which in turn affects the maximum throughput of the network. The packet delivery ratio of the three protocols is shown in Figure 5.1. The Figure depicts the variation of the packet delivery ratio as a function of velocity of nodes.

As the velocity of the nodes increases, the probability of link failure increases and hence the number of packet drops also increases. ODMRLB has much higher packet delivery ratio than both NDMR and DSR. More than 95% data packets of NDMR can be delivered to specified destinations in all of mobility conditions in the 50-node network. DSR have a similar low delivery ratio situation in that only 65% sent packets are received at higher speeds. The reason is that ODMRLB has multiple paths with node-disjoint ness. As the load is distributed through multiple node-disjoint paths in the inverse ratio of their path length, more bandwidth is available. Hence the packet delivery ratio is high.

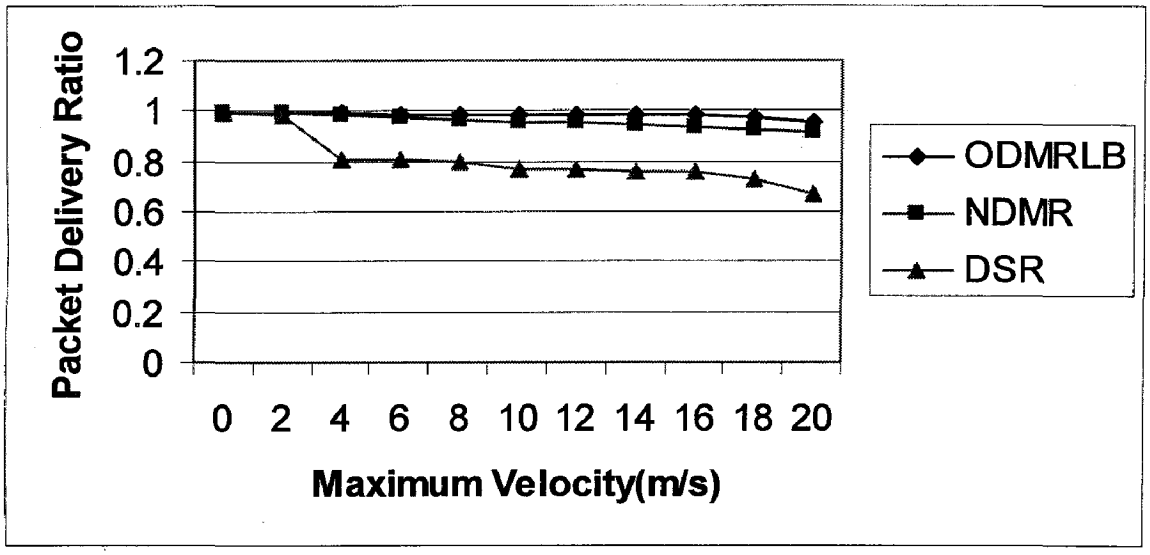


Figure 5.1 Maximum Velocity VS Packet Delivery Ratio

5.1.2 Average end-to-end delay of data packets

The average end-to-end delay includes all possible delays from the moment the packet is generated to the moment it is received by the destination node.

Generally, there are three factors affecting end-to-end delay of a packet:

(1) Route discovery time, which causes packets to wait in the queue before a route path is found;

(2) Buffering waiting time, which causes packets to wait in the queue before they can be transmitted;

(3) The length of routing path. The more number of hops a data packet has to go through, the more time it takes to reach its destination node.

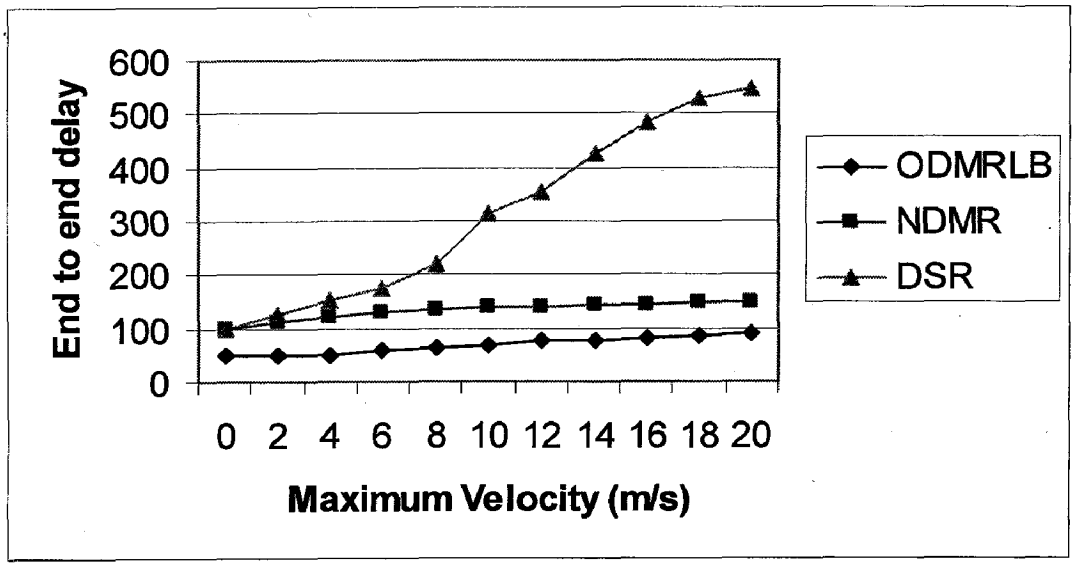


Figure 5.2 Maximum Velocity VS End to end delay

Figure 5.2 depicts the variation of the average end-to-end delay as a function of velocity of nodes. It can be seen that the general trend of all curves is an increase in delay with the increase of velocity of nodes. The reason is mainly that high mobility of nodes results in an increased probability of link failure that causes an increase in the number of routing rediscovery processes. This makes data packets have to wait for more time in its queue until a new routing path is found. The delay of ODMRLB remains approximately equal at all mobile velocities. Delay in DSR increases quickly as velocity increases. This is because availability of alternate node-disjoint routing paths in ODMRLB eliminates route discovery latency that contributes to the delay when active route fails. In addition, the source node distributes data packets in the available node-disjoint routing paths to avoid congestion. This reduces the waiting time of data packets in queue.

5.2 Varying Number of Sources

The second set of experiments varies the number of sources with a random velocity of 0-20 m/s for 50 nodes. The network load is varied by changing the number of sources. The packet sending rate is still fixed at 10 packets / second. The number of sources is varied from 5 to 25 in intervals of 5 for 50 nodes.

5.2.1 Packet Delivery Ratio

The packet delivery ratio of the three protocols is shown in Figure 5.3. The Figure describes the variation of the packet delivery ratio as a function of the number of sources. It can be seen that the packet delivery ratio for ODMRLB has better performance than those of both NDMR and DSR with the increase in the number of sources. When the number of sources increases, DSR drops a larger fraction of the packets.

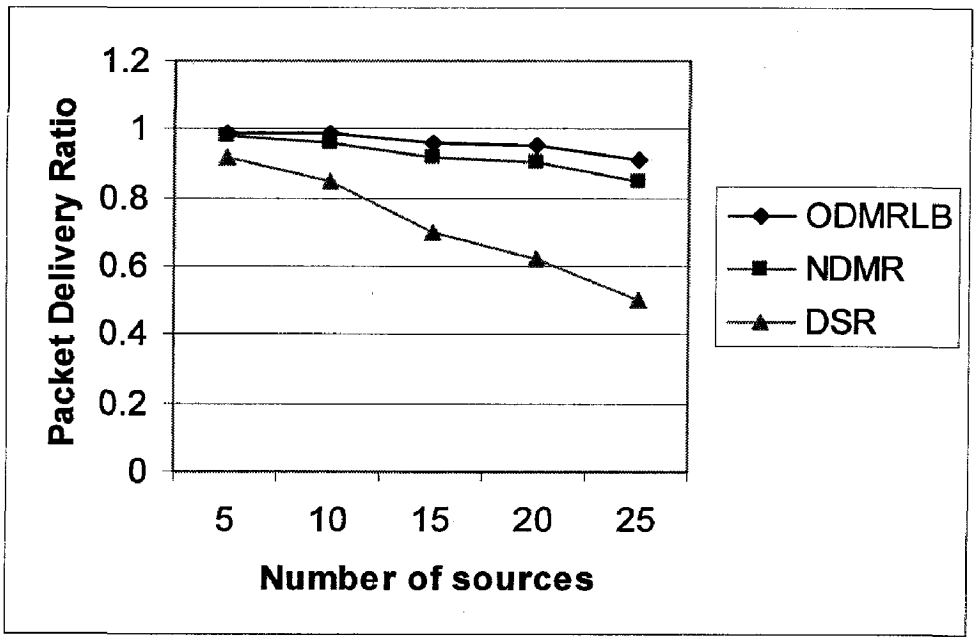


Figure 5.3 Number of sources VS Packet Delivery Ratio

5.2.2 Average end-to-end delay of data packets

Figure 5.4 depicts the variation of the average end-to-end delay as a function of the number of sources. It can be seen that ODMRLB has a lower average delay than both NDMR and DSR. The primary reason is that the number of route discoveries is reduced in ODMRLB. Although ODMRLB has a low number of route discoveries, its delay also increases gradually with the increase of number of source. The reason is that increase of the numbers of sources leads to higher network load traffic in the ad hoc networks. Because of the limitation of a constrained wireless bandwidth, packets that will be sent or forwarded have to stay in buffers and wait for a longer time to get a radio channel available in order to avoid collisions in the air.

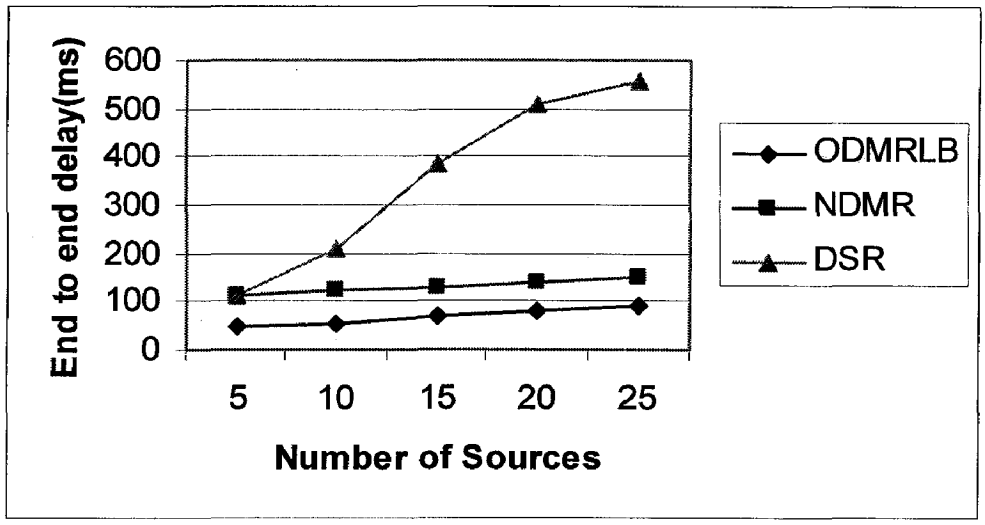


Figure 5.4 Number of sources VS End to end delay

6.1 Conclusions

An ad hoc wireless network is a collection of mobile nodes that communicate with each other by forming a multi-hop radio network and maintaining connectivity management without an existing network infrastructure. Such networks are expected to play increasingly important roles in future civilian and military applications. Design of efficient and reliable routing protocols in such network are challenging issues. The goal of this research is to explore efficient multipath routing in mobile ad hoc networks.

ODMRLB protocol is designed and implemented to overcome the shortcomings of on-demand existing unipath and multipath routing protocols. It is evident from simulation results that ODMRLB outperforms NDMR, DSR. ODMRLB has a higher packet delivery ratio, lower end-to-end delay than NDMR, DSR. These characteristics make the protocol suitable for reliable real time data and multimedia communication applications in ad hoc networks.

6.2 Future Scope

The research work focuses on node-disjoint multipath routing in mobile ad hoc networks. Other important aspects, which need to be further investigated, are:

• Multicast Routing

Multicast is the process of sending packets from a transmitter to multiple destinations identified by a single address. The packets of each multicast group are forwarded according to a multicast tree. Multicast routing in MANET is also hard since the network topology changes quite frequently. Therefore, frequent maintenance of the multicast tree will result in a substantial amount of control overhead. How to reduce routing overhead has to be considered when designing multicast routing.

• Distributed Security

Due to the broadcast nature of radio communication, wireless networks are susceptible to eavesdropping, malicious jamming and interference, which a well-designed physical layer should be able to avoid. Because usually there are no central control and no trusted authorities in an ad hoc network, how to secure key distribution and manage data encryption and authentication has to be considered when designing a secure mechanism of ad hoc networks.

• Effect of quality of wireless links

Because nodes move in and out of each other's range, the network topology changes frequently. The network's dynamic nature, combined with adverse wireless link's effects, raises issues that are difficult to address. In the physical layer, some techniques are needed to adapt to rapidly changing channel characteristics to make wireless link quality less sensitive to node performance.

References

[1] Charles E. Perkins and Elizabeth M. Royer, "Ad-Hoc On Demand Distance Vector Routing", In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, IEEE Computer Society, February 1999, pp. 90–100.

[2]David B. Johnson and David A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks", In *Mobile Computing*, edited by Tomasz Imielinski and Hank Korth, chapter 5, Kluwer Academic Publishers, 1996 , pp. 153–181.

[3] Charles E. Perkings, Elizabeth M.Royer, and Samir R.Das, "Performance Comparison of Two On-Demand Routing Protocols for Ad Hoc Networks", *IEEE Personal Communications*, February 2001, pp. 16 - 28.

[4] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J.Jetcheva, "A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols", In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking* (MobiCom'98), Dallas, Texas, USA, October 25-30, 1998, pp. 1 – 13.

[5]S. R. Das, R. Castaeda, J. Yan, and R. Sengupta, "Comparative performance evaluation of routing protocols for mobile ad hoc networks," in *Proceedings of 7th International Conference on Computer Communications and Networks (IC3N)*, October 1998, pp. 153–161.

[6]Xuefei Li and Laurie Cuthbert, "On-demand Node-Disjoint Multipath Routing in Wireless Ad hoc Network," 29th Annual IEEE International Conference on Local Computer Networks (LCN'04), 2004, pp. 419-420.

[7]Mario Gerla, Lokesh Bajaj, Mineo Takai, Rajat Ahuja, and Rajive Bagrodia, "GloMoSim: A Scalable Network Simulation Environment", Technical Report 990027, University of California, 13, 1999

[8] D.Estrin, R.Govindan, J.Heidemann, and S.Kumar, "Next Century Challenges: Scalable Coordination in Sensor Networks", *ACM Mobicom,* 1999, pp. 263 - 270.

[9] C. E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers", In *Proceedings of ACM SIGCOMM,* 1994, pp. 234-244.

[10] S. Murthy and J.J. Garcia-Luna-Aceves, "An efficient routing protocol for wireless networks", *ACM Mobile Networks and Applications Journal,* 1996, pp. 183-197.

[11] Z.Ye, S.V. Krishnamurthy, and S.K. Tripathi, "A Framework for Reliable Routing in Mobile Ad Hoc Networks", *IEEE INFOCOM (2003),* vol.1, 30 March-3 April 2003, pp. 270 - 280.

[12] Mahesh K. Marina and Samir R. Das, "On-demand Multipath Distance Vector Routing in Ad Hoc Networks", In *Proceedings of the 9th IEEE International Conference on Network Protocols (ICNP),* November 2001, pp. 14 – 23.

[13] S.J.Lee and M.Gerla, "Split Multipath Routing with Maximally Disjoint Paths in Ad Hoc Networks", In *Proceedings of the IEEE ICC,* 2001, pp. 3201-3205.

[14] L. Wang, Y. Shu, M. Dong, L. Zhang, and O. Yang, "Adaptive Multipath Source Routing in Ad hoc Networks", *IEEE ICC 2001,* vol.3, June 2001, pp. 867-871.

[15] L. Wang, Y. Shu, Z. Zhao, L. Zhang, and O. Yang, "Load Balancing of Multipath Source Routing in Ad hoc Networks", in *Proceedings of IEEE ICCC'02,* vol.5, April 2002, pp. 3197 - 3201.

APPENDIX: SOURCE CODE LISTING

Odmrlb.h

```
#ifndef _ODMRLB_H_
#define _ODMRLB_H_

#include "ip.h"
#include "nwcommon.h"
#include "main.h"

#define ODMRLB_MAX_SR_LEN                    9

/* Broadcast jittering time to avoid collisions */
#define ODMRLB_BROADCAST_JITTER              10 * MILLI_SECOND

/* Max time between route requests */
#define ODMRLB_MAX_REQUEST_PERIOD            10 * SECOND

/* Length of one backoff period */
#define ODMRLB_REQUEST_PERIOD                500 * MILLI_SECOND

/* TO for non propagating request */
#define ODMRLB_RING0_REQUEST_TO              30 * MILLI_SECOND

/* Saved in Request table for this amount of time */
#define ODMRLB_FLUSH_INTERVAL                30 * SECOND

// Maximum time a packet will be buffered waiting for a route.
#define ODMRLB_MAX_PACKET_BUFFER_TIME        11110 * SECOND

#define ODMRLB_MAX_TTL                       255

#define ROUTE_MAX 5

#define IPOPT_ODMRLB  217

/* ODMRLB option fields for IP header */
typedef struct {
   unsigned char salvagedBit;
   unsigned char segmentLeft;
} ODMRLBIpOptionType;

/* Type of packet */
typedef enum {
    ODMRLB_ROUTE_REQUEST,
    ODMRLB_ROUTE_REPLY,
    ODMRLB_ROUTE_ERROR
} ODMRLB_PacketType;

typedef struct
{
    ODMRLB_PacketType pktType;
    NODE_ADDR srcAddr;
```

```
    NODE_ADDR targetAddr;
    int seqNumber;
    int hopCount;
    NODE_ADDR path[ODMRLB_MAX_SR_LEN];
} ODMRLB_RouteRequest;

typedef struct
{
    ODMRLB_PacketType pktType;
    NODE_ADDR targetAddr;                    /* Source of the route */
    NODE_ADDR srcAddr;                       /* Destination of the route
*/
    int hopCount;
    int segLeft;
    NODE_ADDR path[ODMRLB_MAX_SR_LEN];
} ODMRLB_RouteReply;

typedef struct
{
    ODMRLB_PacketType pktType;
    NODE_ADDR srcAddr;                   /* Originator of the Route Error
*/
    NODE_ADDR destAddr;                  /* Source of the broken route */
    NODE_ADDR unreachableAddr;           /* Immediate downstream of
broken link */
    int hopCount;
    BOOL salvaged;
    NODE_ADDR path[ODMRLB_MAX_SR_LEN];
} ODMRLB_RouteError;

typedef struct RCE
{
    NODE_ADDR destAddr;
    int hopCount;                          /* Hop length to the destAddr
*/
    NODE_ADDR path[ODMRLB_MAX_SR_LEN];
    struct RCE *prev;
    struct RCE *next;
} ODMRLB_RouteCacheEntry;

typedef struct
{
    ODMRLB_RouteCacheEntry *head;
    int count;                           /* Count of current entries */
} ODMRLB_RouteCache;

typedef struct RRCE
{
    NODE_ADDR destAddr;
    int hopCount;                          /* Hop length to the destAddr
*/
    NODE_ADDR path[ODMRLB_MAX_SR_LEN];
    clocktype destReached;
    struct RRCE *prev;
    struct RRCE *next;
} ODMRLB_RouteReplyCacheEntry;
```

```
typedef struct
{
    ODMRLB_RouteReplyCacheEntry *head;
    int count;                              /* Count of current entries */
} ODMRLB_RouteReplyCache;

typedef struct RQE
{
    NODE_ADDR destAddr;
    clocktype lastRequest;                  /* Time when last sent a request
*/
    clocktype backoffInterval;              /* No additional Req for this
time */
    int ttl;
    struct RQE *next;
} ODMRLB_RequestTableEntry;

typedef struct
{
    ODMRLB_RequestTableEntry *head;
    int count;
} ODMRLB_RequestTable;

typedef struct STE
{
    NODE_ADDR srcAddr;
    int seqNumber;
    NODE_ADDR prevNode;
    int hopCount;
    struct STE *next;
} ODMRLB_RequestSeenEntry;

typedef struct
{
    ODMRLB_RequestSeenEntry *front;
    ODMRLB_RequestSeenEntry *rear;
    int count;
} ODMRLB_RequestSeen;

typedef struct fifo
{
    NODE_ADDR destAddr;
    clocktype timestamp;
    Message *msg;
    struct fifo *next;
} ODMRLB_BUFFER_Node;

typedef struct
{
    ODMRLB_BUFFER_Node *head;
    int size;
} ODMRLB_BUFFER;
typedef struct
{
    NODE_ADDR destAddr;
    int ttl;
} ODMRLB_CR;
```

```
typedef struct
{
    /* Total number of route request packets transmitted */
    int numRequestSent;

    /* Total number of route reply packets transmitted */
    int numReplySent;

    /* Total number of route error packets transmitted */
    int numErrorSent;

    /* Total number of data packets originated as the source */
    int numDataSent;

    /* Total number of data packets tranmitted */
    int numDataTxed;

    /* Total number of data packets received as the destination */
    int numDataReceived;

    int numRoutes;

    int numHops;

    int numLinkBreaks;

    int numSalvagedPackets;
    int numDroppedPackets;


} ODMRLB_Stats;

typedef struct glomo_network_ODMRLB_str {
    ODMRLB_RouteCache routeCacheTable;
    ODMRLB_RouteReplyCache routeReplyCacheTable;
    ODMRLB_RequestTable requestTable;
    ODMRLB_RequestSeen requestSeenTable;
    ODMRLB_BUFFER buffer;
    int seqNumber;
    ODMRLB_Stats stats;
} GlomoRoutingODMRLB;

void RoutingODMRLBInit(
    GlomoNode *node,
    GlomoRoutingODMRLB **ODMRLBPtr,
    const GlomoNodeInput *nodeInput);

void RoutingODMRLBFinalize(GlomoNode *node);

void RoutingODMRLBHandleRequest(GlomoNode *node, Message *msg,
                                int ttl);
void RoutingODMRLBHandleReply(
    GlomoNode *node, Message *msg, NODE_ADDR destAddr);

void RoutingODMRLBHandleError( GlomoNode *node, Message *msg, NODE_ADDR
                               srcAddr, NODE_ADDR destAddr);
```

```
void RoutingODMRLBInitRouteCache(ODMRLB_RouteCache *routeCache);

void RoutingODMRLBInitRequestSeen(ODMRLB_RequestSeen *requestSeen);

void RoutingODMRLBInitRequestTable(ODMRLB_RequestTable *requestTable);

void RoutingODMRLBInitSeq(GlomoNode *node);

void RoutingODMRLBInitBuffer(ODMRLB_BUFFER *buffer);

void RoutingODMRLBInitStats(GlomoNode *node);

void RoutingODMRLBDeleteSeenTable(ODMRLB_RequestSeen *requestSeen);

BOOL RoutingODMRLBCheckRouteExist(NODE_ADDR destAddr,
                                  ODMRLB_RouteCache *routeCache);

BOOL RoutingODMRLBLookupRequestSeen(NODE_ADDR srcAddr, int seq,
                                ODMRLB_RequestSeen *requestSeen);

BOOL RoutingODMRLBLookupRequestTable(NODE_ADDR destAddr,
                                  ODMRLB_RequestTable *requestTable);

void RoutingODMRLBInsertRequestSeen(GlomoNode *node, NODE_ADDR srcAddr,
                                int seq, NODE_ADDR prevNode,
                          int hopCount, ODMRLB_RequestSeen *requestSeen);

void RoutingODMRLBInsertRouteCache(NODE_ADDR destAddr,
   int hopCount,NODE_ADDR *path, ODMRLB_RouteCache *routeCache);

ODMRLB_RouteCacheEntry *RoutingODMRLBInsertRCInOrder(
                NODE_ADDR destAddr,  int hopCount, NODE_ADDR *path,
          ODMRLB_RouteCacheEntry *old, ODMRLB_RouteCacheEntry *last);

void RoutingODMRLBInsertRequestTable(NODE_ADDR destAddr,
                                  ODMRLB_RequestTable *requestTable);
ODMRLB_RequestTableEntry *RoutingODMRLBInsertRTInOrder(
          NODE_ADDR destAddr,ODMRLB_RequestTableEntry *old);
void RoutingODMRLBInsertBuffer(Message *msg, NODE_ADDR destAddr,
                               ODMRLB_BUFFER *buffer);
ODMRLB_BUFFER_Node *RoutingODMRLBInsertBufInOrder(Message *msg,
                            NODE_ADDR destAddr, ODMRLB_BUFFER_Node *old);
BOOL RoutingODMRLBCompareRoute(NODE_ADDR destAddr,
    int hopCount, NODE_ADDR *path, ODMRLB_RouteCache *routeCache);
void RoutingODMRLBDeleteRouteCache(GlomoNode *node,
  NODE_ADDR fromHop, NODE_ADDR nextHop, ODMRLB_RouteCache *routeCache);


void RoutingODMRLBRemoveOldPacketsFromBuffer(ODMRLB_BUFFER *buffer);

BOOL RoutingODMRLBDeleteBuffer(NODE_ADDR destAddr,
                                  ODMRLB_BUFFER *buffer);

void RoutingODMRLBDeleteRequestTable(NODE_ADDR destAddr,
                                ODMRLB_RequestTable *requestTable);
```

```
BOOL RoutingODMRLBCheckDataSeen(
    GlomoNode *node, NODE_ADDR *header, int currentHop);

BOOL RoutingODMRLBCheckRequestPath(
    GlomoNode *node, NODE_ADDR *path, int currentHop);

NODE_ADDR *RoutingODMRLBGetRoute(NODE_ADDR destAddr,
                                 ODMRLB_RouteCache *routeCache);

int RoutingODMRLBGetHop(NODE_ADDR destAddr,
                  ODMRLB_RouteCache *routeCache);

int RoutingODMRLBGetSeq(GlomoNode *node);

BOOL RoutingODMRLBCheckUnprocessedPath(GlomoNode *node,
                 int currentHop,int segmentLeft, NODE_ADDR *header);

Message *
RoutingODMRLBGetBufferedPacket(NODE_ADDR destAddr,
                                   ODMRLB_BUFFER *buffer);
BOOL RoutingODMRLBLookupBuffer(NODE_ADDR destAddr,
                                   ODMRLB_BUFFER *buffer);
void RoutingODMRLBUpdateRequestTable(NODE_ADDR destAddr,
                                 ODMRLB_RequestTable *requestTable);
void RoutingODMRLBUpdateTtl(NODE_ADDR destAddr,
                            ODMRLB_RequestTable *requestTable);
BOOL RoutingODMRLBCheckRequestTable(NODE_ADDR destAddr,
                                  ODMRLB_RequestTable *requestTable);

clocktype RoutingODMRLBGetBackoff(NODE_ADDR destAddr,
                                  ODMRLB_RequestTable *requestTable);

void RoutingODMRLBHandleProtocolPacket(
    GlomoNode *node, Message *msg, NODE_ADDR srcAddr,
    NODE_ADDR destAddr, int ttl);

void RoutingODMRLBHandleProtocolEvent(GlomoNode *node, Message *msg);



void RoutingODMRLBRouterFunction(
                          GlomoNode *node,    Message *msg,
                          NODE_ADDR destAddr,BOOL *packetWasRouted);

void RoutingODMRLBPeekFunction(GlomoNode *node, const Message *msg);
void RoutingODMRLBPacketDropNotificationHandler(
 GlomoNode *node, const Message* msg, const NODE_ADDR nextHopAddress);
void RoutingODMRLBSendReply(GlomoNode *node, Message *msg);
int RoutingODMRLBCheckDisjointRouteExist(GlomoNode *node,
                                                 Message *msg);
int RoutingODMRLBComparePath(int hopCount1,NODE_ADDR *path1,
                                   int hopCount2,NODE_ADDR *path2);
int selectRouteNumber(NODE_ADDR destAddr,
                  ODMRLB_RouteCache *routeCache,int numSent[]);
int RoutingODMRLBGetHop2(NODE_ADDR destAddr,
                  ODMRLB_RouteCache *routeCache,int routeNum);
NODE_ADDR *RoutingODMRLBGetRoute2(NODE_ADDR destAddr,
```

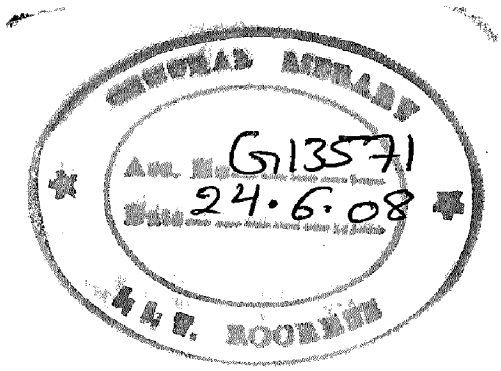
```
                    ODMRLB_RouteCache *routeCache,int routeNum);
void RoutingODMRLBTransmitData2(GlomoNode *node, Message *msg,
                                NODE_ADDR destAddr, int routeNum);
void RoutingODMRLBStartTransmission(GlomoNode *node, Message *msg,
                                                    NODE_ADDR destAddr);
void RoutingODMRLBSetTimer(
    GlomoNode *node, long eventType, ODMRLB_CR cr, clocktype delay);
void RoutingODMRLBInitiateRREQ(GlomoNode *node, NODE_ADDR destAddr);
void RoutingODMRLBRetryRREQ(GlomoNode *node, NODE_ADDR destAddr,
                                                              int ttl);
void RoutingODMRLBTransmitData(GlomoNode *node, Message *msg,
                                                    NODE_ADDR destAddr);
void RoutingODMRLBRelayRREQ(GlomoNode *node, Message *msg, int ttl);
void RoutingODMRLBInitiateRREP(GlomoNode *node, Message *msg);
void RoutingODMRLBInitiateRREPbyIN(GlomoNode *node, Message *msg);
void RoutingODMRLBRelayRREP(GlomoNode *node, Message *msg);
void RoutingODMRLBInitiateRERR(GlomoNode *node, NODE_ADDR destAddr,
                  NODE_ADDR unreachableAddr, NODE_ADDR *errorPath);
void RoutingODMRLBRelayRERR(GlomoNode *node, Message *msg);
void RoutingODMRLBSalvageData(GlomoNode *node, Message *msg);
void RoutingODMRLBSalvageRERR(GlomoNode *node, NODE_ADDR targetAddr,
                    NODE_ADDR srcAddr, NODE_ADDR unreachableAddr);
void RoutingODMRLBGratuitousRREP(GlomoNode *node, NODE_ADDR srcAddr,
            NODE_ADDR destAddr, NODE_ADDR *old, int count, int length);
void AddCustomODMRLBIpOptionFields(GlomoNode* node, Message* msg);
ODMRLBIpOptionType* GetPtrToODMRLBIpOptionField(Message* msg);
extern double ceil(double x);
BOOL RoutingODMRLBNeednotForwardRequest(NODE_ADDR srcAddr,
                          int seq,NODE_ADDR prevNode,int hopCount,
                                    ODMRLB_RequestSeen *requestSeen);
void RoutingODMRLBInitRouteReplyCache(
                          ODMRLB_RouteReplyCache  *routeReplyCache);
void RoutingODMRLBInsertRouteReplyCache(NODE_ADDR destAddr,
     int hopCount,NODE_ADDR *path,ODMRLB_RouteReplyCache *routeCache);

ODMRLB_RouteReplyCacheEntry  *RoutingODMRLBInsertRCReplyInOrder(
                      NODE_ADDR destAddr,int hopCount,NODE_ADDR *path,
                              ODMRLB_RouteReplyCacheEntry *old,
                              ODMRLB_RouteReplyCacheEntry *last);

void RoutingODMRLBDeleteRouteReplyCache(
      ODMRLB_RouteReplyCache *routeCache, NODE_ADDR destAddr);


#endif /* _ODMRLB_H_ */
```

Odmrlb.pc




```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <math.h>

#include "api.h"
#include "structmsg.h"
#include "fileio.h"
#include "message.h"
#include "network.h"
#include "odmrlb.h"
#include "ip.h"
#include "nwip.h"
#include "nwcommon.h"
#include "application.h"
#include "transport.h"
#include "java_gui.h"

void RoutingODMRLBHandleRequest(GlomoNode *node, Message *msg, int ttl)
{
      GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                                          networkData.networkVar;

      GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                                             routingProtocol;
      ODMRLB_RouteRequest *rreq = (ODMRLB_RouteRequest*)
                                    GLOMO_MsgReturnPacket(msg);
    IpHeaderType *ipHdr = (IpHeaderType *)GLOMO_MsgReturnPacket(msg);

      /* If destination of the route (dest sends Reply to every
      requests) */
      if (rreq->targetAddr == node->nodeAddr)
      {
            RoutingODMRLBSendReply(node,msg);
      }


      /* Not a destination; if the request can be forwarded */
      else if (rreq->hopCount==1)
      {
      /* Insert request info into request seen table */
      RoutingODMRLBInsertRequestSeen(node,
                  rreq->srcAddr, rreq->seqNumber, rreq->srcAddr,
                        rreq->hopCount, &ODMRLB->requestSeenTable);

/* Check if its address is in the path of the packet */
if (!RoutingODMRLBCheckRequestPath(node,rreq->path,rreq->hopCount-1))
{

      /*  Relay the packet if ttl > 0 */
      if (ttl > 0 && rreq->hopCount < ODMRLB_MAX_SR_LEN)
      {
            RoutingODMRLBRelayRREQ(node, msg, ttl);
```

```
        } /*  if ttl > 0 */

        else
        {
              GLOMO_MsgFree(node, msg);
        }

} /* if check request path */
else
{
        GLOMO_MsgFree(node, msg);
}
} /* else if lookup request seen */


else if (  (rreq->hopCount>1)&&
(!RoutingODMRLBNeednotForwardRequest(rreq->srcAddr,
                  rreq->seqNumber, rreq->path[rreq->hopCount -2],
                  rreq->hopCount, &ODMRLB->requestSeenTable) ) )
{
        /* Insert request info into request seen table */
        RoutingODMRLBInsertRequestSeen(node,
                  rreq->srcAddr,rreq->seqNumber,
                  rreq->path[rreq->hopCount -2],rreq->hopCount,
                   &ODMRLB->requestSeenTable);

        /* Check if its address is in the path of the packet */
        if (!RoutingODMRLBCheckRequestPath(node,
                  rreq->path,rreq->hopCount - 1))
        {

        /*  Relay the packet if ttl > 0 */
        if (ttl > 0 && rreq->hopCount < ODMRLB_MAX_SR_LEN)
        {
              RoutingODMRLBRelayRREQ(node, msg, ttl);
        } /*  if ttl > 0 */

else
{
        GLOMO_MsgFree(node, msg);
}

} /* if check request path */

else
{
        GLOMO_MsgFree(node, msg);
}
} /* else if lookup request seen */

else
{
        GLOMO_MsgFree(node, msg);
}
} /* Handle Request */
```

```
/*
* RoutingODMRLBHandleReply
*
* Processing procedure when Route Reply is received
*/
void RoutingODMRLBHandleReply(
GlomoNode *node, Message *msg, NODE_ADDR destAddr)
{
       GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                                          networkData.networkVar;

      GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                                       routingProtocol;
      Message *newMsg;
      ODMRLB_CR cr;
      ODMRLB_RouteReply *rrep = (ODMRLB_RouteReply *)
                                GLOMO_MsgReturnPacket(msg);
      NODE_ADDR newPath[ODMRLB_MAX_SR_LEN];
      int segLeft;
      int i, j, k;

      segLeft = rrep->segLeft - 1;

     /* I'm the destination of the packet (source of the route) */
if (rrep->targetAddr == node->nodeAddr && destAddr == node->nodeAddr)
{
      /* A new (and first) route to the destination */
      if (!RoutingODMRLBCheckRouteExist(rrep->srcAddr, &ODMRLB->
            routeCacheTable) &&
          !RoutingODMRLBCheckDataSeen(node,rrep->path,rrep->hopCount)
             && rrep->srcAddr != ANY_DEST)
      {
      RoutingODMRLBInsertRouteCache(rrep->path[rrep->hopCount - 1],
            rrep->hopCount, rrep->path, &ODMRLB->routeCacheTable);

      cr.destAddr = destAddr;
      cr.ttl = 0;
      RoutingODMRLBSetTimer(node, MSG_NETWORK_CheckRequest, cr,
                                (clocktype)5);


      ODMRLB->stats.numRoutes++;
      ODMRLB->stats.numHops += rrep->hopCount;

      RoutingODMRLBDeleteRequestTable(rrep->srcAddr,&ODMRLB->
                                                   requestTable);

      // Remove packets expired waiting for route.

      RoutingODMRLBRemoveOldPacketsFromBuffer(&ODMRLB->buffer);

      /* Send buffered data packets that waited for a route */
while (RoutingODMRLBLookupBuffer(rrep->srcAddr, &ODMRLB->buffer))
{
      newMsg = RoutingODMRLBGetBufferedPacket(rrep->srcAddr,
                                              &ODMRLB->buffer);
```

```
        RoutingODMRLBTransmitData(node, newMsg, rrep->srcAddr);

        RoutingODMRLBDeleteBuffer(rrep->srcAddr, &ODMRLB->buffer);

} /* while */

/* Optimization: Adding routes to intermediate nodes */
for (i = 0; i < rrep->hopCount - 1; i++)
{
        for (j = 0; j <= i; j++)
        {
                newPath[j] = rrep->path[j];
        }
        for (j = i + 1; j < ODMRLB_MAX_SR_LEN; j++)
        {
                newPath[j] = ANY_DEST;
        }

        /* Check if the route is new */
        if (!RoutingODMRLBCompareRoute(rrep->path[i],i + 1, newPath,
                                        &ODMRLB->routeCacheTable) &&
                !RoutingODMRLBCheckDataSeen(node, newPath, i + 1))
        {
                RoutingODMRLBInsertRouteCache(rrep->path[i],i + 1,
                                newPath, &ODMRLB->routeCacheTable);

        } /* if a new route */
        } /* for */
} /* if check route exist */

/* routes to the destination already exist */
else
{
        /* if the route is not the same as one in the cache */
        if (!RoutingODMRLBCompareRoute(rrep->path[rrep->hopCount - 1],
                rrep->hopCount,rrep->path,&ODMRLB->routeCacheTable) &&
        !RoutingODMRLBCheckDataSeen(node, rrep->path, rrep->hopCount))
        {
        RoutingODMRLBInsertRouteCache(rrep->path[rrep->hopCount - 1],
                  rrep->hopCount, rrep->path,&ODMRLB->routeCacheTable);

        }

        /* Optimization: Adding routes to intermediate nodes */
        for (i = 0; i < rrep->hopCount - 1; i++)
        {
                for (j = 0; j <= i; j++)
                {
                        newPath[j] = rrep->path[j];
                }
                for (j = i + 1; j < ODMRLB_MAX_SR_LEN; j++)
                {
                        newPath[j] = ANY_DEST;
                }

                /* Check if new route is the same as one in cache */
                if (!RoutingODMRLBCompareRoute(rrep->path[i], i + 1,
```

```
                        newPath, &ODMRLB->routeCacheTable) &&
                    !RoutingODMRLBCheckDataSeen(node, newPath, i + 1))
                {
                    /* Insert the route into cache */
                    RoutingODMRLBInsertRouteCache(rrep->path[i],i + 1,
                            newPath,  &ODMRLB->routeCacheTable);

                } /* if a new route */
            } /* for */
} /* else */

GLOMO_MsgFree(node, msg);

} /* if dest */

/* Node is the intended intermediate node;
cache the routes and relay the packet*/


else if (destAddr == node->nodeAddr)
{
        /* Insert the routes into cache */
        for (i = 0; i < rrep->hopCount; i++)
        {
            newPath[i] = rrep->path[segLeft + i];
        }
        for (i = rrep->hopCount; i < ODMRLB_MAX_SR_LEN; i++)
        {
            newPath[i] = ANY_DEST;
        }

        /* Check if the route is new */
        if (!RoutingODMRLBCompareRoute(rrep->srcAddr,
            rrep->hopCount,newPath,&ODMRLB->routeCacheTable) &&
           !RoutingODMRLBCheckDataSeen(node, newPath, rrep->hopCount))
        {
            RoutingODMRLBInsertRouteCache(rrep->srcAddr,
             rrep->hopCount, newPath, &ODMRLB->routeCacheTable);

        } /* if compare route */

        /* Optimization: Adding routes to intermediate nodes */
        for (j = segLeft; j< rrep->hopCount+segLeft - 1;j++)
        {
            for (k = 0; k <= j - segLeft; k++)
            {
                newPath[k] = rrep->path[k + segLeft];
            }
            for (k = j + 1 - segLeft; k < ODMRLB_MAX_SR_LEN; k++)
            {
                newPath[k] = ANY_DEST;
            }

            /* Check if the route is new */
            if (!RoutingODMRLBCompareRoute(rrep->path[j],
            j + 1 - segLeft,    newPath, &ODMRLB->routeCacheTable) &&
          !RoutingODMRLBCheckDataSeen(node, newPath, j + 1 - segLeft))
```

```
            {
              RoutingODMRLBInsertRouteCache(rrep->path[j],
                        j + 1 - segLeft, newPath, &ODMRLB->
                        routeCacheTable);
            } /* if compare route */
        } /* for */

        RoutingODMRLBRelayRREP(node, msg);

} /* else if intended receiver */

else
{
        GLOMO_MsgFree(node, msg);
}
} /* Handle Reply */

/*
* RoutingODMRLBHandleError
*
* Processing procedure when Route Error is received
*/

void RoutingODMRLBHandleError(GlomoNode *node, Message *msg,
            NODE_ADDR srcAddr, NODE_ADDR destAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                                        networkData.networkVar;
        GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                                        routingProtocol;
        Message *newMsg;
        ODMRLB_RouteError *rerr = (ODMRLB_RouteError *)
                                            GLOMO_MsgReturnPacket(msg);

        /* Delete routes in cache that use the broken link */
        RoutingODMRLBDeleteRouteCache(node, rerr->srcAddr,
                  rerr->unreachableAddr, &ODMRLB->routeCacheTable);

        /* If intermediate node of the broken route, then forward the
        packet */
        if (rerr->path[rerr->hopCount - 1] == node->nodeAddr &&
                        destAddr == node->nodeAddr &&
                        rerr->destAddr != node-> nodeAddr)
        {
              RoutingODMRLBRelayRERR(node, msg);
        } /* if intended receiver */

        /* if source of the broken route */
        else if (rerr->destAddr == node->nodeAddr)
        {
              GLOMO_MsgFree(node, msg);
        }

} /* Handle Error */
```

```
/*
 * RoutingODMRLBCompareRoute
 *
 * Check if new route is the same as the one in cache
 * return TRUE if new route is the same; FALSE otherwise
 */
BOOL RoutingODMRLBCompareRoute(NODE_ADDR destAddr,
            int hopCount, NODE_ADDR *path,
            ODMRLB_RouteCache *routeCache)
{
      int i, j;
      BOOL found = FALSE;
      ODMRLB_RouteCacheEntry *current;

      /*
      printf("COMPARE ROUTE: hop count = %d\n", hopCount);
      */

      for (current = routeCache->head;
            current != NULL && current->destAddr <= destAddr;
            current = current->next)
      {
            if (current->destAddr == destAddr && current->hopCount ==
                                                      hopCount)
            {
                  for (i = 0; i < hopCount; i++)
                  {
                        if (current->path[i] != path[i])
                        {
                              found = FALSE;
                              break;
                        }
                        else
                        {
                              found = TRUE;
                        }
                  }
                  if (found == TRUE)
                  {
                        return (found);
                  }
            }
      }

      return (found);

} /* Compare route */

/*
* RoutingODMRLBGetRoute
*
* Extract a route from the route cache table
*/
NODE_ADDR *RoutingODMRLBGetRoute(NODE_ADDR destAddr,
                              ODMRLB_RouteCache *routeCache)
{
```

```
        ODMRLB_RouteCacheEntry *current;

        for (current = routeCache->head;
              current != NULL && current->destAddr <= destAddr;
              current = current->next)
        {
              if (current->destAddr == destAddr)
              {
              return(current->path);
              }
        }

        printf("ERROR: Get Route - No route can be retrived from
                                          Cache\n");
        return(NULL);
} /* Get route */


/*
* RoutingODMRLBHandleProtocolPacket
*
* Called when packet is received from MAC
*/
void RoutingODMRLBHandleProtocolPacket(
            GlomoNode *node, Message *msg, NODE_ADDR srcAddr,
                  NODE_ADDR destAddr, int ttl)
{

        ODMRLB_PacketType *ODMRLBHeader =
                        (ODMRLB_PacketType*)GLOMO_MsgReturnPacket(msg);

        switch (*ODMRLBHeader)
        {
              case ODMRLB_ROUTE_REQUEST:
              {
                    RoutingODMRLBHandleRequest(node, msg, ttl);

                    break;
              } /* RREQ */

              case ODMRLB_ROUTE_REPLY:
              {
                    RoutingODMRLBHandleReply(node, msg, destAddr);
                    break;
              } /* RREP */

              case ODMRLB_ROUTE_ERROR:
              {
              RoutingODMRLBHandleError(node, msg, srcAddr, destAddr);
              break;
              } /* RERR */
        } /* switch */
} /* RoutingODMRLBHandleProtocolPacket */
```

```
/*
* RoutingODMRLBHandleProtocolEvent
*
* Handles all the protocol events
*/
void RoutingODMRLBHandleProtocolEvent(GlomoNode *node, Message *msg)
{
      GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                                networkData.networkVar;

      GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                                routingProtocol;

      switch (msg->eventType) {

            /* Remove an entry from the request seen table */
      case MSG_NETWORK_FlushTables: {
            RoutingODMRLBDeleteSeenTable(&ODMRLB->requestSeenTable);
            GLOMO_MsgFree(node, msg);
            break;
      }

      /* check if a route is obtained after sending a Route
       Request */
      case MSG_NETWORK_CheckReplied: {
      ODMRLB_CR *cr = (ODMRLB_CR *)GLOMO_MsgReturnInfo(msg);
      int ttl;
      if (!RoutingODMRLBCheckRouteExist(
      cr->destAddr, &ODMRLB->routeCacheTable))
      {
      if (cr->ttl == 1)
      {
            ttl = ODMRLB_MAX_TTL;
      }

      else
      {
            ttl = 1;
      }

      RoutingODMRLBRetryRREQ(node, cr->destAddr, ttl);

      } /* if no route */

      GLOMO_MsgFree(node, msg);

      break;
}
/*added*/
case MSG_NETWORK_CheckRequest: {
      ODMRLB_CR *cr = (ODMRLB_CR *)GLOMO_MsgReturnInfo(msg);
      RoutingODMRLBStartTransmission(node,msg,cr->destAddr);
      break;

}
```

```
default:
      fprintf(stderr, "RoutingODMRLB: Unknown MSG type %d!\n",
                                    msg->eventType);
      assert(FALSE);

} /* switch */


} /* RoutingODMRLBHandleProtocolEvent */

/*
* RoutingODMRLBInitiateRREQ
*
* Initiate a Route Request packet when no route to destination is known
*/
void RoutingODMRLBInitiateRREQ(GlomoNode *node, NODE_ADDR destAddr)
{
      GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                           networkData.networkVar;
      GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                          routingProtocol;
      Message *newMsg;
      ODMRLB_RouteRequest *rreq;
      ODMRLB_CR cr;
      char *pktPtr;
      int pktSize = sizeof(ODMRLB_RouteRequest);
      int i;

      newMsg = GLOMO_MsgAlloc(node, GLOMO_MAC_LAYER, 0,
                                    MSG_MAC_FromNetwork);
      GLOMO_MsgPacketAlloc(node, newMsg, pktSize);

      pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
      rreq = (ODMRLB_RouteRequest *) pktPtr;

      rreq->pktType = ODMRLB_ROUTE_REQUEST;
      rreq->srcAddr = node->nodeAddr;
      rreq->targetAddr = destAddr;
      rreq->seqNumber = RoutingODMRLBGetSeq(node);
      rreq->hopCount = 1;
      for (i = 0; i < ODMRLB_MAX_SR_LEN; i++)
      {
            rreq->path[i] = ANY_DEST;
      }

      NetworkIpSendRawGlomoMessage(
            node, newMsg, ANY_DEST, CONTROL, IPPROTO_ODMRLB, 1);

      ODMRLB->stats.numRequestSent++;

   if(RoutingODMRLBCheckRequestTable(destAddr,&ODMRLB->requestTable))
   {
      RoutingODMRLBUpdateTtl(destAddr, &ODMRLB->requestTable);
   }
   else
   {
```

```
    RoutingODMRLBInsertRequestTable(destAddr,&ODMRLB->requestTable);
    }

     if(rreq->hopCount==1)
       RoutingODMRLBInsertRequestSeen(node, node->nodeAddr, rreq->
                   seqNumber, rreq->srcAddr,rreq->hopCount,
                   &ODMRLB->requestSeenTable);
     else if(rreq->hopCount>1)
       RoutingODMRLBInsertRequestSeen(node, node->nodeAddr,
       rreq->seqNumber, rreq->path[rreq->hopCount -2],rreq->hopCount,
       &ODMRLB->requestSeenTable);
     else
       ;

       cr.destAddr = destAddr;
       cr.ttl = 1;

       RoutingODMRLBSetTimer(node, MSG_NETWORK_CheckReplied, cr,
                     (clocktype)ODMRLB_RING0_REQUEST_TO);

} /* RoutingODMRLBInitiateRREQ */

/*
* RoutingODMRLBRetryRREQ
*
* Send RREQ again after not receiving any RREP
*/
void RoutingODMRLBRetryRREQ(GlomoNode *node, NODE_ADDR destAddr, int
ttl)
{
      GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                              networkData.networkVar;
      GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                           routingProtocol;
      Message *newMsg;
      ODMRLB_RouteRequest *rreq;
      ODMRLB_CR cr;
      char *pktPtr;
      clocktype backoff;
      int pktSize = sizeof(ODMRLB_RouteRequest);
      int i;

      newMsg = GLOMO_MsgAlloc(node, GLOMO_MAC_LAYER, 0,
                    MSG_MAC_FromNetwork);
      GLOMO_MsgPacketAlloc(node, newMsg, pktSize);

      pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
      rreq = (ODMRLB_RouteRequest *) pktPtr;

      rreq->pktType = ODMRLB_ROUTE_REQUEST;
      rreq->srcAddr = node->nodeAddr;
      rreq->targetAddr = destAddr;
      rreq->seqNumber = RoutingODMRLBGetSeq(node);
      rreq->hopCount = 1;
      for (i = 0; i < ODMRLB_MAX_SR_LEN; i++)
      {
            rreq->path[i] = ANY_DEST;
```

```
        }

        NetworkIpSendRawGlomoMessage(
              node, newMsg, ANY_DEST, CONTROL, IPPROTO_ODMRLB, ttl);

        ODMRLB->stats.numRequestSent++;


        if(rreq->hopCount==1)
              RoutingODMRLBInsertRequestSeen(node, node->nodeAddr,
              rreq->seqNumber, rreq->srcAddr,rreq->hopCount,
              &ODMRLB->requestSeenTable);
        else if(rreq->hopCount>1)
              RoutingODMRLBInsertRequestSeen(node, node->nodeAddr,
                    rreq->seqNumber,rreq->path[rreq->hopCount -2],
                    rreq->hopCount,&ODMRLB->requestSeenTable);
        else
        ;

        if (ttl == ODMRLB_MAX_TTL)
        {
              RoutingODMRLBUpdateRequestTable(destAddr, &ODMRLB->
                                                      requestTable);
              backoff = RoutingODMRLBGetBackoff(destAddr, &ODMRLB->
                                                      requestTable);
        }
        else
        {
              RoutingODMRLBUpdateTtl(destAddr, &ODMRLB->requestTable);
              backoff = ODMRLB_RING0_REQUEST_TO;
        }

        cr.destAddr = destAddr;
        cr.ttl = ttl;

        RoutingODMRLBSetTimer(node, MSG_NETWORK_CheckReplied, cr,
                        backoff);

} /* RoutingODMRLBRetryRREQ */

/*
* RoutingODMRLBTransmitData
*
* Obtain the route from the cache and send the data thru the source
route
*/
void RoutingODMRLBTransmitData(GlomoNode *node, Message *msg,
                NODE_ADDR destAddr)
{

      GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                networkData.networkVar;
GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
routingProtocol;
ODMRLBIpOptionType *option;
NODE_ADDR *route;
int hopCount;
```

```
GLOMO_MsgSetLayer(msg, GLOMO_MAC_LAYER, 0);
GLOMO_MsgSetEvent(msg, MSG_MAC_FromNetwork);

route = RoutingODMRLBGetRoute(destAddr, &ODMRLB->routeCacheTable);
hopCount = RoutingODMRLBGetHop(destAddr, &ODMRLB->routeCacheTable);

if (route != NULL)
{
AddCustomODMRLBIpOptionFields(node, msg);
option = GetPtrToODMRLBIpOptionField(msg);
option->segmentLeft = hopCount;
option->salvagedBit = FALSE;

NetworkIpSendPacketToMacLayerWithNewStrictSourceRoute(
node, msg, route, hopCount, TRUE);

ODMRLB->stats.numDataSent++;
ODMRLB->stats.numDataTxed++;
}
else
{
GLOMO_MsgFree(node, msg);
}

} /* RoutingODMRLBTransmitData */

/*
* RoutingODMRLBRelayRREQ
*
* Forward (re-broadcast) the Route Request
*/
void RoutingODMRLBRelayRREQ(GlomoNode *node, Message *msg, int ttl)
{
GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                          networkData.networkVar;
GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                          routingProtocol;
Message *newMsg;
ODMRLB_RouteRequest *oldRreq;
ODMRLB_RouteRequest *newRreq;
char *pktPtr;
int pktSize = sizeof(ODMRLB_RouteRequest);
clocktype delay;
int i;

oldRreq = (ODMRLB_RouteRequest *) GLOMO_MsgReturnPacket(msg);

newMsg = GLOMO_MsgAlloc(node, GLOMO_MAC_LAYER, 0,
                        MSG_MAC_FromNetwork);
GLOMO_MsgPacketAlloc(node, newMsg, pktSize);

pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
newRreq = (ODMRLB_RouteRequest *) pktPtr;

newRreq->pktType = oldRreq->pktType;
newRreq->srcAddr = oldRreq->srcAddr;
```

```
newRreq->targetAddr = oldRreq->targetAddr;
newRreq->seqNumber = oldRreq->seqNumber;
newRreq->hopCount = oldRreq->hopCount + 1;
for (i = 0; i < oldRreq->hopCount - 1; i++)
{
      newRreq->path[i] = oldRreq->path[i];
}
newRreq->path[oldRreq->hopCount - 1] = node->nodeAddr;
for (i = oldRreq->hopCount; i < ODMRLB_MAX_SR_LEN; i++)
{
      newRreq->path[i] = ANY_DEST;
}
delay = pc_erand(node->seed) * ODMRLB_BROADCAST_JITTER;
NetworkIpSendRawGlomoMessageWithDelay(
      node, newMsg, ANY_DEST, CONTROL, IPPROTO_ODMRLB, ttl, delay);
ODMRLB->stats.numRequestSent++;
GLOMO_MsgFree(node, msg);

} /* RoutingODMRLBRelayRREQ */

/*
* RoutingODMRLBCheckDisjointRouteExist
*
* Check for route replies
*/
int RoutingODMRLBCheckDisjointRouteExist(GlomoNode *node, Message *msg)
{

GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)node->
                                networkData.networkVar;
GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                                routingProtocol;
ODMRLB_RouteRequest *rreq = (ODMRLB_RouteRequest *)
                          GLOMO_MsgReturnPacket(msg);
ODMRLB_CR cr;
ODMRLB_RouteReplyCache *routeCache=&ODMRLB->routeReplyCacheTable;

NODE_ADDR destAddr=rreq->targetAddr;

ODMRLB_RouteReplyCacheEntry *current;


if (routeCache->count == 0)
{
return 0;
}

for (current = routeCache->head;
            current != NULL ;
            current = current->next)
{


if (current->destAddr == destAddr)
{
if(RoutingODMRLBComparePath(rreq->hopCount,rreq->path,current->
              hopCount,current->path)!=0)
```

```
            return 1;
}
}
return 0;

}/*RoutingODMRLBCheckDisjointRouteExist*/

/*
*RoutingODMRLBDeleteRouteReplyCache
*/
void RoutingODMRLBDeleteRouteReplyCache(ODMRLB_RouteReplyCache
*routeCache,NODE_ADDR destAddr)
{

ODMRLB_RouteReplyCacheEntry *toFree;
ODMRLB_RouteReplyCacheEntry *current;


if (routeCache->count == 0)
{
return;
}
else if (routeCache->head->destAddr == destAddr)
{
while(routeCache->head->destAddr == destAddr)
{
toFree = routeCache->head;
routeCache->head = toFree->next;
if(routeCache->count>1)
toFree->next->prev=NULL;

pc_free(toFree);
--(routeCache->count);
}
}
else
{
for (current = routeCache->head;
      current->next != NULL && current->next->destAddr < destAddr;
                        current = current->next)
{
}

while (current->next != NULL && current->next->destAddr == destAddr)
{
toFree = current->next;
current->next = toFree->next;
toFree->next->prev=current;
pc_free(toFree);
--(routeCache->count);
}
}
}/*RoutingODMRLBDeleteRouteReplyCache*/

/*
* RoutingODMRLBSendReply
*/
```

```
void RoutingODMRLBSendReply(GlomoNode *node, Message *msg)
{

GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                    node->networkData.networkVar;
GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                          routingProtocol;
ODMRLB_RouteRequest *rreq = (ODMRLB_RouteRequest *)
                         GLOMO_MsgReturnPacket(msg);
ODMRLB_CR cr;
ODMRLB_RouteReplyCache *routeCache=&ODMRLB->routeReplyCacheTable;

if(RoutingODMRLBCheckDisjointRouteExist(node,msg)==0)
{

RoutingODMRLBInsertRouteReplyCache(rreq->srcAddr, rreq->hopCount,
                          rreq->path, &ODMRLB->routeReplyCacheTable);
RoutingODMRLBInitiateRREP(node, msg);
}
else
GLOMO_MsgFree(node, msg);
}/*RoutingODMRLBSendReply*/


/*
* RoutingODMRLBStartTransmission
*
*
*/
void RoutingODMRLBStartTransmission(GlomoNode *node, Message *msg,
                                                  NODE_ADDR destAddr)
{

GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->
                                              networkData.networkVar;
GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *) ipLayer->
                                             routingProtocol;
Message *newMsg;
int numSent[ROUTE_MAX+1]={0};
int count=0,routeNum;

while (RoutingODMRLBLookupBuffer(destAddr, &ODMRLB->buffer))
{
newMsg = RoutingODMRLBGetBufferedPacket(destAddr,
                                               &ODMRLB->buffer);
routeNum=selectRouteNumber(destAddr,&ODMRLB-> routeCacheTable,
                                                   numSent);
RoutingODMRLBTransmitData2(node,newMsg, destAddr,routeNum);
RoutingODMRLBDeleteBuffer(destAddr, &ODMRLB->buffer);

} /* while */

}/*RoutingODMRLBStartTransmission*/
```

```
/*
* RoutingODMRLBInitiateRREP
*
* Destination of the route sends Route Reply in reaction to Route
Request
*/
void RoutingODMRLBInitiateRREP(GlomoNode *node, Message *msg)
{
GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)node->
                          networkData.networkVar;
GlomoRoutingODMRLB* ODMRLB = (GlomoRoutingODMRLB *)ipLayer->
                           routingProtocol;
Message *newMsg;
ODMRLB_RouteRequest *rreqPkt;
ODMRLB_RouteReply *rrepPkt;
char *pktPtr;
int pktSize = sizeof(ODMRLB_RouteReply);
int i;
clocktype delay;
rreqPkt = (ODMRLB_RouteRequest *) GLOMO_MsgReturnPacket(msg);
newMsg =GLOMO_MsgAlloc(node,GLOMO_MAC_LAYER, 0, MSG_MAC_FromNetwork);
GLOMO_MsgPacketAlloc(node, newMsg, pktSize);
pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
rrepPkt = (ODMRLB_RouteReply *) pktPtr;
rrepPkt->pktType = ODMRLB_ROUTE_REPLY;
rrepPkt->targetAddr = rreqPkt->srcAddr;
rrepPkt->srcAddr = node->nodeAddr;
rrepPkt->hopCount = 1;
rrepPkt->segLeft = rreqPkt->hopCount;
for (i = 0; i < rreqPkt->hopCount - 1; i++)
{
      rrepPkt->path[i] = rreqPkt->path[i];
}
rrepPkt->path[rreqPkt->hopCount - 1] = node->nodeAddr;
for (i = rreqPkt->hopCount; i < ODMRLB_MAX_SR_LEN; i++)
{
      rrepPkt->path[i] = ANY_DEST;
}
delay = pc_erand(node->seed) * ODMRLB_BROADCAST_JITTER;
if (rreqPkt->hopCount > 1)
{
      NetworkIpSendRawGlomoMessageToMacLayerWithDelay(
            node, newMsg, rrepPkt->path[rreqPkt->hopCount - 2],
            CONTROL, IPPROTO_ODMRLB, 1, DEFAULT_INTERFACE,
                  rrepPkt->path[rreqPkt->hopCount -2], delay);
}
else
{
      NetworkIpSendRawGlomoMessageToMacLayerWithDelay(
            node, newMsg, rrepPkt->targetAddr, CONTROL, IPPROTO_ODMRLB,
            1,DEFAULT_INTERFACE, rrepPkt->targetAddr, delay);
}

ODMRLB->stats.numReplySent++;
GLOMO_MsgFree(node, msg);
} /* RoutingODMRLBInitiateRREP */
```