DOA: DSR OVER AODV ROUTING FOR MOBILE AD-HOC NETWORKS

A PROJECT REPORT

Submitted by

ARTHI.N.R	**71204205003**
INDUMATHI.P	**71204205011**

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

KUMARAGURU COLLEGE OF TECHNOLOGY, COIMBATORE

ANNA UNIVERSITY: CHENNAI 600 025

APRIL 2008

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report **"DOA: DSR OVER AODV ROUTING FOR MOBILE AD-HOC NETWORKS"** is the bonafide work of **"ARTHI.N.R and INDUMATHI.P"** who carried out the project work under my supervision.

SIGNATURE

Dr.Thangasamy, B.E(hons).,Ph.D.

HEAD OF THE DEPARTMENT

Dept of Information Technology,

Kumaraguru College of Technology,

Coimbatore – 641 006.

SIGNATURE

Ms.L.S.Jayashree M.E.

SUPERVISOR

Dept of Information Technology,

Kumaraguru College of Technology,

Coimbatore – 641 006.

The candidates with University Register No 71204205003 and 71204205011 were examined by us in the project viva-voice examination held on.....

INTERNAL EXAMINER

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We express our sincere thanks to our Chairman **Padmabhushan Arutselvar Dr. N. Mahalingam B.Sc, F.I.E** and Correspondent **Shri. Balasubramanian** for all their support and ray of strengthening hope extended. We are immensely grateful to our Principal **Dr. Joseph V. Thanikal M.E., Ph.D., PDF., CEPIT.,** for his invaluable support to the outcome of this project.

We also thank **Dr. Thangasamy,** Head of the department of Information Technology for his valuable advice and useful suggestions throughout this project.

We express our heartiest thanks to our project coordinator **Mr. K.R.Baskaran M.S.,** Assistant Professor, Department of Information Technology who has helped us to perform our project work extremely well.

We also extend our heartfelt thanks to our project guide **Ms. L.S.Jayashree M.E.,** Assistant Professor, Department of Information Technology who rendered her valuable guidance and support to perform our project work extremely well.

We thank the teaching and non-teaching staffs of our Department for providing us the technical support in the duration of our project.

We express our humble gratitude and thanks to our beloved parents and family members who have supported and helped us to complete the project and our friends, for lending us valuable tips, support and co-operation throughout the project work.

DECLARATION

We,

ARTHI.N.R **71204205003**

INDUMATHI.P **71204205011**

hereby declare that the project entitled **"DOA: DSR OVER AODV ROUTING FOR MOBILE AD-HOC NETWORKS"**, submitted in partial fulfillment to Anna University as the project work of Bachelor of Technology (Information Technology) degree, is a record of original work done by us under the supervision and guidance of Department of Information Technology, Kumaraguru college of Technology, Coimbatore.

Place: Coimbatore

Date: 22.04.08

N.R. Arthi.

[Arthi.N.R]

P. Indumathy

[Indumathi.P]

Project Guided by

[Ms. L.S. Jayashree M.E.]

ABSTRACT

The DOA Protocol implements a Lightweight Hierarchical Routing Model, Way Point Routing(WPR) in which a number of intermediate nodes are selected as Waypoints and the route is divided into segments by the waypoints. Waypoints, including the source and the destination, run a high-level intersegment routing protocol, while the nodes on each segment run a low-level intrasegment routing protocol. One distinct advantage of this model is that when a node on the route moves out or fails, instead of discarding the whole original route and discovering a new route from the source to the destination, only the two waypoint nodes of the broken segment have to find a new segment. In addition, this model is lightweight because it maintains a hierarchy only for nodes on active routes.

WPR use DSR as the intersegment routing protocol and AODV as the intrasegment routing protocol. This instantiation is termed DSR over AODV (DOA) routing protocol. Thus, DSR and AODV—two well-known on-demand routing protocols for MANETs—are combined into one hierarchical routing protocol and become two special cases of this protocol. Furthermore, one novel technique for DOA presented is an efficient loop detection method.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVATIONS

MANET	: Mobile Ad hoc Network
DOA	: DSR Over AODV routing protocol
DSR	: Dynamic Source Routing
AODV	: Ad hoc On-demand Distance Vector routing
WPR	: Way Point Routing model
CGSR	: Clusterhead-Gateway Switch Routing
ZRP	: Zone Routing Protocol
RREQ	: Route Request
RREP	: Route Reply
RERR	: Route Error
RACT	: Route Activate
TTL	: Time To Live
CBR	: Constant Bit Rate

1. INTRODUCTION

1.1 GENERAL:

An Ad Hoc network is a cooperative engagement of collection of Mobile Hosts without the intervention of any centralized Access Point. Mobile ad-hoc networks (MANETs) are infrastructure-free network of mobile nodes that communicate with each other wirelessly. Since the nodes are mobile, the network topology may change rapidly and unpredictably over time.

Routing has always been one of the key challenges in MANETs and the challenge becomes more difficult when the network size increases. The design of routing protocols for these networks is a complex issue. These routing protocols are classified into different categories. If classified by the manner in which they react to network topology changes, these can be grouped into proactive (or table-driven) protocols and reactive (or on-demand) protocols (though several hybrid protocols exist). If classified by the role of routing nodes and the organization of the network, these protocols can be grouped into flat protocols and hierarchical protocols.

Proactive protocols propagate topology information periodically and find routes continuously, while reactive protocols find routes on demand.

In Flat routing protocols, all the nodes are assigned the same functionality. In Hierarchical routing protocols, the network is divided into groups and a subset of nodes are assigned special functionalities, typically, coordinating roles.

DSR and AODV are On-demand Flat Routing protocols where as CGSR and ZRP are Hierarchical protocols.

1.2. PROBLEM DEFINITION

As a network grows, routes between sources and destinations become longer. When a route breaks due to node mobility or node failure, flat routing protocols like DSR and AODV typically discard the whole original route and initiate another round of route discovery to establish a new route from the source to the destination. When a route breaks, usually only a few hops are broken, but other hops are still intact. Thus, this approach wastes the knowledge of the original route and may cause significant overhead in global route discoveries. An optimization to AODV is local repair. However, the local repair is suitable for situations where link failures occur near the destination.This is because in AODV, intermediate nodes only know the destination and the next hop for a route and the target of the local repair has to be the destination. If a link failure occurs far from the destination, it would be better for the source to discover a new route directly.So as the whole, these protocols does not solve routing overheads, scalability problem and so performance degrade.

1.3. OBJECTIVE OF PROJECT

The main objective of the project is as follows: By establishing and maintaining active routes hierarchically, a broken route can be repaired locally wherever the link failure occurs along the route and fewer global route discoveries need to be initiated by the source node. Consequently, this new routing protocol incurs less routing overhead and exhibits better scalability and performance.

This new routing protocol implements scalable routing model, Way Point Routing (WPR), which maintains a hierarchy only for active routes. The instantiation of WPR, where it use DSR as the intersegment routing protocol and AODV as the intrasegment routing protocol. This instantiation is termed DSR over AODV (DOA) routing protocol. In WPR, a number of intermediate nodes on a route are selected as waypoints and the route is divided into segments by the waypoints. The source and the destination are also considered as waypoints. One distinct advantage of our model is that when a node on the route moves out or fails, instead of discarding the whole original route and discovering a new route from the source to the destination, only the two waypoints of the broken segment have to find a new segment.

2. LITERATURE REVIEW

2.1. FEASIBILITY STUDY

2.1.1 CURRENT STATUS OF THE PROBLEM

2.1.1.1 CLUSTERHEAD-GATEWAY SWITCH ROUTING

Clusterhead-Gateway Switch Routing (CGSR) is based on the Destination-Sequence Distance-Vector (DSDV) routing protocol and works proactively. CGSR organizes the network into clusters and elects the cluster head in each cluster by running the Least Clusterhead Change(LCC) clustering algorithm. Other nodes in the cluster are one hop away from the clusterhead. Nodes that belong to more than one cluster are gateways. Each node maintains a cluster member table and a distance vector table. The cluster Member table records which cluster each node belongs to by storing the clusterhead of that cluster,while the distance vector table records next hops to all clusters. CGSR reduces the size of the routing table by maintaining a route entry for each cluster instead of an entry for each node. Typically a route in CGSR is of the form clusterhead-gateway-clusterhead, and so on.

DISADVANTAGES:

1. Selection of clusterheads includes all nodes-nodes work non-uniformly
2. Traffic Bottlenecks-drain in battery
3. Hierarchy is built in two dimensions

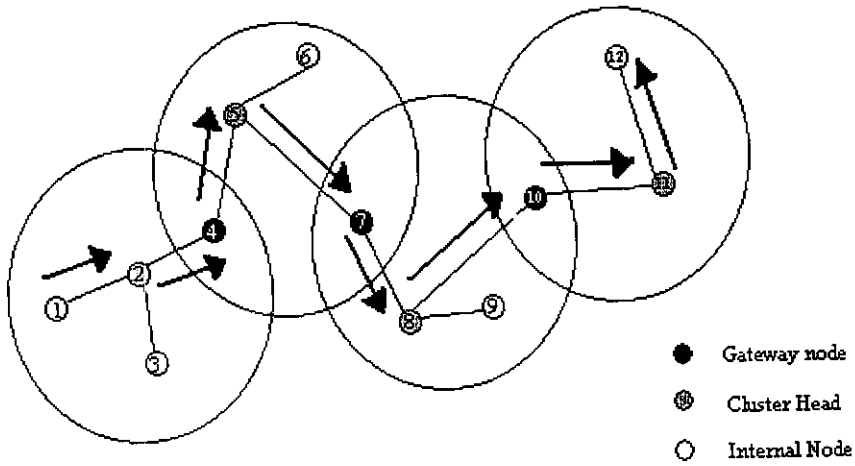


Fig.2.1 Hierarchy in two dimensions

2.1.1.2 DYNAMIC SOURCE ROUTING

Dynamic source routing is a Source routed On-Demand routing protocol in Ad Hoc networks. It uses Source Routing, which is a technique in which the sender of a packet determines the complete sequence of nodes through which the node has to travel. The sender of the packet explicitly mentions the list of all nodes in the packet's header, identifying each forwarding 'hop' by the address of the next node to which to transmit the packet on its way to destination host.

In this protocol the nodes don't need to exchange the Routing table information periodically and thus reduces the bandwidth overhead in the network. Each Mobile node participating in the protocol maintain a 'routing cache' which contains the list of routes that the node has learnt. Whenever the node finds a new route it adds the new route in its 'routing cache'. Each mobile node also maintains a sequence counter 'request id' to uniquely identify the requests generated by a mobile host. The pair < source address,

request id > uniquely identifies any request in the Ad Hoc network. The protocol does not need transmissions between hosts to work in bi-direction.

ADVANTAGES:

1. Routes maintained only between nodes who need to communicate- reduces overhead of route maintenance
2. Route caching can further reduce route discovery overhead
3. Learn more routing information from traffic than AODV
4. Easier to secure than AODV

DISADVANTAGES

1. Packet header size grows with route length due to source routing
2. Potential collisions between route requests propagated by neighboring nodes
3. Stale caches will lead to increased overhead

2.1.1.3 ADHOC ON-DEMAND DISTANCE VECTOR ROUTING

AODV is an extension of Distance Sequenced Distance Vector (DSDV) routing protocol, a Table Driven routing protocol for Ad Hoc networks. AODV is designed to improve the performance characteristics of DSDV in the creation and maintenance of routes. AODV decreases the control overhead by minimizing the number of broadcasts using a pure on-demand route acquisition method. AODV uses only symmetric links between neighboring nodes. In AODV, each node maintains two separate counters: 1. Sequence Number, a monotonically increasing counter used to

maintain freshness information about the reverse route to the Source and 2.Broadcast-ID, which is incremented whenever the source issues a new Route Request message. Each node also maintains information about its reachable neighbors with bi-directional connectivity.

ADVANTAGES:

1. Routes need not to be included in packet header-reduces header overhead
2. Able to run larger networks-avoids Scalability problem
3. Sequence numbers are used to avoid old/broken routes

DISADVANTAGES:

1. Less secure than DSR
2. Less routing information compared to DSR
3. Slower in finding route compared to DSR

2.1.1.4 ZONE ROUTING PROTOCOL

The Zone Routing Protocol (ZRP) is a hybrid routing protocol. Each node dynamically maintains a zone centered at itself. A zone is a collection of neighbors and links within a predefined number of hops, which is termed the zone radius. IntrAzone Routing Protocol (IARP) works proactively and is used for the routing inside the zone, while IntErzone Routing Protocol (IERP) works reactively and is used for the routing outside the zone. To establish a route to the destination outside the source's zone, RREQ/RREP is used like other on-demand routing protocols.

2.1.2 PROPOSED SYSTEM AND ADVANTAGES

The DOA Protocol inherits the strength of both DSR and AODV and thus exhibit better scalability and performance. The instantiation of WPR choose DSR and AODV as the intersegment and the intrasegment protocols, respectively.This instantiation as DSR over AODV (DOA) routing protocol. Thus, DSR and AODV—two wellknown routing protocols for MANETs—are combined hierarchically. Moreover, they become two special cases of our scheme: When the segment length is set to 1, DOA works in DSR mode because each hop is a segment and the intersegment routing protocol (DSR) dominates; when the segment length is set to a large number, DOA works in AODV mode because the whole route is one segment and the intrasegment routing protocol (AODV) dominates. In this protocol,two novel techniques for DOA is presented: one is an efficient loop detection method and the other is a multitarget route discovery.

A route is divided into segments by selecting a number of waypoint nodes from the route. Therefore, this hierarchical routing model Way Point Routing (WPR). The source and the destination are also waypoint nodes. Other nodes on the route are termed forwarding nodes. Each segment starts with a waypoint node called start node and ends with a waypoint node called end node. The start and end nodes of a segment are generally connected by a number of forwarding nodes. Two neighboring segments share a common waypoint node, which acts as the end node of the upstream segment and the start node of the downstream segment.

ADVANTAGES:

1. Since routes are maintained at a finer grain (i.e., a segment), a broken route can be fixed locally at the level of a segment. When a route breaks, usually only a few hops are broken and other hops are still intact. Fixing a broken route within a segment extends the lifetime of the route and saves expensive, time-consuming global route discoveries. Thus, WPR will substantially reduce the routing overhead and improve the performance.
2. The length (hop count) of each segment on a route can be different and lengths of segments on different routes can be different. This makes WPR an adaptive routing scheme, which is important for MANETs where various network scenarios exist. For example, if a network is stable and nodes move slowly, longer segments can be used to reduce the overhead of maintaining the hierarchy; if nodes move faster, shorter segments can be used to facilitate route repairs.

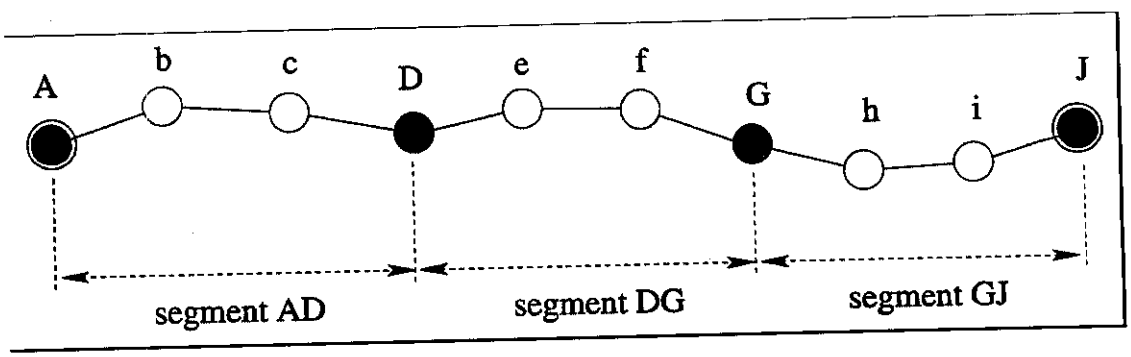


Fig.2.2 Route divided into segments

2.2 HARDWARE REQUIREMENTS

Processor	:	Pentium IV
Speed	:	Above 500 MHz
RAM capacity	:	128 MB
Floppy disk drive	:	1.44 MB
Hard disk drive	:	20 GB
Key Board	:	108 keys
Mouse	:	Optical Mouse
CD Writer	:	Optional
Printer	:	Optional
Motherboard	:	Intel
Monitor	:	17"

2.3 SOFTWARE REQUIREMENTS

Operating System	:	Windows 2000/XP
Front end used	:	Java
Simulator used	:	Glomosim
Software needed	:	MS Visual C++ 6.0
Coding language	:	Parsec C

2.4 SOFTWARE OVERVIEW

2.4.1.GLOMOSIM

GloMoSim (for Global Mobile Information System Simulator) is used to simulate the DOA protocol that effectively utilizes parallel execution to reduce the simulation time of detailed high-fidelity models of large communication networks. GloMoSim has been designed to be extensible and composable: the communication protocol stack for wireless networks is divided into a set of layers, each with its own API. Models of protocols at one layer interact with those at a lower (or higher) layer only via these APIs. The modular implementation enables consistent comparison of multiple protocols at a given layer. The parallel implementation of GloMoSim can be executed using a variety of conservative synchronization protocols, which include the null message and conditional event algorithms.

GloMoSim is a mobile simulator built using C language. All message transfers and other network elements are handled by the individual layer coding built in C. To make the concepts clear, GloMoSim provides users with a Visualization Tool (VT) built using Java. The VT helps us understand the network environment, the node positions, message transfers, clustering details, etc.

GloMoSim Architecture

The networking stack is decomposed into a number of layers as shown in Figure 2.3. A number of protocols have been developed at each layer and models of these protocols or layers can be developed at different levels of granularity.

In our project we deal with all the layers, but most of the coding has been implemented in the Routing layers. The coding present in the Routing layer has been largely modified.

The communication between the various layers is accomplished by means of the various APIs available. A common API between two layers helps in the communication between those two layers.

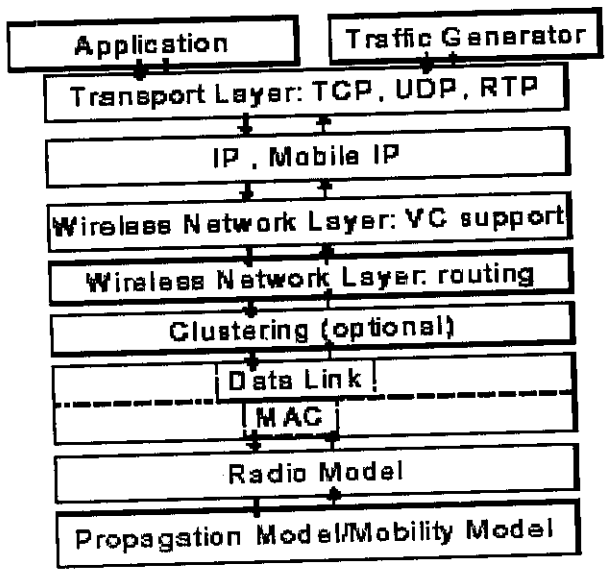


Fig.2.3. GloMoSim Architecture

2.4.2.PARSEC COMPILER

PARSEC (for PARallel Simulation Environment for Complex systems) is a C-based simulation language developed by the Parallel Computing Laboratory at UCLA, for sequential and parallel execution of discrete-event simulation models. It can also be used as a parallel programming language. PARSEC runs on several platforms, including most recent UNIX variants as well as Windows. PARSEC adopts the process interaction approach to discrete-event simulation. An object (also referred to

as a physical process) or set of objects in the physical system is represented by a logical process. Interactions among physical processes (events) are modeled by time-stamped message exchanges among the corresponding logical processes. One of the important distinguishing features of PARSEC is its ability to execute a discrete-event simulation model using several different asynchronous parallel simulation protocols on a variety of parallel architectures. PARSEC is designed to cleanly separate the description of a simulation model from the underlying simulation protocol, sequential or parallel, used to execute it. Thus, with few modifications, a PARSEC program may be executed using the traditional sequential (Global Event List) simulation protocol or one of many parallel optimistic or conservative protocols. In addition, PARSEC provides powerful message receiving constructs that result in shorter and more natural simulation programs.

3. DETAILS OF METHODOLOGY EMPLOYED

3.1 ROUTE DISCOVERY

Intersegment Route Request

When a source node requires a new route to a destination node, it broadcasts an RREQinter message. A RREQinter can be uniquely identified by the combination of the source address and the source's broadcast ID number. Similarly to DSR routing protocol, the RREQinter message records the list of nodes it has traversed. When an intermediate node receives the RREQinter, it first determines whether this request is a duplicate by looking up its request seen table. Duplicate requests are discarded. If the RREQinter is new and its TTL is greater than zero, the intermediate node appends its address to the path recorded in the RREQinter and rebroadcasts the request to its neighbors. Route reply by intermediate nodes is not enabled in DOA because the routes provided by intermediate nodes may be outdated. When the RREQinter message reaches the destination, the destination node replies to the request.

Intersegment Route Reply

In DSR, for one route discovery identified by a source address and a broadcast id, there is no limit on the number of RREPs that the destination can send to the source; the destination will send a RREP whenever it receives a RREQ. For each intersegment route discovery, we limit the number of replies that the destination can send to the source by the parameter MAX_REPLY_TIMES. If the length of the path recorded in RREQinter exceeds a threshold value (DEFAULT_SEGMENT_LEN), the destination divides the path into segments by selecting waypoint nodes from

the path. It is worth noting that there are several ways to select waypoint nodes. A naive method is selecting waypoint nodes that divide the path into segments evenly, i.e., the length (hop count) of each segment is roughly equal to DEFAULT_SEGMENT_LEN.

For example, suppose the path from a source A to a destination J is

A _ B _ C _ D _ E _ F _ G _ H _ I _ J;

and suppose the value of DEFAULT_SEGMENT_LEN is 3.

A possible path division would be

A _ b _ c _ D _ e _ f _ G _ h _ i _ J;

which contains three segments: AD, DG, and GJ.

The destination generates a RREPinter message after the path has been divided. The path that includes both waypoint nodes and forwarding nodes is placed into the RREPinter. This detailed path is used by intermediate nodes to establish the route and relay the RREPinter back to the source. There are two tables maintained by all nodes in the network: route cache and routing table.

Route cache is used for intersegment DSR routing, while routing table is used for intrasegment AODV routing. An entry in the route cache uses *<destination : source route>* structure as in the DSR protocol. The route cache stores source routes from the current node to destination nodes. A source route only contains waypoint nodes. Therefore, any two neighboring nodes on a source route in DOA are one segment away from each other, instead of one hop away as in DSR. Typically, one segment contains multiple hops.

An entry in the routing table uses *<end node : next hop>* structure similarly to the AODV protocol. An end node corresponds to a destination in AODV. The routing table stores the next hop to the end node of a segment,

where the current node is either a forwarding node or the start node of this segment. The difference between the AODV routing table and the DOA routing table is that the destination field in the AODV routing table is the final destination of data packets, while in DOA, it is the end node of a segment and is not necessarily the final destination. When an intermediate node receives the RREPinter, it first determines if it is a waypoint node or a forwarding node on the replied route. If it is a waypoint node, it updates both its route cache and its routing table. If it is a forwarding node, it updates only its routing table.

3.2 DATA FORWARDING

To transmit a data packet, the source node first sets the source route option in the data packet for intersegment routing. The source gets a source route to the destination from its route cache and inserts the source route into the header of the data packet, as in the DSR protocol. The difference is that in DOA, a source route only contains waypoint nodes. Therefore, compared to DSR, our approach reduces the header overhead of data packets and improves the scalability of source routing.

After setting the source route option for the data packet, the source node processes intrasegment routing for the first segment on the route. It gets the next hop to the end node of the first segment by looking up its routing table, like the AODV protocol. The difference is that AODV gets the next hop to the final destination, while DOA gets the next hop to the end node of the current segment. Next, the source node sends the packet to the next hop, which, typically, is a forwarding node.

The source route option of a data packet contains a field named *current_seg*, which is the index of the current segment on the source route.

This index lets intermediate nodes on the route know which segment the data packet is being routed in. The value of *current_seg* is maintained by waypoint nodes. When the data packet leaves an upstream segment and enters a downstream segment at a waypoint node, the value of *current_seg* is increased by 1. This operation belongs to intersegment routing. For intrasegment routing, intermediate nodes get the next hop to the end node of the current segment from their routing tables and send the data packet to that next hop. Intersegment routing (occurs only at waypoint nodes) and intrasegment routing (occurs at both waypoint and forwarding nodes) continue until the data packet reaches the destination.

3.3 ROUTE MAINTENANCE

Intrasegment Route Repair

Intrasegment route repair works in AODV mode. When a node finds the next hop node is unreachable, it sends a RERRintra message to its precursor nodes, which use the current node as the next hop for some segments. Multiple segments will be broken if they all use the broken link as a hop. The RERRintra contains the broken link and the end nodes of the broken segments. Upon receiving the RERRintra, a precursor node searches for the broken segments from its routing table and sets the state of the broken segments to invalidated. If the current node is not the start node for some of the broken segments, it relays the RERRintra until the message reaches the start nodes of all broken segments.

When the start node of a broken segment knows the intrasegment route error, it tries the intrasegment route repair and becomes the initiator of the route repair. The initiator broadcasts an intrasegment route request (RREQintra) looking for the end node of the broken segment, and the TTL

of the RREQintra is set to MAX_SEGMENT LEN. Intermediate nodes that receive the RREQintra forward the message if it is not a duplicate and its TTL value is greater than zero and discard the message otherwise. When the target receives the first RREQintra message, it sends a RREPintra message to the initiator. The RREPintra is relayed as a unicast message along the reverse path from the target to the initiator. Intrasegment route repair succeeds if the initiator receives the RREPintra.

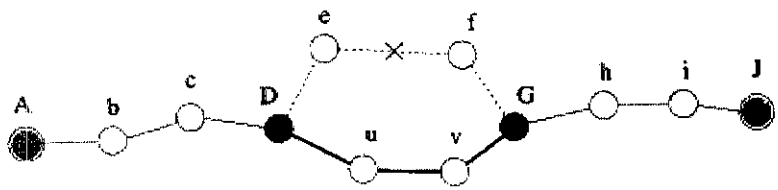


Fig. 3.1 Intrasegment route repair

Fig. 3.1 shows an example in which the original path for segment DG is

D _ e _ f _ G

and the link between e and f breaks. An intrasegment route discovery succeeds and the new path for the segment DG is

D _ u _ v _ G.

During the period from initiating the intrasegment route repair to knowing the result of the repair, the initiator buffers data packets it receives that use the route under repair. If the intrasegment repair succeeds, the initiator transmits the buffered data packets; otherwise, it tries intersegment repair and keeps buffering undeliverable data packets. If the intrasegment route repair fails, the initiator will try the intersegment route repair.

Intersegment Route Repair

The initiator knows the failure of an intrasegment route repair by using a timer called Intra_CheckReplied, which is set when the intrasegment route repair is initiated. If the timer expires and no path to the end node of the broken segment exists, the initiator concludes an unsuccessful intrasegment route repair and starts an intersegment route repair. Briefly, the process of the intersegment route repair
includes discovering routes to downstream waypoint nodes on the route, repairing the broken route if the route discovery succeeds, and sending the repair result to the source.

A RERRinter message can be any of the following three types:

REPAIR: It informs that the route was broken, and the intersegment route repair succeeded. The repaired route is contained in the message.

BROKEN: It informs that the route was broken and the intersegment route repair failed. The source node may need to start another global route discovery.

LOOP: It removes a loop on the route. This type of RERRinter message is used for loop detection.

These three types of RERRinter messages are denoted as RERRrepair_inter , RERRbroken_inter and RERRloop_inter, respectively.

As for the details of the intersegment route repair, the initiator starts a localized intersegment route discovery by broadcasting a RREQinter message. The discovery process is similar to the global intersegment route discovery. While the intrasegment route repair discovers a path to the end node of the broken segment in the range of one segment (MAX_SEGMENT_ LEN), the intersegment route repair discovers a path to the end node of the segment after the broken segment in the range of two

segments (2 _ MAX_SEGMENT_LEN). If the broken segment is the last segment of the route, the target of the route discovery will be the destination.

Upon receiving the RREQinter message, intermediate nodes handle the message as before. The RREQinter is propagated if it is not a duplicate and its TTL is greater than zero. When the target receives the RREQinter, it divides the path recorded in the RREQinter into segments if the path is longer than DEFAULT_ SEGMENT_LEN hops. The target sends an intersegment route reply (RREPinter) to the initiator. After receiving the RREPinter, the initiator repairs the route by replacing the broken segment and a few old downstream segments, which are no longer used by the repaired route, with the new segments returned by the RREPinter. Then the initiator updates its route cache, sends a RERRrepair_inter message containing the repaired route to the source through upstream waypoint nodes, and transmits buffered data packets using the repaired route. Upon receiving the RERRrepair_inter , upstream waypoint nodes update their route cache and transmit buffered data packets using the repaired route.

4. PERFORMANCE EVALUATION

4.1 SIMULATION ENVIRONMENT

An extensive simulations are conducted to evaluate the performance of DOA and compare it with AODV and DSR. The simulations were implemented on GloMoSim 2.03, a scalable simulation environment for wireless network systems. GloMoSim uses the parallel discrete-event simulation capability provided by PARSEC.

The simulation parameters used are as follows:

PARAMETERS	VALUES
Network size	10nodes
Radio Range	250m
Packet size	512bytes
Terrain-Dimensions	(500,500)
Node-Placement	Uniform
GUI Option	YES
Routing Protocols	AODV and DSR

Table.4.1 Simulation Parameters

The MAC layer protocol used in the simulations was the IEEE standard 802.11 Distributed Coordination Function (DCF) [24]. DCF uses Request-to-Send (RTS) and Clear-to- Send (CTS) control packets for unicast data transmissions. Broadcast data packets and RTS control packets are sent using the unslotted Carrier Sense Multiple Access protocol with Collision Avoidance (CSMA/CA). The simulations used the two ray propagation

model, 2 Mb/sec radio bandwidth, and 250m radio range. The traffic was constant bit rate (CBR). The source and the destination of each CBR flow were randomly selected but not identical. Each flow did not change its source and destination for the lifetime of a simulation run. Each source transmitted data packets at a rate of four 512-byte data packets per second. Exceptions will be noted otherwise. We ran simulations for networks with 5 to 10 nodes to see collisions, end-to-end delay and control overhead.

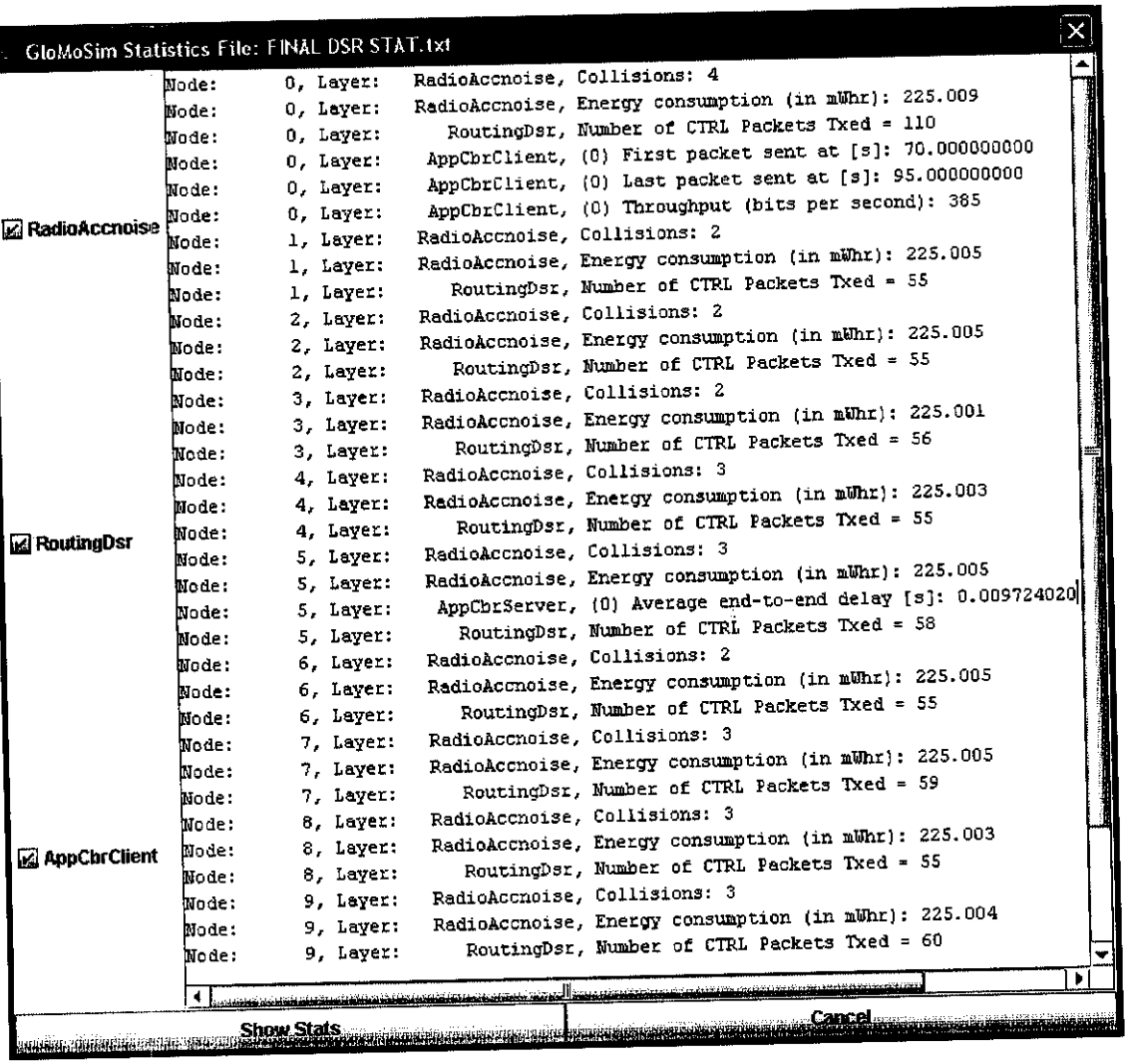
The primary purpose of the Java VT is to help network designers debug their protocols. It is written in Java to provide portability across multiple platforms. The GloMoSim simulation can be run with or without the VT. If run without the VT, it can just be executed from the command line. However, if run with the VT, it must be executed through the GUI provided by the VT rather than from the command line.

There are basically two ways to run GloMoSim with the VT real time and play back. If we choose "Real Time" from the Simulate menu, then the VT will display the results of GloMoSim while GloMoSim is running. If we wish to run GloMoSim first and then play the results of the simulation back later, then we need to do the following.

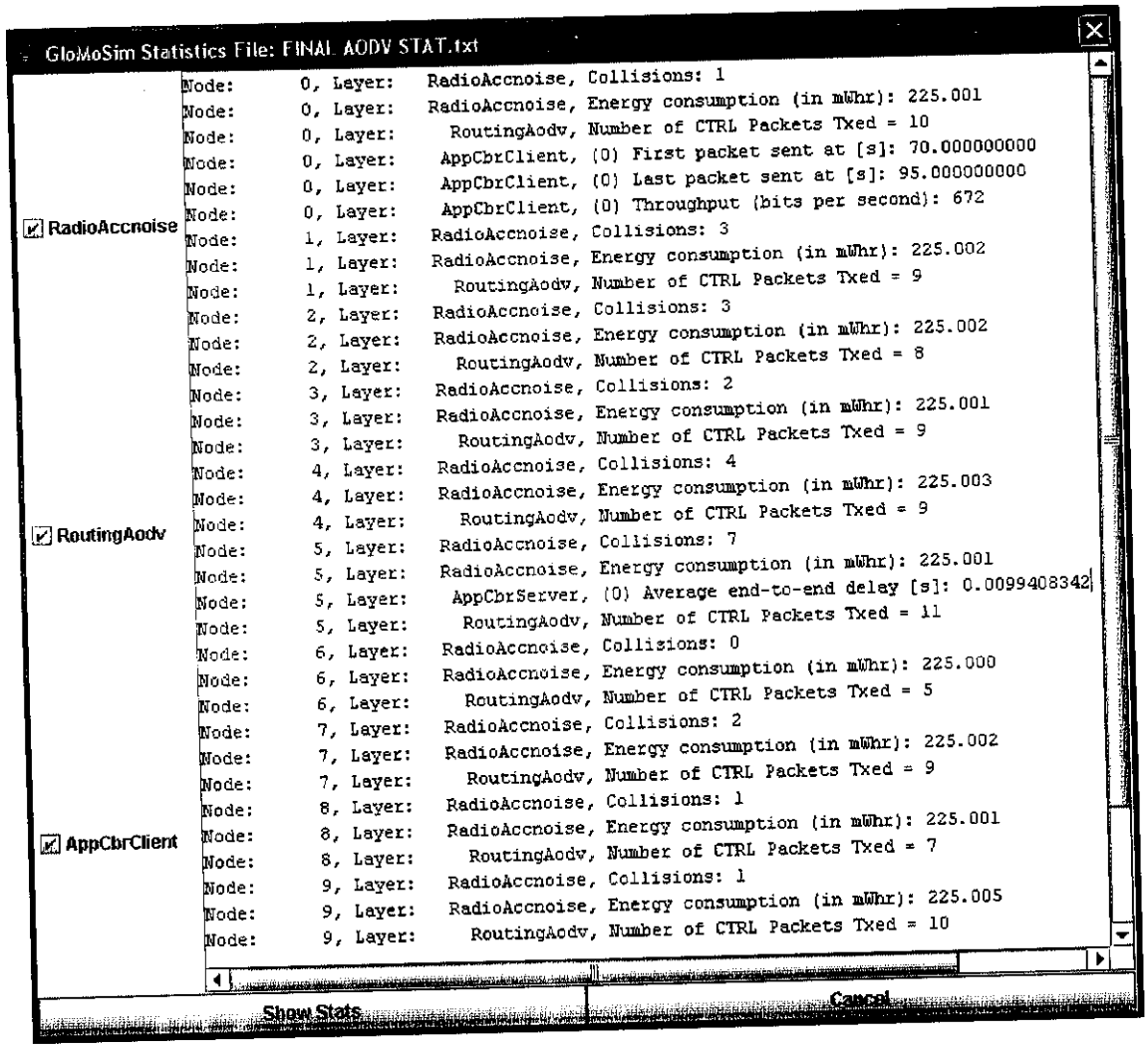
- Choose "Write Trace" from the Simulate menu. A dialog will pop up asking for the name of the executable and the name of the trace file to write the output from GloMoSim to.
- Once the trace file has been written to, you can play it back by choosing "Play Back" from the Simulate menu. A dialog will pop up asking for the name of the trace file to play back.

4.2 SIMULATION RESULTS

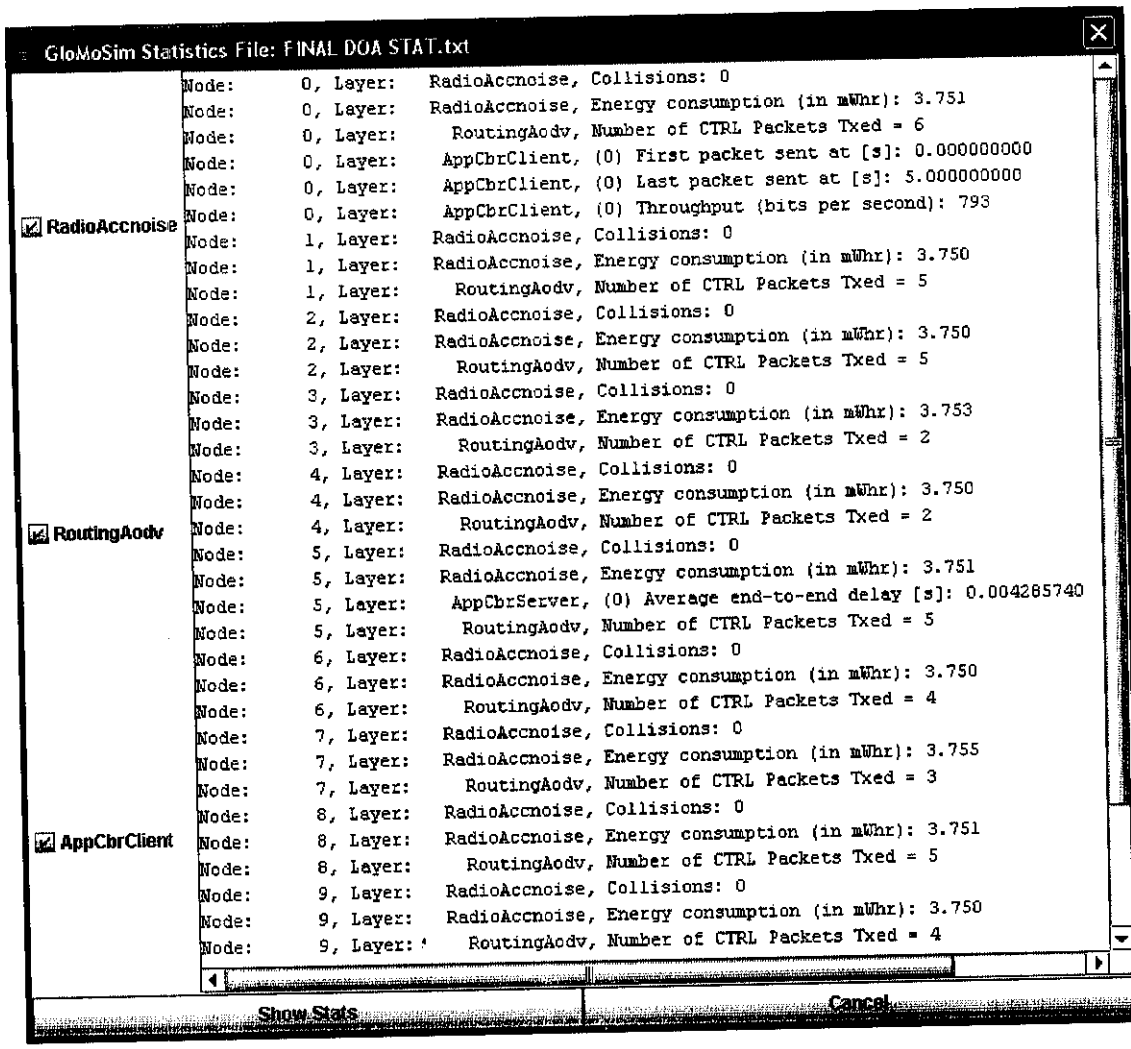
STATISTICS FOR THE SIMULATION OF DSR PROTOCOL:



STATISTICS FOR THE SIMULATION OF AODV PROTOCOL:



STATISTICS FOR THE SIMULATION OF DOA PROTOCOL:



5. CONCLUSION

This protocol uses way point routing model which has a great potential for real time applications. WPR divides an active route into segments by selecting a number of nodes on the route as waypoint nodes. An intersegment routing protocol runs globally, while an intrasegment routing protocol runs locally. Thus, route maintenance can be localized to one or a few neighboring segments. One distinct advantage of WPR is that when a route is broken because of node mobility or node failure, instead of discarding the whole route and discovering a new route from the source to the destination, only the two waypoint nodes of the broken segment need to find a new segment. This will have a clear performance advantage in terms of routing overhead, end-to-end delay, etc. This is the first work to combine DSR and AODV, two well-known on-demand routing protocols, in a hierarchical manner.

6. FUTURE ENHANCEMENTS

In this model, waypoints nodes are selected as just the end nodes of each segment. So our future work includes utilizing heuristic methods to select waypoint nodes, e.g., selecting nodes with lower speed; using common waypoint nodes for different active routes to facilitate the hierarchy maintenance; securing the WPR routing model and particularly DOA routing protocol; and considering other routing protocols for intersegment routing and intrasegment routing in WPR. Also Loop detection and Multi-target route discovery are done in future to enhance this model.

7. APPENDICES

APPENDIX 1

SOURCE CODE

```
#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 "aodv.h"
#include "ip.h"
#include "nwip.h"
#include "nwcommon.h"
#include "application.h"
#include "transport.h"
#include "java_gui.h"
#include "aodv.h"

#define max(a,b)        ((a > b) ? a : b)

//RoutingAodvHandleProtocolPacket

void RoutingAodvHandleProtocolPacket(GlomoNode *node, Message *msg,
        NODE_ADDR srcAddr, NODE_ADDR destAddr, int ttl)
{
        AODV_PacketType *aodvHeader =
        (AODV_PacketType*)GLOMO_MsgReturnPacket(msg);
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

        switch (*aodvHeader)
        {
                case AODV_RREQ:
                {
                        RoutingAodvHandleRequest(node, msg, ttl);
                        if(RoutingAodvCheckNbrExist(node,srcAddr))
                        {
```

```
                RoutingAodvUpdateLastPacketTime(node, srcAddr);
        }
        break;
}

case AODV_RREP:
{
        if(destAddr==ANY_DEST)
        {
                assert(ttl==0);
                RoutingAodvHandleHello(node,msg,srcAddr);
        }
        else
        {
                aodv->lastpkt = simclock();
                RoutingAodvHandleReply(node, msg, srcAddr, destAddr);
                if(RoutingAodvCheckNbrExist(node,srcAddr))
                {
                        RoutingAodvUpdateLastPacketTime(node, srcAddr);
                }
        }
        break;
}

case AODV_RREP_ACK:
{
        aodv->lastpkt = simclock();
        RoutingAodvHandleReplyAck(node,msg);
        if(RoutingAodvCheckNbrExist(node,srcAddr))
        {
                RoutingAodvUpdateLastPacketTime(node, srcAddr);
        }
        break;
}
case AODV_RERR:
{
        RoutingAodvHandleRouteError(node, msg, srcAddr);
        if(RoutingAodvCheckNbrExist(node,srcAddr))
        {
                RoutingAodvUpdateLastPacketTime(node, srcAddr);
        }
        break;
}
default:
assert(FALSE); abort();
break;
```

```
        }
}

// RoutingAodvHandleProtocolEvent

void RoutingAodvHandleProtocolEvent(GlomoNode *node, Message *msg)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        Message *destUnrchMsg;
        NODE_ADDR *unrchDest;
        char clockdisplay[100];
        GLOMO_PrintClockInSecond(simclock(),clockdisplay);
        switch (msg->eventType)
        {
                case MSG_NETWORK_FlushTables:
                {
                        RoutingAodvDeleteSeenTable(&aodv->seenTable);
                        GLOMO_MsgFree(node, msg);
                        break;
                }
                case MSG_AODV_DeleteRouteEntry:
                {
                        NODE_ADDR *destAddr = (NODE_ADDR
                                *)GLOMO_MsgReturnInfo(msg);
                        RoutingAodvDeleteRouteTable(node,*destAddr);
                        GLOMO_MsgFree(node, msg);
                        break;
                }
                case MSG_AODV_CheckLocalRepairSuccessful:
                {
                        AODV_LR_TimerInfo *destHop = (AODV_LR_TimerInfo
                                *)GLOMO_MsgReturnInfo(msg);
                        if(RoutingAodvCheckRouteExist(destHop->destAddr,
                                &aodv->routeTable))
                        {
                                aodv->stats.numSuccessfulLocalRepair++;
                                GLOMO_MsgFree(node,msg);
                        }
                        else
                        {
                        while (RoutingAodvLookupBuffer(destHop->destAddr,
                         &aodv->buffer))
                        {
                                Message* messageToDelete =
                                RoutingAodvGetBufferedPacket(destHop->destAddr,
```

```
                    &aodv->buffer);
                    RoutingAodvDeleteBuffer(destHop->destAddr,
                     &aodv->buffer);
                    GLOMO_MsgFree(node, messageToDelete);
                    aodv->stats.numPacketsDropped++;
                }
                RoutingAodvResetBeingRepaired(destHop->destAddr,
                        &aodv->routeTable);
                RoutingAodvInitiateRERROnLinkBreak(node,destHop->nextHop);
               GLOMO_MsgFree(node,msg);
        }
        break;
}
case MSG_NETWORK_CheckReplied:
{
        NODE_ADDR *destAddr = (NODE_ADDR
                *)GLOMO_MsgReturnInfo(msg);
        if (!RoutingAodvCheckRouteExist(*destAddr, &aodv->routeTable))
        {
        if (RoutingAodvGetTimes(*destAddr, &aodv->sent) < RREQ_RETRIES)
        {
                RoutingAodvRetryRREQ(node, *destAddr);
        }
        else
        {
                while (RoutingAodvLookupBuffer(*destAddr, &aodv->buffer))
                {
                        Message* messageToDelete =
                                RoutingAodvGetBufferedPacket(*destAddr,
                                 &aodv->buffer);
                    RoutingAodvDeleteBuffer(*destAddr, &aodv->buffer);
                                                GLOMO_MsgFree(node,
                    messageToDelete);
                     aodv->stats.numPacketsDropped++;
                }
          #ifdef CBR_TRAFFIC
             destUnrchMsg =
             GLOMO_MsgAlloc(node,GLOMO_APP_LAYER,APP_CBR_CLIENT,
             MSG_AODV_CBR_DestinationUnreachable);
             GLOMO_MsgInfoAlloc(node,destUnrchMsg,sizeof(NODE_ADDR));
             unrchDest =(NODE_ADDR *) GLOMO_MsgReturnInfo(destUnrchMsg);
             *unrchDest = *destAddr;
             GLOMO_MsgSend(node,destUnrchMsg,0);
             aodv->stats.numDestUnrchSent++;
           #endif
        }
```

```
        }
            GLOMO_MsgFree(node, msg);
            break;
        };

        case MSG_AODV_HELLO_EVENT:
        {
            #ifdef HELLO_PACKETS
            if((aodv->lastbcast + HELLO_INTERVAL <= simclock())&&
                    (RoutingAodvIfMePartOfActiveRoute(node)))
            {
                    RoutingAodvInitiateHELLO(node);
            }
            else
                    RoutingAodvSetTimer(node, MSG_AODV_HELLO_EVENT,
                    ANY_DEST, (clocktype)HELLO_INTERVAL);
            #endif
            GLOMO_MsgFree(node,msg);
            break;
        }
        default:
            fprintf(stderr, "RoutingAodv: Unknown MSG type %d!\n",
                        msg->eventType);
            abort();
        }

}

//RoutingAodvHandleRequest

void RoutingAodvHandleRequest(GlomoNode *node, Message *msg, int ttl)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        AODV_RREQ_Packet *rreqPkt = (AODV_RREQ_Packet *)
                GLOMO_MsgReturnPacket(msg);
        Message *newMsg;
        IpHeaderType *ipHeader;
        RoutingAodvReplaceInsertRouteTable(node, rreqPkt->lastAddr,-
                1,FALSE,TRUE,1,rreqPkt->lastAddr,simclock()+
                (clocktype)ACTIVE_ROUTE_TIMEOUT);
        if(RoutingAodvLookupSeenTable(rreqPkt->origAddr,rreqPkt->bcastId,
                &aodv->seenTable))
        {
                GLOMO_MsgFree(node,msg);
```

```
        return;
}
aodv->lastpkt = simclock();
rreqPkt->hopCount++;
RoutingAodvInsertSeenTable(node, rreqPkt->origAddr, rreqPkt->bcastId,
        &aodv->seenTable);
if(!RoutingAodvCheckRouteExist(rreqPkt->origAddr,&aodv->routeTable))
{
        clocktype lifetime = RoutingAodvGetMinimalLifetime(
        rreqPkt->hopCount);
        RoutingAodvReplaceInsertRouteTable(node, rreqPkt->origAddr,
                rreqPkt->origSeq,TRUE,TRUE,rreqPkt->hopCount,
                rreqPkt->lastAddr,lifetime);
}
else
{
        clocktype lifetime = max(RoutingAodvGetLifetime(rreqPkt->origAddr,
                &aodv->routeTable),RoutingAodvGetMinimalLifetime(
                rreqPkt->hopCount));
        int seq = RoutingAodvGetSeq(rreqPkt->origAddr,&aodv->routeTable);
        RoutingAodvReplaceInsertRouteTable(node,
                rreqPkt->origAddr,max(seq,rreqPkt->origSeq),
                TRUE,TRUE,rreqPkt->hopCount,rreqPkt->lastAddr,lifetime);
}

if(rreqPkt->destAddr==node->nodeAddr)
{
        RoutingAodvInitiateRREP(node,msg);
}
else
if((rreqPkt->destinationOnly==FALSE)&&
        (RoutingAodvCheckRouteExist(rreqPkt->destAddr,
        &aodv->routeTable))&&(RoutingAodvIfSeqValid(rreqPkt->destAddr,
        &aodv->routeTable))&&(RoutingAodvGetSeq(rreqPkt->destAddr,
        &aodv->routeTable)>=rreqPkt->destSeq))
{
        RoutingAodvInitiateRREPbyIN(node,msg);
}
else
if(ttl==0)
{
        GLOMO_MsgFree(node,msg);
}
else
{
        RoutingAodvRelayRREQ(node,msg,ttl);
```

```
        }
        while(RoutingAodvLookupBuffer(rreqPkt->lastAddr, &aodv->buffer))
        {
                newMsg = RoutingAodvGetBufferedPacket(rreqPkt->lastAddr,
                        &aodv->buffer);
                ipHeader = (IpHeaderType *) newMsg->packet;
                if(ipHeader->ip_src==node->nodeAddr)
                aodv->stats.numDataSent++;
                RoutingAodvTransmitData(node, newMsg, rreqPkt->lastAddr);
                RoutingAodvDeleteBuffer(rreqPkt->lastAddr, &aodv->buffer);
        }
        while(RoutingAodvLookupBuffer(rreqPkt->origAddr, &aodv->buffer))
        {
                newMsg = RoutingAodvGetBufferedPacket(rreqPkt->origAddr,
                        &aodv->buffer);
                ipHeader = (IpHeaderType *) newMsg->packet;
                if(ipHeader->ip_src==node->nodeAddr)
                aodv->stats.numDataSent++;
                RoutingAodvTransmitData(node, newMsg, rreqPkt->origAddr);
                RoutingAodvDeleteBuffer(rreqPkt->origAddr, &aodv->buffer);
        }
}

//RoutingAodvCheckNbrExist

BOOL RoutingAodvCheckNbrExist(GlomoNode *node, NODE_ADDR nbrAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        AODV_NT nbrTable = aodv->nbrTable;
        AODV_NT_Node *current;
        if (nbrTable.size == 0)
        {
                return (FALSE);
        }
        for (current = nbrTable.head;current != NULL && current->nbrAddr <=
 nbrAddr;
                current = current->next)
        {
                if (current->nbrAddr == nbrAddr)
                {
                        return(TRUE);
                }
        }
        return (FALSE);
  }
```

```
// RoutingAodvHandleHello

void RoutingAodvHandleHello(GlomoNode *node,Message *msg, NODE_ADDR
nbrAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        AODV_RREP_Packet* helloPkt =
        (AODV_RREP_Packet*)GLOMO_MsgReturnPacket(msg);
        clocktype oldlifetime;
        assert(nbrAddr==helloPkt->destAddr);
        if(RoutingAodvCheckRouteExist(nbrAddr,&aodv->routeTable))
        {
                oldlifetime = RoutingAodvGetLifetime(nbrAddr,&aodv->routeTable);
                if(oldlifetime < simclock() +
                        ALLOWED_HELLO_LOSS * HELLO_INTERVAL)
                {
                        oldlifetime = simclock() +
                        ALLOWED_HELLO_LOSS * HELLO_INTERVAL;
                }
                RoutingAodvReplaceInsertRouteTable(node,nbrAddr,helloPkt->destSeq,
                        TRUE,TRUE,1,nbrAddr,oldlifetime);

         }
        else
        {
                RoutingAodvReplaceInsertRouteTable(node,nbrAddr,helloPkt->destSeq,
                        TRUE,TRUE,1,nbrAddr,simclock() +
                        ALLOWED_HELLO_LOSS * HELLO_INTERVAL);

         }
        if(RoutingAodvCheckNbrExist(node,nbrAddr))
        {
                RoutingAodvUpdateLastHelloTime(node,nbrAddr);
                RoutingAodvUpdateLastPacketTime(node,nbrAddr);

         }
         else
         {
                RoutingAodvInsertNbrTable(node,nbrAddr);

         }
         GLOMO_MsgFree(node,msg);
}

//RoutingAodvHandleReply
```

```
void RoutingAodvHandleReply(GlomoNode *node, Message *msg, NODE_ADDR
srcAddr,
        NODE_ADDR destAddr)
        {
                GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
                GlomoRoutingAodv* aodv = (GlomoRoutingAodv *)
                ipLayer->routingProtocol;
                Message *newMsg;
                AODV_RREP_Packet*rrepPkt=(AODV_RREP_Packet *)
                GLOMO_MsgReturnPacket(msg);
                BOOL causedNewRoute = FALSE;
                BOOL wasBeingRepaired = FALSE;
                int oldHopCount = -1;
                char clockdisplay[100];
                IpHeaderType *ipHeader;
                NODE_ADDR newPath[AODV_MAX_SR_LEN];
                int segLeft;
                int i, j, k;
                segLeft = rrepPkt->segLeft - 1;
                assert(destAddr==node->nodeAddr);
                #ifdef LOCAL_REPAIR
                wasBeingRepaired=RoutingAodvCheckBeingRepaired(
                rrepPkt->destAddr,&aodv->routeTable);
                if(wasBeingRepaired)
                {
                        oldHopCount=RoutingAodvGetHopCount(rrepPkt->destAddr,
                        &aodv->routeTable);
                }
                #endif
                if(rrepPkt->ackReqd==TRUE)
                {
                        RoutingAodvInitiateRREPACK(node,msg,srcAddr);
                }
                RoutingAodvReplaceInsertRouteTable(node,srcAddr,-1,
                        FALSE,TRUE,1,srcAddr,simclock()+ rrepPkt->lifetime);
                if(srcAddr==rrepPkt->destAddr&&
                        RoutingAodvIfSeqValid(srcAddr,&aodv->routeTable))
                {
                        causedNewRoute = TRUE;
                }
                rrepPkt->hopCount++;
                if(!RoutingAodvCheckRouteEntryExist(rrepPkt->destAddr,
                        &aodv->routeTable))
                {
                        RoutingAodvReplaceInsertRouteTable(node,rrepPkt->destAddr,
```

```
                    rrepPkt->destSeq,TRUE,TRUE,
                    rrepPkt->hopCount,srcAddr,simclock()+rrepPkt->lifetime);

                causedNewRoute = TRUE;
            }
            else
            {
                BOOLseqInvalid=FALSE,seqGreater=FALSE,routeInactive=
                FALSE,smallerHopCount=FALSE;seqInvalid=
                !RoutingAodvIfSeqValid(rrepPkt->destAddr,&aodv->routeTable);
                if(!seqInvalid)
                {
                    int seq = RoutingAodvGetSeq(rrepPkt->destAddr,
                    &aodv->routeTable);
                    if(seq < rrepPkt->destSeq)
                    {
                        seqGreater = TRUE;
                    }
                    if((seq==rrepPkt->destSeq)&&
                    (RoutingAodvIfRouteInactive(rrepPkt->destAddr,
                    &aodv->routeTable)))
                    {
                        routeInactive = TRUE;
                    }
                    if((seq==rrepPkt->destSeq)&&(rrepPkt->hopCount<
                    RoutingAodvGetHopCount(rrepPkt->destAddr,
                    &aodv->routeTable)))
                    {
                        smallerHopCount = TRUE;
                    }
                }
                if(seqInvalid||seqGreater||routeInactive||smallerHopCount)
                {
                 RoutingAodvReplaceInsertRouteTable(node,
                        rrepPkt->destAddr,rrepPkt->destSeq,
                        TRUE,TRUE,rrepPkt->hopCount,
                        srcAddr,simclock()+rrepPkt->lifetime);
                  causedNewRoute = TRUE;
                 }
            }

        GLOMO_MsgFree(node, msg);
}
void RoutingAodvHandleReplyAck(GlomoNode *node, Message *msg)
{
        AODV_RREP_ACK_Packet*rrepAckPkt=AODV_RREP_ACK_Packet*)
```

```
            GLOMO_MsgReturnPacket(msg);
        GLOMO_MsgFree(node,msg);

}

// RoutingAodvHandleRouteError

void RoutingAodvHandleRouteError(GlomoNode *node, Message *msg, NODE_ADDR
srcAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        AODV_RERR_Packet* rerrPkt
                =(AODV_RERR_Packet*)GLOMO_MsgReturnPacket(msg);
        AODV_RERR_Packet newRerrPacket;
        int I;
        AODV_PL precursorList;
        precursorList.size=0;
        precursorList.head = NULL;
        precursorList.tail = NULL;
        newRerrPacket.pktType = AODV_RERR;
        newRerrPacket.destinationCount = 0;
        newRerrPacket.N = rerrPkt->N;
        for(I = 0; I < rerrPkt->destinationCount; I++)
        {
                NODE_ADDR destAddr;
                int seqNum;
                destAddr = rerrPkt->destinationPairArray[I].destinationAddress;
                seqNum = rerrPkt->destinationPairArray[I].destinationSequenceNumber;
                if(RoutingAodvCheckRouteExist(destAddr,&aodv->routeTable))
                {
                        if(RoutingAodvGetNextHop(destAddr,
                        &aodv->routeTable)==srcAddr)
                        {
                                AODV_RT_Node *current;
                                if(rerrPkt->N==FALSE)
                                {
                                        current = aodv->routeTable.head;
                                        while(current->destAddr!=destAddr)
                                        {
                                                current=current->next;
                                        }
                                        if(seqNum!=-1)
                                        {
                                                current->destSeqValid = TRUE;
                                                current->destSeq = seqNum;
```

```
                        }
                        current->valid = FALSE;
                        current->lifetime = simclock() +
                        RoutingAodvGetDeletePeriod();
                    NetworkIpDeleteOutboundPacketsToANode
                        (node, srcAddr, destAddr, FALSE);
                    }

    newRerrPacket.destinationPairArray[newRerrPacket.destinationCount].
                        destinationAddress = destAddr;

    newRerrPacket.destinationPairArray[newRerrPacket.destinationCount].
                        destinationSequenceNumber = seqNum;
                    newRerrPacket.destinationCount++;

    RoutingAodvGetPrecursors(node,destAddr,&precursorList);

                }
            }
        }
        if (newRerrPacket.destinationCount > 0)
        {
            if(precursorList.size>0)
            {
                SendRouteErrorPacket(node, &newRerrPacket,&precursorList);
                aodv->stats.numRerrSent++;
            }
        }
        GLOMO_MsgFree(node, msg);
}

void RoutingAodvInitRouteCache(AODV_RouteCache *routeCache)
{
    routeCache->head = NULL;
    routeCache->count = 0;
}

//RoutingAodvDeleteSeenTable

void RoutingAodvDeleteSeenTable(AODV_RST *seenTable)
{
    AODV_RST_Node *toFree;
    toFree = seenTable->front;
    seenTable->front = toFree->next;
    pc_free(toFree);
    --(seenTable->size);
```

```
    if (seenTable->size == 0)
    {
        seenTable->rear = NULL;
    }
}

// RoutingAodvDeleteRouteTable

void RoutingAodvDeleteRouteTable(GlomoNode *node,NODE_ADDR destAddr)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
        node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    AODV_RT *routeTable = &aodv->routeTable;
    AODV_RT_Node *toFree;
    AODV_RT_Node *current;
    AODV_PL_Node *plnode;
    if (routeTable->size == 0 || routeTable->head == NULL)
    {
        return;
    }
    else if (routeTable->head->destAddr == destAddr)
    {
        if ((routeTable->head->lifetime <= simclock())&&
        (routeTable->head->valid==FALSE))
        {
            toFree = routeTable->head;
            routeTable->head = toFree->next;
            plnode = toFree->precursorList.head;
            while(plnode!=NULL)
            {
                toFree->precursorList.head = plnode->next;
                pc_free(plnode);
                plnode = toFree->precursorList.head;
            }
            pc_free(toFree);
            --(routeTable->size);
        }
    }
    else
    {
        for (current = routeTable->head;current->next != NULL &&
            current->next->destAddr < destAddr;current = current->next)
        {
        }
        if (current->next != NULL && current->next->destAddr == destAddr &&
```

```
                current->next->lifetime <= simclock() &&
                (current->next->valid==FALSE))
        {
                toFree = current->next;
                current->next = toFree->next;
                plnode = toFree->precursorList.head;
                while(plnode!=NULL)
                {
                        toFree->precursorList.head = plnode->next;
                        pc_free(plnode);
                        plnode = toFree->precursorList.head;
                }
                pc_free(toFree);
                --(routeTable->size);
        }
    }
}

//RoutingAodvCheckRouteExist

BOOL RoutingAodvCheckRouteExist(NODE_ADDR destAddr, AODV_RT
*routeTable)
{
    AODV_RT_Node *current;
    if (routeTable->size == 0)
    {
        return (FALSE);
    }

    for (current = routeTable->head;current != NULL &&
        current->destAddr <= destAddr;
        current = current->next)
    {
        if ((current->destAddr == destAddr) &&
            (current->lifetime > simclock()) && (current->valid == TRUE))
        {
            return(TRUE);
        }
    }
    return (FALSE);
}

// RoutingAodvCheckRouteEntryExist
```

```
BOOL RoutingAodvCheckRouteEntryExist(NODE_ADDR destAddr,
        AODV_RT *routeTable)
{
        AODV_RT_Node *current;
        if(routeTable->size==0)
        {
                return(FALSE);
        }
        for(current=routeTable->head;current !=NULL &&
                current->destAddr <=destAddr;
                current = current->next)
        {
                if(current->destAddr==destAddr)
                {
                        return TRUE;
                }
        }
        return FALSE;
}

//RoutingAodvCheckBeingRepaired

BOOL RoutingAodvCheckBeingRepaired(NODE_ADDR destAddr,
        AODV_RT *routeTable)
{
        AODV_RT_Node *current;
        if(routeTable->size==0)
        {
                return(FALSE);
        }
        for(current=routeTable->head;current !=NULL &&
                current->destAddr <=destAddr;
                current = current->next)
        {
                if(current->destAddr==destAddr)
                {
                        if(current->beingRepaired==TRUE)
                        {
                                return TRUE;
                        }
                        else
                                return FALSE;
                }
        }
        return FALSE;
}
```

```
// RoutingAodvLookupBuffer

BOOL RoutingAodvLookupBuffer(NODE_ADDR destAddr, AODV_BUFFER *buffer)
{
        AODV_BUFFER_Node *current;
        if (buffer->size == 0)
        {
                return (FALSE);
        }
        for (current = buffer->head;current != NULL &&
                current->destAddr <= destAddr;
                current = current->next)
        {
                if (current->destAddr == destAddr)
                {
                        return(TRUE);
                }
        }
        return (FALSE);
  }

// RoutingAodvDeleteBuffer

BOOL RoutingAodvDeleteBuffer(NODE_ADDR destAddr, AODV_BUFFER *buffer)
{
        AODV_BUFFER_Node *toFree;
        AODV_BUFFER_Node *current;
        BOOL deleted;
        if (buffer->size == 0)
        {
                deleted = FALSE;
        }
        else if (buffer->head->destAddr == destAddr)
        {
                toFree = buffer->head;
                buffer->head = toFree->next;
                pc_free(toFree);
                --(buffer->size);
                deleted = TRUE;
        }
        else
        {
                for (current = buffer->head;
                current->next != NULL && current->next->destAddr < destAddr;
                current = current->next)
```

```
            {
            }
            if (current->next != NULL && current->next->destAddr == destAddr)
            {
                    toFree = current->next;
                    current->next = toFree->next;
                    pc_free(toFree);
                    --(buffer->size);
                    deleted = TRUE;
            }
            else
            {
                     deleted = FALSE;
            }
        }
        return (deleted);
}

//RoutingAodvStartLocalRepair

void RoutingAodvStartLocalRepair(GlomoNode *node, NODE_ADDR origAddr,
NODE_ADDR destAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        NODE_ADDR nextHop;
        clocktype discovery_period;
         int ttl;
        Message *newMsg,*timerMsg;
        char *pktPtr;
        int pktSize = sizeof(AODV_RREQ_Packet);
        int MIN_REPAIR_TTL, hopstoorig;
        AODV_LR_TimerInfo *info;
        char clockdisplay[100];
        AODV_RREQ_Packet *rreqPkt;
        MIN_REPAIR_TTL = RoutingAodvGetHopCount(destAddr,
                &aodv->routeTable);
        hopstoorig = RoutingAodvGetHopCount(origAddr,&aodv->routeTable);
         ttl = max(MIN_REPAIR_TTL, 0.5 * hopstoorig) + LOCAL_ADD_TTL;
        discovery_period = RoutingAodvGetRingTraversalTime(ttl);
        nextHop = RoutingAodvGetNextHop(destAddr,&aodv->routeTable);
        RoutingAodvIncreaseSeq(node);
        newMsg = GLOMO_MsgAlloc(node, GLOMO_MAC_LAYER, 0,
                MSG_MAC_FromNetwork);
        GLOMO_MsgPacketAlloc(node, newMsg, pktSize);
```

```
    aodv->stats.numAttemptsLocalRepair++;
    pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
    rreqPkt = (AODV_RREQ_Packet *) pktPtr;
     rreqPkt->pktType = AODV_RREQ;
     rreqPkt->bcastId = RoutingAodvGetBcastId(node);
    rreqPkt->destAddr = destAddr;
     rreqPkt->destSeq = RoutingAodvGetSeq(destAddr, &aodv->routeTable);
    rreqPkt->unknownSeqNo = FALSE;
    rreqPkt->origAddr = node->nodeAddr;
    rreqPkt->origSeq = RoutingAodvGetMySeq(node);
    rreqPkt->lastAddr = node->nodeAddr;
    rreqPkt->hopCount = 0;
    #ifdef GRATUITOUS
            rreqPkt->gratuitousRREP=TRUE;
    #else
    rreqPkt->gratuitousRREP=FALSE;
    #endif

     #ifdef DESTINATION_ONLY
            rreqPkt->destinationOnly=TRUE;
     #else
            rreqPkt->destinationOnly=FALSE;
     #endif
     #ifdef HELLO_PACKETS
            aodv->lastbcast = simclock();
     #endif
     NetworkIpSendRawGlomoMessage(node, newMsg, ANY_DEST,
            CONTROL, IPPROTO_AODV, ttl);
     timerMsg =
            GLOMO_MsgAlloc(node,GLOMO_NETWORK_LAYER,
            ROUTING_PROTOCOL_AODV,MSG_AODV_CheckLocalRepairSucce
     ssful);
      GLOMO_MsgInfoAlloc(node, timerMsg, sizeof(AODV_LR_TimerInfo));
     info = (AODV_LR_TimerInfo *) GLOMO_MsgReturnInfo(timerMsg);
     info->destAddr = destAddr;
     info->nextHop = nextHop;
     GLOMO_MsgSend(node, timerMsg, discovery_period);
     aodv->stats.numRequestSent++;
     RoutingAodvInsertSeenTable(node, node->nodeAddr, rreqPkt->bcastId,
            &aodv->seenTable);
}

//RoutingAodvInsertBuffer

static
```

```
void RoutingAodvInsertBuffer(Message* msg,NODE_ADDR destAddr,
AODV_BUFFER* buffer)
{
        AODV_BUFFER_Node* current;
        AODV_BUFFER_Node* previous;
        AODV_BUFFER_Node* newNode =(AODV_BUFFER_Node
                *)checked_pc_malloc(sizeof(AODV_BUFFER_Node));
        newNode->destAddr = destAddr;
        newNode->msg = msg;
        newNode->timestamp = simclock();
        newNode->next = NULL;
        ++(buffer->size);
        previous = NULL;
        current = buffer->head;
        while ((current != NULL) && (current->destAddr <= destAddr))
        {
                previous = current;
                current = current->next;
        }
        if (previous == NULL)
        {
                newNode->next = buffer->head;
                buffer->head = newNode;
        }
        else {
                newNode->next = previous->next;
                previous->next = newNode;
            }
}

//RoutingAodvDeleteNbrTable

void RoutingAodvDeleteNbrTable(GlomoNode *node, NODE_ADDR nbrAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        AODV_NT* nbrTable = &aodv->nbrTable;
        AODV_NT_Node *toFree;
        AODV_NT_Node *current;
        if (nbrTable->size == 0)
        {
                return;
        }
        else if ((nbrTable->head->nbrAddr == nbrAddr) &&
                        (nbrTable->head->lastHello +
                        RoutingAodvGetDeletePeriod() <= simclock()))
```

```
    {
            toFree = nbrTable->head;
            nbrTable->head =toFree->next;
            pc_free(toFree);
            --(nbrTable->size);
            return;
     }
     else
     {
             for (current = nbrTable->head;((current->next != NULL) &&
                     (current->next->nbrAddr < nbrAddr));current = current->next)
              {
              }
             if ((current->next != NULL )&& (current->next->nbrAddr == nbrAddr)
                     && (current->next->lastHello +
                     RoutingAodvGetDeletePeriod() <= simclock()))
              {
                       toFree = current->next;
                       current->next = toFree->next;
                       pc_free(toFree);
                       --(nbrTable->size);
                       return;
               }
        }
}

//RoutingAodvGetPrecursors
 void RoutingAodvGetPrecursors(GlomoNode *node, NODE_ADDR
destAddr,AODV_PL* precursorList)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
        node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        AODV_PL_Node *plnode;
        AODV_RT_Node *current;
        current = aodv->routeTable.head;
        while((current!=NULL)&&(current->destAddr!=destAddr))
        {
                current = current->next;
        }
        assert(current!=NULL);

        plnode = current->precursorList.head;
        while(plnode!=NULL)
        {
                if(RoutingAodvCheckPrecursorList
                (plnode->precursor,precursorList->head) == FALSE)
```

```
            {
                AODV_PL_Node *newNode;
                newNode =
                (AODV_PL_Node*)pc_malloc(sizeof(AODV_PL_Node));
                newNode->precursor = plnode->precursor;
                newNode->next = NULL;

                if(precursorList->head==NULL)
                {
                    precursorList->head = newNode;
                    precursorList->tail = newNode;
                }
                else
                {
                    precursorList->tail->next = newNode;
                    precursorList->tail = newNode;
                }
                precursorList->size++;
            }
            plnode = plnode->next;
        }
}

//RoutingAodvInit
void RoutingAodvInit(GlomoNode *node,GlomoRoutingAodv **aodvPtr,
                const GlomoNodeInput *nodeInput)
{
        GlomoRoutingAodv *aodv =(GlomoRoutingAodv *)checked_pc_malloc
                (sizeof(GlomoRoutingAodv));
        clocktype delay;
        int plusMinus;
        (*aodvPtr) = aodv;
        if (aodv == NULL)
        {
                fprintf(stderr, "AODV: Cannot alloc memory for AODV struct!\n");
                assert (FALSE);
        }
        RoutingAodvInitStats(node);
        RoutingAodvInitRouteTable(&aodv->routeTable);
        RoutingAodvInitNbrTable(&aodv->nbrTable);
        RoutingAodvInitSeenTable(&aodv->seenTable);
        RoutingAodvInitRouteCache(&aodv->routeCacheTable);
        RoutingAodvInitBuffer(&aodv->buffer);
        RoutingAodvInitSent(&aodv->sent);
        RoutingAodvInitSeq(node);
        RoutingAodvInitBcastId(node);
```

```
        NetworkIpSetPacketDropNotificationFunction(node,
                &RoutingAodvPacketDropNotificationHandler);
        #ifdef HELLO_PACKETS
        aodv->lastbcast = 0;
        if(pc_erand(node->seed)<=0.5) plusMinus = 1;else plusMinus = -1;delay =
                pc_erand(node->seed)
                * BROADCAST_JITTER;RoutingAodvSetTimer(node,
                MSG_AODV_HELLO_EVENT,ANY_DEST,HELLO_INTERVAL+plus
                Minus * delay);
                aodv->lastpkt = 0;
        #endif
        NetworkIpSetRouterFunction(node, &RoutingAodvRouterFunction);
}

//RoutingAodvTransmitData
void RoutingAodvTransmitData(GlomoNode *node, Message *msg, NODE_ADDR
destAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)
         node->networkData.networkVar;
        GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
        NODE_ADDR nextHop;
        GLOMO_MsgSetLayer(msg, GLOMO_MAC_LAYER, 0);
        GLOMO_MsgSetEvent(msg, MSG_MAC_FromNetwork);
        nextHop = RoutingAodvGetNextHop(destAddr, &aodv->routeTable);
        assert(nextHop != ANY_DEST);
        aodv->lastpkt = simclock();
        NetworkIpSendPacketToMacLayer(node, msg, DEFAULT_INTERFACE,
nextHop);
        aodv->stats.numDataTxed++;
        RoutingAodvUpdateLifetime(destAddr, &aodv->routeTable);
        RoutingAodvSetTimer(node, MSG_NETWORK_CheckRouteTimeout,
                destAddr, (clocktype)ACTIVE_ROUTE_TIMEOUT);
        RoutingAodvUpdateLifetime(nextHop,&aodv->routeTable);
        RoutingAodvSetTimer(node, MSG_NETWORK_CheckRouteTimeout,
                        nextHop,(clocktype)ACTIVE_ROUTE_TIMEOUT);
}


//RoutingAodvHandleData

void RoutingAodvHandleData(GlomoNode *node, Message *msg, NODE_ADDR
destAddr)
{
        GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node-
>networkData.networkVar;
```

```
GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
IpHeaderType *ipHeader = (IpHeaderType *)GLOMO_MsgReturnPacket(msg);
NODE_ADDR sourceAddress = ipHeader->ip_src;
NODE_ADDR lastHop;
assert(sourceAddress  != node->nodeAddr);
 if(RoutingAodvCheckRouteExist(sourceAddress,&aodv->routeTable)==TRUE)
{
        RoutingAodvUpdateLifetime(sourceAddress, &aodv->routeTable);
        RoutingAodvSetTimer(node, MSG_NETWORK_CheckRouteTimeout,
                sourceAddress, (clocktype)ACTIVE_ROUTE_TIMEOUT);
}
lastHop = RoutingAodvGetNextHop(sourceAddress,&aodv->routeTable);
if(RoutingAodvCheckRouteExist(lastHop,&aodv->routeTable)==TRUE)
{
        RoutingAodvUpdateLifetime(lastHop,&aodv->routeTable);
        RoutingAodvSetTimer(node,MSG_NETWORK_CheckRouteTimeout,
                lastHop,(clocktype)ACTIVE_ROUTE_TIMEOUT);
}

if(RoutingAodvCheckNbrExist(node,lastHop))
{
        RoutingAodvUpdateLastPacketTime(node, lastHop);
 }
 if (destAddr == node->nodeAddr)
 {
        aodv->stats.numDataReceived++;
        aodv->stats.numHops += (64-ipHeader->ip_ttl);
 }
 else if (destAddr != ANY_DEST)
 {
   if (RoutingAodvCheckRouteExist(destAddr, &aodv->routeTable))
   {
        RoutingAodvTransmitData(node, msg, destAddr);
   }
 else
   {
        AODV_RERR_Packet newRerrPacket;
        AODV_PL precursorList;
        #ifdef LOCAL_REPAIR
        if(RoutingAodvCheckBeingRepaired(destAddr,&aodv->routeTable))
        {
                RoutingAodvInsertBuffer(msg, destAddr, &aodv->buffer);
                return;
        }
         if(RoutingAodvCheckRepairable(destAddr,&aodv->routeTable))
         {
```

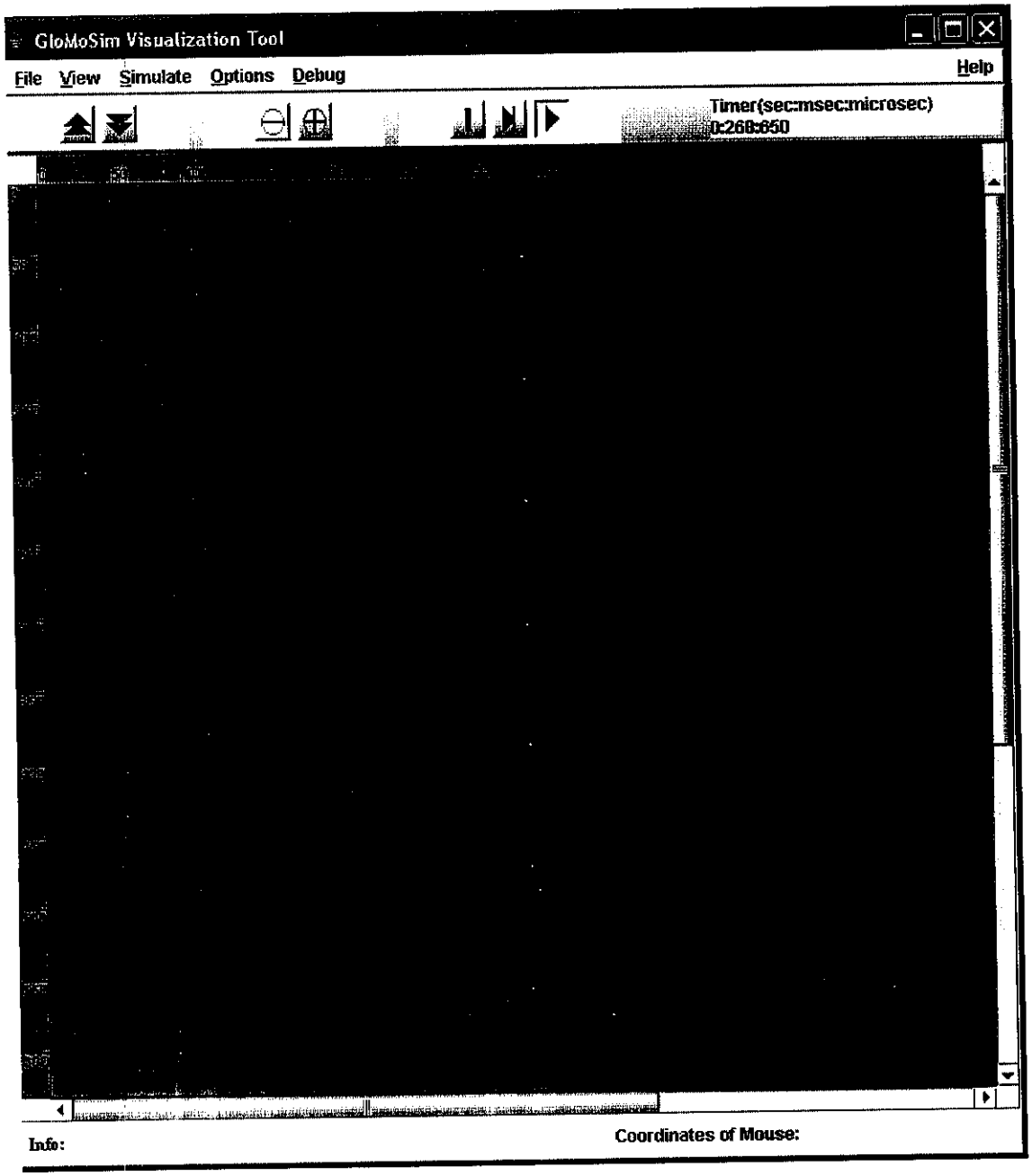
```
            if(RoutingAodvCheckSent(destAddr, &aodv->sent))
            {
                RoutingAodvInsertBuffer(msg,destAddr,&aodv->buffer);
                return;
            }
            RoutingAodvInsertBuffer(msg,destAddr,&aodv->buffer);

RoutingAodvResetRepairableSetBeingRepairedAndIncSeq(destAddr,
            &aodv->routeTable);
RoutingAodvStartLocalRepair(node,sourceAddress,destAddr);
            return;
        }
        #endif
        precursorList.size = 0;
        precursorList.head=NULL;
        precursorList.tail=NULL;
        if(RoutingAodvCheckRouteEntryExist(destAddr,&aodv->routeTable))
        {
            AODV_RT_Node *current;
            current = aodv->routeTable.head;
            while(current->destAddr!=destAddr)
            {
                current = current->next;
            }
            if(current->destSeqValid==TRUE)
            {
                current->destSeq++;
            }
            current->lifetime =  simclock() + RoutingAodvGetDeletePeriod();
            RoutingAodvSetTimer(node, MSG_AODV_DeleteRouteEntry,
                    current->destAddr,RoutingAodvGetDeletePeriod());
            newRerrPacket.pktType = AODV_RERR;
            newRerrPacket.N = FALSE;
            newRerrPacket.destinationCount = 1;
            newRerrPacket.destinationPairArray[0].destinationAddress =
                destAddr;

newRerrPacket.destinationPairArray[0].destinationSequenceNumber
            = RoutingAodvGetSeq(destAddr, &aodv->routeTable);
            RoutingAodvGetPrecursors(node,destAddr,&precursorList);
            if(precursorList.size>0)
            {
                SendRouteErrorPacket(node,
                        &newRerrPacket,&precursorList);
                aodv->stats.numRerrNoNSent++;
                aodv->stats.numRerrSent++;
```

```
                }
                aodv->stats.numPacketsDropped++;
                GLOMO_MsgFree(node,msg);
            }
            else
            {
                GLOMO_MsgFree(node,msg);
            }
            }
        }

}

//RoutingAodvFinalize

void RoutingAodvFinalize(GlomoNode *node)
{
        GlomoNetworkIp *ipLayer = (GlomoNetworkIp *)
                node->networkData.networkVar;
        GlomoRoutingAodv *aodv = (GlomoRoutingAodv *)ipLayer->routingProtocol;
        FILE *statOut;
        float avgHopCnt;
        char buf[GLOMO_MAX_STRING_LENGTH];
        sprintf(buf, "Number of CTRL Packets Txed = %d",
                        aodv->stats.numRequestSent + aodv->stats.numReplySent+
                        aodv->stats.numReplyAckSent +
                        aodv->stats.numRerrSent);
        GLOMO_PrintStat(node, "RoutingAodv", buf);

        if(aodv->stats.numDataReceived!=0)
        {
                avgHopCnt  = (float)aodv->stats.numHops/(float)
                        aodv->stats.numDataReceived;
        }

        #ifdef HELLO_PACKETS
        printf(buf,"Number of Hello packets sent = %d",aodv->stats.numHelloSent);
        LOMO_PrintStat(node,"RoutingAodv",buf);
        #endif
  }
```

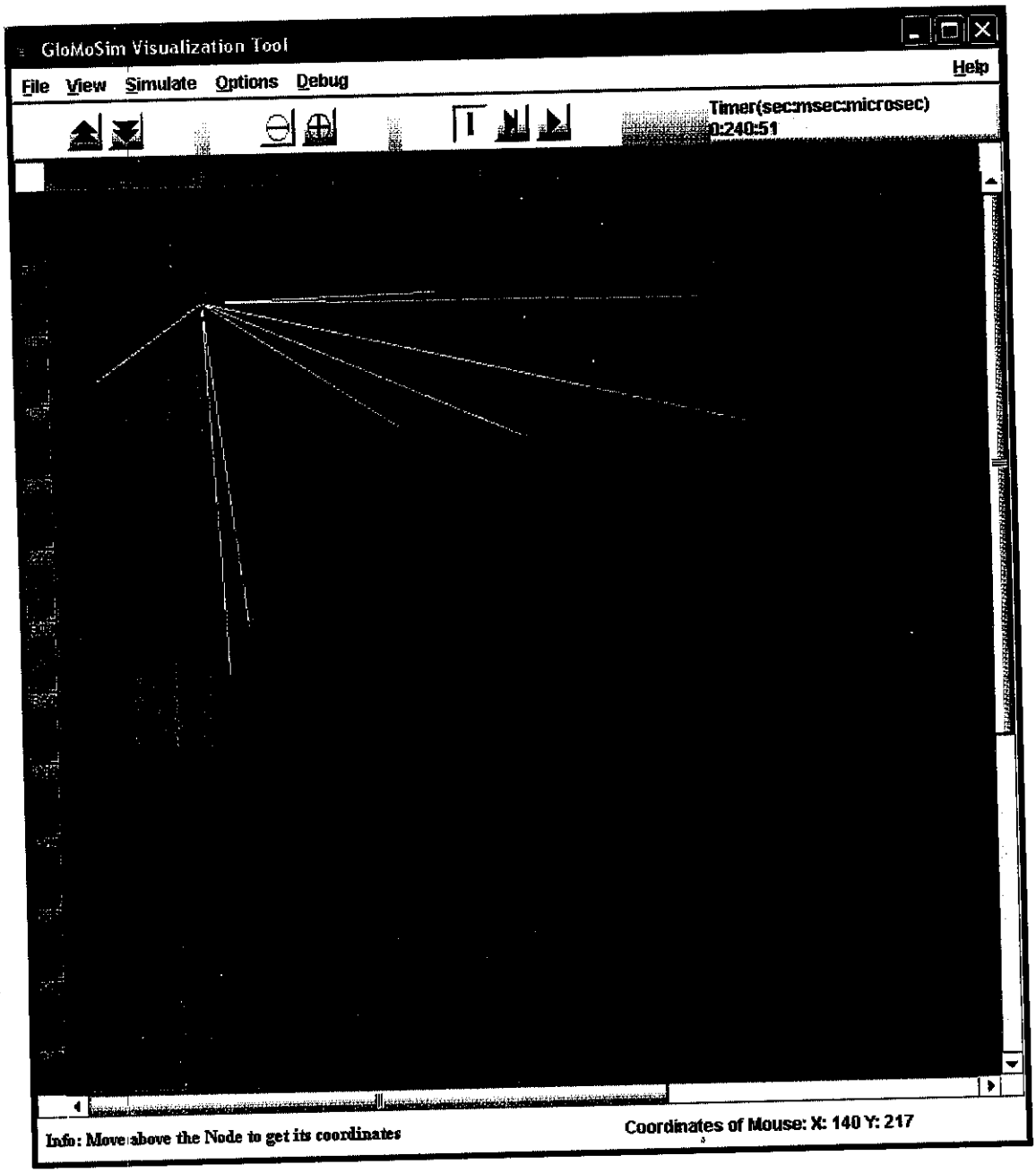
APPENDIX 2

SCREEN SHOTS

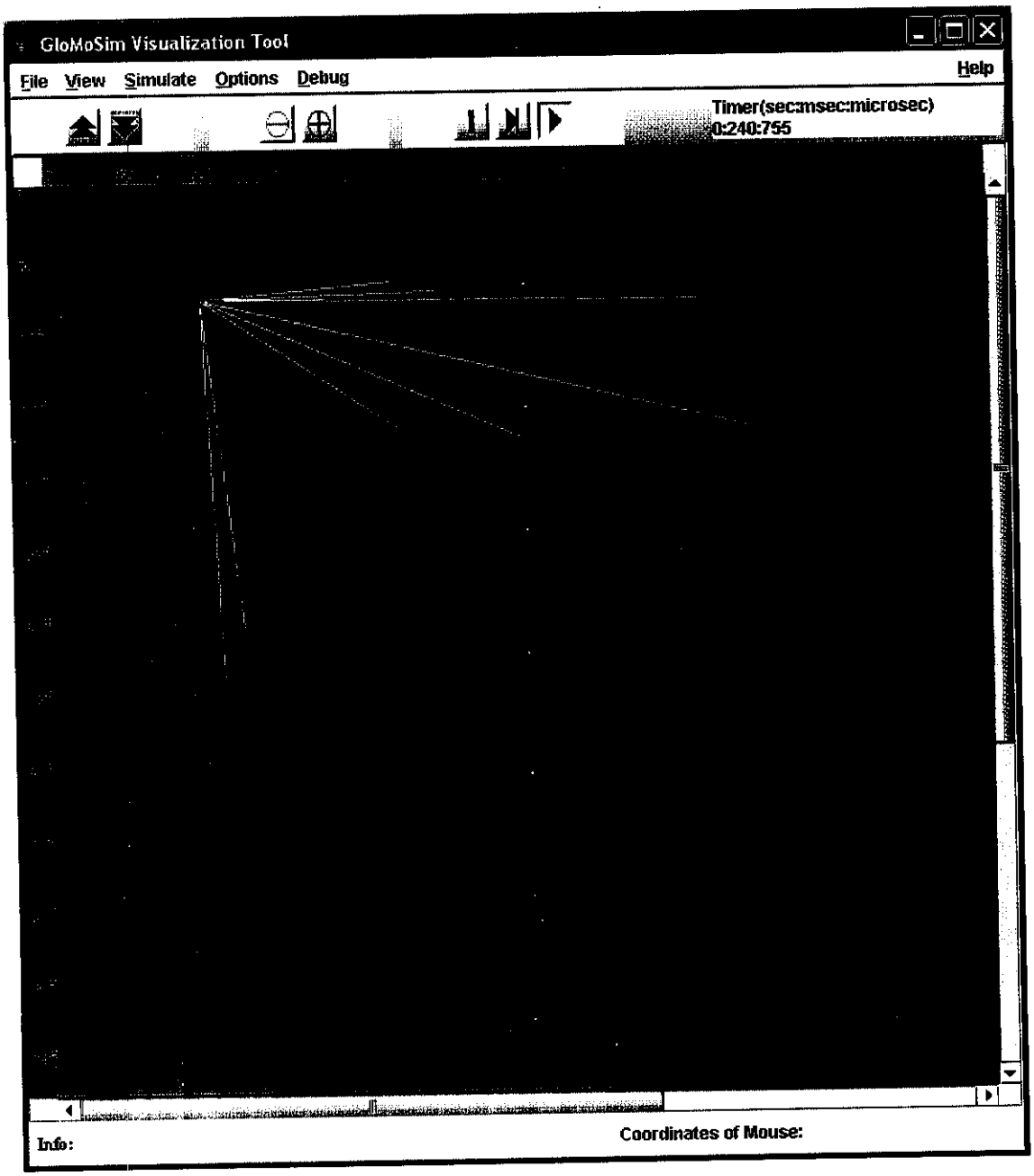
1. NODE PLACEMENT



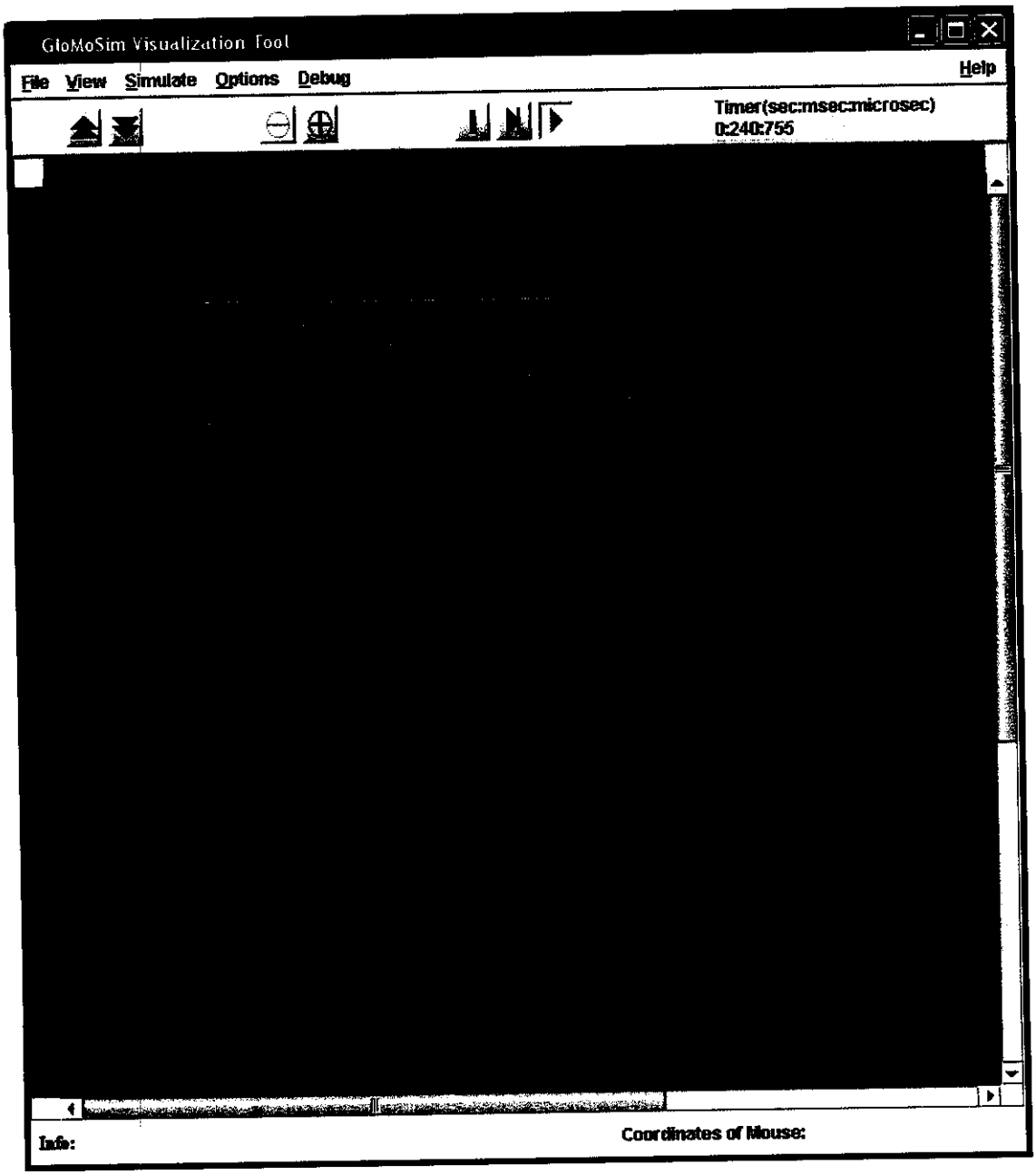
2. CONNECTION ESTABLISHMENT



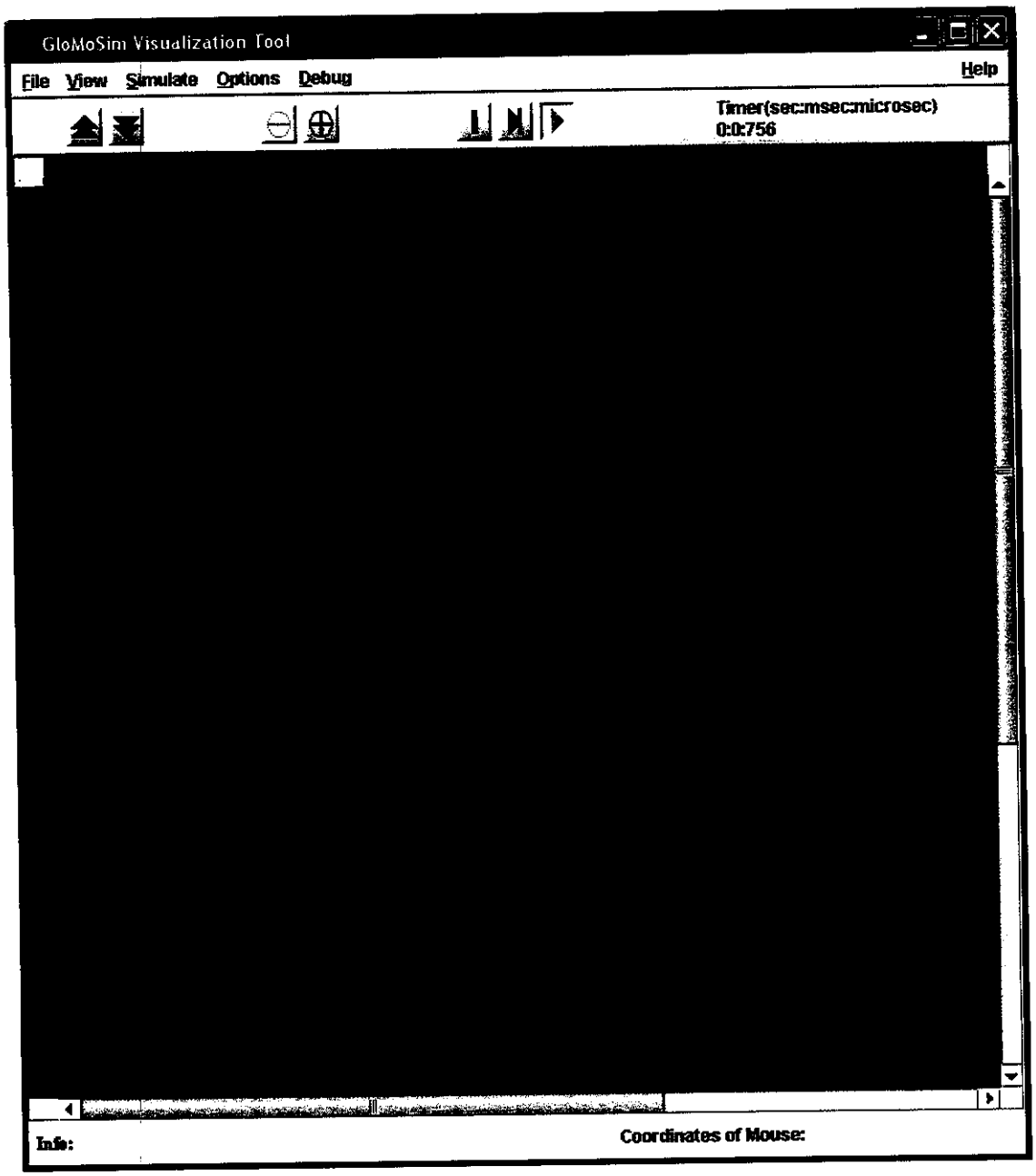
2. DATA TRANSMITTED TO THE NEAREST NODE



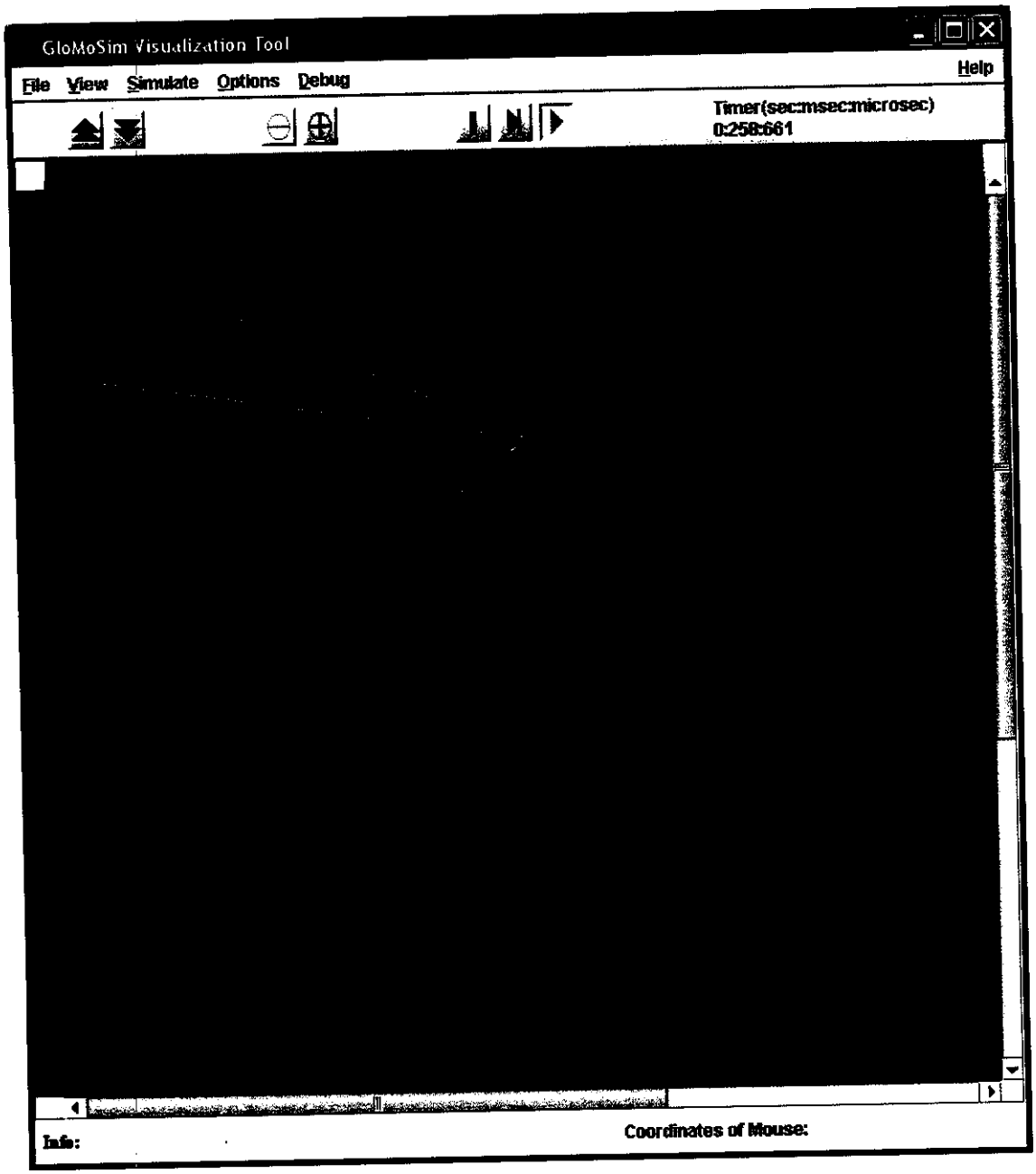
4. DATA TRANSMITTED TO NODE 6



5. ERROR IN DATA TRANSMISSION



6. SUCCESSFUL LOCAL REPAIR



REFERENCES

[1] S.R. Das, C.E. Perkins, and E.E. Royer, "Performance Comparison of Two On-Demand Routing Protocols for Ad Hoc Networks," Proc. INFOCOM, pp. 3-12, 2000.

[2] Y.-C. Hu, A. Perrig, and D.B. Johnson, "Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks," Proc. Mobi- Com '02, pp. 12-23, 2002.

[3] D.B. Johnson and D.A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," Mobile Computing, vol. 353. Kluwer Academic, 1996.

[4] C.E. Perkins and E.M. Royer, "Ad-Hoc On-Demand Distance Vector Routing," Proc.Workshop Mobile Computing Systems and Applications (WMCSA '99), Feb. 1999.

[5] E. Royer and C. Toh, "A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks," IEEE Personal Comm., pp. 46- 55, Apr. 1999