6-2-2005

Secure routing and trust computation in multihop infrastructureless networks

Tirthankar Ghosh
Florida International University

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

SECURE ROUTING AND TRUST COMPUTATION IN MULTIHOP INFRASTRUCTURELESS NETWORKS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

by

Tirthankar Ghosh

2005

To: Dean Vish Prasad
College of Engineering and Computing

This dissertation, written by Tirthankar Ghosh, and entitled Secure Routing and Trust Computation in Multihop Infrastructureless Networks, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Kia Makki

Kang Yen

Shih-Ming Lee

Niki Pissinou, Major Professor

Date of Defense: June 2, 2005

The dissertation of Tirthankar Ghosh is approved.

Dean Vish Prasad
College of Engineering and Computing

Dean Douglas Wartzok
University Graduate School

Florida International University, 2005

DEDICATION

I dedicate this dissertation to my wife and my parents whose love and sincere support gave me inspiration to complete the work.

ACKNOWLEDGMENTS

I take this opportunity to thank my advisor and co-advisor, Dr. Niki Pissinou and Dr. Kia Makki who helped me successfully finish my dissertation. I am deeply indebted to them for professional guidance, encouragement, constructive criticism and thoughtful insights into every step of my dissertation. This work would not have been possible without their able guidance and support. I also thank the other members of my dissertation committee, Dr. Kang Yen, and Dr. Shih-Ming Lee who helped me in every step in completing my dissertation.

I acknowledge National Science Foundation (Grant Nos. ANI-0123950 and CCR-0196557), Department of Transportation (Project No. FL-26-7102- 00), Department of Defense (Award No. H98230-04-C-0460) and IBM-SUR grant for supporting my research. I sincerely extend my warm regards to all my fellow colleagues for their help and co-operation in carrying out this research.

I thank Dr. Subbarao Wunnava and Dr. Tadeusz Babij for serving on my PhD qualifying exam committee.

Last, but not the least, I sincerely acknowledge the inspiration I received from my wife, Ms. Sukanya Ghosh, my parents Mr. Kamalesh K. Ghosh and Ms. Anjana Ghosh and my parents in law Mr. Rajat K. Dasgupta and Ms. Supriya Dasgupta. Without their inspiration and support it would not have been possible for me to carry out my research with enough motivation and hard work.

ABSTRACT OF THE DISSERTATION

SECURE ROUTING AND TRUST COMPUTATION IN MULTIHOP

INFRASTRUCTURELESS NETWORKS

by

Tirthankar Ghosh

Florida International University, 2005

Miami, Florida

Professor Niki Pissinou, Major Professor

Today's wireless networks rely mostly on infrastructural support for their operation. With the concept of ubiquitous computing growing more popular, research on infrastructureless networks have been rapidly growing. However, such types of networks face serious security challenges when deployed. This dissertation focuses on designing a secure routing solution and trust modeling for these infrastructureless networks.

The dissertation presents a trusted routing protocol that is capable of finding a secure end-to-end route in the presence of malicious nodes acting either independently or in collusion, The solution protects the network from active internal attacks, known to be the most severe types of attacks in an ad hoc application. Route discovery is based on trust levels of the nodes, which need to be dynamically computed to reflect the malicious behavior in the network. As such, we have developed a trust computational model in conjunction with the secure routing protocol that analyzes the different malicious behavior and quantifies them in the model itself. Our work is the first step towards protecting an ad hoc network from colluding internal attack. To demonstrate the feasibility of the approach, extensive simulation has been carried out to evaluate the protocol efficiency and scalability with both network size and mobility.

This research has laid the foundation for developing a variety of techniques that will permit people to justifiably trust the use of ad hoc networks to perform critical functions, as well

as to process sensitive information without depending on any infrastructural support and hence will enhance the use of ad hoc applications in both military and civilian domains.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

LIST OF ABBREVIATIONS

ACK: Acknowledgement

AODV: Ad Hoc On Demand Distance Vector Routing

ARAN: Authenticated Routing for Ad Hoc Network

CA: Certificate Authority

CBR: Constant Bit Rate

CTS: Clear To Send

DH: Diffie Hellman

DOS: Denial of service

DSDV: Destination Sequence Distance Vector

DSR: Dynamic Source Routing

FTP: File Transfer Protocol

HF: Has Forwarded

IP: Internet Protocol

LAN: Local Area Network

LER: Local Evaluation Record

MAC: Media Access Control

MOCA: Mobile Certificate Authority

NNL: Neighbor Node List

OER: Overall Evaluation Record

PDA: Personal Digital Assistant

PK: Public Key

RERR: Route Error

RF: Request to Forward

RREP: Route Reply

RREQ: Route Request

RSA: Rivest, Shamir, Adleman algorithm

RTS: Request To Send

RWARN: Route Warning

SK: Private Key

SORI: Secure and Objective Reputation-based Incentive

SRP: Secure Routing Protocol

T-AODV: Trust-embedded AODV

TCP: Transmission Control Protocol

Chapter 1

Introduction

Mobile computing has experienced a very sharp rise since the last decade. The continued increase in the processing power of mobile devices, together with competitive prices and attractive design, has made them available to a growing number of the population around the globe. There is an increasing interest among the industries as well as academia to bring wireless voice and data networks together. With the concept of ubiquitous computing growing more and more popular, people now want to remain connected "anytime, anywhere". The growth of wireless Internet access has also been phenomenal in the past ten years.

With the advent of wireless communications and mobile computing, another form of networking has also emerged. This is known as "infrastructureless" or "ad hoc" networking. This form of peer-to-peer (or even multicast), multihop networking is growing more popular in an infrastructureless environment (like the absence of access points or base stations). These ad hoc networks have given rise to active research issues since their evolution [Zho99, Hie01, Yi01, Sta99]. The research community has been actively working on various challenges and problems arising out of the effective implementation of ad hoc networks.

Most of the research so far has been done in the area of routing protocols [Hu02a, Hu02b, Pap03, Pap02], although in recent years security issues have also been explored. Although the basic security goals and requirements of an ad hoc network are very similar to those of a wireless network, some inherent characteristics of the former make security issues more challenging. These include absence of infrastructure, high probability of node compromise, frequent and dynamic topology change and low level of trust among the nodes.

1.1 Background and Motivation

In the following subsections we are going to elaborate on the principles of ad hoc networking with an emphasis on their routing protocols and security challenges and finally will discuss about the motivation behind this research.

1.1.1 An Insight into Ad Hoc Networks

Ad hoc networks are formed without any centralized administration or infrastructure, and depend upon the mutual agreement between all the nodes to cooperate with one another for their operation. The vision for implementing an ad hoc network can be supported by many applications, many of which can be temporary but exchanging highly sensitive information. The implementations can range from totally independent application and network formation (Figure 1.1) to a peripheral ad hoc zone in integration with existing wireless networks to form a hybrid ad hoc wireless LAN (Figure 1.2).

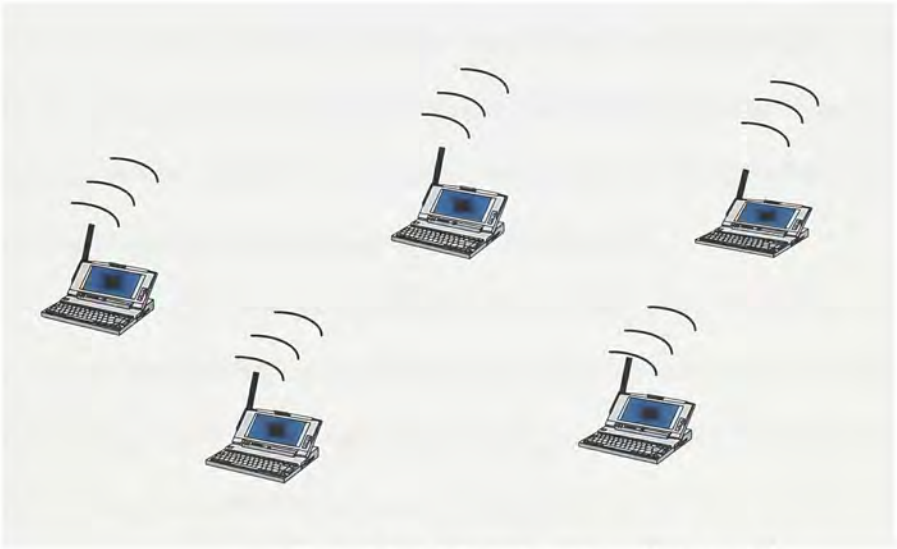

Figure 1.1 A fully independent ad hoc network

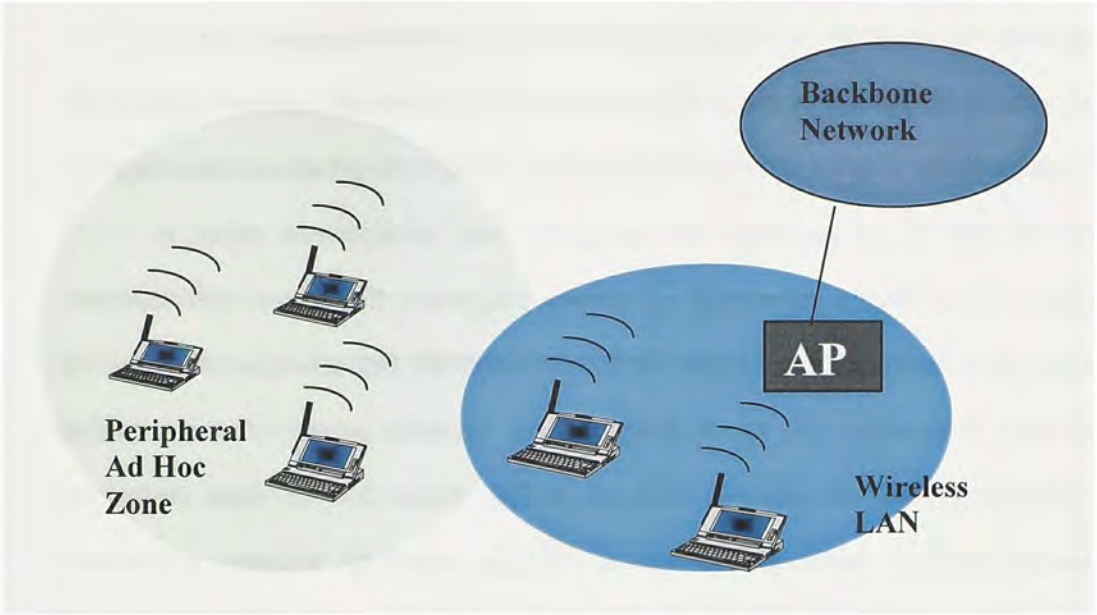

Figure 1.2 A hybrid ad hoc wireless LAN

There are several design challenges when implementing an ad hoc network, some of which have been mentioned in [Per01]. The first consideration is whether the transmission range of the ad hoc nodes should be within the range of one another [Per01]. In that case, there is no need for routing in the network and moreover, considerable amount of power is needed for the nodes to remain connected to one another, thus wasting precious battery life. Hence, there is a need of routing protocols in such an ad hoc implementation.

The second consideration should be whether there would be any location-centric approach for the ad hoc nodes where each node can measure its own relative position and use the status information of the appropriate link. This gives rise to the question of geographic routing where "routing decisions are to be made based not on destination addresses, but on destination attributes of packet content" [Zha04]. This is better to avoid as it compromises user convenience is using and configuring an ad hoc network.

Another design consideration should be concerned with the directional characteristics of the wireless channels. The routing and MAC protocols in any ad hoc implementation have to be designed based on whether the wireless channels used are bidirectional or unidirectional.

A major consideration when designing and deploying an ad hoc network is its communication security. It is always preferable to incorporate security at the desired layers during the development and standardization of the protocols at that layer. This was seriously lacking when the routing protocols were designed for ad hoc applications. Later on many researchers came up with secure routing solutions to make the existing protocols robust. However, a seamless inter-layer approach is still missing when a secure communication infrastructure has been designed.

1.1.2 Security Challenges in Ad Hoc Networks

Before discussing the security challenges that a typical ad hoc network has, we should give an insight into the different threats that such networks face. Broadly, threats can be divided into two types – passive and active. In a passive threat an attacker quietly listens to the ongoing communication without taking part in it with an intention to capture the packets and read their contents. These types of passive threats can easily be eliminated by using basic cryptographic mechanisms to encrypt the message contents that are flowing into the network.

In an active threat an attacker not only tries to capture packets, but also takes part in the active communication in the network. Active threats can be carried out either by malicious inclusion of the attacker(s) into the active routes or by injecting bogus messages in the network with an intention to flood the networks and wasting precious bandwidth. These threats can either be external where the attack is carried out by somebody not initially a part of the network, or internal where the attacker is already a part of the network. External threats can be prevented by reverting to traditional cryptographic algorithms [Des87, Dif76, Riv78] or designing new

cryptographic schemes suitable for the typical application. The deadliest of all the threats are from the internal ones, where the attacker has already been an active part of the network, for example in the form of compromised nodes. An internal attacker has all the secrets used in the network in his/her disposal and can use those secrets to effectively authenticate him(her)self to their peers or take part in the ongoing communication. Hence, the main goal of designing a secure communication mechanism in an ad hoc application should be to protect the network from the active threats in the form of internal attacks.

Although the basic security needs of an ad hoc network are the same as in conventional networks, namely authentication, integrity, confidentiality, availability, non-repudiation and access control, there are certain characteristics of such a network, which make the security issues challenging. Ad hoc networks are formed without the aid of any infrastructure. Hence, unlike an infrastructured wireless network, the ad hoc networks cannot rely on any central entity for security issues. This lack of infrastructure has posed serious threats so far as key distribution [Hie01, Yi03, Yi02] authentication [Nga04, Pir04c] and trust computation [Dav04, Esc02] are concerned.

Nodes forming an ad hoc network are vulnerable to physical compromise. This lack of physical security gives rise to internal[1] threats within the network, which make the issues of authentication, integrity and confidentiality even more challenging than in conventional wireless networks. Security must also be incorporated in the network layer to make the routing protocols robust enough to withstand attack from compromised[2] or disloyal[3] nodes trying to inject malicious information into the network. The third most important characteristic of an ad hoc

[1] We define an internal threat in a network as an active attack by a compromised node or a disloyal node which actively takes part in the ongoing communication.

[2] We use the term *compromised node* to indicate a node which has been physically taken over by an intruder thus giving access to all its stored secrets and system codes.

[3] We use the term *disloyal node* to indicate a node which has ended its loyalty to the network and has decided to disrupt the network operation by non-cooperation of some means.

network is that the topology of the network changes dynamically. Hence, any security model based on a fixed architecture cannot be used in such a scenario.

In addition to the above three network centric features, the nodes in an ad hoc network are characterized by their low battery power and limited computational abilities. These are even more prominent in applications like personal computing and small sensor networks. These restrictions seriously limit the ability of the nodes to perform intensive public key computations like RSA. As the nodes are characterized by minimum trust for each other, key distribution and secure routing have been challenging research issues. Most of the work on key distribution in ad hoc networks is based on threshold cryptography [Zho99, Bec98, Bur95, Des87, Sha79, Ste00], and assumes the use of public key cryptosystem which involves intensive computation. Even secure routing solutions proposed by many [Pap03, Pap02, Zap02, San02] are based on public key infrastructure. Eventually, all routing protocols in ad hoc networks tend to find the shortest path to the destination, irrespective of the presence of a malicious node in between.

1.1.3 Routing in Ad Hoc Networks

Most routing protocols have been divided into two broad types: proactive or table-driven and reactive or on-demand [Per01]. In proactive or table-driven protocols each node stores the routes to other nodes in its routing table, and uses these routes during communication. These routing protocols, although having the advantage of selecting the routes quickly, suffer from the major drawback of generating more control traffic into the network by exchanging large number of messages needed to update the route entries. This is even more critical in ad hoc networks with high degree of mobility where the existing routes are more frequently broken.

To solve this problem, reactive or on-demand routing protocols have been designed. In these protocols each node seeks a route to the destination only when it is needed, thus generating lesser number of control packets. These routing protocols normally take more time to kick-off,

because of the initiation of the route discovery process at the start of the application, but they have significantly lower overhead as compared to the proactive routing protocols. Below we briefly describe two of the reactive routing protocols that are used most often in ad hoc networks. For details in the design of these routing protocols interested readers may refer to [Per99, Per01, Joh99].

1.1.3.1 Ad Hoc On Demand Distance Vector Routing (AODV)

The ad hoc on demand distance vector routing protocol was proposed by Perkins and Royer in [Per99] and has been the most widely used since its standardization. It's performance has been found to be superior as compared to the other routing protocols designed, especially with higher mobility. A detailed comparison has been carried out in later sections. The characteristics of the protocol are:

- It responds to any change in the network topology in a quick and timely manner. This makes the protocol suitable for an ad hoc network application as it demands a routing protocol to adapt quickly to topological changes;
- It is capable of building routes with small overheads in terms of control messages;
- It stores only next hop information in the route tables, thus reducing storage space significantly;
- It does not place any additional overhead on data packets, as it does not utilize source routing.

AODV has two phases – route discovery phase and route maintenance phase. We describe each phase separately.

1.1.3.1.1 Route Discovery

A node initiates a route discovery when no route to a destination is found on its route table. It broadcasts a route request (RREQ) message containing source IP address, destination IP address, source sequence number, destination sequence number, broadcast ID and hop count. The

source address and broadcast ID together form a unique identifier to the RREQ packet. On receiving a RREQ, a node first checks whether the packet is duplicate. In case it is duplicate, the node drops it without taking any further action. If it is a new RREQ, the node checks whether it is the destination of the packet, in case of which it unicasts a route reply (RREP) back to the source node. If the node is any intermediate node, it searches its route table to look for a fresh route to the specified destination having sequence number greater than that specified in the RREQ. If a fresh route is found, it sends the route to the source of the RREQ. If no fresh route is found, the node increments the hop count and re-broadcasts the packet.

Each node also maintains a reverse route entry on receiving the RREQ packet containing the source IP address, sequence number, number of hops to the source node and the address of the node from which it has received the RREQ. The reverse route is used to forward the RREP corresponding to the RREQ.

1.1.3.1.2 Route Maintenance

The route that has been discovered by any node needs to be maintained. When a source node moves out, it re-initiates the route discovery phase to discover a route to the destination. When a destination node or any intermediate node moves out, the link breakage is reported by the node upstream to the break. The node reporting the breakage broadcasts route error (RERR) packets. On receiving a RERR packet, a node marks the route to the destination invalid in its route table. It re-initiates route discovery when it needs to communicate with the destination.

1.1.3.2 Dynamic Source Routing (DSR)

Dynamic source routing protocol was developed by Johnson and Maltz in [Joh99]. It is a reactive routing protocol similar to AODV, where the source node initiates route discovery only when the route is needed. The major difference that DSR has from AODV is in the fact that the former is a source routing protocol where the source node appends the entire route to the packet

header when sending data. Similar to AODV, DSR also has two phases – route discovery and route maintenance.

1.1.3.2.1 Route Discovery

In DSR a node initiates route discovery when no route to the destination is found in its route table (route cache). The node broadcasts a route request (RREQ) packet containing source and destination addresses, a unique request ID and also a sequence of addresses of each intermediate node through which the RREQ packet has been forwarded. This sequence of intermediate addresses are all initialized to an empty set. When a node receives a RREQ, it checks whether the packet is duplicate by matching the request ID and source address pair with that anything seen previously. If it is not a duplicate packet and if the node itself is the destination of the packet, it sends a route reply (RREP) back to the source node. If the node is any intermediate node, it appends its own address in the route record and re-broadcasts the packet.

In forwarding a RREP, each node either looks in its route cache to find a route to the source node, or initiates a route request to discover the route. It can also simply reverse the sequence of hops in the packet's route record and use this as the source route on the packet carrying the RREP itself.

1.1.3.2.2 Route Maintenance

In DSR each node forwarding a RREQ or RREP is responsible for the confirmation that the packet is delivered properly. This is ensured either by MAC layer acknowledgement or by a passive acknowledgement where each node overhears the next node's transmission of the packet. Alternatively, a node may set a bit in the packet header to ask for an application specific acknowledgement. If a node fails to receive any acknowledgement from its next node, it sends a route error (RERR) message to the source node indicating a broken link, which is duly removed by the source node from its route cache.

1.1.3.3 Comparison Between AODV and DSR

Extensive comparison between the two on demand routing protocols, AODV and DSR have been carried out in [Das00] through simulation. The difference in the performance can be attributed to several fundamental design differences between the two protocols which are briefly highlighted below. Interested readers may refer to [Das00] for further details.

The most fundamental difference between AODV and DSR lies in the route request-reply cycle. DSR route requests are designed to accumulate the whole route in the packet itself, thus allowing the nodes to learn the routes to multiple nodes in the network other than the source and the destination. This results in lower number of route request generation in DSR. AODV, however, allows only few routes to be discovered in a route request-reply cycle, essentially routes to source nodes are only discovered. This results in larger route request floods in AODV. However, in DSR more route replies are generated than that in AODV, as in the former all copies of route requests received by the destination are replied back. In AODV, however, only one route reply is sent back to the source node, the one which corresponds to the first request received. Because of this difference in the design of a request-reply cycle, each node in AODV has at most one route to the destination, while DSR allows the nodes to maintain extensive route caches with multiple routes to a particular destination. Thus any link break will trigger a new route discovery phase in AODV, while in DSR a route discovery is triggered only when all the routes in the cache become invalid. Moreover, in AODV, stale routes are deleted from the route table by setting an expiration timer, while DSR does not allow the expiration of stale routes from route cache. Route error messages are also propagated in different ways in the two protocols. In AODV, route errors are essentially broadcast messages, while in DSR it is a unicast, sent out by backtracking the data packet.

The simulation carried out in [Das00] shows that the routing load is almost always lower in DSR. The routing load has been defined by the authors as the number of routing packets

transmitted per data packet delivered. As DSR uses extensive route cache, it generates much less route request packets than that in AODV, as the possibility of finding a route is almost always higher in DSR. Most of the routing load in AODV comes from the route request packets, while in DSR the routing load is mainly generated by route reply and route error packets. In summary, it can be said that DSR always generates more route reply and route error packets than in AODV, but has lower route request packets in higher proportion. This brings down the routing load in DSR.

In spite of the above fact, it has been observed that DSR performance goes down when the whole network load is considered, as it has significantly higher MAC load than in AODV. The MAC load has been defined by the authors as the number of routing, ARP (Address Resolution Protocol) and layer 2 control packets generated per data packet delivered. As DSR generates more route reply messages and each route reply message is a unicast packet with MAC layer RTS/CTS/Data/ACK exchange, it imparts higher MAC layer load than in AODV. Route error messages are also unicast in DSR unlike in AODV where they are broadcast messages. This also increases the MAC load in DSR.

Mobility also has different effects on the performance of DSR and AODV. It has been observed that AODV performs better than DSR in higher mobility conditions. When the mobility is low, the possibility of link failure is also low. But low mobility results in some localized node concentration because of which the network may get congested. Due to this congestion, packets may get lost in the MAC layer triggering false route error messages indicating broken link, although the physical links still exist. This will trigger the route discovery phase generating more route request messages in AODV. DSR, however, remains unaffected due to this problem because of its extensive route cache. Hence, the performance of AODV degrades with low mobility condition. In contrast, higher mobility causes higher possibility of link failure, and hence trigger more route request packets in AODV. But higher mobility also causes stale routes in the

route cache in DSR. Thus, when route requests are initiated in DSR, they generate higher route replies and also higher MAC load. This results in the degradation of DSR performance with higher mobility in the network.

1.1.3.4 Attacks on Routing Protocols

An adversary can carry out active attacks on the routing protocols in different forms. The main goals of attacking the routing protocols are: to be a part of the active end-to-end route; and to inject bogus information in the form of routing overhead with an intention to flood the network. In order to carry out attacks in the routing layer, an adversary has to manipulate the metrics used in the control packets and deceive the source nodes to believe in either a non-existent route or a route through the adversary itself. Below we briefly describe some of the attacks carried out in AODV and DSR routing protocols:

An adversary decreases the hop count information in the route request packet in AODV with an intention to make the destination and source believe that the shortest path exists through itself. Subsequently, the route through the adversary will be selected as it will have the lowest hop count metric maliciously injected by the adversary.

An adversary advertises a fresh route through itself in AODV by sending a route reply back to the source node having a high destination sequence number. The source will be forced to believe that a fresh route exists through the adversary and uses that route to send the data.

An adversary places itself in the active route and discards all data packets received for forwarding. This attack is known as the *black hole* attack as the adversary behaves as a black hole in the sense that it absorbs all data packets. This attack can be carried out in any routing protocol, be it AODV or DSR.

More than one adversary collude together to carry out a *wormhole attack* in the DSR routing protocol. When the first adversary receives a route request packet in DSR, it unicasts the

packet to the second adversary who then broadcasts it to the destination. The destination is thus forced to believe that the shortest path exists through these two adversaries and effectively the route is selected.

Other than the attacks discussed above, there are several other ways by which an adversary can disrupt the network operation. Denial of service (DOS) attacks can be carried out in MAC layer or application layer, channel jamming can be implemented in physical layer, Sybil attack [Dou02, New04] can be carried out with address spoofing, Man-in-the-middle attack [Sta02] can be launched to fool the source and the destination, or even the transport layer protocols can be mishandled by manipulating the metrics. In our research we only concentrate on the attacks in the routing layer and leave the rest as a future extension with an intention to develop a cross-layer approach towards designing a secure communication mechanism.

1.1.4 Motivational Example

While there is a plethora of applications to support our proposed environments, especially in the military, emergency crisis management, homeland security, and medicine, we will look at a simple medical scenario in this section. In this scenario, let us consider the formation of a hospital medical board, where doctors and staff create a network "on the fly" without the aid of any infrastructure. They share substantial confidential information and data which they want to keep restricted within themselves. An ad hoc multihop network is formed using their laptops or PDAs. The members can actually communicate among themselves from different locations inside the hospital, with each member appropriately forwarding data. All the data flowing through the network is cryptographically protected using shared secrets. Shared secrets are formed by the collaborative effort of some trusted members within the network. A person who is not authorized to join the network cannot take part as he/she is unable to encrypt or decrypt the necessary information. But he/she can easily get hold of a member's laptop and steal all network secrets.

Using these secrets, he/she can communicate with the others without getting detected. It is easy for the person to include himself/herself in the routing path by injecting malicious routing information into the network. The malicious information goes undetected as appropriate keys are used to encrypt and decrypt the headers. In the face of such an attack, even the cryptographically protected network breaks down. In view of the above fact, it is of itmost importance to find a trusted route in the network to disseminate confidential information. The trusted route must include nodes who are are proven to be trustworthy, which gives rise to further issues like trust computation and distribution in the network.

A similar example can be cited in a war front situation where army officials form a multihop ad hoc network to exchange confidential information. A similar attack can be launched by compromising any of the nodes comprising the network. Although the network is cryptographically protected by secrets formed by collaborative efforts, an internal attack in the form of a compromised node can completely disrupt it.

Secured communication is also desired in applications where an ad hoc network is used in an archeological mission. The nature of the network demands exchange of confidential data between trusted hosts. A secure end-to-end path needs to be set up to ensure safe data exchange. Efficient use of cryptographic keys can provide a feasible solution, but an internal compromise can lead to disclosure of confidential data. Consider the following scenario as depicted in Figure1.3.

Let us assume that the nodes a and b have somehow managed to share a secret. When node a wants to communicate with node b, it broadcasts a route request packet; its format being the same as in AODV [Per99]. Now, let node c get compromised and get hold of the secret group key. It will place malicious routing information into the network and wants all the packets to go through itself. This will disrupt the network operation since the compromised node can drop

important information, refuse to forward them or can even forward information after altering them suitably.

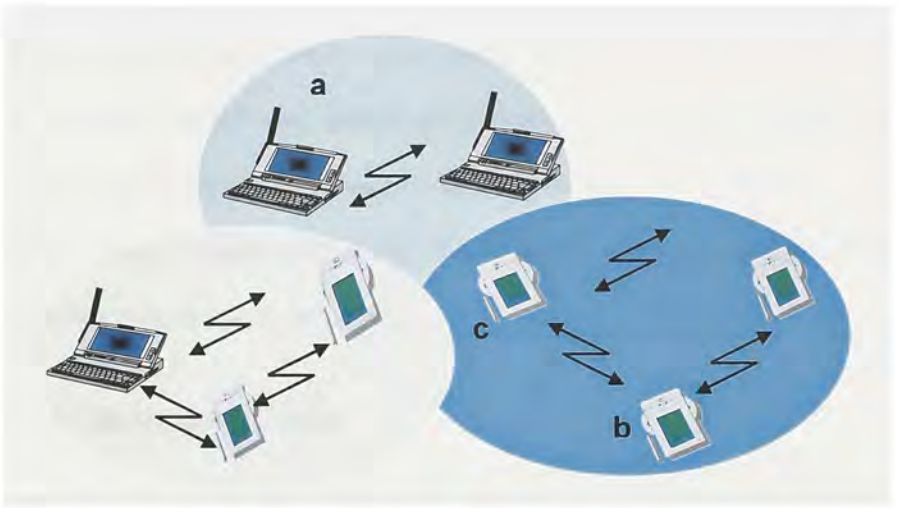

Figure 1.3 An example scenario

In view of the above discussion, our research will focus mainly on securing an ad hoc network from an active internal attack. We mainly consider the threats posed by compromised nodes, acting either independently or in collusion, and propose to secure the network from such attack. As we have already discussed, we will carry out the research in two steps: designing a trusted routing solution that will find a trusted end-to-end route free of any malicious entity; and developing a framework for computing, distributing and updating trusts in ad hoc applications.

1.2 Problem Statement

Based on our discussions in the previous sections, we can emphasize that the overall goal of this research is to provide a secure solution for communication in ad hoc networks that will be strong enough to withstand the most serious type of internal threat. None of the solutions, proposed so far, can resist an internal attack in the form of a compromised or disloyal node. This

demands the design of an efficient and secure solution in ad hoc communications based on the issues and challenges arising out of their inherent characteristics and applications.

Our motivation for this proposed research is twofold:

1. Finding a secure end-to-end route in an ad hoc network, which will be able to withstand active internal attacks from compromised nodes, either acting independently or in collusion, trying to inject malicious routing information.
2. Designing a trust computational model that will analyze the psychology of the attacker and quantify them in the model itself.

1.3 Research Goals and Issues

The overall goal of this research is to provide a secure solution for communication in ad hoc network applications strong enough to withstand an active internal threat within the network. None of the solutions, proposed so far, can actually resist an internal attack in the form of a compromised or disloyal node. If an efficient cryptographic algorithm is used, a compromised node can always give away the secrets to the attacker. The same is true for a disloyal node. It can be argued, however, that the disclosure of secrets can be prevented by making the nodes (or some of their selected modules) tamper-proof, which is easier to visualize than to implement. The discussion of tamper-proof components are outside the scope of this thesis and we consider it a separate research issue altogether.

The first aspect of the research will be to design a trusted routing protocol that will be able to find a trusted end-to-end route free of any malicious node. The malicious nodes willing to take part in ongoing communication in the network by trying to put themselves into the active routes will be detected and isolated by collaborative effort of their neighbors. This demands the determination of a suitable trust metric in the routing protocol which will play an active role in the final route selection.

Another aspect of this research is to develop a framework for computing, distributing and updating trust in an ad hoc network application. Modeling and computing trusts in such an application is a challenging problem. It is very difficult to form a true and honest opinion about the trustworthiness of the nodes, as they can be engaged in malicious activities in different ways. This intricacy in trust computation, together with frequent topology changes among nodes, quite often causes the whole network to get compromised or disrupted. Different malicious activities of the nodes can very well be misinterpreted as the regular erratic behavior of the wireless networks in general and ad hoc networks in particular, thus making trust computation all the more difficult. In this paper we have proposed a framework for modeling and computing trusts that take into account different malicious behavior of the nodes. Our proposed model tries to explore the psychology of the attacker in different ways and quantifies those behaviors to form a computing framework.

Selfish behavior in ad hoc networks has been prevented by proposed schemes that used either a reputation-based incentive mechanism [Buc02, He04, Mic02], or a price-based incentive mechanism [But02]. In both the mechanisms, nodes are given incentives to suppress their malicious intention in favor of the network. But nodes with malicious intention at their subconscious self always try to find ways to bypass these incentive mechanisms.

In view of the above issues and challenges and keeping in mind the severity of an internal threat, we intend to incorporate two components into our research.

1. A network layer security scheme in the form of a robust routing protocol which will address the following issues:
    - Can the routing protocol be robust enough to withstand all forms of internal attack?
    - Can the protocol find an end-to-end secure and reliable path (which may not be the shortest) free of malicious nodes without the latter altering information and including itself in such a path?

- Can the protocol successfully isolate a compromised node trying to inject malicious routing information?
- Can the isolation be done by the collaborative effort of the neighboring nodes, instead of relying on a single node?

2. An efficient trust management system that takes into account different malicious behavior of the nodes in trying to disrupt the network operation. The trust computational framework should address the following issues:
   - Can the computational framework incorporate different malicious behavior and quantify them in the model?
   - Can the algorithm successfully isolate a non-trusted entity with the collaborative effort from all its neighbors?
   - Can the model form a true opinion about malicious entities colluding together to disrupt the network?

We propose to combine these two components to come up with a robust security solution against an active internal attack in the network. The trusted routing protocol, when designed, will be able to find a trusted end-to-end route free of any malicious entity, effectively isolating any node trying to inject malicious information into the network. A trust computational model will be developed and integrated with the routing protocol to act as a basis for selecting end-to-end trusted path based on trust metric. Message integrity, authenticity and confidentiality will be incorporated in this solution by efficiently encrypting the messages using either shared secrets or reverting to a public key infrastructure. We will not address these issues now as they have been well addressed in past studies. In the next section we will briefly discuss the routing protocols most commonly used in ad hoc networks.

1.4 Significance and Contribution

This research will lay a foundation to develop a variety of techniques that will permit people to justifiably rely on ad hoc networks to perform critical functions, as well as rely on ad-hoc networks to process sensitive information. This will in turn potentially allow mobile ad hoc networks to gain even more attention in both national scale infrastructures and localized systems, since they could securely serve as the primary communications networks where no infrastructure can be deployed. The solutions proposed in this research will be one of the first of its kinds to secure the network from colluding malicious nodes actively carrying out internal attacks. In addition, the trust computational model proposed in this work will also provide a solid foundation for developing a policy-based autonomous system to develop a trusted communication infrastructure in the absence of any support.

1.5 Methodology

Our work involves the combination of model development, protocol design, simulation and experimentation. Model development involves a critical assessment of the requirements and challenges of the security needs. The secure protocol has been designed to satisfy those challenges and needs with concentration to minute details. To evaluate the design of our protocol we are going to use simulation techniques. Although we know that simulation is not quite foolproof, and only implementation in a real environment can assure us of the effectiveness of the design, we will first turn our attention to simulation. Since mobile ad hoc networks are characterized by the lack of a centralized entity, dynamically changing topologies due to the mobility of the nodes, and a hostile wireless communication medium, it is difficult to obtain theoretical analysis on these types of networks. The system performance is primarily measured and evaluated through data statistics, which can be obtained through simulation, emulation, or a real life network. Despite the recent surge in research activities in ad hoc wireless networks,

software simulation remains the primary approach to evaluate the network performance, as it is easy to implement and manipulate. We observe that among all the protocols proposed for ad hoc wireless networks, few are practical for implementation and operation. Some are too complicated to implement, and some use parameters that are not available in practical systems. In order to evaluate the trustworthiness and overhead of the proposed protocol, we will adopt an experimental approach, and an obvious extension of the work will be to create an ad hoc testbed and evaluate our designed protocol with real-time data. In addition to simulation, we will also analyze the security of the routing protocol by evaluating different threat scenarios and will show that the protocol is secure against those scenarios.

1.6 Organization of the Dissertation

The remaining chapters are organized as follows. In chapter 2 we give an overview of the related work that have been done to secure communication in ad hoc networks. Chapter 3 discusses the design of the trusted routing algorithm with a detailed analysis of results obtained from extensive simulation. A security analysis of the protocol has also been carried out by evaluating different threat scenarios Chapter 4 describes the trust computational model designed with a detailed analysis of the results obtained. Finally, in chapter 5 we conclude with a summary of our research follwed by future extensions.

Chapter 2

Related Work

The discussions presented in this chapter are inspired by the recent developments and also the ongoing work that are being done in the area of security in ad hoc networking. The chapter summarizes in details the state of the art in the area of key management, routing security and trust computation in ad hoc networks.

Ever since the evolution of ad hoc networks researchers focused on designing communication protocols suitable for typical ad hoc applications to fit into the TCP/IP protocol stack. Security considerations for those types of networks did not play a major role in the research activities in the designing stages of the communication protocols. Some effort has been given to securing such networks in the past few years [Zho99, Hie01, Yas02, Yi02, Sta99, Alb02, Den02, Yi01, Pap02, Pap03, Hu02a, Hu02b, Zap02, San02, Yan02], arising out of the need for their effective implementation [Hub01, Wro02, Sta99]. Different means of securing ad hoc networks have been proposed, which involve key management [Hie01, Yas02, Yi03, Yi02, Zho99], routing security [Buc02c, Den02, Hu02a, Hu02b, Pap03, Pap02, Pir04b, San02, Smi97, Yi01, Zap02] and trust computation [Dav04, Esc02, He04, Li04, Nga04, Pir04a, The04, Ver01, Yan03].

Although security issues in ad hoc networks have drawn considerable attention over the past few years, no solution has been proposed so far to secure the network against an internal attack by colluding malicious nodes by discovering a trusted end-to-end route. Most of the work done so far [Den02, Yi01, Pap02, Pap03, Hu02a, Hu02b, Zap02, San02, Smi97, Pis04, Buc02c] fail in the face of an active internal attack where the adversaries have the network secrets in their possession. A collusion of more than one malicious node increases the severity of the threat manifold.

2.1 Key Management in Ad Hoc Networks

Sharing of secrets [Des87, Sha79] and secure key exchange [Bec98, Bur95, Ste00, Blu83] have been actively explored by researchers over decades, but very little effort has been given to extend the concepts to ad hoc networks. After the discovery of the Diffie-Hellman key exchange in a Public key scenario [Dif76], some researchers tried to implement the concept in group communication and even in ad hoc networks [Zho99, Ste96, Hie01, Bur95, Des87, Sha79, Ste00, Yas02, Yi02]. One of the first extensions of the Diffie-Hellman key exchange protocol to group communication was proposed by Steiner *et. al.* which is referred to as the generalized Diffie-Hellman protocol [Ste96]. Each member contributes its own part of the key by performing an exponentiation. The group key is given by:

$$K = \alpha^{\prod_{k=1}^{n} N_k} \qquad \text{----------} \qquad (1)$$

where

K = group key

α = exponentiation base or the generator

N_k = random exponent generator by member k

n = total number of members in the group

[Ste96] discusses three protocols, each one optimizing on the number of messages exchanged and the number of exponentiations. However, all the protocols suffer from one big disadvantage; they allow all the group members to have the entire key. Thus the group key can be obtained by compromising any of its members.

There are other protocols which follow the generalized Diffie-Hellman protocol. Some of them are the *Hypercube* protocol and the *Octopus* protocol presented by Becker and Wille in

[Bec98]. In the *Hypercube* protocol, the 2^n members are arranged in hypercube. Key exchange is done in steps, one for each square. In the *Octopus* protocol, four of the participants form a square at the center and the remaining members form tentacles that are attached to the central nodes. First, the key exchange is done between the central nodes by the 4-node hypercube protocol and then the key is distributed to the surrounding nodes. Another protocol, called the *tree-based* protocol, is presented by Burmester and Desmedt in [Bur95]. This protocol is applicable for networks having the topology of a binary tree. The root of the tree generates the key and distributes it along the tree. This protocol is not contributory and hence can not be applied to an ad hoc network. All of these protocols for distributing a group key are based on a fixed architecture and are not suitable for an ad hoc network scenario.

Some work on key distribution in ad hoc networks is based on threshold cryptography [Zho99, Bec98, Bur95, Des87, Sha79, Ste00, Gen96a, Gen96b]. The concept of threshold cryptography was first proposed by Desmedt in [Des87, Des97]. He proposed a protocol, which accepts k out of n players to create a valid signature, but ignores the signature if less than k players take part in the signing process. This was based upon the concept of sharing a secret as proposed by Shamir in 1979 [Sha79]:

"Consider, for example, a company that digitally signs all checks. If each executive is given a copy of the company's secret signature key, the system is inconvenient but easy to misuse. If the cooperation of all the company's executives is necessary in order to sign each check, the system is safe but inconvenient. The standard solution requires at least three signatures per check, and it is easy to implement with a (3,n) threshold scheme."

Another implementation of threshold cryptography is the CLIQUES protocol proposed by Steiner *et al* [Ste00]. In this protocol, the key computation proceeds from node to node, with each node doing an exponentiation with its private Diffie-Hallman (DH) value. The final node computes the key and broadcasts it to all other members. The main drawbacks of this protocol,

when applied to an ad hoc network, are that key generation has to be in serial order, which violates the dynamic configuration change in ad hoc networks, and that nodes have the final group key, which can be hacked by compromising any of the nodes.

Alec Yasinsac *et al* modified this CLIQUES proposal in [Yas02] to make it more suitable for an ad hoc network, by optimizing the number of messages and taking care of the serialization problem. The protocol works as follows:

- One member announces the formation of a group;
- The potential group members ($i = 1...n$) select and publish a coordinator (member #0), the DH base g and modulus p, with base and modulus having all the necessary properties to ensure that the impending DH computations are secure;
- Each i^{th} member (except the coordinator) chooses a random x_i as their private DH number and broadcasts their public DH number g^{r_i};
- The coordinator generates the random numbers z and x_0, g^{r_0}, $g^{r_i x_0}$ for each i, and encrypts $e[z]\, g^{r_i x_0}$ for each i. The coordinator then concatenates g^{r_0} with all the encrypted values and broadcasts the concatenated message;
- After receiving the broadcast from the coordinator, each member computes $g^{r_i x_0}$ using their private x_i, decrypts z, and computes a combining function $F = f\left(g^{r_1}, g^{r_2}, ..., g^{r_i}\right)$, and the group key, $K = g^{F \square z}$.

Although this modified protocol solves the serialization problem, the problem of the nodes having access to the entire key remains unresolved. Any node can still be compromised, giving away the entire group key. This is a major threat in an ad hoc scenario where the physical security of a node is always threatened.

Yi and Kravets [Yi02] proposed a key management service in an ad hoc scenario whereby the functionalities of a Certificate Authority (CA) are distributed among a number of

mobile nodes (called MOCA). The proposed key management service satisfies the three broad criteria, namely fault-tolerance, vulnerability and availability. The paper extended the concept of threshold cryptography by sharing the digital signature among a number of MOCAs. A client requiring a certification service, contacts at least k number of MOCAs, each of which generates its own share of signature and sends it back. The client reconstructs the full signature after receiving k partial signatures. This proposed key management service overcomes the disadvantage of a single CA in an ad hoc scenario by distributing the functionalities to a number of them. But it fails to overcome the threat of key disclosure by compromising a node. An intruder can easily compromise the client to get the key and hack into the network. A similar approach is proposed in [Kon01]. The authors proposed a model for intrusion detection based on the distribution of certificate authority functionalities using a threshold secret sharing mechanism. Their model works under the assumption that the network elements have some information which is untrustworthy or unknown to any intruder. This assumption does not hold true for a compromised node or a disloyal node. A certificate distribution algorithm is also proposed in [Hub01] based on a public key infrastructure where users are responsible for storing and distributing the certificates by building local certificate repositories.

In [Zho99], Zhou and Haas have proposed a key management model again based on threshold cryptography. Their model is built upon the distribution of trust and the assumption of using a public key infrastructure. Public key infrastructure is assumed because of its superior key distribution and its ability to achieve message integrity and non-repudiation. The functionalities of a certification authority are distributed among n number of nodes called servers. A threshold level of $(t + 1)$ has been used so that at least $(t + 1)$ servers can combine to form the final signature. With $(t + 1)$ valid partial signatures, a combiner can compute the full signature for the certificate. However, compromised servers cannot create a valid signature as they can generate at most t partial signatures. The protocol also takes care of mobile adversaries. The concept of

mobile adversaries was first proposed by Ostrovsky and Yung [Dif76]. This type of adversary moves on compromising each server until it compromises at least $(t + 1)$ of them. It can gather at least $(t + 1)$ partial signatures to compute the entire key. The authors proposed a proactive scheme to counter this mobile adversary. The scheme uses share refreshing through which each server computes a new share from the old one. However, the protocol works under the assumption that the shares from each server are put together by the combiner to compute the full signature. This combiner, being any server, can always be a single point of attack. A combiner, if compromised, can give out the full signature and because of this will not work in an ad hoc scenario.

2.2 Secure Routing in Ad Hoc Networks

Routing protocols have been designed for ad hoc networks since their evolution. None of these routing protocols incorporated security into them during their design stage. However, researchers have started looking into the vulnerabilities of these routing protocols for the past few years and have proposed solutions to make them secure and robust. Most of these protocols revert to cryptographic techniques for their security and essentially find the shortest path from source to destination.

There have been some work [Den02, Yi01, Pap02, Pap03, Hu02a, Hu02b, Zap02, San02, Yan02, Smi97] to secure the distance vector routing protocols in ad hoc networks. In [San02], the authors have proposed an authenticated routing for ad hoc applications. This protocol named ARAN uses digital signature by each node to authenticate itself, which is duly verified by the next node. Their work has also assumed the existence of a trusted certificate server. Both assumptions may not be applicable in an ad hoc network scenario. The presence of a trusted certificate server cannot be assumed because it violates the basic norms of an ad hoc paradigm. Also, the use of digital signature in a public key infrastructure is of very high computational

complexity and hence is not suitable for ad hoc nodes with limited computational power. In this context, some points need to be emphasized when using a public key communication system for ad hoc application.

First, if a message is encrypted with the receiver's public key, then the authentication of the sender can be questioned. This is more applicable in an ad hoc scenario where there is little or no trust between nodes. Thus, every message must be signed digitally by the sender. Second, if the message is signed by the sender (i.e., encrypted with the sender's private key), then any node in the network can decrypt the message with the sender's public key which is known to all. Consequently, the confidentiality and integrity of the message are questioned. The ideal system would be to use both digital signature as well as public key encryption.

The most widely used public key cryptosystem is the RSA algorithm. Although this algorithm is widely implemented, its computational complexity is very high. The computational aspects of RSA involve the following:

Generation of two large prime numbers- it has been undoubtedly proved that primes near N are spaced, on an average, one in every (lnN) integer. So, to generate a prime number of order of magnitude 2^{200} would require $ln(2^{200})/2$ (to discard the even integers) = 70 trials. It can be shown that for a 100-digit number, 1 in every 230 is a prime.

To test the primality of a number, an algorithm was proposed based on the well-known Fermat's theorem which states that:

$$\text{If } p \text{ is prime, then, } a^{p-1} = 1 \bmod p \quad \text{where } 0 < a < p$$

By this algorithm, if (a^{p-1}) is not equal to 1, then the number p is definitely not prime. But if the result is 1, then the probability the number is prime would be $1/10^{13}$. Even if the probability is very low, it proves a great risk for a high security ad hoc application. If the primality test does not work, the RSA algorithm will totally fail.

Computing exponentiation of a large number by another large number also takes much computational time. To ensure authentication and message confidentiality and integrity, it has to be done twice, once for computing digital signature, and again for public key encryption.

To avoid using a public key cryptosystem in an ad hoc scenario, some researchers have come up with secure routing solutions using symmetric key systems. In [Hu02b], the authors have proposed a new on-demand secure routing protocol called Ariadne. The protocol is based on symmetric key cryptography and its security depends on the secrecy and authenticity of secret keys stored in nodes. A source node performs route discovery, based on the assumption that it already shares a secret key with the destination node. The source node simply includes a message authentication code (MAC)[4] computed with the shared key which the destination node can easily verify. Thus the protocol relies on the distribution of shared secret keys between source and destination, which itself is a burning research problem in an ad hoc network, without the presence of any trusted entity.

While Ariadne uses an end to end security solution, a hop by hop approach is proposed in [Hu02b]. The authors have proposed a secure routing protocol based on the destination sequence distance vector (DSDV) routing. The security is based on the efficient use of one-way hash function, unlike the use of MAC in Ariadne. The source node generates the elements of its hash chain upon initialization and uses some elements from the chain to secure its routing update over time. The authentication protocol works under the assumption that a secure means of distributing the elements of the hash chain is already there, an assumption which itself defies the MANET paradigm because of the absence of any trusted entity.

In [Zap02], the authors have proposed to extend the existing AODV [Per99] routing protocol to make it secure. They have proposed to use digital signature to secure the non-mutable

[4] We have used MAC to represent both Media Access Control and Message Authentication Code interchangeably. We have specified the meaning of MAC during each use.

fields of the AODV messages and hash chains to secure the hop count information, which is the only mutable field in AODV. The protocol works under the assumption of the existence of an efficient key management system enabling all the ad hoc nodes to obtain public key information of all other nodes. The authors also did not consider the problem of compromised nodes, which they think is not critical in non-military application. However, this assumption is too strong in securing ad hoc communication as compromised nodes can disrupt network operation in sensitive applications outside the military environment. The security of the proposed scheme is also limited by an attacker who deliberately keeps the information unchanged and can force the source node to select the path.

Another extension of AODV is proposed in [Yan02]. The proposed protocol requires each node to carry a token signed with a secret system key, which can be appropriately verified by its neighbors. A node without a valid token is singled out in the network and all other nodes stop communicating with it. They have referred to the threshold cryptographic scheme to secretly distribute the token among nodes.

Some work has also been done to secure routing protocols based on the existing DSR [Joh99] protocol. In [Buc02b] the authors have proposed a CONFIDANT protocol based on DSR. It aims at isolating the misbehaving nodes, making non-cooperation unattractive. The monitoring mechanism is implemented by a neighborhood watch concept where the no-forwarding behavior of the nodes are monitored and reported. No-forwarding behavior or so-called selfishness of the nodes is also taken care of by a mechanism proposed in [But02]. The authors have proposed a protocol based on having simple counters at each node, called nuglets, which will encourage the nodes to forward the packets. However, this protocol only ensures the selfless act of nodes in forwarding others' packets. It does not ensure such an act in forwarding packets with malicious routing information. Although the protocol secures the network from the presence of selfish

nodes, it does not secure it from malicious nodes which enthusiastically forward others' packets by modifying information in it.

The above protocols use a secure way of route discovery, but do not consider any secure means to discover the topology of the network [Pap03, Pap02]. In [Pap03], the authors have proposed a protocol for securely discovering the network topology in a public key infrastructure. The protocol is responsible for securing the discovery and distribution of link state information. Each node broadcasts signed hello messages to its neighbors giving its MAC address, IP address pair. The receiving node retains that information after validating the signature. However, the protocol fails in face of a colluding attack, and also does not scale well to frequent topology changes. Another protocol to achieve a similar goal is proposed in [Pap02]. The authors have proposed a source-routing protocol that can securely discover correct connecting information. It works under the assumption of an already established shared secret between the source and the destination. The security of the protocol is based upon the computation of a Message Authentication Code (MAC) using the source, destination, unique query identifiers and the shared secret. However, an internal attacker in the network, who can get hold of the shared secret, can easily place itself on the end-to-end route, and get hold of all the data packets. Moreover, the protocol does not secure the network from an attack by colluding malicious nodes.

All the above solutions have used either a symmetric or a public key cryptosystem, intending to use either a shared secret key or a digital signature to authenticate and protect the routing information. They still tend to find the shortest path from source to destination, irrespective of some untrustworthy links in between. In [Yi01], the authors have proposed a secured routing protocol based upon the trust level of the nodes. Their protocol is based on an on-demand protocol like AODV [Per99] or DSR [Joh99]. The authors have defined a security metric and embedded it into the RREQ packet. When an intermediate node receives a RREQ with a specified security metric or trust level, it can only process or forward the packet if it meets the

required security level, otherwise it drops it. If an end to end path with the required security metric is found, then only a RREP is sent back. However, this solution does not prevent a compromised node from changing its trust level to match the higher trust level of the packet, and getting all the messages it is not supposed to. The authors also did not discuss any model for computing and distributing the trust levels.

A similar approach to compute trusted routes based on trust levels has been proposed in [Pir04b]. The trust computation is based on the actions of three components, namely the trust agent, the reputation agent and the combiner. Each node in the network computes a direct trust value for its immediate neighbor based on the latter's honesty in executing the routing protocol. The nodes share their trust opinions about other nodes through an effective reputation exchange protocol. The trust values are used as the metric to compute end-to-end routes. However, it has not been clearly mentioned how these trust levels are used in the route computation. Also the proposed protocol does not secure the network from colluding malicious entities.

All the secure routing protocols proposed so far lack any formal model to prove their security. The first such attempt to develop a formal model to prove the security of a routing protocol was done in [But04]. The authors have suggested two attacks on SRP [Pap02] and Ariadne [Hu02b] based on the method. However the authors assumed that the adversary only advertises non-existent routes. This type of attack, although degrades network performance, is not detrimental to the confidentiality of the network secrets. There is however no discussion that the proposed method can prove the security of the routing protocols in face of other attacks such as the modification of routing metrics by a malicious entity.

2.3 Trust Computation in Ad Hoc Networks

All the security solutions proposed so far and discussed in the last section have at least one direct point of weakness. This point, when compromised, can disclose the group key and,

thus, threaten some of the basic security requirements like authentication, confidentiality and integrity. The use of a trust computational model can be thought of as an alternative approach to cryptographic solutions. Establishing security associations based on distributed trust among nodes is an important consideration while designing a secure routing solution, though not much work has been done to develop a trust model to build-up, distribute and manage trust levels among the ad hoc nodes. Most of the proposed schemes talk about the general requirement of trust establishment [Buc02c, Ver01, Esc02, Kag01], but do not come up with any specific model or computational framework to do so. None of the models proposed so far have tried to understand and analyze different malicious behavior of the attacker and quantify those behaviors in a policy-based computational framework.

Modeling and computing trust for a distributed environment has been actively researched for quite a long time [Zhu03, Bet94, Abd97], though not much work has been done to extend the concept in ad hoc networks. Most of these distributed trust models combine direct and recommended trusts to come up with trust computations. The concept of direct and recommended trusts was given in [Bet94]. The authors defined a direct trust relationship as:

$$P\ trusts_x\ ^{seq}\ Q\ value\ V$$

Where x is the trust class, V is the value of trust relationship, which is an estimation of the probability that Q behaves well when being trusted. Thus, a direct trust relationship exists between P and Q if P has all positive experiences with Q. seq is the sequence of entities that mediated the experiences. In a similar way, a recommendation trust has been defined as follows:

$$P\ trusts.rec_x\ ^{seq}\ Q\ when.path\ S_p\ when.target\ S_t\ \ value\ V$$

A recommendation trust exists between P and Q if P is willing to accept from Q reports about third entities with a specific trust class x. This trust is restricted to the experiences with entities in S_t (target constraint set) mediated by entities in S_p (path constraint set). seq is the sequence of entities that mediated the recommendation and V is the value of the trust relationship.

The computation of trust based on values of trust relationships and sequence of mediating entities results into continuous values which necessitates the specification of a threshold, which is not easy to contemplate in a sensitive and dynamic application. When extending the concept of direct and recommendation trusts in ad hoc networks, it is desirable to avoid the later as it encourages the collusion between malicious nodes. Moreover, there is a need to understand and analyze different malicious behavior and quantify them in the computational model.

A similar approach of direct and recommendation trust has been taken by the authors in [Abd97]. They have suggested a recommendation protocol to formalize the propagation of trust information by issuing *recommendation request* and *recommendation* messages. However, the proposed model lacks any mathematical basis to calculate the trust values. The authors have not discussed how the trust values are computed and updated. In addition, this model, when extended to an ad hoc network with frequent topology changes, results in generating a large routing overhead in the form of control packets with each node generating more and more *recommendation request* messages.

A pairwise trust establishment based on self trust and group trust has been proposed in [Vir05]. The authors proposed to use the trust establishment model for establishing pairwise keys between nodes. However, the problem of key distribution still exists which cannot be solved from a trust establishment angle. The trust model proposed here also does not take into account different forms of malicious behavior.

In [Gra02] the authors have proposed a trust formation and risk assessment schemes based on the small world concept [Mil67]. The small world concept suggests that "any pair of entities in a seemingly vast, random network can actually connect in a predictable way through relatively short paths of mutual acquaintances" [Gra02]. The authors have proposed a trust-based security architecture consisting of four components namely, entity recognition, trust-based

admission control, risk assessment and trust management. The trust formula, as given by the authors, is as follows:

$$T_{p_0\ (p_m)} = \frac{\sum_{k=1}^{m} w_k \left(T_{p_{k-1}}\right)_{(p_k)}}{m} \qquad \text{----------} \qquad (2)$$

where

$T_{p_0\ (p_m)}$ = trust value p_0 forms for any p_m

p_0 = principal making admission control decision

p_m = principal m steps away from p_0 and requesting admission

m = total steps between p_0 and p_m

w_k = discounting factor (decreases as k increases)

In a similar way, the authors presented the risk assessment formula as follows:

$$R_{p_0(p_m)} = 1 - \left(\left(1 - R_{p_0(p_1)}\right)\left(1 - R_{p_1(p_2)}\right).....\left(1 - R_{p_n(p_m)}\right)\right) \qquad \text{---------} \qquad (3)$$

where

$R_{p_0(p_m)}$ = risk assessment p_o forms for interaction with any p_m;
the other parameters being same as the previous formula.

The above two formulae for computing the trust factor and the risk assessment factor are based upon the recommendation of each entity about its next entity in the path chain. This model, when extended to a security-sensitive ad hoc application can attract collusion among different malicious entities, and hence is not welcome. The trust management component, as designed by the authors, is based upon the small world assumption that an entity, when roaming in a particular environment, knows specific information about that environment. This assumption is primarily

based upon the Small World Clustering algorithm described in [Mat02] as "*a cluster often shows the particular context*". However, this assumption is too strong when applied to an security-sensitive ad hoc application.

A policy based approach has been proposed in [Bla96], based on a simple language to specify trust actions and relationships. The authors proposed a trust management system called PolicyMaker which binds Public Keys to the predicates defining actions for which they are used. PolicyMaker accepts as input a set of policy statements, a collection of credentials and a description of a proposed trusted action. It then evaluates the proposed actions by interpreting the policy statements and credentials. A simple example of the PolicyMaker language is shown below:

$key_1, key_2, \ldots\ldots, key_n$ **REQUESTS** *ActionString*

Where the *ActionString* describes a trusted action requested by a sequence of (or a single) public keys. This PolicyMaker language is too simplistic to use in an ad hoc scenario where the actions of an entity can be forcefully implemented by an intruder. For example, when a node is compromised it can carry out the same set of trusted actions with the key(s) which will be authorized as per the PolicyMaker language specification.

Watchdog mechanism [Mar00], based on promiscuous mode operation of the ad hoc nodes, has been the fundamental assumption in any trust computational model. In [Yan03] the authors have proposed a trust evaluation-based secure routing solution. The trust evaluation is done based on a trust matrix stored at each ad hoc node. The matrix consists of several parameters on which the final trust evaluation is computed. However, the mechanism for collecting the required parameters was not discussed by the authors. They also did not discuss the means of measuring communication success or failure pertaining to the parameter *experience statistics*. Also, some of the parameters suggested by the authors are not realistic in a highly sensitive application; for example, the parameter *personal preference* may attract colluding attack in the

network. In [Pir04a] a similar concept has been proposed. The authors have defined different trust categories based on the effectiveness of the protocol functionalities. The final trust computation has been formulated as follows:

$$T_x(y) = \sum_i W_x(i) * T_x(i) \quad \text{----------} \quad (4)$$

where

$T_x(y)$ is the trust of node x in node y

$W_x(i)$ is the weight of the i^{th} trust category

$T_x(i)$ is the situational trust of x in the i^{th} trust category

The trust computation is based only on the success and failure of transmission of different packets and does not take into account different forms of malicious behavior. In [Nga04] the authors have proposed an authentication scheme based on Public Key infrastructure and distributed trust relationship. The trust relationship is established by direct as well as recommended trusts. Composite trust is computed by combining both direct and recommended trust relationships. Some work has also been done to establish trust based on distribution of certificates. In [Dav04] the authors have proposed such a trust management scheme. However, the proposed scheme lacks any specific framework for computing the indices.

Another model has been proposed based on subjective logic [Li04]. The concept of subjective logic was first proposed by Josang [Jos01, Jos98, Jos97]. Subjective logic is "*a logic which operates on subjective beliefs about the world, and uses the term opinion to denote the representation of a subjective belief*" [Jos01]. An opinion towards another entity x is represented by three states: *belief [b(x)]*, *disbelief [d(x)]* and *uncertainty [u(x)]*, with the following equality:

$b(x) + d(x) + u(x) = 1$

The concept of subjective logic has been extended to propose a trusted routing solution in [Li04]. The opinion of a node about another node is represented in a 3-dimensional matrix representing *trust*, *distrust* and *uncertain* opinions. The opinions are updated by a positive or a negative feedback from the node in question. The proposed model, however, fails to protect the network from an internal attack, where a malicious node either refuses to forward the packets and duly authenticates itself to the source, or it cooperates with the source node and acts as a black hole. The vulnerabilities are discussed in details in the following cases:

Case 1: A compromised node B does not forward node A's message. A's opinion towards B changes to (0, 0.33, 0.67) from (0, 0, 1), where the values represent *trust*, *distrust* and *uncertainty* respectively. A subsequently asks for B's digital signature, which is duly supplied by B. Consequently, A's opinion towards B changes towards positive.

Case 2: B duly forwards all messages sent by A, hence A's opinion towards B changes towards positive. B can inject false routing information to place itself into the route and subsequently act as a black hole.

Some mechanisms have been proposed to give incentives to the nodes for acting unselfishly. In [He04] authors have proposed a secure reputation-based incentive scheme (SORI) that prevents the nodes from behaving in a selfish way. The scheme, however, does not prevent a malicious node from selectively forwarding packets or from other malicious behavior.

In view of the above, we prefer to model a trust computational framework based on different malicious behavior. The problem is not easy, as it requires intricate understanding of the different malicious intentions and actions and model them in the computational framework. This trust model can then be used with the trust based routing solution designed by us which will eventually be discussed in later chapters.

Chapter 3

Collaborative Trust-based Secure Routing Protocol

This chapter highlights the design of the trusted routing protocol, which we call Trust-embedded AODV (T-AODV). We will discuss our assumptions towards designing the protocol, give an overview of the design and finally talk about the protocol in details with analysis of the results obtained from extensive simulation. We will also carry out a case by case analysis to demonstrate the security of T-AODV against different threats in the network.

3.1 Introduction

In our quest for developing a solution for the problems defined in chapter 1, we have developed a trusted routing solution for ad hoc network applications. The solution has been developed with an extension of the Ad Hoc On Demand Distance Vector (AODV) [Per99] routing protocol. The selection of AODV was motivated by its superior performance over the Dynamic Source Routing (DSR) [Joh99] protocol as highlighted in chapter 1, especially with increased mobility in the network. Our trusted protocol is unique from other solutions proposed in the literature in that it is capable of finding a trusted end-to-end route free of any malicious entity acting either independently or in collusion. It is the first such solution proposed to counter an internal attack from colluding malicious nodes. The protocol does not encourage shortest path route discovery as in traditional AODV, but relies on a trust metric to do so. To formulate and distribute the trust values, we have developed a trust computational model that takes into account different malicious behavior typical to an ad hoc application. The model is unique and different from other solutions proposed in the sense that it analyzes different malicious behavior of the nodes and quantifies them in the model itself.

The remainder of this chapter is organized as follows. In the next section we will discuss the assumptions for designing the protocol, followed by an overall description and detailed overview. Extensive simulation resilts will follow with analysis of the results. Finally, we will carry out a security analysis of the protocol by evaluating different threat scenarios.

3.2 Assumptions

Our trusted routing protocol is based on the following assumptions that we think are justified. First, all the nodes communicate via a shared wireless channel and all communication channels are bi-directional. Second, all the nodes operate in a promiscuous mode. Third, our main focus is on the network layer and the protocol that we propose here is an extension of the Ad hoc On Demand Distance Vector (AODV) [Per99] routing protocol which we call Trust-embedded AODV (T-AODV). We have assumed a reliable link layer protocol to be in place. Fourth, we assume that all the nodes are identical in their physical characteristics, i.e., if node A is within the transmission range of B, then B is also within the transmission range of A.

The above assumptions are very fundamental and used for all the solutions proposed so far. Finally, we do not encourage the notion of trust transitivity, i.e., 'if A trusts B and B trusts C, then A also trusts C'. This trust transitivity encourages more colluding attack in the network from multiple malicious nodes. For example, if a trusted routing solution [Gho04b] is in use in the network, then more than one malicious nodes can collude and make their combined trust high enough to put them in the active route. Instead, our trust model is based on collaborative effort of all the nodes and analysis of different malicious behavior.

3.3 Design of Trust-embedded AODV (T-AODV)

Essentially all routing protocols in the ad hoc community tend to find the shortest path to the destination irrespective of the presence of any malicious node in that path. We can argue that,

as internal threat in the network in the form of a compromised or disloyal node is of significant concern, a path free of malicious node is more important than the shortest path. In the following section we present a detailed description of T-AODV.

3.3.1 Overall Protocol Description

The motivation for designing T-AODV comes from finding a trusted end-to-end path free of malicious nodes. The basic idea behind the protocol is for a node to append the trust level of its predecessor from which has received the route request packet. Trust levels are defined to be unique values of the level of trustworthiness of a node on another node, a detailed modeling of which will be discussed in chapter 4. A path with maximum trust level will eventually be selected by the destination node and will be sent to the source as the end-to-end active path to be used. A node with malicious intention will try to put itself into that active route by trying to inject malicious trust information. The protocol will ensure that all the trust level information provided by a node will be checked by its predecessor node in order to ensure information authenticity. The preliminary design of T-AODV has also been extended to protect the network from colluding attack where any accusation about a node will also be checked. This is ensured by computing a signature with the Private Key of the node, alongwith the trust level computation. In the next sections we are going to discuss the protocols in depth.

3.3.2 High Level Description of T-AODV

When a node wants to find a route to another node, it initiates a route discovery. The route request packet header contains a *trust_level* field which is concatenated with the IP address of the node whose trust level is being appended, in addition to the other fields in AODV route request, as shown in figure 3.1. The header also contains a *cumulative trust level* field which reflects the sum of the accumulated trust level of all the nodes in the path. When an intermediate node receives the route request packet, it rebroadcasts it after modifying the *trust_level* field to

include the trust level of the node that sends it the route request and also increases the *cumulative trust level* field by the trust level of its previous node. Every node checks back the rebroadcasted route request packet from its next node to see whether it has provided the proper information. If not, it immediately broadcasts a warning message questioning the intended malicious action of that node. Our protocol does not encourage any intermediate node to send a route reply. The final route selection is based upon the *trust_level* metric where the destination node selects the path with the maximum *Cumulative Trust Level*. *Hop_count* plays a role in deciding the final route only when more than one packet has same *trust_level*.

Source Address	*Destination Address*	*Source Sequence Number*	*Destination Sequence Number*	*Last Address*	*Broadcast ID*	*Hop Count*	*Previous node IP : Trust Level*	*Cumulative Trust Level*

Figure 3.1 Route Request packet structure in T-AODV

where

- *Source Address* is the address from which the RREQ packet originates;
- *Destination Address* is the address to which source node wants to send data;
- *Source Sequence Number* is the latest sequence number received in the past by the source for any route towards the destination;
- *Destination Sequence Number* is the most recent sequence number used by the source;
- *Last Address* is the address of the node from which its next node receives RREQ;
- *Broadcast ID* is a sequence number uniquely identifying the particular RREQ when taken in conjunction with the originating node's IP address;
- *Hop Count* is the number of hops from the source node to the node handling the RREQ;
- *Previous Node IP* is the IP address of the previous node from which a node receives the RREQ;

- *Trust Level* is a unique value identifying the level of trust of a node on another node;
- *Cumulative Trust Level* is the sum of trust levels of all the nodes in the path.

The route reply packet has the next hop information. This is in line with the solution given in [Den02] to counter the black hole problem. When the source node gets back the first route reply, it waits for a specified amount of time before using that route. If within that time another route reply comes, the source node queries the next hops of the two route replies. The next hop of the malicious route reply will obviously not have the same route to the destination. Thus, malicious route injection into the network can be prevented. The procedure below shows the action of a node after it receives a route request packet.

```
// when a node receives a Route Request packet

Receice_RREQ( ) {
// check whether it is the destination of the route request
if destination
   compute_highest_trust_level( )
   // in case more than one RREQ has same trust_level
   // decides on the basis of lowest hop_count
   sends_RREP_to_source( )
else (not destination)
   if duplicate packet
      cross_checks_trust_level( )
      if found correct
       drops the packet
     else
       broadcasts roure_warning message( )
      end if
   else (not duplicate)
      concatenates the previous node trust level with
        previous node IP address
      increases the cumulative trust level
      increments hop_count
      rebroadcasts RREQ
   end if
end if
}  // end of function Receice_RREQ
```

Figure 3.2 Procedure for the action of a node after receiving the RREQ packet

A node first checks whether it is the destination of the packet. If it is the destination, it creates a route reply and sends it back along the reverse route. If it is not the destination, it checks whether the packet is duplicate. If found duplicate, the node cross checks its trust level provided by its neighbor and takes action according to the correctness of the information. If the packet is not a duplicate, it appends the necessary information and rebroadcasts it.

The procedure below shows the detailed action of the source node after it receives the first route reply.

```
// when the source node gets back the first Route Reply
 Receive_RREP( ) {
   waits for a specified period
   if receives another RREP
     queries next_hop( )
   else
     sends data( )
   end if
 } // end of function Receive_RREP
```

Figure 3.3 Procedure for the action of the source node after receiving the first route reply

The function *cross_checks_trust_level* can be implemented in two ways. When an intermediate node receives a duplicate route request packet, it extracts the concatenation of (IP Address : Trust level). If it finds that the IP addess matches its own address, it cross checks the trust level appended by the node. The following algorithm implements this function.

```
// Extracts (IP Address : Trust Level)
if (IP Adress = = own address)
    cross_checks_trust_level( )
     if (trust level does not match)
         broadcasts roure_warning message( )
     else
         drops the packet( )
else
         drops the packet( )
end if
```

Figure 3.4 Procedure for the Cross checks trust level function

The second possible implementation takes care of the above assumption. An intermediate node, on receiving a duplicate route request packet, extracts the address stored in the *lastaddr* field (the *lastaddr* field contains the address of the node from which the next node receives a route request packet) and checks from the neighbor table whether it is from any of its neighbor. The neighbor table contains a list of all the nodes in the one-hop neighborhood of a node. The table is populated by the *Hello* messages received in regular intervals.The algorithm works as follows:

```
if (lastaddr = = neighbortable->addr)
    cross_checks_trust_level( )
else
    drops the packet( )
end if
```

Figure 3.5 Procedure for the alternative implementation of Cross checks trust level function

The above implementation can actually increase the computational overhead in each node. However, the computational overhead can be reduced by efficient searching of the neighbor table.

3.3.3 Proof of Protocol Security under Attack from Independent Malicious Nodes

Below we present a simple proof to show the security of the protocol. The proof uses method of contradiction and shows that the protocol is secure under the assumption of an independent attack[5] from malicious nodes.

***Theorem**: In the presence of malicious nodes acting independently, our protocol (T-AODV) is secure.*

***Proof**:* Let us assume that the protocol is not secure in presence of malicious nodes acting independently. In that case, any malicious node will inject faulty information into the network to

[5] An independent attack is carried out by one or more malicious nodes without the aid of any other node.

include itself in the routing path. Eventually, all information will be forwarded through it. The malicious node will be able to do this successfully by putting a very high trust level for its previous node. This act will go unnoticed and the route selected by the destination will eventually include the malicious node.

Let us assume a *n-node* network, where $n > 2$. Let S_j be the set of nodes in the neighborhood of node j. In case node j has 4 nodes in its neighborhood we can denote S_j as:

$$S_j = \{(j-i) : -2 \leq i \leq +2 , i \neq 0 \}, \quad where \ j \in \mathrm{N}$$

Let $s_j \subset S_j$ be the set of nodes from which node j receives route request. Let,

$$s_j = \{(j-i) : i = 1, 2\}$$

Now, if node j gets compromised, it will want to put malicious routing information. So. after receiving the route request, j puts the wrong *trust_level* for either or all of its predecessors, $j-i$ $(j-i \in s_j)$ and rebroadcasts it. When the nodes $j-i$ $(j-i \in s_j)$ receive copies of the rebroadcasted route request from node j, they cross check the information. If either of them finds that j is trying to put malicious information, it immediately broadcasts a warning message to all its neighbors about the sanctity of node j. Thus, injection of any faulty information by a malicious node will be detected by its predecessor. Hence, we can argue that our assumption is not correct. Thus, our protocol is secure in presence of independent malicious nodes. *Q.E.D.*

We recognize that, the protocol that we have designed so far, has one possible vulnerability. It fails to secure the network against multiple malicious nodes colluding together. In the following section we discuss a threat model with multiple colluding malicious nodes and design solutions to secure against it.

3.3.4 Threat Model

As we have discussed in the earlier section, our secure routing protocol fails to secure the network against multiple malicious nodes colluding together. In this section we describe the

threat model and propose a secure solution for it. Let us consider the following example shown in Figure1 below.

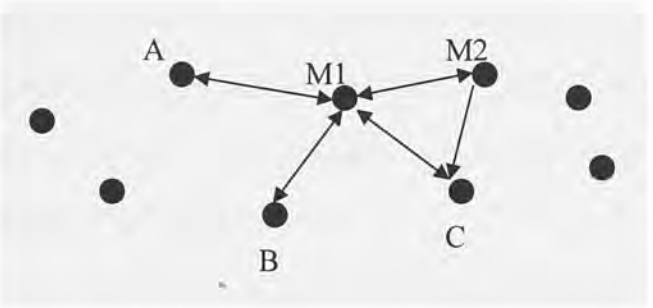

Figure 3.6 An example of the threat model

M1 gets a RREQ from A (it also gets a copy from B which is redundant). It rebroadcasts the RREQ packet to M2 and C. Let us assume that M1 and M2 are malicious and colluding to disrupt the routing operation. M2 appends a high trust level for M1 and rebroadcasts the packet. C, however, appends the right trust level. M1 broadcasts a RWARN message that C is malicious. C subsequently gets isolated from the network and the route through M1 and M2 is selected, as it has a high trust level. Thus, the malicious nodes collude with each other to bring down the entire network. The sheer purpose of finding a secure end-to-end route is defied and trusted nodes are isolated from the network as malicious.

3.3.5 Preventing Colluding Attack

The fact that ad hoc nodes are characterized by low level of trust among themselves, motivated us to design a secure algorithm based on internal spying and verification. We recognize the need of developing a mechanism to verify the claim of an accuser accusing another node of malicious behavior. For example, when M1 broadcasts a RWARN message to A and B about C's trustworthiness, there must be a mechanism by which both A and B check back M1's accusation.

Our algorithm to counter the colluding attack assumes the existence of a Public Key infrastructure. Each node has a <*Public Key (PK), Private Key (SK)*> pair, the generation of which can be done by any existing algorithm. Further assumptions are already discussed in section 3.1. We extended the T-AODV protocol discussed in section 3.1.1 to incorporate the security needed to counter the colluding attack. Each node, before broadcasting the RREQ packet, not only computes the *trust_level* field, but also computes a signature and appends it to the RREQ packet header. The signature is computed as follows:

$Sign_i$ = (*Source_Address, Broadcast ID,* $trust\_level_{i-1}$, $IP\_Address_i$: SK_i), $i \in N$

where

Source_Address is the address of the node from which RREQ originates,

Broadcast ID is a sequence number uniquely identifying the particular RREQ when taken in conjunction with the originating node's IP address,

$trust\_level_{i-1}$ is the trust level of the node from which node i receives the RREQ, $i \in N$

$IP\_Address_i$ is the IP address of node i $i \in N$,

SK_i is the private key of node *i*, $i \in N$,

N is the number of nodes

The (*Source_Address, Broadcast ID*) pair has been used in the signature to prevent replay attack. A malicious node can store a copy of the signature from another node and use it at a later point of time to accuse the honest node. If the trust level of the malicious node has already been changed, then the re-used signature will reflect its previous trust level and the honest node will be misinterpreted as dishonest and subsequently will be isolated.

After receiving the RREQ packet, a node, besides computing the *trust_level*, concatenates the source address, broadcast ID and its own IP address with the trust level of the

node from which it receives the RREQ and signs it with its private key to compute the signature. It then appends the signature in the RREQ packet before broadcasting it to its neighbors. When a node questions about another node's trustworthiness and broadcasts a route warning (RWARN) message, it not only sends the IP address of the accused node, but also the signature provided by the later. The RWARN message structure is shown below:

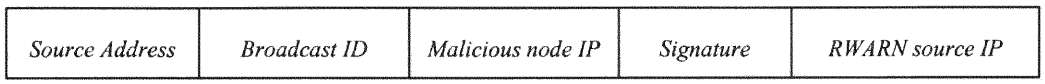

Figure 3.7 The RWARN message structure

where

- *Source Address* is the address from which the RREQ packet originates;
- *Broadcast ID* is a sequence number uniquely identifying the particular RREQ when taken in conjunction with the originating node's IP address;
- *Maliciosu Node IP* is the IP address of the node being accused;
- *Signature* is the Signature generated by a node;
- *RWARN source IP* is the IP address of the accusing node.

The inclusion of source address and broadcast ID into the signature is to prevent any replay attack that a malicious node can carry out. A malicious node can copy the signature field from the RREQ packet and use it at a later time to falsely accuse another node after its own trust level has changed. The inclusion of the source address and broadcast ID fields in the signature generation can successfully prevent such a replay attack.

When a node receives a RWARN message, it verifies the accusation by the sender of the message. It decrypts the signature with the public key of the accused node and checks whether the trust level matches the trust level of the accuser. If the two trust levels match, the node concludes that the accused node has provided right information, and hence cannot be malicious. Thus the

trustworthiness of the accuser is in question. However, if the trust levels are different, then it concludes that the accused node is malicious. The procedure is shown below:

```
receive RWARN ( ) {
decrypt_Sign ( ) {
if (decrypt = = yes)
     Check trust level provided by the accused node
     if (trust_level provided = trust_level of the accuser)
       the accuser is malicious
     else
       the accused node is malicious
     end if
else (decrypt = = no)
   the accuser is malicious
end if
    } //end of function decrypt_MAC
} //end of function receive RWARN
```

Figure 3.8 The receive RWARN function

We have carried out extensive simulation to show the effectiveness of the protocols that we have designed. In the next chapter we are going to highlight the simulation scenario and discuss the results in details.

3.4 Simulation Model

We have used Glomosim [Zen98] for our simulation. Glomosim is a scalable simulation software used for mobile ad hoc networks. We have carried out the simulation with two different scenarios. The mobility model selected for the simulation is random waypoint mobility. In this model a node randomly selects a destination from the physical terrain. It moves towards the chosen destination with a speed uniformly chosen between a minimum and a maximum speed limit. After reaching the destination, the node stays there for a certain pause time before it selects another destination and starts moving in that direction.

We have chosen two types of traffic for our simulation - CBR (Constant Bit Rate) and FTP (File Transfer Protocol). For each CBR traffic we have used 10000 packets each of length 512 bytes. In FTP tcplib has been used to simulate the file transfer protocol. In each FTP traffic we have used either 10 or 5 items to be sent to the destination node.

We defined a region of 2 Km by 2 Km and placed the nodes randomly within that region. In the first scenario, the nodes moved with uniform speed chosen between 0 to 10 meters/sec with 30 seconds pause between each successive movement. We increased the number of nodes and studied the network performance. In the second scenario, we have increased the node speed, keeping the similar infrastructure, to carry out our analysis. With these two scenarios, we are able to evaluate the scalability of our protocol with increased network size and increased mobility. The parameters for both the scenarios are shown in the table below.

Scenario 1	Independent variable	Set of parameters compared		
	Number of nodes	Routing overhead	Number of routes selected	Number of route errors
Scenario 2	Independent variable	Set of parameters compared		
	Node speed	Routing overhead	Number of routes selected	Number of route errors

Table 3.1 Parameters Chosen For Simulation

3.5 Analysis of Results

In our earlier work, when we designed the T-AODV routing protocol, we found that it had a very small increase in routing overhead than AODV (Figure 3.9), which we think can be traded off with the incorporation of security into the protocol.

This increase in overhead is due to retransmission of some route request packets because of delayed receipt of route reply by the source nodes as we do not encourage intermediate nodes to send route reply in our protocol. We can also see from Figure 3.9 that the percentage variation in overhead decreases with increasing number of nodes, which can be explained as follows. In AODV, as more and more nodes join the network, the probability of sending route replies by intermediate nodes from their caches increases, which accounts for more increase in overhead. In our protocol, as we do not encourage intermediate nodes to send route replies, the overhead does not increase much with increase in number of nodes. Hence the percentage variation of routing overhead actually decreases in our protocol from that in AODV.

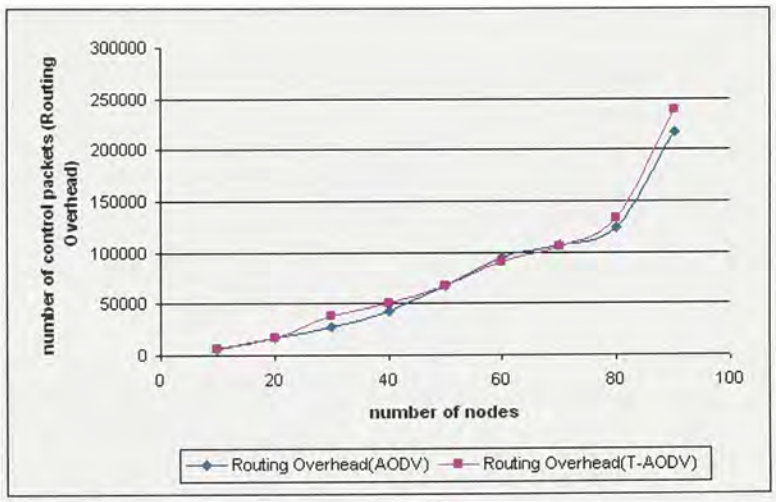

Figure 3.9 Comparison of routing overhead between AODV and T-AODV

Figure 3.10 shows the comparison between the routing overhead for AODV, T-AODV and modified T-AODV. We can conclude that modified T-AODV also has a small increase in its overhead. This increase in overhead is again due to retransmission of some route request packets because of delayed receipt of route reply by the source nodes. The average percentage increase in overhead for modified T-AODV than AODV has been found to be 5.5 %.

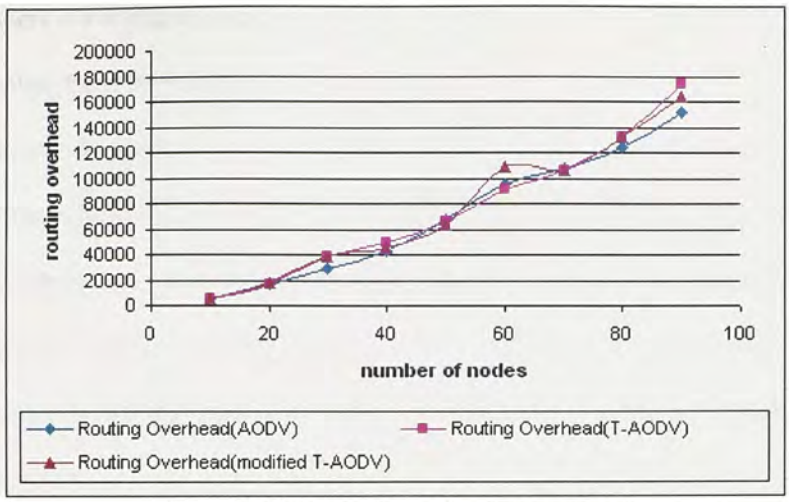

Figure 3.10 Comparison of routing overhead

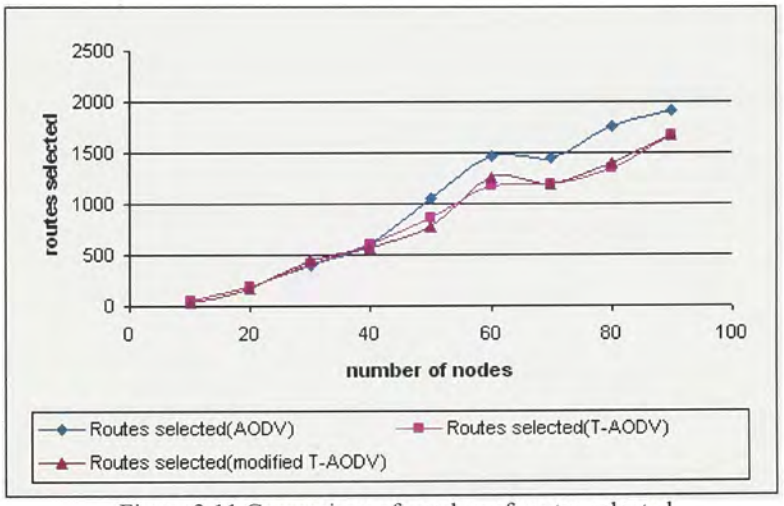

Figure 3.11 Comparison of number of routes selected

As no intermediate node is encouraged to come up with route replies, we obviously have lesser number of routes selected in T-AODV and modified T-AODV than that in AODV. This can be seen from Figure 3.10, which compares the number of routes selected for all the three protocols. However, this should not give any misconception that some of the routes are not properly selected. In fact, both T-AODV and modified T-AODV have lesser number of route errors reported than that in AODV as can be seen from Figure 3.12. If less number of routes are

selected, it renders lower processing overhead for the source nodes, as they do not have to process all the route replies from the intermediate nodes.

The reason for getting lesser number of route errors can be explained as follows. In AODV as the intermediate nodes come up with route replies, more MAC layer load is generated because of the unicast nature of route reply packets. However, in modified T-AODV, as no intermediate node is allowed to send a route reply, this leads to a lower MAC layer load and hence lower number of MAC layer collisions. This accounts for lower number of route error packets in our protocol.

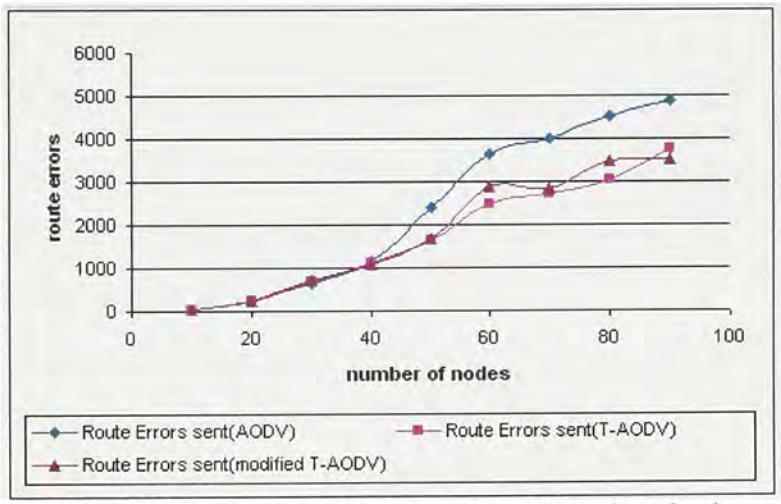

Figure 3.12 Comparison of route errors sent with number of nodes

Figure 3.13 compares the average end-to-end delay for AODV and modified T-AODV. The variation of the delay has been found to be random, as quite naturally expected because of the ad hoc nature of the network. However, the delay for modified T-AODV can be found to be in close proximity to that of the original AODV, which also establishes the efficiency of our trusted routing protocol.

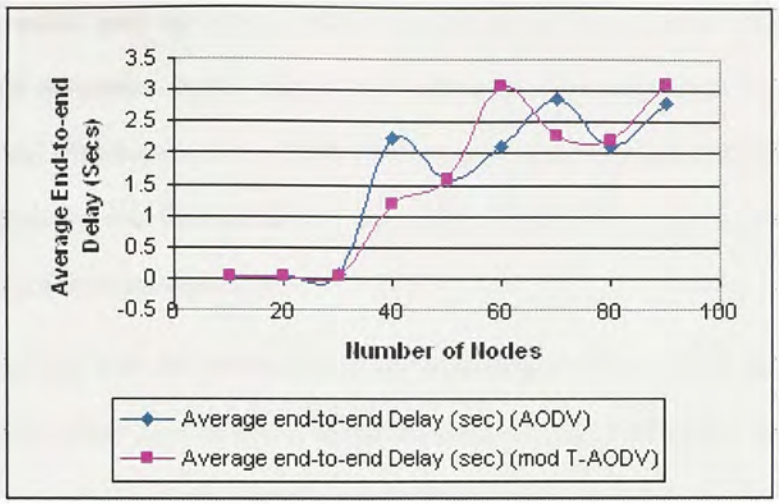

Figure 3.13 Comparison of average end-to-end delay

Figure 3.14 compares the throughput in bits/sec for the two protocols. We can see from the figure that the throughput for modified T-AODV is sometimes lower than that of AODV. This is because, for random waypoint mobility model, the speed is chosen from a range specified; and for higher speed the throughput for T-AODV is higher. This will be discussed in more details in the our next set of simulations.

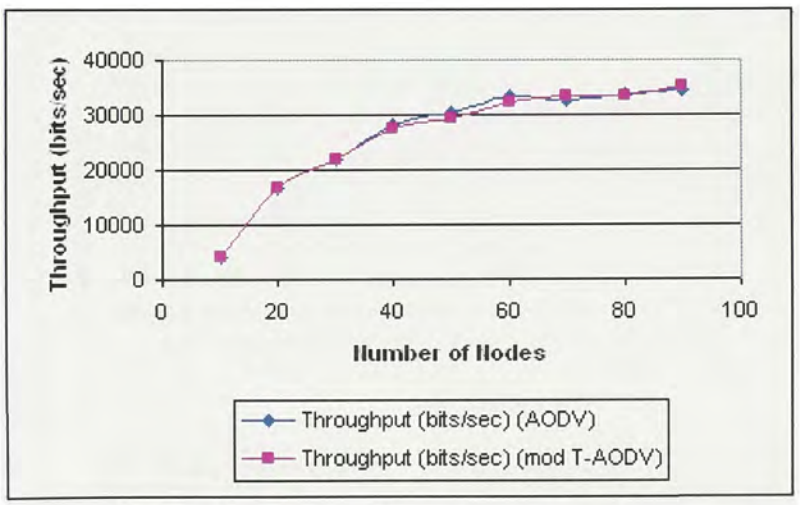

Figure 3.14 Comparison of throughput with number of nodes

The results from our second set of simulations are shown below. We have varied the speed of node movement from 5 meters/sec to 30 meters/sec and compared our modified T-AODV protocol with the original AODV. This range of speed has been selected to simulate the moderate human speed as well as that of a high speed vehicle. The range can give us a very good approximation of real time movement.

We can see from the comparison of routing overhead (Figure 3.15) that the modified T-AODV performs better than AODV with higher speed of node movement. The explanation for this is that, the topology of the network would change when the nodes move faster resulting in more route requests to be generated. In AODV more and more intermediate nodes come up with route replies, which increases the overhead. On the other hand, in modified T-AODV, only destination nodes come up with replies, hence the overhead is smaller. The small increase in overhead for modified T-AODV with lesser speed is because of the retransmission of some route request packets due to delayed receipt of route reply by the source nodes.

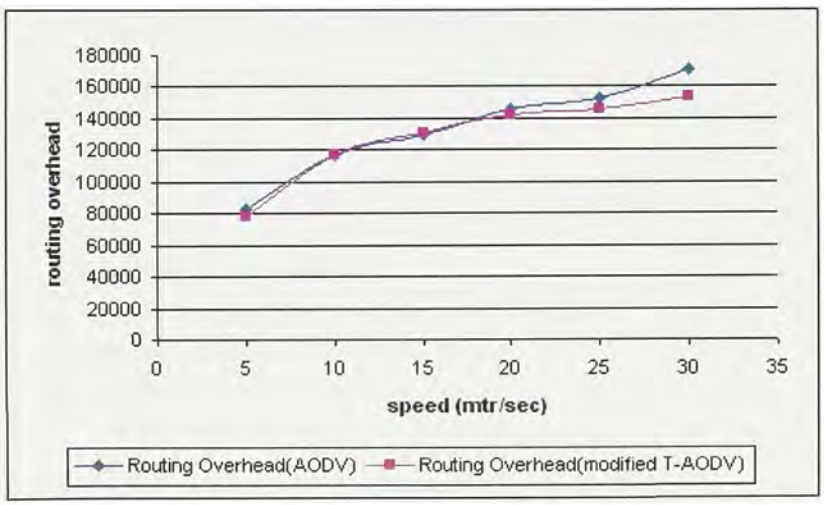

Figure 3.15 Comparison of routing overhead with node speed

Figure 3.16 and 3.17 respectively compare the number of routes selected and the number of route errors sent with node speed. We can see the efficiency of our protocol from these two parameters also. Significantly lesser number of routes selected by modified T-AODV at higher node speed imposes lesser processing overhead at the nodes. A significant reduction of route errors can also be observed for modified T-AODV at higher speed, which can be attributed to the lower MAC layer load as discussed before.

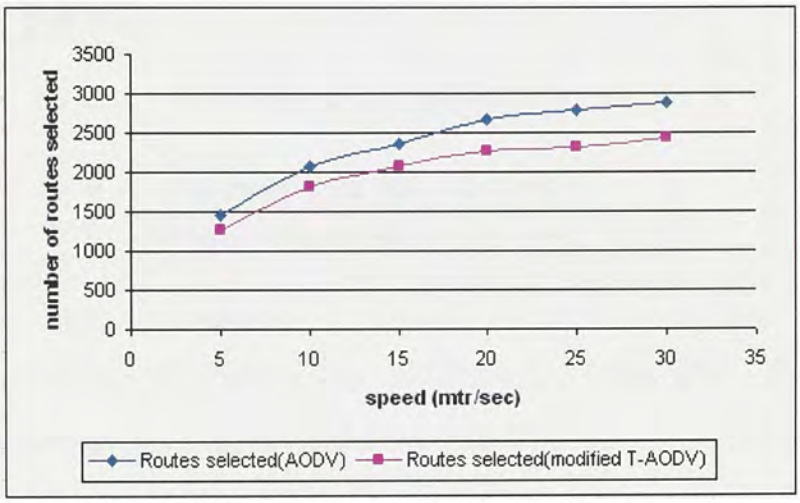

Figure 3.16 Comparison of routes selected with node speed

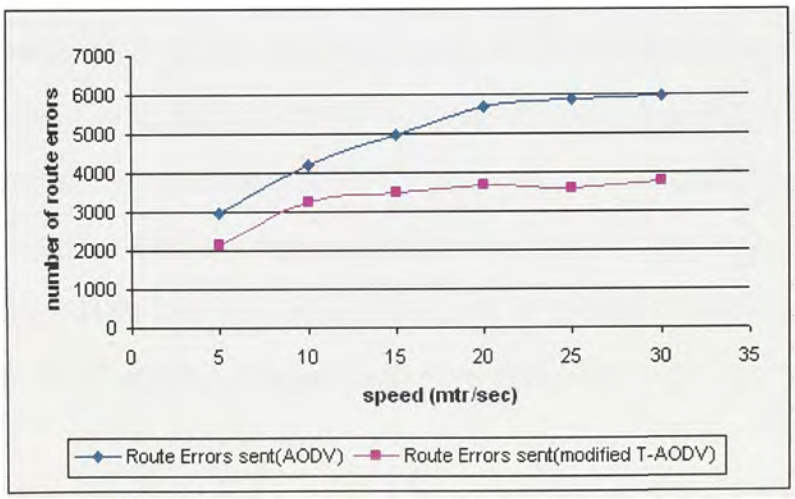

Figure 3.17 Comparison of route errors with node speed

Finally, figure 3.18 compares the average end-to-end delay with increasing node speed for the two routing protocols. Again, the delay has been found to be extremely random in nature, as predicted by the ad hoc characteristic of the network.

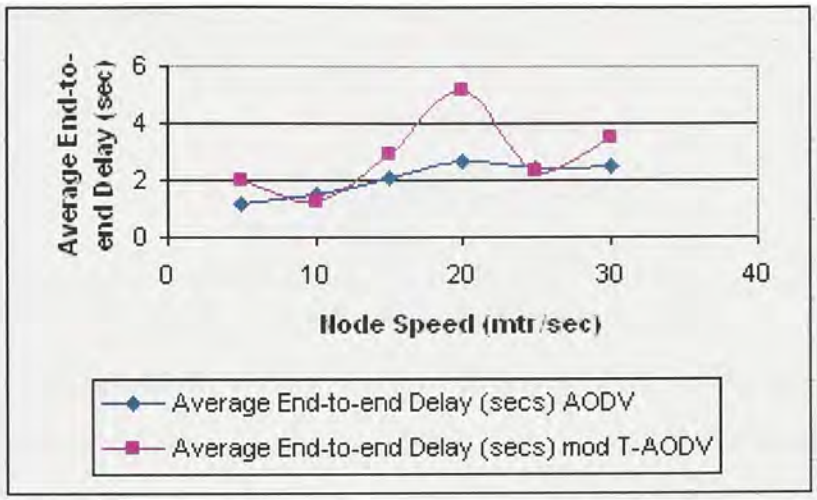

Figure 3.18 Comparison of average end-to-end delay with node speed

Figure 3.19 compares throughput for modified T-AODV and AODV. We can see from the figure that T-AODV has higher throughput than AODV for higher node speed, which can be explained as follows. With the increase in node speed, network topology changes faster and hence the probability of broken routes increases. This will result in more number of route request packets being generated. In AODV, more route replies will result which will increase the MAC load, as route replies, being unicast packets will require MAC layer handshaking in the form of RTS/CTS/Data/ACK. Thus throughput for AODV comes down with higher node speed. On the other hand, in modified T-AODV, lesser number of route reply messages are generated which will result in lower MAC layer load. Hence throughput for modified T-AODV decreases at a slower rate than that of AODV, resulting in higher throughput than AODV with increased node speed.

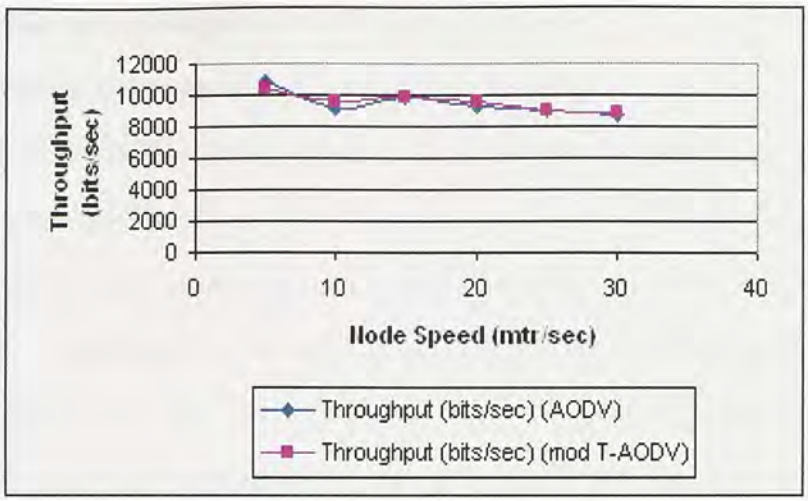

Figure 3.19 Comparison of throughput with node speed

To conclude briefly, we have also noted that the average running time for modified T-AODV is about 11 % higher than that of T-AODV. This increase in the running time is because of the incorporation of cryptographic operations into modified T-AODV protocol. In the following section, we analyze the security of the protocol by evaluating different threat scenarios.

3.6 Security Analysis

The security of our protocol lies in the verification of the information provided by other nodes. We evaluate different scenarios of attack by a malicious entity, acting either independently or in collusion, and show that the protocol is secure against these attacks.

Scenario 1: A malicious node wants to include itself into the path and provides wrong information in the RREQ packet – this attack has already been prevented while designing the T-AODV protocol. The malicious node is effectively isolated by the collaborative effort of its neighbors.

Scenario 2: A node falsely accuses another node and alters the information provided by the later – the accuser has to append the signature computed by the accused node. In order to alter the information, it has to decrypt the signature, change the original information and recompute it. But

it fails to recompute the signature as it lacks the knowledge of the accused node's Private key. Thus any attempt to alter the original information gets detected.

Scenario 3: A node falsely accuses another node, alters the information provided by the later and recomputes the signature with its own Private key – this malicious act gets detected, as the nodes receiving the warning messages cannot decrypt the signature using the accused node's Public key.

Scenario 4: A node falsely accuses another node and provides the signature of a different node other than the accused one – this act also gets detected, as the neighboring nodes receiving the warning messages cannot decrypt the signature using the accused node's Public key.

Scenario 5: A node whose trust level has changed, falsely accuses another node by using a copy of the old Signature field that the later used at some earlier point of time. This false accusation gets detected as the decryption of the signature will reveal the actual source address and broadcast ID pair.

We recognize that these scenarios are not exhaustive but at least demonstrate that the protocol is secure under these threats.

3.7 Conclusion

From the above discussion and analysis of the results obtained from extensive simulation, we can conclude that the secure routing solution developed by us in course of our research scales extremely well to both network size and mobility. It has been observed that the routing protocol performs even better than the original AODV routing protocol with increased mobility in the network. In the next chapter we are going to discuss the design of the trust computational model and carry out simulation analysis to prove its efficiency.

Chapter 4

Trust Modeling against Selfish and Malicious Behavior

Although it is extremely difficult to put forward a formal definition of trust, many people have tried to do so since past several years. Generally speaking, trust is looked upn as a belief in the honesty and truthfulness of an entity in carrying out certain protocols that have already been agreed upon mutually. A precise definition of trust is given in [Gam90] as "*...trust (or, symmetrically, distrust) is a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such action (or independently of his capacity ever to be able to monitor it) and in a context in which it affects his own action*" [Liu04]. This definition of trust has led to the concept of *confidence*[6] into the action of an entity. Hence, the terms *trust* and *confidence* have been linked with each other since the concepts evolved in our society.

Trust has been an important concept behind the civilized evolution of our society. A sociological interpretation of trust has been given in [Szt00] as follows: It is a "*clear and simple fact that, without trust, the everyday social life which we take for granted is simply not possible*". Most people would now agree that "*the existence of trust is an essential component of all enduring social relationships*".

This social interpretation of trust can be mapped exactly to the context of our research. Indeed the basic motivation of our work in developing a trut model has been evolved from this fundamental concept. In the following section, we talk about this motivation.

In chapter 3 we have designed a trusted routing solution that depends on the trust levels to compute end-to-end routes. The solution was developed under the assumption of static and pre

[6] Confidence has been defined to be the level of belief on an entity's action.

distributed trust levels among the nodes in the network. In reality, this would not work in many situations, as described below:

1. In a network deployed by the military in a war front there is a high possibility of nodes getting compromised, in which case the trust levels of the nodes should change. But this would not be reflected if static truts levels are used in the network;
2. In any commercial sensor networking application, a node may act selfish in forwarding other nodes' data to conserve its own power and eventually bring down the network performance. This selfish act must also be reflected in the trust computation which should dynamically compute the trust levels;
3. In multimedia applications for video and audio file sharing, nodes sometimes show selfish behavior for which they are accused by others. This also changes the trust levels of the nodes dynamically.

Hence, realistically speaking, the trust levels used in route computation should be dynamically computed based on the behavior of the nodes. In this chapter we develop a framework for computing, distributing and updating trust in an ad hoc network application. Modeling and computing trusts in such an application has been a challenging problem since the concept of trust has been extended for infrastructureless scenario. It is very difficult to form a true and honest opinion about the trustworthiness of the nodes, as they can be engaged in malicious activities in different ways. This intricacy in trust computation, together with frequent topology changes among nodes, quite often causes the whole network to get compromised or disrupted. Different malicious activities of the nodes can very well be misinterpreted as the regular erratic behavior of the wireless networks in general and ad hoc networks in particular, thus making trust computation all the more difficult.

As we have already discussed in chapter 2, most of the trust computations proposed for ad hoc networks talk about the general requirement of trust establishment [Buc02c, Ver01, Esc02,

Kag01], but do not come up with any specific model or computational framework to do so. Some researchers have proposed to use trust tables populated by different parameters [Pir04a, Yan03] collected by the nodes in promiscuous mode. However, none of the models proposed so far have tried to understand and analyze different malicious behavior of the attacker and quantify those behaviors in a policy-based computational framework. Some research has been done to prevent selfish behavior in ad hoc networks by using either a reputation-based incentive mechanism [Buc02b, He04, Mic02], or a price-based incentive mechanism [But02]. In both the mechanisms, nodes are given incentives to suppress their malicious intention in favor of the network. But nodes with malicious intention at their subconscious self always try to find ways to bypass these incentive mechanisms.

Keeping in view the above facts, our objective is to design an efficient trust management system that takes into account different malicious behavior of the nodes in trying to disrupt the network operation. The trust computational framework should address the following issues:

1. Incorporation of different malicious behavior and quantifying them in the model;
2. Isolation of a non-trusted entity with the collaborative effort from all its neighbors;
3. Formation of true opinion about malicious entities colluding together to disrupt the network.

This chapter focuses on the design of the trust computation model that we have developed to integrate with the trusted routing protocol discussed in the previous chapter. A simulated analysis of the model has been carried out to prove the efficiency and scalability of the model.

4.1 Trust Issues in Infrastructureless Networks

As we have already discussed, our motivation for developing the trust model is to form a true and honest impression about the trustworthiness of the nodes and to punish the nodes with

the slightest malicious intention. To do this we need to understand clearly the ways a node can engage itself in different malicious acts. Below we highlight three different malicious behavior:

1. A node engaging in selfish behavior by not forwarding packets meant for other nodes, or selectively forwarding smaller packets while discarding larger ones;
2. A node falsely accusing another node for not forwarding its packets, thus isolating the node from normal network operation;
3. A node placing itself in active route and then coming out to break the route, thus forcing more route request packets to be injected into the network. By repeating this malicious act, a large number of routing overhead is forcefully generated wasting valuable bandwidth and disrupting normal network operation.

Lets go back to the example of the medical board in chapter 1. We concluded from that example that a trusted route establishment is of utmost importance where confidential data dissemination is in use. Even if a trusted route is established, some nodes may act selfish by not forwarding other nodes' data or even selectively forwarding them. This will slow down the performance of the network, eventually defeating the purpose of setting up the medical board. In forming a true opinion about other nodes' trustworthiness, it is essential to have a trust model in place by which nodes can compute the levels of trust they can have on others. Some nodes may act malicious in accusing other nodes falsely and thus be a good guy in the eyes of others. Even some nodes, after putting themselves in the active routes, may break the route frequently, thus forcing the injection of more control packets in the network.

All the malicious acts discussed above, if undetected, bring down the network performance. This has been found from simulation analysis where we implemented the malicious behaviors in the AODV routing protocol. We have used Glomosim [Zen98] and incorporated the malicious behaviors in the network layer. We selected a network size of 50 nodes and increased the number of malicious nodes from 10 % to 50 % exhibiting the malicious behaviors. The

network size has been kept to a moderate level to remain at par with any real-life ad hoc implementation. It has been found that theree is a substantial degradation of the (packets received / packets sent) ratio with the increase in the number of malicious nodes (Figure 4.1).

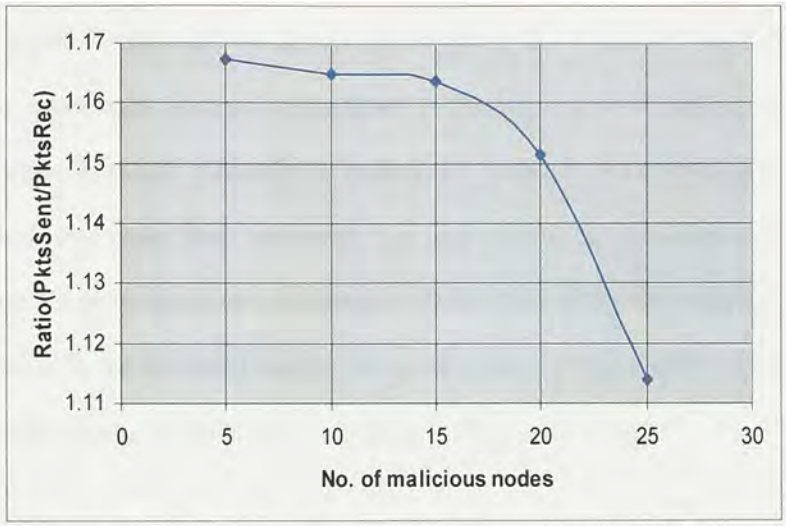

Figure 4.1 Variation of (Packets received / Packets sent) with malicious nodes

A more severe degradation has been found in the network throughput as the number of malicious nodes increases. We have found that the network throughput decreases by 28% as the malicious nodes increase from 10 % to 50 %, whereas the throughput of the honest nodes decreases by 61% with the similar increase in malicious nodes.

In view of the above, we can argue that there is a need to develop a trust computational model that not only depends upon the history of the nodes' behavior, but also is able to analyze different froms of malicious behavior and quantify them in the model itself.

The rest of the chapter is organized as follows. In the next sections we will discuss the trust model that has been built up based on different malicious behaviors alongwith the assumptions and parameters used towards developing the model. Next, we will talk about the simulation setup and analyze the results. The final section will conclude the chapter.

4.2 Design of the Trust Model

Our model has been developed with a view to form a true and honest opinion about the trustworthiness of the nodes with collaborative effort from their neighbors. Our trust model is not transitive, i.e., we do not consider the notion "if A trusts B and B trusts C, then A trusts C". This trust transitivity encourages more colluding attack in the network from multiple malicious nodes. For example, when T-AODV [Gho04b] is used in the network, them more than one malicious nodes can collude and make their combined trust high enough to put them in the active route. Instead, our trust model is based on collaborative effort of all the nodes and analysis of different malicious behavior. In the following section we analyze some of the possible malicious behaviors exhibited by the nodes and quantify them to gradually develop the model.

4.2.1 Assumptions

While designing the trust model, we have made the following assumptions which are realistic. First, all the nodes communicate via a shared wireless channel and all communication channels are bi-directional. This has been the fundamental assumption while designing any standard MAC layer protocol for wireless networks. Second, all the nodes operate in a promiscuous mode. Third, our main focus is on the network layer and we have incorporated trust computations in the network layer to avoid any unwanted inter-layer cross functioning. We have assumed a reliable link layer protocol to be in place. The above assumptions are very fundamental and used for all the solutions proposed so far. Finally, we do not encourage the notion of trust transitivity, i.e., 'if A trusts B and B trusts C, then A also trusts C'. This trust transitivity encourages more colluding attack in the network from multiple malicious nodes. For example, when our trusted routing solution [Gho04b] is in use in the network, then more than one malicious nodes can collude and make their combined trust high enough to put them in the active

route. Instead, our trust model is based on collaborative effort of all the nodes and analysis of different malicious behavior.

4.2.2 Trust Model Against Selfish Behavior

The development of the model to punish a node for selfish behavior is based on the Secure and Objective Reputation-based Incentive (SORI) scheme proposed in [He04] with several modifications. We will elaborate more on these modifications as we describe the trust model.

4.2.2.1 Parameters used in the Model

Below we describe the parameters used in our trust computational model. We have represented nodes by N, X and i. A detailed description can be found in [He04].

(i) NNL_N = Neighbor Node List (each node maintains a list of its neighbors, either by receiving *Hello* messages, or by learning from overhearing).

(ii) $RF_N(X)$(Request for Forwarding) = total number of packets node N has forwarded to node X for further forwarding.

(iii) $HF_N(X)$(Has Forwarded) = total number of packets that have been forwarded by X and noticed by N.

(iv) $LER_N(X)$= Local Evaluation Record of node N of node X. It reflects the evaluation of the behavior of node X by another node N.

(v) $G_N(X)$= Forwarding ratio of node N on node X.

(vi) $C_N(X)$= Confidence level of N on X.

(vii) $OER_N(X)$= Overall Evaluation Record of node N on node X. It is the overall evaluation of a node on another node based on its own local evaluation and collaborative evaluation from its neighbors.

4.2.2.2 Model Formulation

With the above parameters, node N can create a *local evaluation record* (denoted by $LER_N(X)$) about X. The record $LER_N(X)$ consists of two parameters shown below:

$$LER_N(X) = \{G_N(X), C_N(X)\}$$

where

$G_N(X)$ is the forwarding ratio given by $G_N(X) = (HF_N(X) / RF_N(X))$

$C_N(X)$ is the confidence level of N on X

In [He04] the authors have set $C_N(X) = RF_N(X)$. This gives quite an accurate estimation about the trustworthiness of a node when weighted by the confidence level. But the trust computation does not take into account a node's "selective forwarding" behavior, where it only forwards small packets while selectively discarding larger ones. To reflect this kind of malicious behavior in our trust model, we compute the confidence level $C_N(X)$ as shown in equation (5).

$$C_N(X) = \frac{\sum_i (HF_N(X)_i / RF_N(X)_i) * (Pkt\_size)_i}{\sum_i (Pkt\_size)_i} \quad \text{----------} \quad (5)$$

Node N computes its confidence level on X after sending a specified number of packets to X. The computation is weighted by the packet size to reflect the "selective forwarding" behavior of a node.

We propose a similar propagation model proposed in SORI [He04]. Each node updates its local evaluation record (*LER*) and sends it to its neighbors. When a node N receives the $LER_i(X)$ from node i, it computes the *overall evaluation record* of X (denoted by $OER_N(X)$), as shown in equation (6).

$$OER_N(X) = \frac{\sum_{i \in NNL, i \neq X} C_N(i) * C_i(X) * G_i(X)}{\sum_{i \in NNL, i \neq X} C_N(i) * C_i(X)} \quad \text{----------} \quad (6)$$

where

$C_N(i)$ = confidence level of node N on node i from which it receives $LER_i(X)$

$C_i(X)$ = confidence level of node i on node X

$G_i(X)$ = forwarding ratio of node i on X

4.2.3 Trust Model Against Malicious Accuser

A malicious accuser is defined to be a node falsely accusing another node for non-cooperation. We foresee a threat where a node falsely accuses another node of not forwarding its packets, eventually to isolate the later as an untrustworthy one. This malicious act should also be reflected in the trust computation, where every node should be given a chance to defend itself. We extend the trust model against selfish behavior developed in section 4.2.1.2 to take into account the malicious accusation of a node about another node.We have modified equation (5) to

reflect such a malicious act in the computation of the confidence level. The modified equation is shown below:

$$C_N(X) = \frac{\sum_i (HF_N(X)_i / RF_N(X)_i) * (Pkt\_size)_i}{\sum_i (Pkt\_size)_i} * \alpha_X(N) \quad \text{----------} \quad (7)$$

where

$$\alpha_N(X) = \textit{accusation index} \text{ of N by X} = \begin{cases} 0 & \text{if X falsely accuses N} \\ 1 & \text{otherwise} \end{cases}$$

Node N keeps a track of the packets it received from X and packets it forwarded. If N finds out that X is falsely accusing it for non-cooperation, it recomputes its confidence level on X by taking into account the accusation index. It then broadcasts the new $LER_N(X)$ with new $C_N(X)$, thus resulting in computation of a new $OER_N(X)$, which is low enough to punish X. Thus, any sort of malicious behavior of X by falsely accusing other nodes gets punished eventually.

4.2.4 Trust Model Against Malicious Topology Change

Malicious topology changes are carried out deliberately by malicious nodes in order to inject more control packets in the network with a view to slow down the network operation. This is a special type of Denial of Service (DoS) attack by which network resources are wasted. In this section our proposed model is extended to reflect this malicious behavior of a node. If such a behavior is detected, the confidence level must be changed in order to punish the malicious node. However, detection of such a behavior is not easy, as any such topology change can be viewed as

a normal characteristic of an ad hoc network. We have tried to capture such a malicious act by statistically modeling the action and reflecting it in the computation of trust.

To develop the model, we require each node to maintain a table called a *neighbor remove table*, where it keeps track of any node moving out of the path. The table is populated by successive *Hello* misses in AODV, or from the *unreachable node address* field in the route error packet in DSR. In AODV each node periodically broadcasts *Hello* messages to its neighbors to ensure connectivity. If successive Hello messages are missed, a node is removed from the neighbor table and an entry is made in the neighbor remove table. On the other hand, if DSR is used, the neighbor remove table can be updated from the *unreachable node address* in the route error message. Each route error message carries the the address of the node that is unreachable. This address is entered in the neighbor remove table for further action.

A snapshot of the *neighbor remove table* is shown below:

Node Address	Time of Leaving	Time Difference
X	T1	t0 = 0
X	T2	t1 = T2 – T1
X	T3	t2 = T3 – T2
X	T4	T3 = T4 – T3
		Mean = μ_t

Table 4.1 Snapshot of Neighbor Remove Table

Each node periodically scans the table to find whether any particular node is leaving at frequent intervals. It computes the mean, μ_t of the time difference of any particular node leaving the network. If μ_t is found lower than a threshold value (denoted by $t_{threshold}$), then the node is identified as malicious and the confidence level is computed as follows:

$$C_N(X) = \frac{\sum_i (HF_N(X)_i / RF_N(X)_i) * (Pkt\_size)_i}{\sum_i (Pkt\_size)_i} * m(X) \quad \text{----------} \quad (8)$$

where

$m(X)$ = malicious index of node X = $\begin{cases} 0 & \text{if } \mu_t <= t_{threshold} \\ 1 & \text{if } \mu_t > t_{threshold} \end{cases}$

The choice of the threshold value can be selected based on the typical application for which the ad hoc network is deployed. A network that demands frequent topology change can have a higher threshold to accommodate the normal network behavior. An example can be a typical application in the military where the deployed network demands frequent movement of the nodes. A similar application can be thought of in an emergency relief operation. On the other hand, a network deployed for medical boards does not demand frequent node movements, and hence can be characterized by a lower threshold value.

Finally, to combine all the malicious behavior discussed earlier and to reflect those behavior in trust computation, the confidence level of node N on X is computed as shown below:

$$C_N(X) = \frac{\sum_i (HF_N(X)_i / RF_N(X)_i) * (Pkt\_size)_i}{\sum_i (Pkt\_size)_i} * \alpha_X(N) * m(X) \quad \text{----------} \quad (9)$$

The final overall evaluation record (*OER*), when computed based on the local LERs, will reflect the different malicious behavior of a node as computed in the confidence level, and finally any malicious act gets detected and punished.

4.3 Simulation Setup and Analysis of Results

Our simulation setup for the trust computational model has been designed in a similar fashion as that designed for T-AODV. The mobility model selected for the simulation is random waypoint mobility. In this model a node randomly selects a destination from the physical terrain. It moves towards the chosen destination with a speed uniformly chosen between a minimum and a maximum speed limit. After reaching the destination, the node stays there for a certain pause time before it selects another destination and starts moving in that direction.

We have chosen two types of traffic for our simulation - CBR (Constant Bit Rate) and FTP (File Transfer Protocol). For each CBR traffic we have used 10000 packets each of length 512 bytes. In FTP tcplib has been used to simulate the file transfer protocol. In each FTP traffic we have used either 10 or 5 items to be sent to the destination node.

We have evaluated the network performance with both increasing network size and mobility. We have selected node speed from 0 to 10 meters/second with 30 seconds pause between each successive movement. We increased the number of nodes from 20 to 100 and studied the network performance with increasing network size. To evaluate the performance of the protocol with increasing mobility, we have increased the node speed from 10 to 60 meters/second. This range of speed can give us a very good approximation of real time movement and has been selected to simulate the moderate human speed as well as that of a high speed vehicle. The parameters are shown in the Table 4.2.

We have incorporated trust computation directly into the routing protocol to avoid any unnecessary layering interoperability. We have extended the Ad Hoc On-Demand Distance Vector (AODV) routing protocol [Per99] to incorporate the trust computation and exchange. The modified protocol has been benchmarked with AODV to study its scalability and efficiency. To

avoid any unwanted overhead we have ensured the trust information exchange to be piggybacked with the route request packet header.

Scenario 1	Independent variable	Set of parameters compared		
	Number of nodes	Routing overhead	Number of routes selected	Number of route errors
Scenario 2	Independent variable	Set of parameters compared		
	Node speed	Routing overhead	Number of routes selected	Number of route errors

Table 4.2 Parameters Chosen For Simulation

Figures 4.2 to 4.10 show the comparison of the base AODV protocol and our modified protocol with the incorporation of trust computations. From Figure 4.2 we can see that our protocol scales as good as the original AODV with increasing number of nodes. Even though we have incorporated extensive trust computation at each node both by its own spying mechanism as well as by exchanging information from its neighbors, we can see that our protocol does not add any significant overhead.

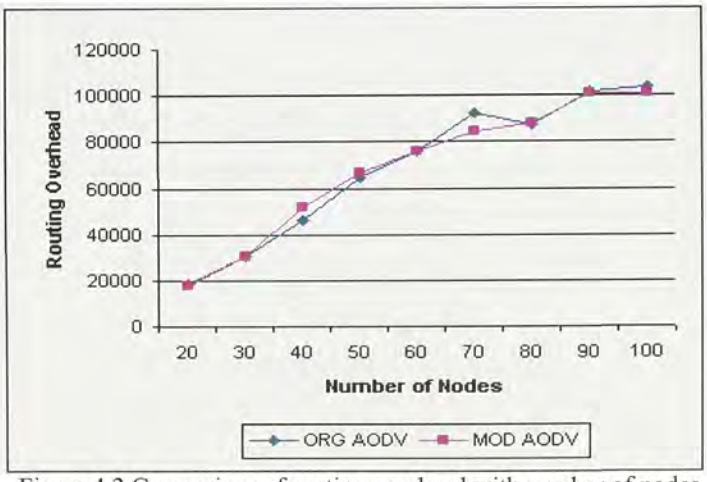

Figure 4.2 Comparison of routing overhead with number of nodes

Similar results can be seen from Figures 4.3 and 4.4 where we have benchmarked our modified protocol with AODV in terms of routes selected and route errors sent. In both the cases we can see that the modified protocol scales as good as AODV even with large network size. Number of routes selected and route errors are dependent on several factors like localized clustering of the nodes, MAC layer load and also routing and transport layer load. The parameters show random variation as quite expected from the ad hoc nature of the whole network. However, both the protocols show very similar variation in the characteristics.

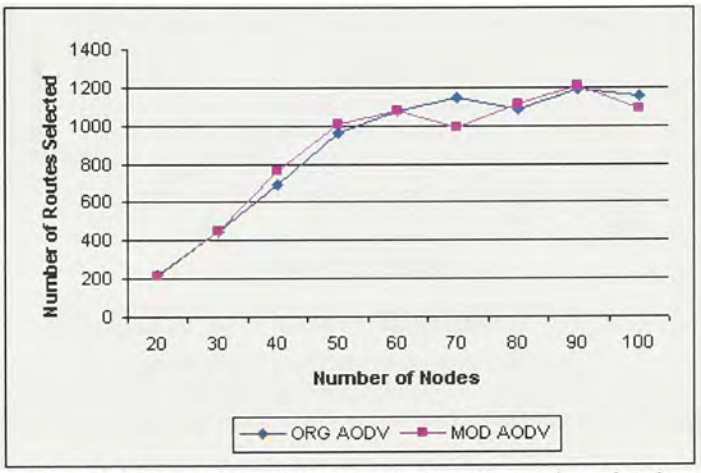

Figure 4.3 Comparison of routes selected with number of nodes

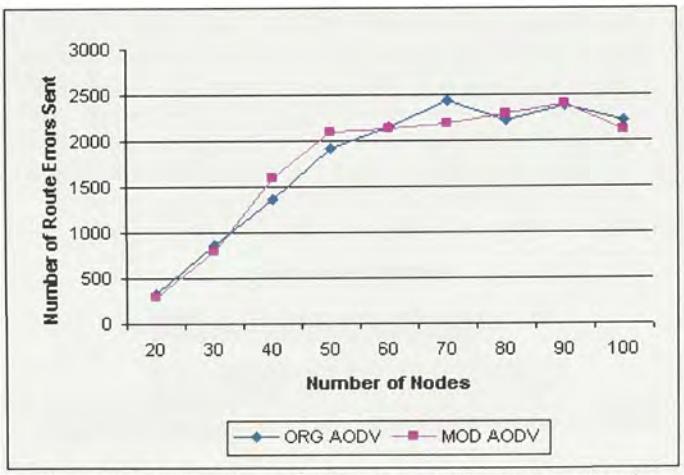

Figure 4.4 Comparison of route errors with number of nodes

Figures 4.5 and 4.6 compare the average end-to-end delay (in seconds) and throughput (in bits per second) respectively for the base AODV and the modified protocol. It can be concluded from the results that the modified protocol scales as good as the original one with respect to these parameters as well. These parameters also depend upon the localized clustering of the ad hoc nodes and overall network load including MAC layer, network layer and transport layer loads. Hence these parameters also show random variation for the two protocols.

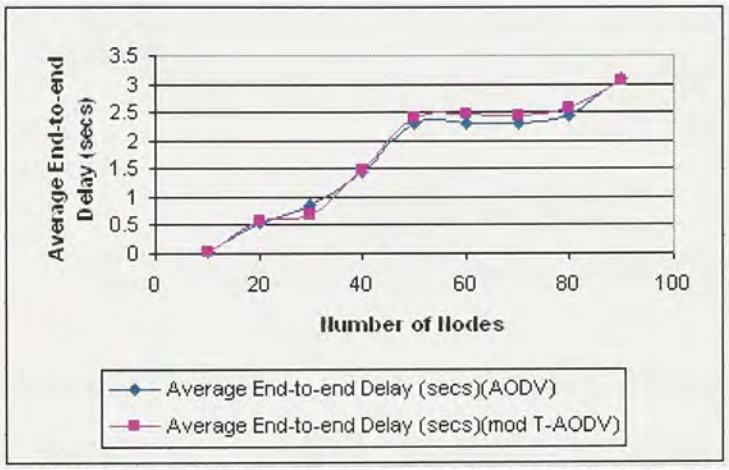

Figure 4.5 Comparison of average end-to-end delay with number of nodes

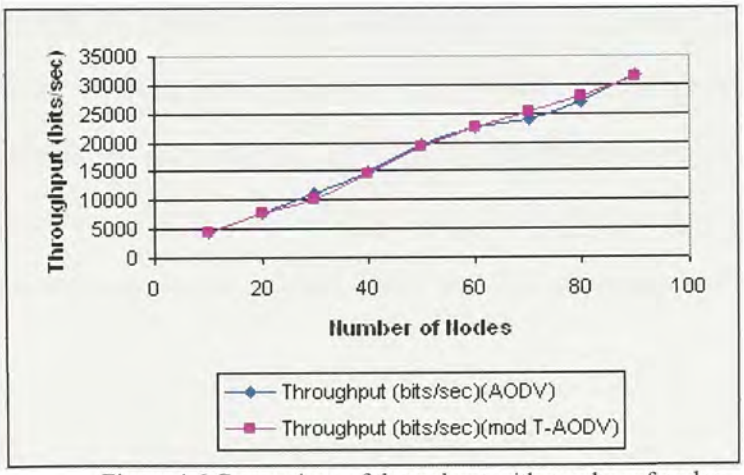

Figure 4.6 Comparison of throughput with number of nodes

Our next set of simulation is to evaluate the modified protocol with increasing node speed. This parameter has been selected to see the protocol scalability and efficiency with

frequent changes in network topology. We can see from Figure 4.7 that our modified protocol does not add any overhead, even with higher node movement.

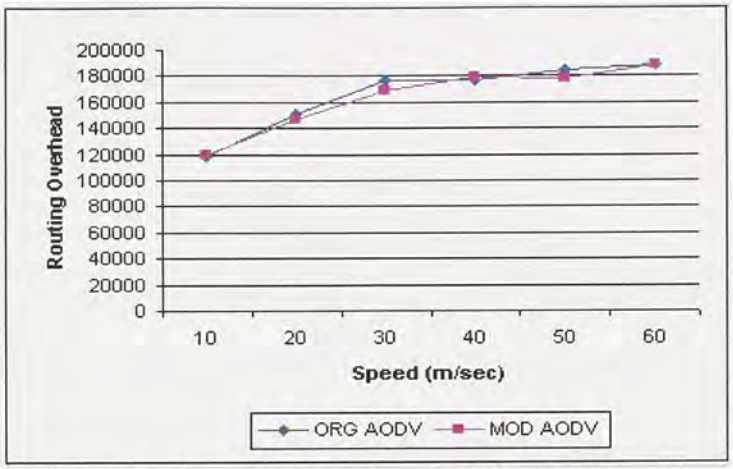

Figure 4.7 Comparison of routing overhead with node speed

As we have piggybacked the confidence information into the route request messages to control routing overhead, we can conclude that mobility will help in updating trust and confidence information in our modified protocol. As the topology of the network changes more frequently necessitating more and more route request packets to be generated, more recent information about the trusts are circulated in the network. Thus, we can conclude that our modified protocol is not only efficient and scalable with network size and node speed, it also gives a better picture of trust and confidence with higher node speed. Figures 4.8 and 4.9 conclude in a similar way that the protocol scales very well in terms of routes selected and route errors sent.

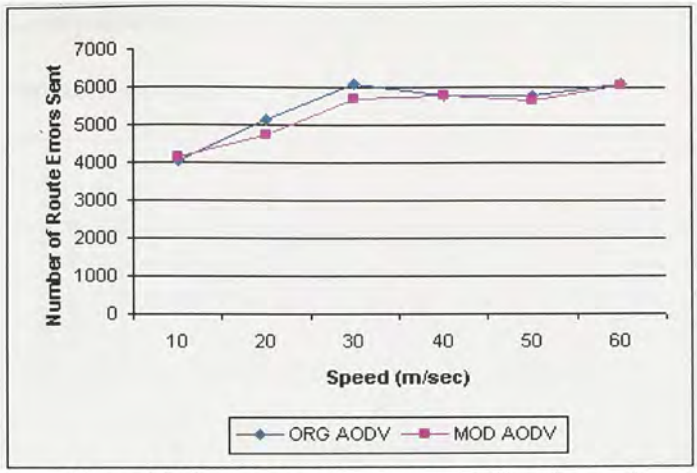

Figure 4.8 Comparison of route errors with node speed

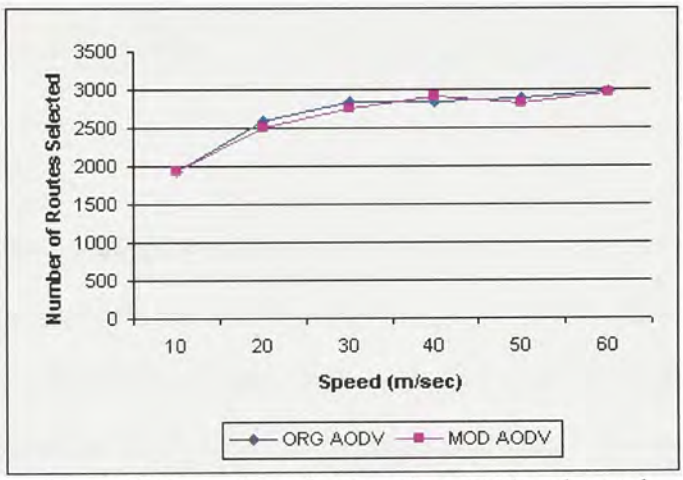

Figure 4.9 Comparison of routes selected with node speed

Figure 4.10 compares the average end-to-end delay (in seconds) for the base AODV and the modified protocol. We can see that the modified protocol scales as good as the original AODV with increasing node speed with respect to the delay.

As we can see from Figures 4.7 to 4.10, the parameters for the modified protocol vary randomly with comparison to the base AODV with sometimes lower and sometimes higher values. This is attributed mainly to the ad hoc nature of the network with random waypoint mobility model. The parameters are dependent upon factors like localized node clustering, MAC layer load and also transport and network layer load, as we have discussed previously. These

factors change with every simulation run with random waypoint mobility, which attributes to the somewhat random variation between the two protocols.

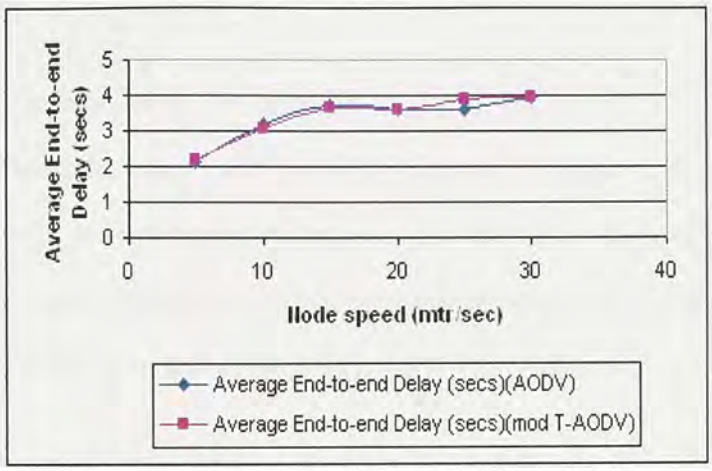

Figure 4.10 Comparison of Average End-to-end Delay

4.4 Conclusion

From the above analysis, we can conclude that the trust model developed by us scales very well to both network size and mobility, even though we have incorporated extensive trust computations in the routing protocol itself. In the next chapter we will conclude our thesis with an emphasis on the possible drawbacks of our designed models and future extension that need to be carried out.

Chapter 5

Conclusion

This dissertation makes two important contributions towards designing a secure communication system for infrastructureless networks. The designs and formulations used can have significant contributions for securing next generation ad hoc or sensor networks. The results obtained can form a solid foundation for future developmental work. This chapter summarizes our main contribution and presents related problems and future extension.

5.1 Trusted Routing Protocol

The quest for finding a secure communication infrastructure led us to design a trusted routing protocol, which we call Trust-embedded AODV (T-AODV) that relies on the collaborative effort of all the nodes to find a trusted end-to-end path. The protocol is capable of securing the network from an active internal attack, be it from malicious node acting independently or in collusion. The working of the protocol depends upon the collaborative effort from all the nodes and any malicious node trying to inject false information gets isolated and punished. This is one of the first major contributions towards designing a secure routing scheme against colluding attack where more than one malicious nodes collude with one another to attack the network. A concept of trust level is introduced which plays a critical role in the final route selection. Our extensive simulation shows that our protocol is efficient as well as scalable with network size and node speed. A case by case threat analysis has also been carried out to show that the protocol is secure against most types of attacks.

5.2 Trust Computational Model

We have also designed a trust computational model and integrated it with the routing protocol. The trust model is unique in the sense that it analyzes different malicious behavior and quantifies those behavior in the model itself. This is one of the first efforts given to formulate trusts by analyzing different forms of malicious behavior. The model can lay down a solid foundation for carrying out future research on trust computations in infrastructureless scenarios. We have carried out extensive simulation to show the efficiency and scalability of our protocol with both network size and mobility.

5.3 Future Direction

Although our work is extensive and forms a solid basis for developing a secure communication infrastructure in multihop ad hoc networks, we believe that there is much more to be done in this area. Up to now, the promiscuous operation of the nodes was assumed to be power efficient. Although this was not the primary focus of our research plan at the time, we realize the need to address the issue of making our protocol power-efficient. In particular, when each node works as a router to help deliver packets to the destination, relaying packets to others can result in the device expending its own energy. In addition, since the nodes are working in a "promiscuous" mode, the amount of energy consumed in this case can be quite significant. Hence, a mobile node should examine its own "well-being" before committing to forwarding packets or sniffing on behalf of the others. Such a limitation in energy supply implies the need for developing power control schemes to prolong the battery life. Some power control techniques are geared at reducing the amount of interference between devices and, therefore, the number of power-consuming retransmissions. Security can also be enhanced by proper power control. To preserve a low probability of being intercepted and being detected (a major issue in military applications), ad hoc nodes have to be controlled so that they transmit as little power as necessary, hence significantly

reducing the probability of being intercepted or being detected. In view of this, we need to develop an innovative framework which takes into consideration power aware secure routing.

The power consideration is more pronounced in sensor network applications, where the nodes are characterized by their power constraints. So far the routing protocols designed for sensor applications are not matured enough to incorporate security and power together into their design. Although the design will be very much application specific, the basic need for trusted routing will be there for multihop applications. However, in broadcast networks, designing reliable MAC protocols will suffice. To extend our trusted routing protocol in sensor applications, we have to give serious considerations to power-constrained design.

Our current "collaborative trust-based secure routing protocol" does not consider the probability of MAC layer collisions. Nodes forming an ad hoc network typically have single antenna to transmit and receive. Thus, it is impossible for the nodes to transmit data, and at the same time, sense the network. This limitation could result in a higher probability of collisions. One possible solution is to use RTS/CTS handshaking to avoid a MAC layer collision. An obvious disadvantage to this approach is the significant increase in overhead. Although we have not considered the probability of MAC layer collisions in our existing solution, the trust interpretation of a node changes when some packets are lost. Even though our existing protocol has a network layer security mechanism, the sheer possibility of losing packets due to layer two collisions makes our protocol ineffective. Therefore, part of our research plan is to consider this issue when designing a secure solution which not only has multiple levels of security, but also a reliable MAC transmission mechanism.

Our solution also finds a secure end-to-end path without considering the reliability of that path. Much work in the field of routing are based on cluster-based solutions [Kri95], where clusters are defined on the basis of mobility, energy and degree parameters. This would allow adopting more reliable solutions other than a flat P2P approach. A still unanswered question is

"what would be preferable, a non-secure reliable path or a secure unreliable one?" Our solution would be the incorporation of path reliability along with security. This will give us a robust and reliable path discovery protocol.

The trust computational model developed by us, although unique in its approach to take different malicious behavior into account, is not exhaustive. With the growing popularity of ubiquitous computing and more and more applications of infrastructureless networking, a wide variety of threats and attacks will be in vogue. To defend the network from those attacks, the trust model has to be updated and designed in accordance to these new threats. The ultimate goal will be to develop a policy-based autonomous trust computational system that will be capable of protecting the network from various threats. In this respect, it can be discussed that a new way of looking at the trust modeling is to develop it from a Markov chain point of view, where the decision to have trust on a node will only depend on the current state of trust on that node and not on the history of its behavior. The motivation for this approach lies in the fact that any infrastructureless application can have a very high probability of instantaneous node compromise, and hence the history of its behavior should carry no impression in deciding the next course of action. However, this thought is only at its preliminary stage, and a more matured approach needs to be designed based on this model.

Our work can also lay a solid foundation for developing a policy-based security management paradigm. The overall goal will be to design and develop rules based on different malicious behavior and threats which will act as a basis for developing policies to secure the network from external and internal attacks.

Finally in the previous section, we have considered security only at the network layer. This can be further extended to incorporate multiple levels of security to make our design strong and robust. We strongly believe that a cross layer approach needs to be taken to design a secure

communication infrastructure which incorporates security into multiple layers and provide a seamless communication between the layers to make the network robust and secure.

Bibliography

[Abd97] Alfarez Abdul-Rahman & Stephen Hailes, "A Distributed Trust Model", ACM New Security Paradigm Workshop, 1997.

[Abe01] Karl Aberer, Zoran Despotovic, "Managing Trust in a Peer-2-Peer Information System", in *Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM '01)*, Atlanta, Georgia, USA, November 5-10, 2001.

[Alb02] Patrick Albers *et. al.*, "Security in Ad Hoc Networks: a General Intrusion Detection Architecture Enhancing Trust Based Approaches", *Wireless Information Systems*, Ciudad Real, Spain, 2002.

[Bec98] Becker and Wille, "Communication complexity of group key distribution", *Proceedings of the 5th ACM conference on Computer and communications security* San Francisco, California, United States Pages: 1 – 6, 1998, ISBN:1-58113-007-4.

[Bet94] Thomas Beth, Malte Borcherding, Bitgit Klein, "Valuation of Trust in Open Networks", *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 1994, Brighton, UK, pp.3-18, LNCS 875, Springer-Verlag.

[Bla96] Matt Blaze, Joan Feigenbaum, Jack Lacy, "Decentralized Trust Management", in *Proc. IEEE Conference on Security and Privacy*, Oakland, CA, May 1996.

[Blu83] Manuel Blum, "How to Exchange (Secret) Keys", *ACM Transactions on Computer Systems*, vol. 1, no. 2, pp. 175-193, May 1983.

[Buc02a] Sonja Buchegger, Jean-Yves Le Boudec, "The Effect of Rumor Spreading in Reputation Systems for Mobile Ad-hoc Networks", in *Proceedings of WiOpt '03, Modeling and Optimization in Mobile Ad Hoc and Wireless Networks*, Sophia-Antipolis, France, March 2003.

[Buc02b] Sonja Buchegger and Jean-Yves Le Boudec, "Performance Analysis of the CONFIDANT Protocol (Cooperation Of Nodes: Fairness In Dynamic Ad-hoc Networks)", in *Proceedings of the Third ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC '02)*, Switzerland, June 9-11, 2002.

[Buc02c] Sonja Buchegger, Jean-Yves Le Boudec, "Nodes Bearing Grudges: Towards Routing Security, Fairness, and Robustness in Mobile Ad Hoc Networks", in *Proceedings of the Tenth Euromicro Workshop on Parallel, Distributed, Network-based Processing*, pages 403-410, Canary Islands, Spain, January 2002.

[Bur95] M.V.D. Burmester and Y. Desmedt, "A Secure and Efficient Conference Key Distribution System", in *A.D.Santis, editor, Advances in Cryptology – EUROCRYPT '94*, volume 950 of Lecture Notes in Computer Science, pp. 275-286, Springer-Verlag, 1995.

[But04] Levente Buttyán and István Vajda, "Towards Provable Security for Ad Hoc Routing Protocols", in *Proceedings of the ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN '04)*, Washington, DC, USA, October 25, 2004.

[But02] Levente Buttyán and Jean-Pierre Hubaux, "Stimulating Cooperation in Self-Organizing Mobile Ad Hoc Networks", *Mobile Networks and Applications(MONET) Journals of Mobile Networks,* 2002.

[Cap03] Srdjan Čapkun, Jean-Pierre Hubaux, Levente Buttyán, "Mobility Helps Security in Ad Hoc Networks", in *Proceedings of the Fourth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc '03)*, Annapolis, Maryland, USA, June 1-3.

[Car03] Marco Carbone, Mogens Nielsen, Vladimiro Sassone, "A Formal Model of Trust in Dynamic Networks", *Basic Research in Computer Dscience (BRICS) Report RS-03-4*, 2003.

[Das00] Samir R. Das, Charles E. Perkins, Elizabeth M. Royer, Mahesh K. Marina, "Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks", in *Proceedings of IEEE Infocom*, 2000.

[Dav04] Carlton R. Davis, "A Localized Trust Management Scheme for Ad Hoc Networks", in *Proceedings of the 3rd International Conference on Networking (ICN '04)*, March 2004.

[Den02] Hongmei Deng, Wei Li and Dharma P. Agrawal, "Routing Security in Wireless Ad Hoc Networks", *IEEE Communications Magazine*, vol. 40, issue 10, pp. 70-75, October 2002.

[Des97] Yvo Desmedt, "Some Recent Research Aspects of Threshold Cryptography", *Proceedings of the First International Workshop on Information Security*, pp. 158 – 173, 1997, ISBN: 3-540-64382-6.

[Des87] Y. Desmedt, "Society and group oriented cryptography: a new concept", in *Advances in Cryptology- Crypto'87*, pp. 120-127, 1987.

[Dif76] W. Diffie and M.E. Hellman, "New Directions in Cryptography", *IEEE Trans. Inform. Theory,* IT-22, (6): pp. 644-654, November 1976.

[Dou02] John R. Douceur, "The Sybil Attack", in *First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, March 2002.

[Esc02] Laurent Eschenauer, Virgil D. Gligor and John Baras, "On Trust Establishment in Mobile Ad Hoc Networks", in *Proceedings of the Security Protocols Workshop,* Cambridge, U.K.: Springer-Verlag, April 2002.

[Gam90] D. Gambetta, "Can We Trust Trust?", in *Trust, Making and Breaking Cooperative Relations, basil Blackwell* , 1990, pp. 213-237.

[Gen96a] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, Tal Rabin, "Robust and Efficient Sharing of RSA Functions", *Crypto '96*, 1996.

[Gen96b] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk and Tal Rabin, "Robust Threshold DSS Signatures", *Advances in Cryptology – Eurocrypt '96*, Springer-Verlag, 1996.

[Gho05] Tirthankar Ghosh , Niki Pissinou, Kia Makki, "Towards Designing a Trusted Routing Solution in Mobile Ad Hoc Networks", to appear in the ACM Journal "Mobile Networks and Applications (MONET)" Special issue on Non-Cooperative Wireless Networking and Computing, 2005.

[Gho04a] Tirthankar Ghosh, Kia Makki, Niki Pissinou, "An Overview of Security Issues for Multihop Mobile Ad Hoc Networks", *Network Security: Technology Advances, Strategies, and Change Drivers*, pp. 149-160, ISBN: 0-931695-25-3, 2004.

[Gho04b] Tirthankar Ghosh, Niki Pissinou, Kia Makki "Collaborative Trust-based Secure Routing Against Colluding Malicious Nodes in Multi-hop Ad Hoc Networks", in Proceedings of the 29th IEEE Annual Conference on Local Computer Networks (LCN), Nov 16-18, Tampa, USA, 2004.

[Gon90] Li Gong, Roger Needham, Raphael Yahalom, "Reasoning about Belief in Cryptographic Protocols", in *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, pp. 234-248, May 1990.

[Gra02] Elizabeth Gray, Jean-Marc Seigneur, Yong Chen, Christian Jensen, "Trust Propagation in Small World", in *Proceedings of First International Conference on Trust Management*, 2002.

[Gra00] Tyrone Grandison and Morris Sloman, "A Survey of Trust in Internet Applications", *IEEE Communications Surveys, http://www.comsoc.org/pubs/surveys*, Fourth Quarter 2000.

[He04] Qi He, Dapeng Wu, Pradeep Khosla, "SORI: A Secure and Objective Reputation-based Incentive Scheme for Ad-hoc Networks", in *Proceedings of the IEEE Wireless Communication and Networking Conference (WCNC)*, 2004.

[Hie01] Maarit Hietalahti, "Key Establishment in Ad hoc Networks", *Technical Report, Lab of Theoretical Computer Science, Helsinki University of Technology*, 2001.

[Hu02a] Yih-Chun Hu, David B. Johnson and Adrian Perrig, "SEAD: Secure Efficient Distance Vector Routing for Mobile Wireless Ad hoc Networks", In *Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '02)*, June 2002, pp. 3-13, June 2002.

[Hu02b] Yih-Chun Hu, Adrian Perrig and David B. Johnson, "Ariadne: A Secure On-Demand Routing Protocol for Ad-hoc Networks", in *Proceedings of the 8th annual international conference on Mobile computing and networking (MobiCom) '02*, September 23-26, 2002, Atlanta, Georgia, USA.

[Hub01] Jean-Pierre Hubaux, Levente Buttyán, Srdan Čapkun, "The Quest for Security in Mobile Ad Hoc Networks", in *Proceedings of the Second ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC)*, 2001.

[Joh99] David B. Johnson and David A. Maltz, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks", *Internet Draft, MANET Working Group, IETF*, October, 1999.

[Jos01] A. Josang, "A Logic for Uncertain Probabilities", *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, 9(3): 279-311, 2001.

[Jos98] A. Josang, "A Subjective Metric of Authentication", in *Proceedings of ESORICS: European Symposium on Research in Computer Security*, LNCS, Springer-Verlag, 1998.

[Jos97] A. Josang, "Prospectives for Modelling Trust in Information Security", in Proceedings of Australasian Conference on Information Security and Privacy, pp. 2-13, 1997.

[Kag01] Lalana Kagal, Tim Finin and Anupam Joshi, "Moving from Security to Distributed Trust in Ubiquitous Computing Environments", *IEEE Computer*, December 2001.

[Kon01] Jiejun Kong, Petros Zerfos, Haiyun Luo, Songwu Lu, Lixia Zhang, "Providing Robust and Ubiquitous Security Support for Mobile Ad-Hoc Networks", *International Conference on Network Protocols(ICNP)*, 2001.

[Kri95] P. Krishna, M. Chatterjee, N. H. Vaidya, D. K. Pradhan, "A Cluster based Approach for Routing in Ad Hoc Networks", second *USENIX Symposium on Mobile and Location Independent Computing*, April 1995.

[Li04] Xiaoqi Li, Michael R. Lyu, Jiangchuan Liu, "A Trust Model Based Routing Protocol for Secure Ad Hoc Networks", *Proceedings 2004 IEEE Aerospace Conference*, Big Sky, Montana, U.S.A., March 6-13 2004.

[Liu04] Jinshan Liu, Valérie Issarny, "Enhanced Reputation Mechanism for Mobile Ad Hoc Networks:, in *Proceedings of iTrust 2004*, Oxford, UK, March 2004.

[Lou04] Wenjing Lou, Wei Liu, Yuguang Fang, "SPREAD: Enhancing Data Confidentiality in Mobile Ad Hoc Networks", *IEEE Infocom*, Hong Kong, March 2004.

[Lou03] Wenjing Lou and Yuguang Fang, "A Survey of Wireless Security in Mobile Ad Hoc Networks: Challenges and Available Solutions", *Ad Hoc Wireless Networking*, X. Cheng, X. Huang and D. Z. Du (Eds.), pp. 319-364, Kluwer Academic Publishers, 2003.

[Mar00] Sergio Marti, T.J. Giuli, Kevin Lai and Mary Baker, "Mitigating Routing Misbehavior in Mobile Ad Hoc Networks", in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom)*, Boston, Massachusetts, United States, August 06 - 11, 2000.

[Mat02] Y. Matsuo, "Clustering Using Small World Structure", in *Knowledge based Intelligent Information and Engineering Systems*, Crema, Italy, 2002.

[Mic02] Pietro Michiardi and Refik Molva, "CORE: A Collaborative Reputation Mechanism to Enforce Node Cooperation in Mobile Ad hoc Networks", in *Proceedings of the 6th IFIP Communications and Multimedia Security Conference*, Portorosz, Slovenia, 2002.

[Mil67] Stanley Milgram, "The Small World Problem", *Psychology Today*, 61, 1967.

[New04] James Newsome, Elaine Shi, Dawn Song, Adrian Perrig, "The Sybil Attack in Sensor Networks: Analysis & Defenses", in *Proceedings of Information Processing in Sensor Networks (IPSN '04)*, Berkeley, California, USA, April 26-27, 2004.

[Nga04] Edith C. H. Ngai and Michael R. Lyu, "Trust and Clustering-Based Authentication Services in Mobile Ad Hoc Networks", *Proceedings of the 2nd International Workshop on Mobile Distributed Computing (MDC '04)*, Tokyo, Japan, March 23-26, 2004.

[Pap03] Panagiotis Papadimitratos and Zygmunt J. Haas, "Secure Link State Routing for Mobile Ad hoc Networks", In *Proc. IEEE Workshop on Security and Assurance in Adhoc Networks,* in conjunction with the *2003 International Symposium on Applications and the Internet,* Orlando, FL, January 28, 2003.

[Pap02] Panagiotis Papadimitratos and Zygmunt J. Haas, "Secure Routing for Mobile Ad hoc Networks", In *Proc. SCS Communication Networks and Distributed Systems Modeling and Simulation Conference(CNDS 2002),* San Antonio, TX, January 27-31, 2002.

[Per03] Charles E. Perkins, E. Belding-Royer, S. Das, "RFC 3561 - Ad hoc On-Demand Distance Vector (AODV) Routing", July 2003.

[Per01] Charles E. Perkins, "Ad Hoc Networking", *Addison-Wesley*, ISBN 0-201-30976-9, 2001.

[Per99] C. Perkins and E. Royer, "Ad hoc On-Demand Distance Vector Routing", In *Proc. IEEE Workshop on Mobile Computing Systems and Applications,* 1999.

[Pir04a] Asad Amir Pirzada and Chris McDonald, "Establishing Trust in Pure Ad-hoc Networks", appeared in *27th Australian Computer Science Conference*, The Univ. of Otago, Dunedin, New Zealand, 2004.

[Pir04b] Asad Amir Pirzada, Amitava Datta, Chris McDonald, "Trustworthy Routing with the AODV Protocol", in *Proceedings of the International Conference on Networking and Communication (INCC 2004)*, 2004.

[Pir04c] Asad Amir Pirzada and Chris McDonald, "Kerberos Assisted Authentication in Mobile Ad-hoc Networks", in the Proceedings of the *27th Australasian Computer Science Conference*, The University of Otago, Dunedin, New Zealand, 2004.

[Pis04] Niki Pissinou, Tirthankar Ghosh, Kia Makki, "Collaborative Trust Based Routing in Multihop Ad Hoc Networks", in *Proceedings of Networking '04: Springer Verlag, Series:Lecture Notes in Computer Science,* vol. 3042, pp. 1446 – 1451, Athens, Greece, May 9-14, 2004

[Riv78] R.L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signature and publickey cryptosystems", Communication of ACM, vol. 21, 1978.

[San02] Kimaya Sanzgiri et al, "A Secure Routing Protocol for Ad hoc Networks", In *Proc. of the 10th IEEE International Conference on Network Protocols (ICNP'02),* 2002.

[Sha79] A. Shamir, "How to share a secret", *Commun. ACM*, 22, pp. 612-613, November 1979.

[Smi97] Bradley R. Smith, Shree Murthy, J.J. Garcia-Luna-Aceves, "Securing Distance-Vector Routing Protocols", In *Proceedings of Internet Society Symposium on Network and Distributed System Security,* San Diego, CA, Feb 1997.

[Son03] Joo-Han Song, Vincent W.S. Wong, Victor C.M. Leung, "Efficient On-Demand Routing for Mobile Ad Hoc Wireless Access Networks", partly in *IEEE Vehicular Technology Conference*, Spring 2003, Jeju, Korea, April 2003 and partly in *IEEE Globecom '03*, SanFrancisco, CA, December 2003.

[Sta02] William Stallings, "Cryptography and Network Security: Principles and Practice", *Prentice Hall*, 3rd Edition, August 2002.

[Sta99] Frank Stajano and Ross Anderson, "The Resurrecting Duckling: Security Issues for Ad hoc Wireless Networks", in *Proceedings of the 7th International Workshop on Security Protocols*, vol. 1796 of LNCS, pp. 172-194, Springer Verlag, September, 1999.

[Ste00] Michael Steiner, Gene Tsudik, and Michael Waidner, "Key Agreement in Dynamic Peer Groups", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, No. 8, pp. 769-80, Aug 2000.

[Ste96] M. Steiner, G. Tsudik, and M. Waidner, "Diffie-Hellman key distribution extended to group communication", in *Proceedingsof the 3rd ACM Conference on Computer and Communications Security (CCS)*, New Delhi, India, pp. 31-37, May 14-16, 1996.

[Szt00] Piotr Sztompka, "Trust: A Sociological Theory", *Cambridge University Press*, February 3, 2000.

[Tan04] Sapon Tanachaiwiwat, Pinalkumar Dave, Rohan Bhindwale, Ahmed Helmy, "Location-centric Isolation of Misbehavior and Trust Routing in Energy-constrained Sensor Networks", in *Proceedings of The Workshop on Energy-Efficient Wireless Communications and Networks (EWCN 04)* in conjunction with *IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2004

[The04] George Theodorakopoulos, John S. Baras, "Trust Evaluation in AdHoc Networks", in *Proceedings of the ACM Workshop on Wireless Security (WiSe'04),* Philadelphia, Pennsylvania, USA, October 1, 2004.

[Tse03] Yu-Chee Tseng, Jehn-Ruey Jiang, Jih-Hsin Lee, "Secure Bootstrapping and Routing in an Ipv6-Based Ad Hoc Network", *Workshop on Wireless Security and Privacy*, 2003 (in conjunction with Int'l Conf. on Parallel Processing, 2003).

[Tsu93] Gene Tsudik and Els Van Herreweghen, "On Simple and Secure Key Distribution", *1st Conf.– Computer & Comm. Security '93-11/93* –VA, USA, 1993.

[Ver01] Raja Rai Singh Verma, Donal O'Mahony and Hitesh Tewari, "NTM – Progressive Trust Negotiation in Ad Hoc Networks", in *Proceedings of the 1st joint IEI/IEE Symposium on Telecommunications Systems Research*, Dublin, November 27, 2001.

[Vir05] Mohit Virendra, *et. al.*, "Quantifying Trust in Mobile Ad-Hoc Networks", in *Proceedings of IEEE International Conference on Integration of Knowledge Intensive Multiagent Systems (KIMAS)*, Weltham, MA, April 18-21, 2005.

[Wro02] Konrad Wrona, "Distributed Security: Ad Hoc Networks & Beyond", *Ad Hoc Networks SecurityPampas Workshop, Rhul*, September 16-17, 2002.

[Yan02] Hao Yang, Xiaoqiao Meng, Songwu Lu, "Self-Organized Network Layer Security in Mobile Ad hoc Networks", in *Proceedings of the ACM Workshop on Wireless Security (WiSe '02)*, Atlanta, Georgia, USA, September 28, 2002.

[Yan03] Zheng Yan, Peng Zhang, Teemupekka Virtanen, "Trust Evaluation Based Security Solution in Ad Hoc Networks", in *Proc. of NordSec 2003*, Norway, 2003.

[Yas02] Alec Yasinsac, *et. al.*, "A Family of Protocols for Group Key Generation in Ad Hoc Networks", *International Conference on Communications and Computer Networks (CCN02)*, Nov 3-4, 2002.

[Yi03] Seung Yi, Robin Kravets, "Composite Key Management for Ad Hoc Networks", *Report No.UIUCDCS-R-2003-2392, UILU-ENG-2003-1778*, 2003.

[Yi02] Seung Yi and Robin Kravets, "Key Management for Heterogeneous Ad hoc Wireless Networks", *Report No. UIUCDCS-R-2002-2290, UILU-ENG-2002-1734,* July, 2002.

[Yi01] Seung Yi, Prasad Naldurg and Robin Kravets, "Security-Aware Ad hoc Routing for Wireless Networks", *Report No. UIUCDCS-R-2001-2241, UILU-ENG-2001-1748*, August 2001.

[Zap02] Manuel Guerro Zapata and N. Asokan, "Securing Ad hoc Routing Protocols", in *Proceedings of the ACM Workshop on Wireless Security (WiSe'02)*, Atlanta, Georgia, USA, September 28, 2002.

[Zen04] Weilin Zeng, Tatsuya Suda, "Path Based Routing Algorithm for Ad Hoc Networks", in *Proceedings of the 13th International Conference on Computer Communication and Networks (ICCCN 2004)*, Chicagi, IL, USA, 11-13 October 2004.

[Zen98] Xiang Zeng, Rajive Bagrodia and Mario Gerla, "Glomosim: A Library for Parallel Simulation of Large-scale Wireless Networks", *Proceedings of the 12th Workshop on Parallel and Distributed Simulations – PADS '98,* Alberta, Canada, May 26-29 1998.

[Zha04] Feng Zhao, Leonidas Guibas, "Wireless Sensor Networks: An Information Processing Approach", *Elsevier Science & Technology Book,* ISBN: 1558609148, May 2004.

[Zho99] Lidong Zhou and Zygmunt J. Haas, "Securing Ad Hoc Networks", *IEEE Network*, vol. 13, issue 6, pp. 24-30, November/December 1999.

[Zhu03] Huafei Zhu, Bao Feng, Robert H. Deng, "Computing of Trust in Distributed Networks", *http://eprint.iacr.org/, 2003/056.*

APPENDIX A
Header File of Simulation for Preventing Colluding Attack

```
/*
 * This modified AODV routing header was written by Tirthankar Ghosh in FIU to make the
protocol secured against colluding attack.
 * The modified code was written in May 2004.
 */

/*
 * GloMoSim is COPYRIGHTED software.  Release 2.02 of GloMoSim is available
 * at no cost to educational users only.
 *
 * Commercial use of this software requires a separate license.  No cost,
 * evaluation licenses are available for such purposes; please contact
 * info@scalable-networks.com
 *
 * By obtaining copies of this and any other files that comprise GloMoSim2.02,
 * you, the Licensee, agree to abide by the following conditions and
 * understandings with respect to the copyrighted software:
 *
 * 1.Permission to use, copy, and modify this software and its documentation
 *   for education and non-commercial research purposes only is hereby granted
 *   to Licensee, provided that the copyright notice, the original author's
 *   names and unit identification, and this permission notice appear on all
 *   such copies, and that no charge be made for such copies. Any entity
 *   desiring permission to use this software for any commercial or
 *   non-educational research purposes should contact:
 *
 *   Professor Rajive Bagrodia
 *   University of California, Los Angeles
 *   Department of Computer Science
 *   Box 951596
 *   3532 Boelter Hall
 *   Los Angeles, CA 90095-1596
 *   rajive@cs.ucla.edu
 *
 * 2.NO REPRESENTATIONS ARE MADE ABOUT THE SUITABILITY OF THE
SOFTWARE FOR ANY
 *   PURPOSE. IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY.
 *
 * 3.Neither the software developers, the Parallel Computing Lab, UCLA, or any
 *   affiliate of the UC system shall be liable for any damages suffered by
 *   Licensee from the use of this software.
 */
```

```
// Use the latest version of Parsec if this line causes a compiler error.
/*
 * Name: aodv.h
 *
 * Implemented by SJ Lee (sjlee@cs.ucla.edu)
 */

/*
 NOTE: The parameter values followed the AODV Internet Draft
      (draft-ietf-manet-aodv-03.txt) and NS2 code by Samir R. Das
      Read the NOTE of aodv.pc for more details
*/

#ifndef _AODV_H_
#define _AODV_H_

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

#define ACTIVE_ROUTE_TO                 10 * SECOND

#define NODE_TRAVERSAL_TIME             40 * MILLI_SECOND

#define NET_DIAMETER                    35

#define RREP_WAIT_TIME          3 * NODE_TRAVERSAL_TIME * NET_DIAMETER / 2

#define BAD_LINK_LIFETIME               2 * RREP_WAIT_TIME

#define BCAST_ID_SAVE                   30 * SECOND

#define REV_ROUTE_LIFE                  RREP_WAIT_TIME

#define MY_ROUTE_TO                     2 * ACTIVE_ROUTE_TO

#define RREQ_RETRIES                    2

#define TTL_START                       1

#define TTL_INCREMENT                   2

#define TTL_THRESHOLD                   7

#define AODV_INFINITY                   255

#define BROADCAST_JITTER                10 * MILLI_SECOND

#define Public                          17              /* added */
```

```
#define Private                 593         /* added */

#define n                       2623        /* added */

#define message                 5           /* added */

/* Packet Types */



typedef unsigned char AODV_PacketType;

#define AODV_RREQ 0
#define AODV_RREP 1
#define AODV_RERR 2
#define AODV_RWARN 3    /*  added  */



typedef struct
{
   AODV_PacketType pktType;
   int bcastId;
   NODE_ADDR destAddr;
   int destSeq;
   NODE_ADDR srcAddr;
   int srcSeq;
   NODE_ADDR lastAddr;          /* address of the node from which next node receives a
RREQ packet  */
   int hopCount;
   int trust_level;                         /*  parameter added  */
   long int MAC;                            /*  parameter added  */
} AODV_RREQ_Packet;



/* ------------------------------------new structure for RWARN packet----------------------------------
*/
typedef struct
{
   AODV_PacketType pktType;
   NODE_ADDR srcAddr;
   int bcastId;
   int srcSeq;
   NODE_ADDR maliciousIP;
   //int trust_level;
   long int MAC;
   NODE_ADDR rwarn_sourceIP;
   clocktype lifetime;
```

```
} AODV_RWARN_Packet;

/* ---------------------------------------------------------------------------------------------- */

typedef struct
{
   AODV_PacketType pktType;
   NODE_ADDR srcAddr;
   NODE_ADDR destAddr;
   int destSeq;
   int hopCount;
   int trust_level;                          /*  parameter added  */
   NODE_ADDR next_hop;                       /*  parameter added  */
   NODE_ADDR lastAddr;            /* address of the node from which next node receives a
RREP packet  */
   clocktype lifetime;
} AODV_RREP_Packet;

typedef struct
{
 NODE_ADDR destinationAddress;
 int destinationSequenceNumber;
} AODV_AddressSequenceNumberPairType;

#define AODV_MAX_RERR_DESTINATIONS 250

typedef struct
{
   AODV_PacketType pktType;        // 1 byte
   unsigned char filling[2];
   unsigned char destinationCount;
   AODV_AddressSequenceNumberPairType
     destinationPairArray[AODV_MAX_RERR_DESTINATIONS];
} AODV_RERR_Packet;


static //inline//
int AODV_RERR_PacketSize(const AODV_RERR_Packet* rerrPacket) {
  return
     (sizeof(rerrPacket->pktType) +
      sizeof(rerrPacket->filling) +
      sizeof(rerrPacket->destinationCount) +
      (rerrPacket->destinationCount *
       sizeof(AODV_AddressSequenceNumberPairType)));
}



typedef struct RTE
```

```
{
   NODE_ADDR destAddr;
   int destSeq;
   int hopCount;
   int trust_level;                          /*  parameter added  */
   int lastHopCount;
   NODE_ADDR nextHop;
   clocktype lifetime;
   BOOL activated;
   BOOL source;
   struct RTE *next;
} AODV_RT_Node;

typedef struct
{
   AODV_RT_Node *head;
   int size;
} AODV_RT;

typedef struct NTE
{
   NODE_ADDR destAddr;
   struct NTE *next;
} AODV_NT_Node;

typedef struct
{
   AODV_NT_Node *head;
   int size;
} AODV_NT;

typedef struct RSE
{
   NODE_ADDR srcAddr;
   int bcastId;
   int hopCount;                             /* parameter added  */
   int trust_level;                          /* parameter added  */
   struct RSE *next;
} AODV_RST_Node;

typedef struct
{
   AODV_RST_Node *front;
   AODV_RST_Node *rear;
   int size;
} AODV_RST;

typedef struct FIFO
{
```

```
    NODE_ADDR destAddr;
    clocktype timestamp;
    Message *msg;
    struct FIFO *next;
} AODV_BUFFER_Node;

typedef struct
{
    AODV_BUFFER_Node *head;
    int size;
} AODV_BUFFER;

typedef struct SE
{
    NODE_ADDR destAddr;
    int ttl;
    int times;
    struct SE *next;
} AODV_SENT_Node;

typedef struct
{
    AODV_SENT_Node *head;
    int size;
} AODV_SENT;

// ------------------------------------------------------------------------------- new structure added
//typedef struct
//{
//  NODE_ADDR node;
//  int trust_level;
//} AODV_Trust_Table[20];

// -------------------------------------------------------------------------------

typedef struct
{
    int numRequestSent;
    int numReplySent;
    int numWarningSent;                                    /* new statistic added */
    int numWarning2Sent;                                   /* new statistic added */
    int numRerrSent;
    int numRerrResent;
    int numDataSent;          /* Data Sent as the source of the route */
    int numDataTxed;
    int numDataReceived;    /* Data Received as the destination of the route */
    int numHops;
    int numRoutes;
    int numPacketsDropped;
```

```
   int numBrokenLinks;
   int numBrokenLinkRetries;
   long int encr_message;                    /* new statistic added to display
encrypted message */
   long int decr_message;                    /* new statistic added to display
decrypted message */
} AODV_Stats;

typedef struct glomo_network_aodv_str
{
   AODV_RT routeTable;
   AODV_NT nbrTable;
   AODV_RST seenTable;
   AODV_BUFFER buffer;
   AODV_SENT sent;
   AODV_Stats stats;
   int seqNumber;
   int bcastId;
} GlomoRoutingAodv;

void RoutingAodvInit(
   GlomoNode *node,
   GlomoRoutingAodv **aodvPtr,
   const GlomoNodeInput *nodeInput);

void RoutingAodvFinalize(GlomoNode *node);

void RoutingAodvHandleData(GlomoNode *node, Message *msg, NODE_ADDR destAddr);

void RoutingAodvHandleRequest(GlomoNode *node, Message *msg, int ttl);

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

void RoutingAodvHandleWarning(GlomoNode *node, Message *msg, NODE_ADDR srcAddr,
NODE_ADDR destAddr, int trust_level, long int MAC);
/*  new function added  */

void RoutingAodvInitRouteTable(AODV_RT *routeTable);

void RoutingAodvInitNbrTable(AODV_NT *nbrTable);

void RoutingAodvInitSeenTable(AODV_RST *seenTable);

void RoutingAodvInitBuffer(AODV_BUFFER *buffer);

void RoutingAodvInitSent(AODV_SENT *sent);

void RoutingAodvInitStats(GlomoNode *node);
```

```
void RoutingAodvInitSeq(GlomoNode *node);
void RoutingAodvInitBcastId(GlomoNode *node);

NODE_ADDR RoutingAodvGetNextHop(NODE_ADDR destAddr, AODV_RT *routeTable);

int RoutingAodvGetBcastId(GlomoNode *node);

int RoutingAodvGetSeq(NODE_ADDR destAddr, AODV_RT *routeTable);

int RoutingAodvGetMySeq(GlomoNode *node);

int RoutingAodvGetHopCount(NODE_ADDR destAddr, AODV_RT *routeTable);

int RoutingAodvGetLastHopCount(NODE_ADDR destAddr, AODV_RT *routeTable);

int RoutingAodvGetTtl(NODE_ADDR destAddr, AODV_SENT *sent);

int RoutingAodvGetTimes(NODE_ADDR destAddr, AODV_SENT *sent);

clocktype RoutingAodvGetLifetime(NODE_ADDR destAddr, AODV_RT *routeTable);

Message *
RoutingAodvGetBufferedPacket(NODE_ADDR destAddr, AODV_BUFFER *buffer);

BOOL RoutingAodvCheckRouteExist(NODE_ADDR destAddr, AODV_RT *routeTable);

BOOL RoutingAodvCheckNbrExist(NODE_ADDR destAddr, AODV_NT *nbrTable);

BOOL RoutingAodvLookupSeenTable(NODE_ADDR srcAddr,
                    int bcastId,
                    AODV_RST *seenTable);

/* ---------------------------------new function ---------------------------------------------- */
//BOOL RoutingAodvLookupSeenTable_WARN(int srcSeq,
//                     int bcastId,
//                     AODV_RST *seenTable_WARN);
/*------------------------------------------------------------------------------------------- */

//void RoutingAodvLookupSeenTable1(GlomoNode *node,Message *msg,              /*
parameter added  */
//                     NODE_ADDR srcAddr,NODE_ADDR lastAddr,int bcastId,int hopCount,
//                     int trust_level,AODV_RST *seenTable, AODV_NT *nbrTable);

void RoutingAodvLookupSeenTable1(GlomoNode *node,Message *msg,              /*
parameter added  */
                    NODE_ADDR srcAddr,NODE_ADDR lastAddr,int bcastId,int hopCount,
                    int trust_level,long int MAC,AODV_RST *seenTable);
```

```
BOOL RoutingAodvLookupBuffer(NODE_ADDR destAddr, AODV_BUFFER *buffer);
BOOL RoutingAodvCheckSent(NODE_ADDR destAddr, AODV_SENT *sent);


void RoutingAodvHandleProtocolPacket(
   GlomoNode *node, Message *msg, NODE_ADDR srcAddr,
   NODE_ADDR destAddr, int ttl, int trust_level, long int MAC);

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

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

void RoutingAodvPacketDropNotificationHandler(
  GlomoNode *node, const Message* msg, const NODE_ADDR nextHopAddress);

void RoutingAodvSetTimer(
   GlomoNode *node, long eventType, NODE_ADDR destAddr, clocktype delay);

void RoutingAodvInitiateRREQ(GlomoNode *node, NODE_ADDR destAddr);

void RoutingAodvRetryRREQ(GlomoNode *node, NODE_ADDR destAddr);

void RoutingAodvTransmitData(GlomoNode *node, Message *msg, NODE_ADDR destAddr);

void RoutingAodvRelayRREQ(GlomoNode *node, Message *msg, int ttl);

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

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

void RoutingAodvRelayRREP(GlomoNode *node, Message *msg, NODE_ADDR destAddr);

void RoutingAodvRelayWarning();                    /* new function added */

void RoutingAodvRelayWarning_1();                  /* new function added */

//void initTrustTable()                            /* new function added */

#endif /* _AODV_H_ */
```

APPENDIX B
Sample Code of Simulation for Preventing Colluding Attack

```
/*
 * This modified AODV routing protocol was written by Tirthankar Ghosh in FIU to make the
protocol secured against colluding attack.
 * The modified code was written in May 2004.
 */

/*
 * GloMoSim is COPYRIGHTED software.  Release 2.02 of GloMoSim is available
 * at no cost to educational users only.
 *
 * Commercial use of this software requires a separate license.  No cost,
 * evaluation licenses are available for such purposes; please contact
 * info@scalable-networks.com
 *
 * By obtaining copies of this and any other files that comprise GloMoSim2.02,
 * you, the Licensee, agree to abide by the following conditions and
 * understandings with respect to the copyrighted software:
 *
 * 1.Permission to use, copy, and modify this software and its documentation
 *   for education and non-commercial research purposes only is hereby granted
 *   to Licensee, provided that the copyright notice, the original author's
 *   names and unit identification, and this permission notice appear on all
 *   such copies, and that no charge be made for such copies. Any entity
 *   desiring permission to use this software for any commercial or
 *   non-educational research purposes should contact:
 *
 *   Professor Rajive Bagrodia
 *   University of California, Los Angeles
 *   Department of Computer Science
 *   Box 951596
 *   3532 Boelter Hall
 *   Los Angeles, CA 90095-1596
 *   rajive@cs.ucla.edu
 *
 * 2.NO REPRESENTATIONS ARE MADE ABOUT THE SUITABILITY OF THE
SOFTWARE FOR ANY
 *   PURPOSE. IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY.
 *
 * 3.Neither the software developers, the Parallel Computing Lab, UCLA, or any
 *   affiliate of the UC system shall be liable for any damages suffered by
 *   Licensee from the use of this software.
 */
```

```
// Use the latest version of Parsec if this line causes a compiler error.
/*
 * Name: aodv.pc
 *
 * Implemented by SJ Lee (sjlee@cs.ucla.edu)
 */

/*
 NOTE: - Followed the specification of AODV Internet Draft
         (draft-ietf-manet-aodv-03.txt)
       - This implements only unicast functionality of AODV.
       - Assumes the MAC protocol sends a signal to the routing protocol
         when it detects link breaks. MAC protocols such as IEEE 802.11
         and MACAW has this functionality. In IEEE 802.11, when no CTS
         is received after RTS, and no ACK is received after retransmissions
         of unicasted packet, it sends the signal to the routing protocol
       - If users want to use MAC protocols other than IEEE 802.11, they
         must implement schemes to detect link breaks. A way to do this is,
         for example, using HELLO packets, as specified in AODV documents.
       - No Precursors (Implemented other mechanism so that the protocol can
         still function the same as when precursors are used)
       - Unsolicited RREPs are broadcasted and forwarded only if the node
         is part of the broken route and not the source of that route
       - If more than one route uses the broken link, send RREP multiple times
         (this should be fixed based on new specification by C. Perkins,
         E. Royer, and S. Das)
       - Rev route of RREQ overwrites the one in the route table
       - May need slight modifications when draft-ietf-manet-aodv-04.txt
         comes out
*/

#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"
```

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

/*
 * RoutingAodvReplaceInsertRouteTable
 *
 * Insert/Update an entry into the route table
 */

static void
RoutingAodvReplaceInsertRouteTable(
    NODE_ADDR destAddr,
    int destSeq,
    int hopCount,
/* ------------------------------------------------------------------------------ parameter added */
    int trust_level,
    NODE_ADDR next_hop,
/* ------------------------------------------------------------------------------------------------ */
    NODE_ADDR nextHop,
    clocktype lifetime,
    BOOL activated,
    BOOL source,
    AODV_RT* routeTable)
{
    AODV_RT_Node* theNode = NULL;
    AODV_RT_Node* current;
    AODV_RT_Node* previous;

    // Find Insertion point.

    previous = NULL;
    current = routeTable->head;
    while ((current != NULL) && (current->destAddr < destAddr)) {
        previous = current;
        current = current->next;
    }//while//


    if ((current == NULL) || (current->destAddr != destAddr)) {
        ++(routeTable->size);

        theNode = (AODV_RT_Node *)checked_pc_malloc(sizeof(AODV_RT_Node));
        theNode->lifetime = lifetime;
        theNode->activated = activated;
        theNode->source = source;
        theNode->destAddr = destAddr;

        if (previous == NULL) {
            theNode->next = routeTable->head;
            routeTable->head = theNode;
```

```
        } else {
          theNode->next = previous->next;
          previous->next = theNode;
        }//if//


      } else {
        assert(current->destAddr == destAddr);

        current->lifetime = max(lifetime, current->lifetime);
        if (!current->activated) {
          current->activated = activated;
        }//if//

        if (!current->source) {
          current->source = source;
        }//if//

        theNode = current;
      }//if//

      theNode->destSeq = destSeq;
      theNode->hopCount = hopCount;
/* ---------------------------------------------------------------------------- parameter added */
      theNode->trust_level = trust_level;
      //theNode->next_hop = next_hop;
/* ------------------------------------------------------------------------------------------- */
      theNode->lastHopCount = hopCount;
      theNode->nextHop = nextHop;

} /* RoutingAodvReplaceInsertRouteTable */



static
void RoutingAodvInsertNbrTable(NODE_ADDR destAddr, AODV_NT* nbrTable)
{
      AODV_NT_Node* current;
      AODV_NT_Node* previous;

      AODV_NT_Node* newNode =
        (AODV_NT_Node *)checked_pc_malloc(sizeof(AODV_NT_Node));

      newNode->destAddr = destAddr;
      newNode->next = NULL;

      ++(nbrTable->size);

      // Find Insertion point.  Insert after all address matches.
```

```
    previous = NULL;
    current = nbrTable->head;
    while ((current != NULL) && (current->destAddr <= destAddr)) {
        previous = current;
        current = current->next;
    }//while//


    if (previous == NULL) {
        newNode->next = nbrTable->head;
        nbrTable->head = newNode;
    } else {
        newNode->next = previous->next;
        previous->next = newNode;
    }//if//
} /* RoutingAodvInsertNbrTable */


/*
 * RoutingAodvInsertSeenTable
 *
 * Insert an entry into the seen table
 */

static void
RoutingAodvInsertSeenTable(
    GlomoNode *node,
    NODE_ADDR srcAddr,
    int bcastId,
    int hopCount,                                   /* parameter added */
    int trust_level,                                /* parameter added */
    AODV_RST *seenTable)
{
    if (seenTable->size == 0)
    {
        seenTable->rear = (AODV_RST_Node *) pc_malloc(sizeof(AODV_RST_Node));
        assert(seenTable->rear != NULL);
        seenTable->front = seenTable->rear;
    }
    else
    {
        seenTable->rear->next = (AODV_RST_Node *)
                        pc_malloc(sizeof(AODV_RST_Node));
        assert(seenTable->rear->next != NULL);
        seenTable->rear = seenTable->rear->next;
    }

    seenTable->rear->srcAddr = srcAddr;
    seenTable->rear->bcastId = bcastId;
```

```
    seenTable->rear->hopCount = hopCount;                    /* parameter added */
    seenTable->rear->trust_level = trust_level;              /* parameter added */

    seenTable->rear->next = NULL;

    ++(seenTable->size);

    RoutingAodvSetTimer(
      node, MSG_NETWORK_FlushTables, ANY_DEST, (clocktype)BCAST_ID_SAVE);

} /* RoutingAodvInsertSeenTable */

/*
 * RoutingAodvInsertBuffer
 *
 * Insert a packet into the buffer if no route is available
 */
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);

    // Find Insertion point.  Insert after all address matches.

    previous = NULL;
    current = buffer->head;
    while ((current != NULL) && (current->destAddr <= destAddr)) {
      previous = current;
      current = current->next;
    }//while//


    if (previous == NULL) {
      newNode->next = buffer->head;
      buffer->head = newNode;
```

```
    } else {
      newNode->next = previous->next;
      previous->next = newNode;
    }//if//
} /* RoutingAodvInsertBuffer */



/*
 * RoutingAodvInsertSent
 *
 * Insert an entry into the sent table if RREQ is sent
 */
static void
RoutingAodvInsertSent(
    NODE_ADDR destAddr,
    int ttl,
    AODV_SENT *sent)
{
    AODV_SENT_Node* current;
    AODV_SENT_Node* previous;

    AODV_SENT_Node* newNode =
      (AODV_SENT_Node *)checked_pc_malloc(sizeof(AODV_SENT_Node));

    newNode->destAddr = destAddr;
    newNode->ttl = ttl;
    newNode->times = 0;
    newNode->next = NULL;

    (sent->size)++;

    // Find Insertion point.  Insert after all address matches.

    previous = NULL;
    current = sent->head;
    while ((current != NULL) && (current->destAddr <= destAddr)) {
      previous = current;
      current = current->next;
    }//while//


    if (previous == NULL) {
      newNode->next = sent->head;
      sent->head = newNode;
    } else {
      newNode->next = previous->next;
      previous->next = newNode;
    }//if//
```

```
} /* RoutingAodvInsertSent */
/*
 * RoutingAodvDeleteRouteTable
 *
 * Remove an entry from the route table
 */
void RoutingAodvDeleteRouteTable(NODE_ADDR destAddr, AODV_RT *routeTable)
{
   AODV_RT_Node *toFree;
   AODV_RT_Node *current;

   if (routeTable->size == 0 || routeTable->head == NULL)
   {
      return;
   }
   else if (routeTable->head->destAddr == destAddr)
   {
      if (routeTable->head->lifetime <= simclock())
      {
         toFree = routeTable->head;
         routeTable->head = toFree->next;
         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())
      {
         toFree = current->next;
         current->next = toFree->next;
         pc_free(toFree);
         --(routeTable->size);
      }
   }

} /* RoutingAodvDeleteRouteTable */

/*
 * RoutingAodvDeleteNbrTable
 *
 * Remove an entry from the neighbor table
```

```
*/
void RoutingAodvDeleteNbrTable(NODE_ADDR destAddr, AODV_NT *nbrTable)
{
   AODV_NT_Node *toFree;
   AODV_NT_Node *current;

   if (nbrTable->size == 0)
   {
      return;
   }
   else if (nbrTable->head->destAddr == destAddr)
   {
      toFree = nbrTable->head;
      nbrTable->head = toFree->next;
      pc_free(toFree);
      --(nbrTable->size);
   }
   else
   {
      for (current = nbrTable->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);
         --(nbrTable->size);
      }
   }

} /* RoutingAodvDeleteNbrTable */

/*
 * RoutingAodvDeleteSeenTable
 *
 * Remove an entry from the seen table
 */
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;
    }

} /* RoutingAodvDeleteSeenTable */

/*
 * RoutingAodvDeleteBuffer
 *
 * Remove a packet from the buffer; Return TRUE if deleted
 */
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);

} /* RoutingAodvDeleteBuffer */

/*
 * RoutingAodvDeleteSent
 *
 * Remove an entry from the sent table
 */
void RoutingAodvDeleteSent(NODE_ADDR destAddr, AODV_SENT *sent)
{
    AODV_SENT_Node *toFree;
    AODV_SENT_Node *current;

    if (sent->size == 0)
    {
        return;
    }
    else if (sent->head->destAddr == destAddr)
    {
        toFree = sent->head;
        sent->head = toFree->next;
        pc_free(toFree);
        --(sent->size);
    }
    else
    {
        for (current = sent->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);
            --(sent->size);
        }
    }

} /* RoutingAodvDeleteSent */


/*
```

```
 * RoutingAodvUpdateLifetime
 *
 * Update the lifetime field of the destination entry in the route table
 */
void RoutingAodvUpdateLifetime(NODE_ADDR destAddr, AODV_RT *routeTable)
{
  AODV_RT_Node *current;

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

} /* RoutingAodvUpdateLifetime */

/*
 * RoutingAodvIncreaseSeq
 *
 * Increase the sequence number
 */
void RoutingAodvIncreaseSeq(GlomoNode *node)
{
  GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
  GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

  aodv->seqNumber++;

} /* RoutingAodvIncreaseSeq */

/*
 * RoutingAodvIncreaseTtl
 *
 * Increase the TTL value
 */
void RoutingAodvIncreaseTtl(NODE_ADDR destAddr, AODV_SENT *sent)
{
  AODV_SENT_Node *current;

  for (current = sent->head;
      current != NULL && current->destAddr <= destAddr;
      current = current->next)
  {
    if (current->destAddr == destAddr)
```

```
      {
        current->ttl += TTL_INCREMENT;

        if (current->ttl > TTL_THRESHOLD)
        {
          current->ttl = NET_DIAMETER;
        }

        return;
      }

    }

} /* RoutingAodvIncreaseTtl */

/*
 * RoutingAodvUpdateTtl
 *
 * Update the ttl value
 */
void RoutingAodvUpdateTtl(NODE_ADDR destAddr, int ttl, AODV_SENT *sent)
{
  AODV_SENT_Node *current;

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

} /* RoutingAodvUpdateTtl */

/*
 * RoutingAodvIncreaseTimes
 *
 * Increase the number of times RREQ sent in TTL = NET_DIAMETER
 */
void RoutingAodvIncreaseTimes(NODE_ADDR destAddr, AODV_SENT *sent)
{
  AODV_SENT_Node *current;

  for (current = sent->head;
       current != NULL && current->destAddr <= destAddr;
       current = current->next)
```

```
    {
      if (current->destAddr == destAddr)
      {
        current->times++;
        return;
      }
    }

} /* RoutingAodvIncreaseTimes */

/*
 * RoutingAodvActivateRoute
 *
 * Activate a route in the route table
 */
void RoutingAodvActivateRoute(NODE_ADDR destAddr, AODV_RT *routeTable)
{
    AODV_RT_Node *current;

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

} /* RoutingAodvActivateRoute */

/*
 * RoutingAodvInactivateRoutesAndGetDestinations
 *
 * Inactivate routes that use the broken link
 * Returns the destAddr and whether the node must relay the RREP
 */

void RoutingAodvInactivateRoutesAndGetDestinations(
    GlomoNode* node,
    AODV_RT* routeTable,
    NODE_ADDR nextHop,
    AODV_AddressSequenceNumberPairType destinationPairs[],
    int maxNumberDestinationPairs,
    int* numberDestinations)
{
    AODV_RT_Node *current;
```

```
    int numDests = 0;

    for (current = routeTable->head;
         current != NULL;
         current = current->next)
    {
        if ((current->nextHop == nextHop) && (current->activated == TRUE))
        {
            current->activated = FALSE;
            current->hopCount = AODV_INFINITY;
            current->lifetime = simclock() + BAD_LINK_LIFETIME;
            current->destSeq++;

            RoutingAodvSetTimer(
                node, MSG_NETWORK_CheckRouteTimeout, current->destAddr,
                (clocktype)BAD_LINK_LIFETIME);

            if (!current->source) {
                destinationPairs[numDests].destinationAddress =
                    current->destAddr;
                destinationPairs[numDests].destinationSequenceNumber =
                    current->destSeq;
                numDests++;
            }//if//
        }//if//
    }//for//

    *numberDestinations = numDests;

} /* RoutingAodvInactivateRoute */


/*
 * RoutingAodvMarkRouteBroken
 *
 * Mark the route with destAddr broken; returns TRUE if relay is required
 */
BOOL RoutingAodvMarkRouteBroken(GlomoNode *node,
                                NODE_ADDR destAddr,
                                AODV_RT *routeTable)
{
    AODV_RT_Node *current;
    BOOL relay = FALSE;

    for (current = routeTable->head;
         current != NULL && current->destAddr <= destAddr;
         current = current->next)
    {
        if (current->destAddr == destAddr && current->activated == TRUE)
```

```
        {
          current->activated = FALSE;
          current->hopCount = AODV_INFINITY;
          current->lifetime = simclock() + BAD_LINK_LIFETIME;
          current->destSeq++;

          RoutingAodvSetTimer(
            node, MSG_NETWORK_CheckRouteTimeout, current->destAddr,
            (clocktype)BAD_LINK_LIFETIME);

          if (current->source == FALSE)
          {
            relay = TRUE;
          }

          return (relay);
        }
      }

      return (relay);

} /* RoutingAodvMarkRouteBroken */


/*
 * RoutingAodvUpdateSeq
 *
 * Update the sequence number of a certain destination
 */
void RoutingAodvUpdateSeq(NODE_ADDR destAddr, int seq, AODV_RT *routeTable)
{
    AODV_RT_Node *current;

    for (current = routeTable->head;
         current != NULL && current->destAddr <= destAddr;
         current = current->next)
    {
      if (current->destAddr == destAddr)
      {
        current->destSeq = seq;
        return;
      }
    }

} /* RoutingAodvUpdateSeq */
```

```
static //inline//
void SendRouteErrorPacket(
   GlomoNode* node,
   const AODV_RERR_Packet* rerrPacket)
{
   Message* newMsg = GLOMO_MsgAlloc(node, 0, 0, 0);
   int packetSize = AODV_RERR_PacketSize(rerrPacket);

   assert(rerrPacket->pktType == (unsigned short)AODV_RERR);
   assert(rerrPacket->destinationCount >= 1);

   GLOMO_MsgPacketAlloc(node, newMsg, packetSize);
   memcpy(GLOMO_MsgReturnPacket(newMsg), rerrPacket, packetSize);
   NetworkIpSendRawGlomoMessage(
      node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, 1);
}



/*
 * RoutingAodvInit
 *
 * Initialization function for AODV protocol
 */
void RoutingAodvInit(
   GlomoNode *node,
   GlomoRoutingAodv **aodvPtr,
   const GlomoNodeInput *nodeInput)
{
   GlomoRoutingAodv *aodv =
      (GlomoRoutingAodv *)checked_pc_malloc (sizeof(GlomoRoutingAodv));

   (*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);
   RoutingAodvInitBuffer(&aodv->buffer);
   RoutingAodvInitSent(&aodv->sent);
   RoutingAodvInitSeq(node);
   RoutingAodvInitBcastId(node);
```

```
    NetworkIpSetPacketDropNotificationFunction(
        node, &RoutingAodvPacketDropNotificationHandler);

    NetworkIpSetRouterFunction(node, &RoutingAodvRouterFunction);

} /* RoutingAodvInit */

/*
 * RoutingAodvFinalize
 *
 * Called at the end of the simulation to collect the results
 */
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 Route Requests Txed = %d",
            aodv->stats.numRequestSent);
    GLOMO_PrintStat(node, "RoutingAodv", buf);

    sprintf(buf, "Number of Replies Txed = %d",
            aodv->stats.numReplySent);
    GLOMO_PrintStat(node, "RoutingAodv", buf);

/* ----------------------------------------------------------------------------------------------------new statistic */
    sprintf(buf, "Number of Warnings Txed = %d",
            aodv->stats.numWarningSent);
    GLOMO_PrintStat(node, "RoutingAodv", buf);
/*
     sprintf(buf, "Encrypted message = %d",
          aodv->stats.encr_message);
     GLOMO_PrintStat(node, "RoutingAodv", buf);

     sprintf(buf, "Decrypted message = %d",
          aodv->stats.decr_message);
     GLOMO_PrintStat(node, "RoutingAodv", buf);
*/
     sprintf(buf, "Number of 2nd Warnings Txed = %d",
            aodv->stats.numWarning2Sent);
     GLOMO_PrintStat(node, "RoutingAodv", buf);
 /* ---------------------------------------------------------------------------------------------------------------- */

    sprintf(buf, "Number of Route Errors (RERR) Txed = %d",
            aodv->stats.numRerrSent);
    GLOMO_PrintStat(node, "RoutingAodv", buf);
```

```
    sprintf(buf, "Number of Route Errors (RERR) Re-sent = %d",
            aodv->stats.numRerrResent);
    GLOMO_PrintStat(node, "RoutingAodv", buf);


    sprintf(buf, "Number of CTRL Packets Txed = %d",
            aodv->stats.numRequestSent + aodv->stats.numReplySent + aodv-
>stats.numWarningSent);
    GLOMO_PrintStat(node, "RoutingAodv", buf);

    sprintf(buf, "Number of Routes Selected = %d", aodv->stats.numRoutes);
    GLOMO_PrintStat(node, "RoutingAodv", buf);
    sprintf(buf, "Number of Hop Counts = %d", aodv->stats.numHops);
    GLOMO_PrintStat(node, "RoutingAodv", buf);

    sprintf(buf, "Number of Data Txed = %d",
            aodv->stats.numDataTxed);
    GLOMO_PrintStat(node, "RoutingAodv", buf);
    sprintf(buf, "Number of Data Packets Originated = %d",
            aodv->stats.numDataSent);
    GLOMO_PrintStat(node, "RoutingAodv", buf);
    sprintf(buf, "Number of Data Packets Received = %d",
            aodv->stats.numDataReceived);
    GLOMO_PrintStat(node, "RoutingAodv", buf);

    sprintf(buf, "Number of Packets Dropped or Left waiting for Route = %d",
            (aodv->stats.numPacketsDropped + aodv->buffer.size));
    GLOMO_PrintStat(node, "RoutingAodv", buf);

    sprintf(buf, "Number of Broken Links = %d", aodv->stats.numBrokenLinks);
    GLOMO_PrintStat(node, "RoutingAodv", buf);
    sprintf(buf, "Number of Broken Link Retries = %d", aodv->stats.numBrokenLinkRetries);
    GLOMO_PrintStat(node, "RoutingAodv", buf);


} /* RoutingAodvFinalize */

/*
 * RoutingAodvHandleData
 *
 * Processing procedure when data is received
 */
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;
```

```
  assert(sourceAddress  != node->nodeAddr);

  /* the node is the destination of the route */
  if (destAddr == node->nodeAddr)
  {
     aodv->stats.numDataReceived++;

     RoutingAodvUpdateLifetime(sourceAddress, &aodv->routeTable);

     RoutingAodvSetTimer(node, MSG_NETWORK_CheckRouteTimeout,
                  sourceAddress, (clocktype)ACTIVE_ROUTE_TO);
  }
  else if (destAddr != ANY_DEST)
  {
     // The node is an intermediate node of the route.
     // Relay the packet to the next hop of the route

     if (RoutingAodvCheckRouteExist(destAddr, &aodv->routeTable)) {
        RoutingAodvTransmitData(node, msg, destAddr);
     } else {
        // Broken Route.  Drop Packet, send RERR again to make them stop
        // sending more.
        AODV_RERR_Packet newRerrPacket;
        newRerrPacket.pktType = AODV_RERR;
        newRerrPacket.destinationCount = 1;
        newRerrPacket.destinationPairArray[0].destinationAddress = destAddr;
        newRerrPacket.destinationPairArray[0].destinationSequenceNumber
          = RoutingAodvGetSeq(destAddr, &aodv->routeTable);

        SendRouteErrorPacket(node, &newRerrPacket);
        aodv->stats.numRerrResent++;

        aodv->stats.numPacketsDropped++;
        GLOMO_MsgFree(node,msg);

     }//if//
  }//if//

} /* RoutingAodvHandleData */

/*
 * RoutingAodvHandleRequest
 *
 * Processing procedure when RREQ is received
 */
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);

  /* Process only if the packet is not a duplicate */
  if (!RoutingAodvLookupSeenTable(
          rreqPkt->srcAddr, rreqPkt->bcastId, &aodv->seenTable))
  {
    RoutingAodvInsertSeenTable(
          node, rreqPkt->srcAddr, rreqPkt->bcastId, rreqPkt->hopCount, rreqPkt->trust_level,
&aodv->seenTable); /* added parameter */

    /* Update the neighbor table if the upstream is new */
    if (!RoutingAodvCheckNbrExist(rreqPkt->lastAddr, &aodv->nbrTable))
    {
      RoutingAodvInsertNbrTable(rreqPkt->lastAddr, &aodv->nbrTable);
      RoutingAodvIncreaseSeq(node);
    }

    /* The node is the destination of the route */
    if (node->nodeAddr == rreqPkt->destAddr)
    {
      RoutingAodvReplaceInsertRouteTable(
        rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount, rreqPkt->trust_level,    /*
added parameter  */
        rreqPkt->lastAddr, rreqPkt->lastAddr, simclock() + ACTIVE_ROUTE_TO, TRUE,
TRUE,
        &aodv->routeTable);

      RoutingAodvSetTimer(
        node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
        (clocktype)ACTIVE_ROUTE_TO);

      /* Send a Route Reply */
      RoutingAodvInitiateRREP(node, msg);

    } /* if dest */

    else
    {
      /* No route to destination is known */
/* -------------------------------------------------------------------------------------------------not reqd
*/

 //       if (!RoutingAodvCheckRouteExist(rreqPkt->destAddr,
 //                                       &aodv->routeTable))
 //       {

/* -------------------------------------------------------------------------------------------------not reqd
*/
```

```
            RoutingAodvReplaceInsertRouteTable(
              rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount, rreqPkt->trust_level,    /*
added parameter  */
              rreqPkt->lastAddr, rreqPkt->lastAddr, simclock() + REV_ROUTE_LIFE, FALSE,
              FALSE, &aodv->routeTable);

            RoutingAodvSetTimer(
              node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
              (clocktype)REV_ROUTE_LIFE);

            if (ttl > 0)
            {
              /* Relay the packet only if TTL is not zero */
              RoutingAodvRelayRREQ(node, msg, ttl);
            } /* if ttl > 0 */
            else
            {
              GLOMO_MsgFree(node, msg);
            }
          //  } /* if no route */

           /* Knows a route to the destination */
/* ----------------------------------------------------------------------------------------------------not reqd
*/

//          else
//          {
//            /* However, the known route is not a fresh one */
//            if (RoutingAodvGetSeq(rreqPkt->destAddr, &aodv->routeTable) <
//              rreqPkt->destSeq)
//            {
//              RoutingAodvReplaceInsertRouteTable(
//                rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount,
//                rreqPkt->lastAddr, simclock() + REV_ROUTE_LIFE,
//                FALSE, FALSE,
//                &aodv->routeTable);

//              RoutingAodvSetTimer(
//                node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
//                (clocktype)REV_ROUTE_LIFE);

//              if (ttl > 0)
//              {
//                /* Relay the packet only if TTL is not zero */
//                RoutingAodvRelayRREQ(node, msg, ttl);
//              } /* if ttl > 0 */
//              else
//              {
//                GLOMO_MsgFree(node, msg);
```

```
//              }
//            } /* if seq no is not fresh */

//            /* has a fresh route to the destination */
//            else
//            {
//              RoutingAodvReplaceInsertRouteTable(
//                rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount,
//                rreqPkt->lastAddr, simclock() + ACTIVE_ROUTE_TO,
//                TRUE, FALSE,
//                &aodv->routeTable);

//              RoutingAodvSetTimer(
//                node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
//                (clocktype)ACTIVE_ROUTE_TO);

//              /* Send a Route Reply */
//              RoutingAodvInitiateRREPbyIN(node, msg);

//            } /* else */
//          } /* else */

/* ----------------------------------------------------------------------------------------------------not reqd
*/
      } /* else (not dest) */
    } /* if new pkt */

    else  // packet is duplicate
    {
      // GLOMO_MsgFree(node, msg);

          RoutingAodvLookupSeenTable1(node,msg,rreqPkt->srcAddr,rreqPkt->lastAddr,rreqPkt-
>bcastId,rreqPkt->hopCount,rreqPkt->trust_level,rreqPkt->MAC,&aodv->seenTable);

    }
} /* RoutingAodvHandleRequest */

/* ----------------------------------------------------------------------------------------added code for
checking trust_level */
void RoutingAodvLookupSeenTable1(GlomoNode *node,Message *msg,NODE_ADDR
srcAddr,NODE_ADDR lastAddr,int bcastId,int hopCount,int trust_level,long int
MAC,AODV_RST *seenTable)
{
  AODV_RST_Node *current;
  int count1;
  struct Struct2                          /* defining the structure for trust table  */
  {
   NODE_ADDR node;
   int trust_level;
```

```
}AODV_Trust_Table[50];

 AODV_Trust_Table[0].node = 0;
 AODV_Trust_Table[0].trust_level = 5;
 for(count1=1; count1<50; count1++)
 {
       AODV_Trust_Table[count1].node = AODV_Trust_Table[count1-1].node + 1;
       AODV_Trust_Table[count1].trust_level = AODV_Trust_Table[count1-1].trust_level;
 }


       for (current = seenTable->front; current != NULL; current = current->next)
       {
         if (current->srcAddr==srcAddr && current->bcastId==bcastId)
         {
               if (current->hopCount==hopCount -1)
               {
                     for(count1=0;count1<50;count1++)
                     {
                           if (AODV_Trust_Table[count1].node == node->nodeAddr)
                           {
                                 if(trust_level - current->trust_level !=
AODV_Trust_Table[count1].trust_level)
                                 {
                                   RoutingAodvRelayWarning(node,MAC,lastAddr,srcAddr);

                                  }
                                 else
                                 {
                                       GLOMO_MsgFree(node, msg);
                                 }

                           }
                     }
               }

         }
       }
} /* RoutingAodvLookupSeenTable1  */
/* ---------------------------------------------------------------------------- added code for
checking trust_level */


/*
 * RoutingAodvHandleReply
 *
 * Processing procedure when RREP is received
 */
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 relay;
  clocktype lifetime;

  /* clocktype must be copied to access the field of that type */
  memmove(&lifetime, &rrepPkt->lifetime, sizeof(clocktype));

  /* Source of the route */

  if (rrepPkt->srcAddr == node->nodeAddr)
  {
    /* The packet is the first reply received */
    if (!RoutingAodvCheckRouteExist(rrepPkt->destAddr,
                          &aodv->routeTable))
    {
      RoutingAodvReplaceInsertRouteTable(
        rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount, rrepPkt->trust_level,
rrepPkt->next_hop,  /* added parameter */
        srcAddr, simclock() + lifetime, TRUE,
        TRUE, &aodv->routeTable);

      aodv->stats.numRoutes++;
      aodv->stats.numHops += rrepPkt->hopCount;

      RoutingAodvDeleteSent(rrepPkt->destAddr, &aodv->sent);

      /* Send any buffered packets to the destination */
      while (RoutingAodvLookupBuffer(
            rrepPkt->destAddr, &aodv->buffer))
      {
        newMsg = RoutingAodvGetBufferedPacket(
                  rrepPkt->destAddr, &aodv->buffer);

        RoutingAodvTransmitData(node, newMsg, rrepPkt->destAddr);

        aodv->stats.numDataSent++;

        RoutingAodvDeleteBuffer(rrepPkt->destAddr, &aodv->buffer);

      } /* while */
    } /* if no route */

    /* The packet contains a better route compared to the one already
      known */
```

```
    else if ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable) <
            rrepPkt->destSeq) ||
           ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable)
            == rrepPkt->destSeq) &&
            (RoutingAodvGetHopCount(rrepPkt->destAddr,
                        &aodv->routeTable) >
             rrepPkt->hopCount)))
    {
      RoutingAodvReplaceInsertRouteTable(
        rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount, rrepPkt->trust_level,
rrepPkt->next_hop,   /* added parameter */
        srcAddr, simclock() + lifetime, TRUE,
        TRUE, &aodv->routeTable);

      /* Send any buffered packet to the destination */
      while (RoutingAodvLookupBuffer(
            rrepPkt->destAddr, &aodv->buffer))
      {
        newMsg = RoutingAodvGetBufferedPacket(
              rrepPkt->destAddr, &aodv->buffer);

        RoutingAodvTransmitData(node, newMsg, rrepPkt->destAddr);

        aodv->stats.numDataSent++;

        RoutingAodvDeleteBuffer(rrepPkt->destAddr, &aodv->buffer);

      } /* while */
    } /* else if */

    GLOMO_MsgFree(node, msg);
  } /* if source */

  /* Intermediate node of the route */
  else
  {
    /* the packet is the first reply received */
    if (!RoutingAodvCheckRouteExist(
          rrepPkt->destAddr, &aodv->routeTable))
    {
      RoutingAodvReplaceInsertRouteTable(
        rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount, rrepPkt->trust_level,
rrepPkt->next_hop,   /* added parameter */
        srcAddr, simclock() + lifetime, TRUE, FALSE,
        &aodv->routeTable);

      RoutingAodvSetTimer(
        node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->destAddr,
        (clocktype)lifetime);
```

```
        RoutingAodvActivateRoute(rrepPkt->srcAddr, &aodv->routeTable);

        RoutingAodvSetTimer(
          node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->srcAddr,
          (clocktype)ACTIVE_ROUTE_TO);

        /* Forward the packet to the upstream of the route */
        RoutingAodvRelayRREP(node, msg, destAddr);

      } /* if new route */

      /* the packet carries a better route compared to the one already
        known */
      else if ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable) <
              rrepPkt->destSeq) ||
             ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable)
              == rrepPkt->destSeq) &&
              (RoutingAodvGetHopCount(rrepPkt->destAddr,
                              &aodv->routeTable) >
              rrepPkt->hopCount)))
      {
        RoutingAodvReplaceInsertRouteTable(
          rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount, rrepPkt->trust_level,
rrepPkt->next_hop,   /* added parameter */
          srcAddr, simclock() + lifetime, TRUE,
          FALSE, &aodv->routeTable);

        RoutingAodvSetTimer(
          node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->destAddr,
          (clocktype)lifetime);

        RoutingAodvActivateRoute(rrepPkt->srcAddr, &aodv->routeTable);

        RoutingAodvSetTimer(
          node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->srcAddr,
          (clocktype)ACTIVE_ROUTE_TO);

        /* Forward the packet to the upstream of the route */
        RoutingAodvRelayRREP(node, msg, destAddr);

      } /* else if newer route or shorter route */

      else
      {
        GLOMO_MsgFree(node, msg);
      }//if//
    }//if//

} /* RoutingAodvHandleReply */
```

```
//
// RoutingAodvHandleRouteError
//
// Processing procedure when RERR is received
//

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;

  newRerrPacket.pktType = (unsigned short)AODV_RERR;
  newRerrPacket.destinationCount = 0;

  for(I = 0; I < rerrPkt->destinationCount; I++) {
    // Mark the route inactive in the route table; Must not remove it
    // right away since the last hop count known is needed for future use
    // Remove destination from packet if it doesn't need to be forwarded
    // further.

    NODE_ADDR destination =
      rerrPkt->destinationPairArray[I].destinationAddress;
    int sequenceNum =
      rerrPkt->destinationPairArray[I].destinationSequenceNumber;

    BOOL mustRelay =
      RoutingAodvMarkRouteBroken(
        node,
        destination,
        &aodv->routeTable);

    RoutingAodvUpdateSeq(destination,
                sequenceNum,
                &aodv->routeTable);

    NetworkIpDeleteOutboundPacketsToANode(
      node, srcAddr, destination, FALSE);


    if (mustRelay) {
      newRerrPacket.destinationPairArray[newRerrPacket.destinationCount] =
        rerrPkt->destinationPairArray[I];
      newRerrPacket.destinationCount++;
    }//if//
```

```
    }//while//

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

    GLOMO_MsgFree(node, msg);
}//RoutingAodvHandleRouteError//

/* ----------------------------------------------------new function to handle warning message----------
------- */
//
// RoutingAodvHandleWarning
//
// Processing procedure when RWARN is received
//

void RoutingAodvHandleWarning(
    GlomoNode *node, Message *msg, NODE_ADDR srcAddr, NODE_ADDR destAddr, int
trust_level, long int MAC)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    AODV_RWARN_Packet* rwarnPkt =
      (AODV_RWARN_Packet*)GLOMO_MsgReturnPacket(msg);
    AODV_RWARN_Packet newRwarnPacket;
    int countbinary, countdecrypt, s2,y2,d2 ;
    int temp_array[16];
    MAC = 2093;
    trust_level = 5;

    //aodv->stats.encr_message = MAC;                              //to show the received MAC

    /* Process only if the packet is not a duplicate */
    if (!RoutingAodvLookupSeenTable(
            rwarnPkt->srcAddr, rwarnPkt->bcastId, &aodv->seenTable))
    {
          // implement the function here

      /* ---------------------- decrypting MAC---------------------------------------------------- */

     s2 = Public;
     for (countbinary = 0; countbinary <= 15; countbinary ++)
     {
      y2 = s2/2;
      temp_array[countbinary] = s2 % 2;
      s2 = y2;
     }
```

```
    d2 = 1;
    for (countdecrypt = 15; countdecrypt >= 0; countdecrypt --)
    {
     d2 = (d2*d2) % n;
     if (temp_array[countdecrypt] == 1)
     {
            d2 = (d2*MAC) % n;
     }
    }
    aodv->stats.decr_message = d2;                    /* to display decrypted message  */
    // RoutingAodvRelayWarning_1(node);
/* ------------------------------------------------------------------------------------------------------ */

    if (d2 == trust_level)
    {
     // concludes that the accusing node is malicious
     RoutingAodvRelayWarning_1(node);
    }
    else   //concludes that the accused node is malicious
    {
     GLOMO_MsgFree(node, msg);
    }

   }

   else  // packet is duplicate
   {
      GLOMO_MsgFree(node, msg);

   }

}//RoutingAodvHandleWarning//

/* ------------------------------------------------------------------------------------------------------ */

/*
 * RoutingAodvInitRouteTable
 *
 * Initialize the route table
 */
void RoutingAodvInitRouteTable(AODV_RT *routeTable)
{
   routeTable->head = NULL;
   routeTable->size = 0;
} /* RoutingAodvInitRouteTable */

/*
 * RoutingAodvInitNbrTable
 *
```

```
 * Initialize the neighbor table
 */
void RoutingAodvInitNbrTable(AODV_NT *nbrTable)
{
   nbrTable->head = NULL;
   nbrTable->size = 0;

} /* RoutingAodvInitNbrTable */

/*
 * RoutingAodvInitSeenTable
 *
 * Initialize the seen table
 */
void RoutingAodvInitSeenTable(AODV_RST *seenTable)
{
   seenTable->front = NULL;
   seenTable->rear = NULL;
   seenTable->size = 0;

} /* RoutingAodvInitSeenTable */

/*
 * RoutingAodvInitBuffer
 *
 * Initialize the buffer
 */
void RoutingAodvInitBuffer(AODV_BUFFER *buffer)
{
   buffer->head = NULL;
   buffer->size = 0;

} /* RoutingAodvInitBuffer */

/*
 * RoutingAodvInitSent
 *
 * Initialize the sent table
 */
void RoutingAodvInitSent(AODV_SENT *sent)
{
   sent->head = NULL;
   sent->size = 0;

} /* RoutingAodvInitBuffer */

/*
 * RoutingAodvInitStats
 *
```

```
 * Initialize all the stat variables
 */
void RoutingAodvInitStats(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

   aodv->stats.numRequestSent = 0;
   aodv->stats.numReplySent = 0;
   aodv->stats.numWarningSent = 0;
   aodv->stats.numWarning2Sent = 0;
   aodv->stats.numRerrSent = 0;
   aodv->stats.numRerrResent = 0;
   aodv->stats.numDataSent = 0;
   aodv->stats.numDataTxed = 0;
   aodv->stats.numDataReceived = 0;
   aodv->stats.numRoutes = 0;
   aodv->stats.numHops = 0;
   aodv->stats.numPacketsDropped = 0;
   aodv->stats.numBrokenLinks = 0;
   aodv->stats.numBrokenLinkRetries = 0;
} /* RoutingAodvInitStats */

/*
 * RoutingAodvInitSeq
 *
 * Initialize the sequence number
 */
void RoutingAodvInitSeq(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

   aodv->seqNumber = 0;

} /* RoutingAodvInitSeq */

/*
 * RoutingAodvInitBcastId
 *
 * Initialize the broadcast id
 */
void RoutingAodvInitBcastId(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

   aodv->bcastId = 0;
```

```
} /* RoutingAodvInitBcastId */

/*
 * RoutingAodvGetNextHop
 *
 * Looks up the routing table to obtain next hop to the destinaton
 */
NODE_ADDR RoutingAodvGetNextHop(NODE_ADDR destAddr, AODV_RT *routeTable)
{
   AODV_RT_Node *current;

   for (current = routeTable->head;
       current != NULL && current->destAddr <= destAddr;
       current = current->next)
   {
      if (current->destAddr == destAddr && current->activated == TRUE)
      {
         return(current->nextHop);
      }
   }

   return (ANY_DEST);
} /* RoutingAodvGetNextHop */

/*
 * RoutingAodvGetBcastId
 *
 * Obtains the broadcast ID for the outgoing packet
 */
int RoutingAodvGetBcastId(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
   int bcast;

   bcast = aodv->bcastId;
   aodv->bcastId++;

   return (bcast);

} /* RoutingAodvGetBcastId */

/*
 * RoutingAodvGetSeq
 *
 * Obtains the sequence number of the destination node
 */
int RoutingAodvGetSeq(NODE_ADDR destAddr, AODV_RT *routeTable)
{
```

```
    AODV_RT_Node *current;

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

    return (-1);

} /* RoutingAodvGetSeq */

/*
 * RoutingAodvGetMySeq
 *
 * Obtains the node's seq number
 */
int RoutingAodvGetMySeq(GlomoNode *node)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

    return (aodv->seqNumber);

} /* RoutingAodvGetMySeq */

/*
 * RoutingAodvGetHopCount
 *
 * Obtains the hop count to the destination node
 */
int RoutingAodvGetHopCount(NODE_ADDR destAddr, AODV_RT *routeTable)
{
    AODV_RT_Node *current;

    for (current = routeTable->head;
        current != NULL && current->destAddr <= destAddr;
        current = current->next)
    {
      if (current->destAddr == destAddr)
      {
        return(current->hopCount);
      }
    }
```

```
    return (-1);

} /* RoutingAodvGetHopCount */

/*
 * RoutingAodvGetLastHopCount
 *
 * Obtains the last hop count known to the destination node
 */
int RoutingAodvGetLastHopCount(NODE_ADDR destAddr, AODV_RT *routeTable)
{
    AODV_RT_Node *current;

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

    return (-1);

} /* RoutingAodvGetLastHopCount */

/*
 * RoutingAodvGetTtl
 *
 * Obtains the ttl value for the outgoing RREQ
 */
int RoutingAodvGetTtl(NODE_ADDR destAddr, AODV_SENT *sent)
{
    AODV_SENT_Node *current;

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

    return (TTL_START);

} /* RoutingAodvGetTtl */
```

```
/*
 * RoutingAodvGetTimes
 *
 * Obtains the number of times the RREQ was sent in TTL = NET_DIAMETER
 */
int RoutingAodvGetTimes(NODE_ADDR destAddr, AODV_SENT *sent)
{
   AODV_SENT_Node *current;

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

   return (0);

} /* RoutingAodvGetTimes */

/*
 * RoutingAodvGetLifetime
 *
 * Obtains the lifetime value of an entry in the route table
 */
clocktype RoutingAodvGetLifetime(NODE_ADDR destAddr, AODV_RT *routeTable)
{
   AODV_RT_Node *current;

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

   return (0);
} /* RoutingAodvGetLifetime */

/*
 * RoutingAodvGetBufferedPacket
 *
 * Extract the packet that was buffered
```

```
*/
Message *
RoutingAodvGetBufferedPacket(NODE_ADDR destAddr, AODV_BUFFER *buffer)
{
  AODV_BUFFER_Node *current;

  for (current = buffer->head;
      current != NULL && current->destAddr <= destAddr;
      current = current->next)
  {
    if (current->destAddr == destAddr)
    {
      return(current->msg);
    }
  }
  assert(FALSE); abort(); return NULL;

} /* RoutingAodvGetBufferedPacket */

/*
 * RoutingAodvCheckRouteExist
 *
 * Returns TRUE if any route to the destination is known
 */
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->hopCount != AODV_INFINITY) &&
        (current->lifetime > simclock()) &&
        (current->activated == TRUE))
    {
      return(TRUE);
    }
  }

  return (FALSE);

} /* RoutingAodvCheckRouteExist */
```

```
/*
 * RoutingAodvCheckNbrExist
 *
 * Returns TRUE if the node is already a neighbor
 */
BOOL RoutingAodvCheckNbrExist(NODE_ADDR destAddr, AODV_NT *nbrTable)
{
   AODV_NT_Node *current;

   if (nbrTable->size == 0)
   {
      return (FALSE);
   }

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

   return (FALSE);

} /* RoutingAodvCheckNbrExist */

/*
 * RoutingAodvLookupSeenTable
 *
 * Returns TRUE if the broadcast packet is processed before
 */
BOOL RoutingAodvLookupSeenTable(NODE_ADDR srcAddr,
                        int bcastId,
                        AODV_RST *seenTable)
{
   AODV_RST_Node *current;

   if (seenTable->size == 0)
   {
      return (FALSE);
   }

   for (current = seenTable->front;
       current != NULL;
       current = current->next)
   {
      if (current->srcAddr == srcAddr && current->bcastId == bcastId)
```

```
        {
          return (TRUE);
        }
      }

    return (FALSE);

} /* RoutingAodvLookupSeenTable */


/*
 * RoutingAodvLookupBuffer
 *
 * Returns TRUE if any packet is buffered to the destination
 *
 */
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);

} /* RoutingAodvLookupBuffer */

/*
 * RoutingAodvCheckSent
 *
 * Check if RREQ has been sent; return TRUE if sent
 */
BOOL RoutingAodvCheckSent(NODE_ADDR destAddr, AODV_SENT *sent)
{
    AODV_SENT_Node *current;

    if (sent->size == 0)
```

```
    {
      return (FALSE);
    }

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

    return (FALSE);

} /* RoutingAodvCheckSent */



/*
 * RoutingAodvHandleProtocolPacket
 *
 * Called when the packet is received from MAC
 */
void RoutingAodvHandleProtocolPacket(
    GlomoNode *node, Message *msg, NODE_ADDR srcAddr,
    NODE_ADDR destAddr, int ttl, int trust_level, long int MAC)
{
    AODV_PacketType *aodvHeader = (AODV_PacketType*)GLOMO_MsgReturnPacket(msg);

    switch (*aodvHeader)
    {
      case AODV_RREQ:
      {
        RoutingAodvHandleRequest(node, msg, ttl);

        break;
      } /* RREQ */

      case AODV_RREP:
      {
        RoutingAodvHandleReply(node, msg, srcAddr, destAddr);

        break;
      } /* RREP */
      case AODV_RERR:
      {
        assert(destAddr == ANY_DEST);
```

```
        RoutingAodvHandleRouteError(node, msg, srcAddr);

        break;
      } /* RERR */


/*---------------------------------- new code for warning handling ----------------------------------
----------------------*/

      case AODV_RWARN:
      {
        //GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
        //GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
            //aodv->stats.encr_message = MAC;                    /* to display the MAC  */
        RoutingAodvHandleWarning(node, msg, srcAddr, destAddr, trust_level, MAC);

        break;
      } /* RWARN */

/*-------------------------------------------------------------------------------------------------------*/

      default:
        assert(FALSE); abort();
        break;
    } /* switch */
} /* RoutingAodvHandleProtocolPacket */

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

    switch (msg->eventType) {

        /* Remove an entry from the RREQ Seen Table */
        case MSG_NETWORK_FlushTables: {
            RoutingAodvDeleteSeenTable(&aodv->seenTable);
            GLOMO_MsgFree(node, msg);
            break;
        }

        /* Remove the route that has not been used for awhile */
        case MSG_NETWORK_CheckRouteTimeout: {
            NODE_ADDR *destAddr = (NODE_ADDR *)GLOMO_MsgReturnInfo(msg);
```

```
        RoutingAodvDeleteRouteTable(*destAddr, &aodv->routeTable);
        GLOMO_MsgFree(node, msg);

        break;
    }

    /* Check if RREP is received after sending RREQ */
    case MSG_NETWORK_CheckReplied: {
        NODE_ADDR *destAddr = (NODE_ADDR *)GLOMO_MsgReturnInfo(msg);

        /* Route has not been obtained */
        if (!RoutingAodvCheckRouteExist(*destAddr, &aodv->routeTable))
        {
            if (RoutingAodvGetTimes(*destAddr, &aodv->sent) < RREQ_RETRIES)
            {
                /* Retry with increased TTL */
                RoutingAodvRetryRREQ(node, *destAddr);
            } /* if under the retry limit */

            /* over the limit */
            else
            {
                while (RoutingAodvLookupBuffer(*destAddr, &aodv->buffer))
                {
                    Message* messageToDelete =
                        RoutingAodvGetBufferedPacket(
                            *destAddr, &aodv->buffer);
                    RoutingAodvDeleteBuffer(*destAddr, &aodv->buffer);

                    GLOMO_MsgFree(node, messageToDelete);
                    aodv->stats.numPacketsDropped++;
                }
            } /* else */
        } /* if no route */

        GLOMO_MsgFree(node, msg);

        break;
    }

    default:
        fprintf(stderr, "RoutingAodv: Unknown MSG type %d!\n",
                msg->eventType);
        abort();

    } /* switch */

} /* RoutingAodvHandleProtocolEvent */
```

```
/*
 * RoutingAodvRouterFunction
 *
 * Determine the routing action to take for a the given data packet
 * set the PacketWasRouted variable to TRUE if no further handling of
 * this packet by IP is necessary
 */
void RoutingAodvRouterFunction(
   GlomoNode *node,
   Message *msg,
   NODE_ADDR destAddr,
   BOOL *packetWasRouted)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
   IpHeaderType *ipHeader = (IpHeaderType *) msg->packet;

   /* Control packets */
   if (ipHeader->ip_p  == IPPROTO_AODV)
   {
      return;
   }

   if (destAddr == node->nodeAddr)
   {
      *packetWasRouted = FALSE;
   }
   else
   {
      *packetWasRouted = TRUE;
   }

   /* intermediate node or destination of the route */
   if (ipHeader->ip_src != node->nodeAddr)
   {
      RoutingAodvHandleData(node, msg, destAddr);
   }

   /* source has a route to the destination */
   else if (RoutingAodvCheckRouteExist(destAddr, &aodv->routeTable))
   {
      RoutingAodvTransmitData(node, msg, destAddr);
      aodv->stats.numDataSent++;
   }

   /* There is no route to the destination and RREQ has not been sent */
   else if (!RoutingAodvLookupBuffer(destAddr, &aodv->buffer))
   {
      RoutingAodvInsertBuffer(msg, destAddr, &aodv->buffer);
```

```
    RoutingAodvInitiateRREQ(node, destAddr);
  }

  /* There is no route but RREQ has already been sent */
  else
  {
    RoutingAodvInsertBuffer(msg, destAddr, &aodv->buffer);
  }

} /* RoutingAodvRouterFunction */



/*
 * RoutingAodvMacLayerStatusHandler
 *
 * Reacts to the signal sent by the MAC protocol after link failure
 */
void RoutingAodvPacketDropNotificationHandler(
  GlomoNode *node, const Message* msg, const NODE_ADDR nextHopAddress)
{
  GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
  GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

  IpHeaderType* ipHeader;
  NODE_ADDR destAddr;
  int numberRouteDestinations;

  ipHeader = (IpHeaderType *) GLOMO_MsgReturnPacket(msg);

  if (ipHeader->ip_p  == IPPROTO_AODV)
  {
    return;
  }//if//


  destAddr = ipHeader->ip_dst;

  if (nextHopAddress == ANY_DEST) {
    aodv->stats.numBrokenLinkRetries++;
    return;
  }//if//

  NetworkIpDeleteOutboundPacketsToANode(
    node, nextHopAddress, ANY_DEST, FALSE);

  aodv->stats.numBrokenLinks++;

  RoutingAodvDeleteNbrTable(nextHopAddress, &aodv->nbrTable);
```

```
  RoutingAodvIncreaseSeq(node);

  do {
    AODV_RERR_Packet newRerrPacket;
    newRerrPacket.pktType = AODV_RERR;

    RoutingAodvInactivateRoutesAndGetDestinations(
      node,
      &aodv->routeTable,
      nextHopAddress,
      newRerrPacket.destinationPairArray,
      AODV_MAX_RERR_DESTINATIONS,
      &numberRouteDestinations);

    newRerrPacket.destinationCount = numberRouteDestinations;

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

  } while (numberRouteDestinations == AODV_MAX_RERR_DESTINATIONS);

}//RoutingAodvMaclayerStatusHandler//



/*
 * RoutingAodvSetTimer
 *
 * Set timers for protocol events
 */
void RoutingAodvSetTimer(
  GlomoNode *node, long eventType, NODE_ADDR destAddr, clocktype delay)
{
  Message *newMsg;
  NODE_ADDR *info;

  newMsg = GLOMO_MsgAlloc(node,
                GLOMO_NETWORK_LAYER,
                ROUTING_PROTOCOL_AODV,
                eventType);

  GLOMO_MsgInfoAlloc(node, newMsg, sizeof(NODE_ADDR));
  info = (NODE_ADDR *) GLOMO_MsgReturnInfo(newMsg);
  *info = destAddr;
  GLOMO_MsgSend(node, newMsg, delay);

} /* RoutingAodvSetTimer */
```

```
/*
 * RoutingAodvInitiateRREQ
 *
 * Initiate a Route Request packet when no route to destination is known
 */
void RoutingAodvInitiateRREQ(GlomoNode *node, NODE_ADDR destAddr)
{
  GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
  GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
  Message *newMsg;
  AODV_RREQ_Packet *rreqPkt;
  char *pktPtr;
  int pktSize = sizeof(AODV_RREQ_Packet);
  int ttl;

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

  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->srcAddr = node->nodeAddr;
  rreqPkt->srcSeq = RoutingAodvGetMySeq(node);
  rreqPkt->lastAddr = node->nodeAddr;
  rreqPkt->hopCount = 1;
  rreqPkt->trust_level = 0;                    /* added parameter */
  rreqPkt->MAC = 0;                            /* added parameter */

  if (RoutingAodvCheckSent(destAddr, &aodv->sent))
  {
    ttl = RoutingAodvGetTtl(destAddr, &aodv->sent);
    RoutingAodvIncreaseTtl(destAddr, &aodv->sent);
  }
  else
  {
    ttl = RoutingAodvGetLastHopCount(destAddr, &aodv->routeTable);

    if (ttl == -1)
    {
      ttl = TTL_START;
    }

    RoutingAodvInsertSent(destAddr, ttl, &aodv->sent);
```

```
    RoutingAodvIncreaseTtl(destAddr, &aodv->sent);
  }

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

  aodv->stats.numRequestSent++;

  RoutingAodvInsertSeenTable(
          node, node->nodeAddr, rreqPkt->bcastId, rreqPkt->hopCount, rreqPkt->trust_level,
&aodv->seenTable);   /* added parameter  */

  RoutingAodvSetTimer(node, MSG_NETWORK_CheckReplied, destAddr,
              (clocktype)2 * ttl * NODE_TRAVERSAL_TIME);

} /* RoutingAodvInitiateRREQ */

/*
 * RoutingAodvRetryRREQ
 *
 * Send RREQ again after not receiving any RREP
 */
void RoutingAodvRetryRREQ(GlomoNode *node, NODE_ADDR destAddr)
{
  GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
  GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
  Message *newMsg;
  AODV_RREQ_Packet *rreqPkt;
  char *pktPtr;
  int pktSize = sizeof(AODV_RREQ_Packet);
  int ttl;

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

  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->srcAddr = node->nodeAddr;
  rreqPkt->srcSeq = RoutingAodvGetMySeq(node);
  rreqPkt->lastAddr = node->nodeAddr;
  rreqPkt->hopCount = 1;
  rreqPkt->trust_level = 0;                    /* added parameter */
  rreqPkt->MAC = 0;                            /* added parameter */
```

```
    ttl = RoutingAodvGetTtl(destAddr, &aodv->sent);

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

    RoutingAodvIncreaseTtl(destAddr, &aodv->sent);

    aodv->stats.numRequestSent++;

    RoutingAodvInsertSeenTable(
            node, node->nodeAddr, rreqPkt->bcastId, rreqPkt->hopCount, rreqPkt->trust_level,
&aodv->seenTable);  /*  added parameter  */

    if (ttl == NET_DIAMETER)
    {
      RoutingAodvIncreaseTimes(destAddr, &aodv->sent);
    }

    RoutingAodvSetTimer(node, MSG_NETWORK_CheckReplied, destAddr,
                (clocktype)2 * ttl * NODE_TRAVERSAL_TIME);

} /* RoutingAodvRetryRREQ */

/*
 * RoutingAodvTransmitData
 *
 * Forward the data packet to the next hop
 */
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);

    NetworkIpSendPacketToMacLayer(node, msg, DEFAULT_INTERFACE, nextHop);
    aodv->stats.numDataTxed++;

    RoutingAodvUpdateLifetime(destAddr, &aodv->routeTable);

    RoutingAodvSetTimer(node, MSG_NETWORK_CheckRouteTimeout,
                    destAddr, (clocktype)ACTIVE_ROUTE_TO);

} /* RoutingAodvTransmitData */
```

```
/*
 * RoutingAodvRelayRREQ
 *
 * Forward (re-broadcast) the RREQ
 */
void RoutingAodvRelayRREQ(GlomoNode *node, Message *msg, int ttl)
{
   struct Struct1                              /* defining the structure for trust table  */
   {
    NODE_ADDR node;
    int trust_level;
   } AODV_Trust_Table[50];

   NODE_ADDR recd_from_node;                                                    /*
added parameter  */
   int count, countbinary, countencrypt, s1,y1,d1 ;                             /*  added
parameter  */
   int temp_array[16];                                                          /*  added
parameter  */
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
   Message *newMsg;
   AODV_RREQ_Packet *oldRreq;
   AODV_RREQ_Packet *newRreq;
   char *pktPtr;
   int pktSize = sizeof(AODV_RREQ_Packet);
   clocktype delay;
   //AODV_Trust_Table *trustTable;                                  /*  added parameter  */
/* ----------------------------------------------------------------------------------------------new function
*/
  // void initTrustTable()
  // {
        AODV_Trust_Table[0].node = 0;
        AODV_Trust_Table[0].trust_level = 5;
        for (count=1; count<50; count++)
        {
              AODV_Trust_Table[count].node = AODV_Trust_Table[count-1].node +1;
              AODV_Trust_Table[count].trust_level = AODV_Trust_Table[count-
1].trust_level;
      }

  //}  //end of function initTrustTable
/* ---------------------------------------------------------------------------------------------------------- */
   oldRreq = (AODV_RREQ_Packet *) 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 = (AODV_RREQ_Packet *) pktPtr;

    recd_from_node = oldRreq->lastAddr;                                    /*
added parameter  */
    newRreq->pktType = oldRreq->pktType;
    newRreq->bcastId = oldRreq->bcastId;
    newRreq->destAddr = oldRreq->destAddr;
    newRreq->destSeq = oldRreq->destSeq;
    newRreq->srcAddr = oldRreq->srcAddr;
    newRreq->srcSeq = oldRreq->srcSeq;
    newRreq->lastAddr = node->nodeAddr;
    newRreq->hopCount = oldRreq->hopCount + 1;
/* ----------------------------------------------------------------------------------------------added code */
    for (count=0; count<50; count++)
    {
     if( AODV_Trust_Table[count].node == recd_from_node)
     {
          newRreq->trust_level = oldRreq->trust_level + AODV_Trust_Table[count].trust_level;
     }
    }
/* ------------------------------------------------------------------------------------------------ */


/* ----------------------new code for computing MAC---------------------------------------------- */

    s1 = Private;
    for (countbinary = 0; countbinary <= 15; countbinary ++)
    {
     y1 = s1/2;
     temp_array[countbinary] = s1%2;
     s1 = y1;
    }

    d1 = 1;
    for (countencrypt = 15; countencrypt >= 0; countencrypt --)
    {
     d1 = (d1*d1) % n;
     if (temp_array[countencrypt] == 1)
     {
          d1 = (d1*message) % n;
     }
    }
    newRreq->MAC = d1;
    // aodv->stats.encr_message = d1;                          /* to display encrypted message  */

 /* ------------------------------------------------------------------------------------------------ */
```

```
    delay = pc_erand(node->seed) * BROADCAST_JITTER;

    NetworkIpSendRawGlomoMessageWithDelay(
                node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, ttl, delay);

    aodv->stats.numRequestSent++;

    GLOMO_MsgFree(node, msg);

} /* RoutingAodvRelayRREQ */

/*
 * RoutingAodvInitiateRREP
 *
 * Destination of the route sends RREP in reaction to RREQ
 */
void RoutingAodvInitiateRREP(GlomoNode *node, Message *msg)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *)ipLayer->routingProtocol;
    Message *newMsg;
    AODV_RREQ_Packet *rreqPkt;
    AODV_RREP_Packet *rrepPkt;
    char *pktPtr;
    int pktSize = sizeof(AODV_RREP_Packet);
    int seq;

    rreqPkt = (AODV_RREQ_Packet *) 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 = (AODV_RREP_Packet *) pktPtr;

    rrepPkt->pktType = AODV_RREP;
    rrepPkt->srcAddr = rreqPkt->srcAddr;
    rrepPkt->destAddr = node->nodeAddr;
    seq = RoutingAodvGetMySeq(node);
    if (seq >= rreqPkt->destSeq)
    {
        rrepPkt->destSeq = seq;
    }
    else
    {
        rrepPkt->destSeq = rreqPkt->destSeq;
        RoutingAodvIncreaseSeq(node);
    }
```

```
    rrepPkt->hopCount = 1;
    rrepPkt->trust_level = rreqPkt->trust_level;            /* parameter added */
    rrepPkt->next_hop = 0;                                          /* parameter added */
    rrepPkt->lastAddr = node->nodeAddr;                     /* parameter added */

    rrepPkt->lifetime = (clocktype)MY_ROUTE_TO;

    NetworkIpSendRawGlomoMessageToMacLayer(
        node, newMsg, rreqPkt->lastAddr, CONTROL, IPPROTO_AODV, 1,
        DEFAULT_INTERFACE, rreqPkt->lastAddr);

    aodv->stats.numReplySent++;

    GLOMO_MsgFree(node, msg);

} /* RoutingAodvInitiateRREP */

/* ----------------------------------------------------------------------------------------------not reqd
*/
/*
 * RoutingAodvInitiateRREPbyIN
 *
 * An intermediate node that knows the route to the destination sends the RREP
 */

//void RoutingAodvInitiateRREPbyIN(GlomoNode *node, Message *msg)
//{
//    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)node->networkData.networkVar;
//    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *)ipLayer->routingProtocol;
//    Message *newMsg;
//    AODV_RREQ_Packet *rreqPkt;
//    AODV_RREP_Packet *rrepPkt;
//    char *pktPtr;
//    int pktSize = sizeof(AODV_RREP_Packet);
//    int seq;

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

//    pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
//    rrepPkt = (AODV_RREP_Packet *) pktPtr;

//    rreqPkt = (AODV_RREQ_Packet *) GLOMO_MsgReturnPacket(msg);

//    rrepPkt->pktType = AODV_RREP;
//    rrepPkt->srcAddr = rreqPkt->srcAddr;
//    rrepPkt->destAddr = rreqPkt->destAddr;
//    rrepPkt->destSeq = RoutingAodvGetSeq(rreqPkt->destAddr, &aodv->routeTable);
```

```
//   rrepPkt->lifetime = RoutingAodvGetLifetime(
//                       rreqPkt->destAddr, &aodv->routeTable) - simclock();
//   rrepPkt->hopCount =  RoutingAodvGetHopCount(
//                        rreqPkt->destAddr, &aodv->routeTable) + 1;

//   NetworkIpSendRawGlomoMessageToMacLayer(
//      node, newMsg, rreqPkt->lastAddr, CONTROL, IPPROTO_AODV, 1,
//      DEFAULT_INTERFACE, rreqPkt->lastAddr);

//   aodv->stats.numReplySent++;

//   GLOMO_MsgFree(node, msg);
//} /* RoutingAodvInitiateRREPbyIN */

/* ----------------------------------------------------------------------------------------------------not
reqd */

/*
 * RoutingAodvRelayRREP
 *
 * Forward the RREP packet
 */
void RoutingAodvRelayRREP(GlomoNode *node, Message *msg, NODE_ADDR destAddr)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
   Message *newMsg;
   AODV_RREP_Packet *oldRrep;
   AODV_RREP_Packet *newRrep;
   char *pktPtr;
   NODE_ADDR nextHop;
   clocktype lifetime;
   int pktSize = sizeof(AODV_RREP_Packet);

   oldRrep = (AODV_RREP_Packet *) GLOMO_MsgReturnPacket(msg);

   memmove(&lifetime, &oldRrep->lifetime, sizeof(clocktype));

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

   pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
   newRrep = (AODV_RREP_Packet *) pktPtr;

   newRrep->pktType = oldRrep->pktType;
   newRrep->srcAddr = oldRrep->srcAddr;
   newRrep->destAddr = oldRrep->destAddr;
   newRrep->destSeq = oldRrep->destSeq;
```

```
    newRrep->hopCount = oldRrep->hopCount + 1;
    newRrep->trust_level = oldRrep->trust_level;                    /* parameter added  */
    newRrep->next_hop = oldRrep->lastAddr;                           /* parameter added  */
    newRrep->lastAddr = node->nodeAddr;                              /* parameter added  */
    newRrep->lifetime = lifetime;

    if (destAddr == ANY_DEST)
    {
      NetworkIpSendRawGlomoMessage(
        node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, 1);
    }
    else
    {
      nextHop = RoutingAodvGetNextHop(oldRrep->srcAddr, &aodv->routeTable);

      if (nextHop != ANY_DEST)
      {
        NetworkIpSendRawGlomoMessageToMacLayer(
          node, newMsg, nextHop, CONTROL, IPPROTO_AODV, 1,
          DEFAULT_INTERFACE, nextHop);
      }
    }

    // aodv->stats.numReplySent++;

    GLOMO_MsgFree(node, msg);

} /* RoutingAodvRelayRREP */

/* ---------------------------------------------------------------------function added----------------------
----*/
/*
 * RoutingAodvRelayWarning
 *
 * Broadcast the warning message
 */
void RoutingAodvRelayWarning(GlomoNode *node, long int MAC, NODE_ADDR lastAddr,
NODE_ADDR srcAddr)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

    Message *newMsg;
    AODV_RWARN_Packet *rwarnPkt;
    char *pktPtr;
    int pktSize = sizeof(AODV_RWARN_Packet);
    // int ttl;
```

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

    pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
    rwarnPkt = (AODV_RWARN_Packet *) pktPtr;

    rwarnPkt->pktType = AODV_RWARN;
    rwarnPkt->bcastId = RoutingAodvGetBcastId(node);
    rwarnPkt->srcSeq = RoutingAodvGetMySeq(node);
    rwarnPkt->maliciousIP = lastAddr;
    //rwarnPkt->trust_level = trust_level;
    rwarnPkt->MAC = MAC;
    rwarnPkt->rwarn_sourceIP = node->nodeAddr;
    rwarnPkt->srcAddr = srcAddr;

    //aodv->stats.encr_message = rwarnPkt->MAC;                    /* to display the MAC  */

    NetworkIpSendRawGlomoMessage(
        node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, rwarnPkt->MAC);

    aodv->stats.numWarningSent++;
} /* RoutingAodvRelayWarning */

/* ----------------------------------------------------------------------------------------------------
------------ */


// ------------------------------------------------------------------------------- function added
/*
 * RoutingAodvRelayWarning_1
 *
 * Broadcast the warning message
 */
void RoutingAodvRelayWarning_1(GlomoNode *node)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

    aodv->stats.numWarning2Sent++;
} /* RoutingAodvRelayWarning_1 */

// -------------------------------------------------------------------------------
```

APPENDIX C
Header File of Simulation for Trust Modeling

```
/*
 *Edited and Modified By Tirthankar Ghosh and Ahmad Farhat
 *
 * GloMoSim is COPYRIGHTED software.  Release 2.02 of GloMoSim is available
 * at no cost to educational users only.
 *
 * Commercial use of this software requires a separate license.  No cost,
 * evaluation licenses are available for such purposes; please contact
 * info@scalable-networks.com
 *
 * By obtaining copies of this and any other files that comprise GloMoSim2.02,
 * you, the Licensee, agree to abide by the following conditions and
 * understandings with respect to the copyrighted software:
 *
 * 1.Permission to use, copy, and modify this software and its documentation
 *   for education and non-commercial research purposes only is hereby granted
 *   to Licensee, provided that the copyright notice, the original author's
 *   names and unit identification, and this permission notice appear on all
 *   such copies, and that no charge be made for such copies. Any entity
 *   desiring permission to use this software for any commercial or
 *   non-educational research purposes should contact:
 *
 *   Professor Rajive Bagrodia
 *   University of California, Los Angeles
 *   Department of Computer Science
 *   Box 951596
 *   3532 Boelter Hall
 *   Los Angeles, CA 90095-1596
 *   rajive@cs.ucla.edu
 *
 * 2.NO REPRESENTATIONS ARE MADE ABOUT THE SUITABILITY OF THE
SOFTWARE FOR ANY
 *   PURPOSE. IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY.
 *
 * 3.Neither the software developers, the Parallel Computing Lab, UCLA, or any
 *   affiliate of the UC system shall be liable for any damages suffered by
 *   Licensee from the use of this software.
 */

// Use the latest version of Parsec if this line causes a compiler error.
/*
 * Name: aodv.h
 *
 * Implemented by SJ Lee (sjlee@cs.ucla.edu)
 */
```

```
/*
 NOTE: The parameter values followed the AODV Internet Draft
     (draft-ietf-manet-aodv-03.txt) and NS2 code by Samir R. Das
     Read the NOTE of aodv.pc for more details
*/

#ifndef _AODV_H_
#define _AODV_H_

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

#define ACTIVE_ROUTE_TO                 10 * SECOND

#define NODE_TRAVERSAL_TIME             40 * MILLI_SECOND

#define NET_DIAMETER                    35

#define RREP_WAIT_TIME          3 * NODE_TRAVERSAL_TIME * NET_DIAMETER / 2

#define BAD_LINK_LIFETIME               2 * RREP_WAIT_TIME

#define BCAST_ID_SAVE                   30 * SECOND

#define REV_ROUTE_LIFE                  RREP_WAIT_TIME

#define MY_ROUTE_TO                     2 * ACTIVE_ROUTE_TO

#define RREQ_RETRIES                    2

#define TTL_START                       1

#define TTL_INCREMENT                   2

#define TTL_THRESHOLD                   7

#define AODV_INFINITY                   255

#define BROADCAST_JITTER                10 * MILLI_SECOND

/* Packet Types */


typedef unsigned char AODV_PacketType;

#define AODV_RREQ 0
#define AODV_RREP 1
```

```
#define AODV_RERR 2

typedef struct
{
   AODV_PacketType pktType;
   int bcastId;
   NODE_ADDR destAddr;
   int destSeq;
   NODE_ADDR srcAddr;
   int srcSeq;
   NODE_ADDR lastAddr;
   int hopCount;
   float conf_Ratio;/*****************************/
   float conf_Level;/*****************************/
   NODE_ADDR conf_Node;/*****************************/

} AODV_RREQ_Packet;

typedef struct
{
   AODV_PacketType pktType;
   NODE_ADDR srcAddr;
   NODE_ADDR destAddr;
   int destSeq;
   int hopCount;
   clocktype lifetime;
} AODV_RREP_Packet;


typedef struct
{
 NODE_ADDR destinationAddress;
 int destinationSequenceNumber;
} AODV_AddressSequenceNumberPairType;

#define AODV_MAX_RERR_DESTINATIONS 250

typedef struct
{
   AODV_PacketType pktType;        // 1 byte
   unsigned char filling[2];
   unsigned char destinationCount;
   AODV_AddressSequenceNumberPairType
     destinationPairArray[AODV_MAX_RERR_DESTINATIONS];
} AODV_RERR_Packet;


static //inline//
int AODV_RERR_PacketSize(const AODV_RERR_Packet* rerrPacket) {
```

```
  return
    (sizeof(rerrPacket->pktType) +
     sizeof(rerrPacket->filling) +
     sizeof(rerrPacket->destinationCount) +
     (rerrPacket->destinationCount *
      sizeof(AODV_AddressSequenceNumberPairType)));
}



typedef struct RTE
{
   NODE_ADDR destAddr;
   int destSeq;
   int hopCount;
   int lastHopCount;
   NODE_ADDR nextHop;
   clocktype lifetime;
   BOOL activated;
   BOOL source;
   struct RTE *next;
} AODV_RT_Node;

typedef struct
{
   AODV_RT_Node *head;
   int size;
} AODV_RT;

typedef struct NTE
{
   NODE_ADDR destAddr;
   struct NTE *next;
} AODV_NT_Node;

typedef struct
{
   AODV_NT_Node *head;
   int size;
} AODV_NT;

typedef struct RSE
{
   NODE_ADDR srcAddr;
   int bcastId;
   struct RSE *next;
   int hopCount;
   NODE_ADDR lastAddr;
} AODV_RST_Node;
```

```
typedef struct
{
   AODV_RST_Node *front;
   AODV_RST_Node *rear;
   int size;
} AODV_RST;

/************************Confidence Table Structure*****************************/
typedef struct CFE
{
  NODE_ADDR nodeAddr;
  float conf_Level;/*C*/
  float conf_Ratio;/*G*/
  float OER;
  struct CFE *next;

} AODV_CFT_Node;

typedef struct
{
  AODV_CFT_Node *front;
  AODV_CFT_Node *rear;
  int size;
} AODV_CFT;
/************************Confidence Table Structure*****************************/

/************************Remove Table Structure*********************************/
typedef struct NRE
{
  NODE_ADDR nodeAddr;
  int TimeOfLeaving;
  struct NRE *next;

}AODV_NRT_Node;

typedef struct
{
  AODV_NRT_Node *front;
  AODV_NRT_Node *rear;
  int size;

}AODV_NRT;
/************************Remove Table Structure*********************************/
typedef struct FIFO
{
   NODE_ADDR destAddr;
   clocktype timestamp;
   Message *msg;
   struct FIFO *next;
```

```
} AODV_BUFFER_Node;

typedef struct
{
   AODV_BUFFER_Node *head;
   int size;
} AODV_BUFFER;

typedef struct SE
{
   NODE_ADDR destAddr;
   int ttl;
   int times;
   struct SE *next;
} AODV_SENT_Node;

typedef struct
{
   AODV_SENT_Node *head;
   int size;
} AODV_SENT;

typedef struct
{
   int numRequestSent;
   int numReplySent;
   int numRerrSent;
   int numRwarnSent;   /* **************************new stat  *************** */
   int numRerrResent;
   int numDataSent;       /* Data Sent as the source of the route */
   int numDataTxed;
   int numDataReceived;   /* Data Received as the destination of the route */
   int numHops;
   int numRoutes;
   int numPacketsDropped;
   int numBrokenLinks;
   int numBrokenLinkRetries;
} AODV_Stats;

typedef struct glomo_network_aodv_str
{
   AODV_CFT confTable;/***********Confidence Table****************/
   AODV_NRT NbrRmvTable;/**********Neighbour Remove Table********/
   AODV_RT routeTable;
   AODV_NT nbrTable;
   AODV_RST seenTable;
   AODV_BUFFER buffer;
   AODV_SENT sent;
   AODV_Stats stats;
```

```
  int seqNumber;
  int bcastId;


} GlomoRoutingAodv;

void RoutingAodvInit(
  GlomoNode *node,
  GlomoRoutingAodv **aodvPtr,
  const GlomoNodeInput *nodeInput);

void RoutingAodvFinalize(GlomoNode *node);

void RoutingAodvHandleData(GlomoNode *node, Message *msg, NODE_ADDR destAddr);

void RoutingAodvHandleRequest(GlomoNode *node, Message *msg, int ttl);

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

void RoutingAodvInitRouteTable(AODV_RT *routeTable);

void RoutingAodvInitNbrTable(AODV_NT *nbrTable);

void RoutingAodvInitSeenTable(AODV_RST *seenTable);

void RoutingAodvInitConfTable(AODV_CFT *confTable);

void RoutingAodvInsertNbrRmvTable(NODE_ADDR destAddr ,AODV_NRT *NbrRmvTable);

float RoutingAodvComputeMean(NODE_ADDR targetAddr, AODV_NRT *NbrRmvTable);

void RoutingAodvInitNbrRmvTable(AODV_NRT *NbrRmvTable);

void RoutingAodvInitBuffer(AODV_BUFFER *buffer);

void RoutingAodvInitSent(AODV_SENT *sent);

void RoutingAodvInitStats(GlomoNode *node);

void RoutingAodvInitSeq(GlomoNode *node);

void RoutingAodvInitBcastId(GlomoNode *node);

NODE_ADDR RoutingAodvGetNextHop(NODE_ADDR destAddr, AODV_RT *routeTable);

int RoutingAodvGetBcastId(GlomoNode *node);

int RoutingAodvGetSeq(NODE_ADDR destAddr, AODV_RT *routeTable);
```

```
int RoutingAodvGetMySeq(GlomoNode *node);

int RoutingAodvGetHopCount(NODE_ADDR destAddr, AODV_RT *routeTable);

int RoutingAodvGetLastHopCount(NODE_ADDR destAddr, AODV_RT *routeTable);

int RoutingAodvGetTtl(NODE_ADDR destAddr, AODV_SENT *sent);

int RoutingAodvGetTimes(NODE_ADDR destAddr, AODV_SENT *sent);

clocktype RoutingAodvGetLifetime(NODE_ADDR destAddr, AODV_RT *routeTable);

Message *
RoutingAodvGetBufferedPacket(NODE_ADDR destAddr, AODV_BUFFER *buffer);

BOOL RoutingAodvCheckRouteExist(NODE_ADDR destAddr, AODV_RT *routeTable);

BOOL RoutingAodvCheckNbrExist(NODE_ADDR destAddr, AODV_NT *nbrTable);

BOOL RoutingAodvLookupSeenTable(GlomoNode *node,
                                NODE_ADDR srcAddr,
                    int bcastId,int hopCount,    /* added parameter   */
                    NODE_ADDR lastAddr,
                    AODV_RST *seenTable,
                    AODV_CFT *confTable);

/************************Conftable function
definition*******************************/
BOOL RoutingAodvComputeConfTable(GlomoNode *node,NODE_ADDR lastAddr, int
HFWD, int RTF,AODV_CFT* confTable);
BOOL RoutingAodvUpdateConfTable(NODE_ADDR address, float confRatio,float
confLevel,NODE_ADDR confNode,AODV_CFT* confTable);
void RoutingAodvInsertConfTable(GlomoNode *node,NODE_ADDR lastAddr,AODV_CFT
*confTable);
/************************Conftable function
definition*******************************/

BOOL RoutingAodvLookupBuffer(NODE_ADDR destAddr, AODV_BUFFER *buffer);

BOOL RoutingAodvCheckSent(NODE_ADDR destAddr, AODV_SENT *sent);


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

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

void RoutingAodvRouterFunction(
```

```
    GlomoNode *node,
    Message *msg,
    NODE_ADDR destAddr,
    BOOL *packetWasRouted);

void RoutingAodvPacketDropNotificationHandler(
  GlomoNode *node, const Message* msg, const NODE_ADDR nextHopAddress);

void RoutingAodvSetTimer(
    GlomoNode *node, long eventType, NODE_ADDR destAddr, clocktype delay);

void RoutingAodvInitiateRREQ(GlomoNode *node, NODE_ADDR destAddr);

void RoutingAodvRetryRREQ(GlomoNode *node, NODE_ADDR destAddr);

void RoutingAodvTransmitData(GlomoNode *node, Message *msg, NODE_ADDR destAddr);

void RoutingAodvRelayRREQ(GlomoNode *node, Message *msg, int ttl);

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

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

void RoutingAodvRelayRREP(GlomoNode *node, Message *msg, NODE_ADDR destAddr);

#endif /* _AODV_H_ */
```

APPENDIX D
Sample Code of Simulation for Trust Modeling

```
/*
 *Edited and Modified By Tirthankar Ghosh and Ahmad Farhat
 *
 * GloMoSim is COPYRIGHTED software.  Release 2.02 of GloMoSim is available
 * at no cost to educational users only.
 *
 * Commercial use of this software requires a separate license.  No cost,
 * evaluation licenses are available for such purposes; please contact
 * info@scalable-networks.com
 *
 * By obtaining copies of this and any other files that comprise GloMoSim2.02,
 * you, the Licensee, agree to abide by the following conditions and
 * understandings with respect to the copyrighted software:
 *
 * 1.Permission to use, copy, and modify this software and its documentation
 *   for education and non-commercial research purposes only is hereby granted
 *   to Licensee, provided that the copyright notice, the original author's
 *   names and unit identification, and this permission notice appear on all
 *   such copies, and that no charge be made for such copies. Any entity
 *   desiring permission to use this software for any commercial or
 *   non-educational research purposes should contact:
 *
 *   Professor Rajive Bagrodia
 *   University of California, Los Angeles
 *   Department of Computer Science
 *   Box 951596
 *   3532 Boelter Hall
 *   Los Angeles, CA 90095-1596
 *   rajive@cs.ucla.edu
 *
 * 2.NO REPRESENTATIONS ARE MADE ABOUT THE SUITABILITY OF THE
SOFTWARE FOR ANY
 *   PURPOSE. IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY.
 *
 * 3.Neither the software developers, the Parallel Computing Lab, UCLA, or any
 *   affiliate of the UC system shall be liable for any damages suffered by
 *   Licensee from the use of this software.
 */

// Use the latest version of Parsec if this line causes a compiler error.
/*
 * Name: aodv.pc
 *
 * Implemented by SJ Lee (sjlee@cs.ucla.edu)
 */
```

```
/*
 NOTE: - Followed the specification of AODV Internet Draft
       (draft-ietf-manet-aodv-03.txt)
     - This implements only unicast functionality of AODV.
     - Assumes the MAC protocol sends a signal to the routing protocol
       when it detects link breaks. MAC protocols such as IEEE 802.11
       and MACAW has this functionality. In IEEE 802.11, when no CTS
       is received after RTS, and no ACK is received after retransmissions
       of unicasted packet, it sends the signal to the routing protocol
     - If users want to use MAC protocols other than IEEE 802.11, they
       must implement schemes to detect link breaks. A way to do this is,
       for example, using HELLO packets, as specified in AODV documents.
     - No Precursors (Implemented other mechanism so that the protocol can
       still function the same as when precursors are used)
     - Unsolicited RREPs are broadcasted and forwarded only if the node
       is part of the broken route and not the source of that route
     - If more than one route uses the broken link, send RREP multiple times
       (this should be fixed based on new specification by C. Perkins,
       E. Royer, and S. Das)
     - Rev route of RREQ overwrites the one in the route table
     - May need slight modifications when draft-ietf-manet-aodv-04.txt
       comes out
*/

//#include <iostream.h>

#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"

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

/* ***********global variables********************************************** */
int requestToForward = 0; /*holds number of requests when RREQ is initiated*/
```

```
int time_of_leaving = 0;/*used in insertNbrRmvTable function*/
/* ***********global variables************************************************** */

/*
 * RoutingAodvReplaceInsertRouteTable
 *
 * Insert/Update an entry into the route table
 */

static void
RoutingAodvReplaceInsertRouteTable(
   NODE_ADDR destAddr,
   int destSeq,
   int hopCount,
   NODE_ADDR nextHop,
   clocktype lifetime,
   BOOL activated,
   BOOL source,
   AODV_RT* routeTable)
{
   AODV_RT_Node* theNode = NULL;
   AODV_RT_Node* current;
   AODV_RT_Node* previous;

   // Find Insertion point.

   previous = NULL;
   current = routeTable->head;
   while ((current != NULL) && (current->destAddr < destAddr)) {
      previous = current;
      current = current->next;
   }//while//


   if ((current == NULL) || (current->destAddr != destAddr)) {
      ++(routeTable->size);

      theNode = (AODV_RT_Node *)checked_pc_malloc(sizeof(AODV_RT_Node));
      theNode->lifetime = lifetime;
      theNode->activated = activated;
      theNode->source = source;
      theNode->destAddr = destAddr;

      if (previous == NULL) {
        theNode->next = routeTable->head;
        routeTable->head = theNode;
      } else {
        theNode->next = previous->next;
        previous->next = theNode;
```

```
    }//if//


  } else {
    assert(current->destAddr == destAddr);

    current->lifetime = max(lifetime, current->lifetime);
    if (!current->activated) {
      current->activated = activated;
    }//if//

    if (!current->source) {
      current->source = source;
    }//if//

    theNode = current;
  }//if//

  theNode->destSeq = destSeq;
  theNode->hopCount = hopCount;
  theNode->lastHopCount = hopCount;
  theNode->nextHop = nextHop;

}//RoutingAodvReplaceInsertRouteTable//



static
void RoutingAodvInsertNbrTable(NODE_ADDR destAddr, AODV_NT *nbrTable)
{
  AODV_NT_Node* current;
  AODV_NT_Node* previous;

  AODV_NT_Node* newNode =
    (AODV_NT_Node *)checked_pc_malloc(sizeof(AODV_NT_Node));

  newNode->destAddr = destAddr;
  newNode->next = NULL;

  ++(nbrTable->size);

  // Find Insertion point.  Insert after all address matches.

  previous = NULL;
  current = nbrTable->head;
  while ((current != NULL) && (current->destAddr <= destAddr)) {
    previous = current;
    current = current->next;
  }//while//
```

```
 if (previous == NULL) {
      newNode->next = nbrTable->head;
      nbrTable->head = newNode;
   } else {
      newNode->next = previous->next;
      previous->next = newNode;
   }//if//
} /* RoutingAodvInsertNbrTable */


/*
 * RoutingAodvInsertSeenTable
 *
 * Insert an entry into the seen table
 */

static void
RoutingAodvInsertSeenTable(
   GlomoNode *node,
   NODE_ADDR srcAddr,
   int bcastId,
   int hopCount,        /* added parameter  */
   NODE_ADDR lastAddr,
   AODV_RST *seenTable)
{
   if (seenTable->size == 0)
   {
      seenTable->rear = (AODV_RST_Node *) pc_malloc(sizeof(AODV_RST_Node));
      assert(seenTable->rear != NULL);
      seenTable->front = seenTable->rear;
   }
   else
   {
      seenTable->rear->next = (AODV_RST_Node *)pc_malloc(sizeof(AODV_RST_Node));
      assert(seenTable->rear->next != NULL);
      seenTable->rear = seenTable->rear->next;
   }

   seenTable->rear->srcAddr = srcAddr;
   seenTable->rear->bcastId = bcastId;
   seenTable->rear->next = NULL;

   ++(seenTable->size);

   RoutingAodvSetTimer(
      node, MSG_NETWORK_FlushTables, ANY_DEST, (clocktype)BCAST_ID_SAVE);

} /* RoutingAodvInsertSeenTable */
```

```
/*
 * RoutingAodvInsertBuffer
 *
 * Insert a packet into the buffer if no route is available
 */
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);

   // Find Insertion point.  Insert after all address matches.

   previous = NULL;
   current = buffer->head;
   while ((current != NULL) && (current->destAddr <= destAddr)) {
      previous = current;
      current = current->next;
   }//while//


   if (previous == NULL) {
      newNode->next = buffer->head;
      buffer->head = newNode;
   } else {
      newNode->next = previous->next;
      previous->next = newNode;
   }//if//
} /* RoutingAodvInsertBuffer */



/*
 * RoutingAodvInsertSent
 *
 * Insert an entry into the sent table if RREQ is sent
```

```
*/
static void
RoutingAodvInsertSent(
   NODE_ADDR destAddr,
   int ttl,
   AODV_SENT *sent)
{
   AODV_SENT_Node* current;
   AODV_SENT_Node* previous;

   AODV_SENT_Node* newNode =
     (AODV_SENT_Node *)checked_pc_malloc(sizeof(AODV_SENT_Node));

   newNode->destAddr = destAddr;
   newNode->ttl = ttl;
   newNode->times = 0;
   newNode->next = NULL;

   (sent->size)++;

   // Find Insertion point.  Insert after all address matches.

   previous = NULL;
   current = sent->head;
   while ((current != NULL) && (current->destAddr <= destAddr)) {
      previous = current;
      current = current->next;
   }//while//


   if (previous == NULL) {
      newNode->next = sent->head;
      sent->head = newNode;
   } else {
      newNode->next = previous->next;
      previous->next = newNode;
   }//if//
} /* RoutingAodvInsertSent */


/*
 * RoutingAodvDeleteRouteTable
 *
 * Remove an entry from the route table
 */
void RoutingAodvDeleteRouteTable(NODE_ADDR destAddr, AODV_RT *routeTable)
{
   AODV_RT_Node *toFree;
   AODV_RT_Node *current;
```

```
  if (routeTable->size == 0 || routeTable->head == NULL)
  {
    return;
  }
  else if (routeTable->head->destAddr == destAddr)
  {
    if (routeTable->head->lifetime <= simclock())
    {
      toFree = routeTable->head;
      routeTable->head = toFree->next;
      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())
    {
      toFree = current->next;
      current->next = toFree->next;
      pc_free(toFree);
      --(routeTable->size);
    }
  }

} /* RoutingAodvDeleteRouteTable */

/*
 * RoutingAodvDeleteNbrTable
 *
 * Remove an entry from the neighbor table
 */
void RoutingAodvDeleteNbrTable(NODE_ADDR destAddr, AODV_NT *nbrTable)
{
  AODV_NT_Node *toFree;
  AODV_NT_Node *current;

  if (nbrTable->size == 0)
  {
    return;
  }
  else if (nbrTable->head->destAddr == destAddr)
```

```
  {
    toFree = nbrTable->head;
    nbrTable->head = toFree->next;
    pc_free(toFree);
    --(nbrTable->size);
  }
  else
  {
    for (current = nbrTable->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);
      --(nbrTable->size);
    }
  }

} /* RoutingAodvDeleteNbrTable */

/*
 * RoutingAodvDeleteSeenTable
 *
 * Remove an entry from the seen table
 */
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;
  }

} /* RoutingAodvDeleteSeenTable */

/*
 * RoutingAodvDeleteBuffer
 *
```

```
* Remove a packet from the buffer; Return TRUE if deleted
*/
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);

} /* RoutingAodvDeleteBuffer */

/*
 * RoutingAodvDeleteSent
 *
```

```
 * Remove an entry from the sent table
 */
void RoutingAodvDeleteSent(NODE_ADDR destAddr, AODV_SENT *sent)
{
   AODV_SENT_Node *toFree;
   AODV_SENT_Node *current;

   if (sent->size == 0)
   {
      return;
   }
   else if (sent->head->destAddr == destAddr)
   {
      toFree = sent->head;
      sent->head = toFree->next;
      pc_free(toFree);
      --(sent->size);
   }
   else
   {
      for (current = sent->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);
         --(sent->size);
      }
    }

 } /* RoutingAodvDeleteSent */


 /*
  * RoutingAodvUpdateLifetime
  *
  * Update the lifetime field of the destination entry in the route table
  */
 void RoutingAodvUpdateLifetime(NODE_ADDR destAddr, AODV_RT *routeTable)
 {
    AODV_RT_Node *current;

    for (current = routeTable->head;
        current != NULL && current->destAddr <= destAddr;
```

```
      current = current->next)
  {
    if (current->destAddr == destAddr)
    {
      current->lifetime = simclock() + ACTIVE_ROUTE_TO;
      return;
    }
  }

} /* RoutingAodvUpdateLifetime */

/*
 * RoutingAodvIncreaseSeq
 *
 * Increase the sequence number
 */
void RoutingAodvIncreaseSeq(GlomoNode *node)
{
  GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
  GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

  aodv->seqNumber++;

} /* RoutingAodvIncreaseSeq */

/*
 * RoutingAodvIncreaseTtl
 *
 * Increase the TTL value
 */
void RoutingAodvIncreaseTtl(NODE_ADDR destAddr, AODV_SENT *sent)
{
  AODV_SENT_Node *current;

  for (current = sent->head;
      current != NULL && current->destAddr <= destAddr;
      current = current->next)
  {
    if (current->destAddr == destAddr)
    {
      current->ttl += TTL_INCREMENT;

      if (current->ttl > TTL_THRESHOLD)
      {
        current->ttl = NET_DIAMETER;
      }

      return;
    }
```

```
  }

} /* RoutingAodvIncreaseTtl */

/*
 * RoutingAodvUpdateTtl
 *
 * Update the ttl value
 */
void RoutingAodvUpdateTtl(NODE_ADDR destAddr, int ttl, AODV_SENT *sent)
{
   AODV_SENT_Node *current;

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

} /* RoutingAodvUpdateTtl */

/*
 * RoutingAodvIncreaseTimes
 *
 * Increase the number of times RREQ sent in TTL = NET_DIAMETER
 */
void RoutingAodvIncreaseTimes(NODE_ADDR destAddr, AODV_SENT *sent)
{
   AODV_SENT_Node *current;

   for (current = sent->head;
       current != NULL && current->destAddr <= destAddr;
       current = current->next)
   {
      if (current->destAddr == destAddr)
      {
         current->times++;
         return;
      }
   }

} /* RoutingAodvIncreaseTimes */

/*
```

```
* RoutingAodvActivateRoute
*
* Activate a route in the route table
*/
void RoutingAodvActivateRoute(NODE_ADDR destAddr, AODV_RT *routeTable)
{
  AODV_RT_Node *current;

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

} /* RoutingAodvActivateRoute */

/*
 * RoutingAodvInactivateRoutesAndGetDestinations
 *
 * Inactivate routes that use the broken link
 * Returns the destAddr and whether the node must relay the RREP
 */

void RoutingAodvInactivateRoutesAndGetDestinations(
  GlomoNode* node,
  AODV_RT* routeTable,
  NODE_ADDR nextHop,
  AODV_AddressSequenceNumberPairType destinationPairs[],
  int maxNumberDestinationPairs,
  int* numberDestinations)
{
  AODV_RT_Node *current;

  int numDests = 0;

  for (current = routeTable->head;
      current != NULL;
      current = current->next)
  {
    if ((current->nextHop == nextHop) && (current->activated == TRUE))
    {
      current->activated = FALSE;
      current->hopCount = AODV_INFINITY;
```

```
        current->lifetime = simclock() + BAD_LINK_LIFETIME;
        current->destSeq++;

        RoutingAodvSetTimer(
          node, MSG_NETWORK_CheckRouteTimeout, current->destAddr,
          (clocktype)BAD_LINK_LIFETIME);

        if (!current->source) {
          destinationPairs[numDests].destinationAddress =
            current->destAddr;
          destinationPairs[numDests].destinationSequenceNumber =
            current->destSeq;
          numDests++;
        }//if//
      }//if//
    }//for//

    *numberDestinations = numDests;

} /* RoutingAodvInactivateRoute */


/*
 * RoutingAodvMarkRouteBroken
 *
 * Mark the route with destAddr broken; returns TRUE if relay is required
 */
BOOL RoutingAodvMarkRouteBroken(GlomoNode *node,
                                NODE_ADDR destAddr,
                                AODV_RT *routeTable)
{
    AODV_RT_Node *current;
    BOOL relay = FALSE;

    for (current = routeTable->head;
         current != NULL && current->destAddr <= destAddr;
         current = current->next)
    {
        if (current->destAddr == destAddr && current->activated == TRUE)
        {
            current->activated = FALSE;
            current->hopCount = AODV_INFINITY;
            current->lifetime = simclock() + BAD_LINK_LIFETIME;
            current->destSeq++;

            RoutingAodvSetTimer(
              node, MSG_NETWORK_CheckRouteTimeout, current->destAddr,
              (clocktype)BAD_LINK_LIFETIME);
```

```
            if (current->source == FALSE)
            {
               relay = TRUE;
            }

            return (relay);
         }
      }

      return (relay);

} /* RoutingAodvMarkRouteBroken */


/*
 * RoutingAodvUpdateSeq
 *
 * Update the sequence number of a certain destination
 */
void RoutingAodvUpdateSeq(NODE_ADDR destAddr, int seq, AODV_RT *routeTable)
{
   AODV_RT_Node *current;

   for (current = routeTable->head;
        current != NULL && current->destAddr <= destAddr;
        current = current->next)
   {
      if (current->destAddr == destAddr)
      {
         current->destSeq = seq;
         return;
      }
   }

} /* RoutingAodvUpdateSeq */

static //inline//
void SendRouteErrorPacket(
   GlomoNode* node,
   const AODV_RERR_Packet* rerrPacket)
{
   Message* newMsg = GLOMO_MsgAlloc(node, 0, 0, 0);
   int packetSize = AODV_RERR_PacketSize(rerrPacket);

   assert(rerrPacket->pktType == (unsigned short)AODV_RERR);
   assert(rerrPacket->destinationCount >= 1);

   GLOMO_MsgPacketAlloc(node, newMsg, packetSize);
   memcpy(GLOMO_MsgReturnPacket(newMsg), rerrPacket, packetSize);
```

```
    NetworkIpSendRawGlomoMessage(
        node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, 1);
}

/*
 * RoutingAodvInit
 *
 * Initialization function for AODV protocol
 */
void RoutingAodvInit(
    GlomoNode *node,
    GlomoRoutingAodv **aodvPtr,
    const GlomoNodeInput *nodeInput)
{
    GlomoRoutingAodv *aodv =
        (GlomoRoutingAodv *)checked_pc_malloc (sizeof(GlomoRoutingAodv));

    (*aodvPtr) = aodv;

    if (aodv == NULL)
    {
        fprintf(stderr, "AODV: Cannot alloc memory for AODV struct!\n");
        assert (FALSE);
    }
    //printf("AODV init");
    RoutingAodvInitStats(node);
    RoutingAodvInitRouteTable(&aodv->routeTable);
    RoutingAodvInitNbrTable(&aodv->nbrTable);
    RoutingAodvInitSeenTable(&aodv->seenTable);
    RoutingAodvInitConfTable(&aodv->confTable);/********initialize confidence
table**********/
    RoutingAodvInitBuffer(&aodv->buffer);
    RoutingAodvInitSent(&aodv->sent);
    RoutingAodvInitSeq(node);
    RoutingAodvInitBcastId(node);

    NetworkIpSetPacketDropNotificationFunction(
        node, &RoutingAodvPacketDropNotificationHandler);
    //printf("AODV b4 routerfunction");
    NetworkIpSetRouterFunction(node, &RoutingAodvRouterFunction);
    //printf("AODV  routerfunction");
} /* RoutingAodvInit */

/*
 * RoutingAodvFinalize
 *
 * Called at the end of the simulation to collect the results
 */
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 Route Requests Txed = %d",
          aodv->stats.numRequestSent);
  GLOMO_PrintStat(node, "RoutingAodv", buf);

  sprintf(buf, "Number of Replies Txed = %d",
          aodv->stats.numReplySent);
  GLOMO_PrintStat(node, "RoutingAodv", buf);

  sprintf(buf, "Number of Route Errors (RERR) Txed = %d",
          aodv->stats.numRerrSent);
  GLOMO_PrintStat(node, "RoutingAodv", buf);

  /* **********new stat ************************************ */
  sprintf(buf, "Number of Route Warnings (RWARN) Txed = %d",
          aodv->stats.numRwarnSent);
  GLOMO_PrintStat(node, "RoutingAodv", buf);
  /* ********************************************************************** */

  sprintf(buf, "Number of Route Errors (RERR) Re-sent = %d",
          aodv->stats.numRerrResent);
  GLOMO_PrintStat(node, "RoutingAodv", buf);


  sprintf(buf, "Number of CTRL Packets Txed = %d",
          aodv->stats.numRequestSent + aodv->stats.numReplySent);
  GLOMO_PrintStat(node, "RoutingAodv", buf);

  sprintf(buf, "Number of Routes Selected = %d", aodv->stats.numRoutes);
  GLOMO_PrintStat(node, "RoutingAodv", buf);
  sprintf(buf, "Number of Hop Counts = %d", aodv->stats.numHops);
  GLOMO_PrintStat(node, "RoutingAodv", buf);

  sprintf(buf, "Number of Data Txed = %d",
          aodv->stats.numDataTxed);
  GLOMO_PrintStat(node, "RoutingAodv", buf);
  sprintf(buf, "Number of Data Packets Originated = %d",
          aodv->stats.numDataSent);
  GLOMO_PrintStat(node, "RoutingAodv", buf);
  sprintf(buf, "Number of Data Packets Received = %d",
          aodv->stats.numDataReceived);
  GLOMO_PrintStat(node, "RoutingAodv", buf);
```

```
    sprintf(buf, "Number of Packets Dropped or Left waiting for Route = %d",
            (aodv->stats.numPacketsDropped + aodv->buffer.size));
    GLOMO_PrintStat(node, "RoutingAodv", buf);

    sprintf(buf, "Number of Broken Links = %d", aodv->stats.numBrokenLinks);
    GLOMO_PrintStat(node, "RoutingAodv", buf);
    sprintf(buf, "Number of Broken Link Retries = %d", aodv->stats.numBrokenLinkRetries);
    GLOMO_PrintStat(node, "RoutingAodv", buf);


} /* RoutingAodvFinalize */

/*
 * RoutingAodvHandleData
 *
 * Processing procedure when data is received
 */
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;

    assert(sourceAddress  != node->nodeAddr);

    /* the node is the destination of the route */
    if (destAddr == node->nodeAddr)
    {
        aodv->stats.numDataReceived++;

        RoutingAodvUpdateLifetime(sourceAddress, &aodv->routeTable);

        RoutingAodvSetTimer(node, MSG_NETWORK_CheckRouteTimeout,
                            sourceAddress, (clocktype)ACTIVE_ROUTE_TO);
    }
    else if (destAddr != ANY_DEST)
    {
        // The node is an intermediate node of the route.
        // Relay the packet to the next hop of the route

        if (RoutingAodvCheckRouteExist(destAddr, &aodv->routeTable)) {
            RoutingAodvTransmitData(node, msg, destAddr);
        } else {
            // Broken Route.  Drop Packet, send RERR again to make them stop
            // sending more.
            AODV_RERR_Packet newRerrPacket;
            newRerrPacket.pktType = AODV_RERR;
            newRerrPacket.destinationCount = 1;
```

```
        newRerrPacket.destinationPairArray[0].destinationAddress = destAddr;
        newRerrPacket.destinationPairArray[0].destinationSequenceNumber
          = RoutingAodvGetSeq(destAddr, &aodv->routeTable);

        SendRouteErrorPacket(node, &newRerrPacket);
        aodv->stats.numRerrResent++;

        aodv->stats.numPacketsDropped++;
        GLOMO_MsgFree(node,msg);

      }//if//
    }//if//

} /* RoutingAodvHandleData */

/*
 * RoutingAodvHandleRequest
 *
 * Processing procedure when RREQ is received
 */
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);

    float meanVal=0.0;/********added mean value to comapare against threshold*********/
                /********computed from the NbrRmvTable**************/

    float threshold_time_leaving = 1.5;/****added threshold******/



    /*********************************************************************/
    //malicious node
    if (node->nodeAddr == 5)
    {
       GLOMO_MsgFree(node,msg);
    }
    else //node is not malicious
    {
          /***************************Check if false accusation*******************/
          if (rreqPkt->conf_Node==node->nodeAddr)
          {
            if(rreqPkt->conf_Level == 0)
            {
          //here broadcast warning
          aodv->stats.numRwarnSent++;
```

```
        }
      }
      /*************************Check if false accusation*******************/

      RoutingAodvInsertConfTable(node,rreqPkt->lastAddr,&aodv->confTable);

      /* Process only if the packet is not a duplicate */
      if (!RoutingAodvLookupSeenTable(
            node,rreqPkt->srcAddr, rreqPkt->bcastId, rreqPkt->hopCount,
              rreqPkt->lastAddr, &aodv->seenTable, &aodv->confTable)) /*added  */
      {
      RoutingAodvInsertSeenTable(
        node, rreqPkt->srcAddr, rreqPkt->bcastId,rreqPkt->hopCount,rreqPkt->lastAddr,
      &aodv->seenTable);


                  /************************Added UpdateConfTable funcation
call**************/
                  RoutingAodvUpdateConfTable(
                  rreqPkt->lastAddr,rreqPkt->conf_Ratio,rreqPkt->conf_Level,rreqPkt-
>conf_Node,
            &aodv->confTable);
      /************************Added UpdateConfTable funcation call**************/

                  /* Update the neighbor table if the upstream is new */
      if (!RoutingAodvCheckNbrExist(rreqPkt->lastAddr, &aodv->nbrTable))
      {
        RoutingAodvInsertNbrTable(rreqPkt->lastAddr, &aodv->nbrTable);
        RoutingAodvInsertNbrRmvTable(rreqPkt->lastAddr,&aodv->NbrRmvTable);
          /*added func call*/
        meanVal=RoutingAodvComputeMean(rreqPkt->lastAddr,&aodv->NbrRmvTable ) ;
          /*added func call************/
                if (meanVal < threshold_time_leaving)
                {
                  //here broadcast warning
                  aodv->stats.numRwarnSent++;

                }

                RoutingAodvIncreaseSeq(node);
        }

        /* The node is the destination of the route */
        if (node->nodeAddr == rreqPkt->destAddr)
        {
              RoutingAodvReplaceInsertRouteTable(
              rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount,
              rreqPkt->lastAddr, simclock() + ACTIVE_ROUTE_TO, TRUE, TRUE,
              &aodv->routeTable);
```

```
        RoutingAodvSetTimer(
        node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
        (clocktype)ACTIVE_ROUTE_TO);

        /* Send a Route Reply */
        RoutingAodvInitiateRREP(node, msg);

} /* if dest */

else
{
        /* No route to destination is known */
        if (!RoutingAodvCheckRouteExist(rreqPkt->destAddr,
                        &aodv->routeTable))
        {
                RoutingAodvReplaceInsertRouteTable(
                rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount,
                rreqPkt->lastAddr, simclock() + REV_ROUTE_LIFE, FALSE,
                FALSE, &aodv->routeTable);

                RoutingAodvSetTimer(
                node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
                (clocktype)REV_ROUTE_LIFE);

                if (ttl > 0)
                {
                        /* Relay the packet only if TTL is not zero */
                        RoutingAodvRelayRREQ(node, msg, ttl);
                } /* if ttl > 0 */
                else
                {
                GLOMO_MsgFree(node, msg);
                }
        } /* if no route */

        /* Knows a route to the destination */
        else
        {
                /* However, the known route is not a fresh one */
                if (RoutingAodvGetSeq(rreqPkt->destAddr, &aodv->routeTable) <
                        rreqPkt->destSeq)
                {
                        RoutingAodvReplaceInsertRouteTable(
                rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount,
                rreqPkt->lastAddr, simclock() + REV_ROUTE_LIFE,
                FALSE, FALSE,
                &aodv->routeTable);

                        RoutingAodvSetTimer(
```

```
                    node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
                    (clocktype)REV_ROUTE_LIFE);

                        if (ttl > 0)
                        {
                        /* Relay the packet only if TTL is not zero */
                        RoutingAodvRelayRREQ(node, msg, ttl);
                        } /* if ttl > 0 */
                        else
                        {
                        GLOMO_MsgFree(node, msg);
                        }
                    } /* if seq no is not fresh */

                    /* has a fresh route to the destination */
                    else
                    {
                            RoutingAodvReplaceInsertRouteTable(
                    rreqPkt->srcAddr, rreqPkt->srcSeq, rreqPkt->hopCount,
                    rreqPkt->lastAddr, simclock() + ACTIVE_ROUTE_TO,
                    TRUE, FALSE,
                    &aodv->routeTable);

                            RoutingAodvSetTimer(
                    node, MSG_NETWORK_CheckRouteTimeout, rreqPkt->srcAddr,
                    (clocktype)ACTIVE_ROUTE_TO);

                            /* Send a Route Reply */
                            RoutingAodvInitiateRREPbyIN(node, msg);

                    } /* else */
                } /* else */
            } /* else (not dest) */
            } /* if new pkt */

            else  // duplicate packet
            {
                    RoutingAodvLookupSeenTable(
                    node,rreqPkt->srcAddr, rreqPkt->bcastId, rreqPkt->hopCount,
                    rreqPkt->lastAddr,&aodv->seenTable,&aodv->confTable);
                    /*************************** added function call
*****************************/
                    //GLOMO_MsgFree(node, msg);

            }


        }//end else node is malicious
        /*********************************added**********************/
```

```
} /* RoutingAodvHandleRequest */

/*
 * RoutingAodvHandleReply
 *
 * Processing procedure when RREP is received
 */
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 relay;
  clocktype lifetime;

  /* clocktype must be copied to access the field of that type */
  memmove(&lifetime, &rrepPkt->lifetime, sizeof(clocktype));

  /* Source of the route */

  if (rrepPkt->srcAddr == node->nodeAddr)
  {
    /* The packet is the first reply received */
    if (!RoutingAodvCheckRouteExist(rrepPkt->destAddr,
                        &aodv->routeTable))
    {
      RoutingAodvReplaceInsertRouteTable(
        rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount,
        srcAddr, simclock() + lifetime, TRUE,
        TRUE, &aodv->routeTable);

      aodv->stats.numRoutes++;
      aodv->stats.numHops += rrepPkt->hopCount;

      RoutingAodvDeleteSent(rrepPkt->destAddr, &aodv->sent);

      /* Send any buffered packets to the destination */
      while (RoutingAodvLookupBuffer(
            rrepPkt->destAddr, &aodv->buffer))
      {
        newMsg = RoutingAodvGetBufferedPacket(
                  rrepPkt->destAddr, &aodv->buffer);

        RoutingAodvTransmitData(node, newMsg, rrepPkt->destAddr);

        aodv->stats.numDataSent++;
```

```
        RoutingAodvDeleteBuffer(rrepPkt->destAddr, &aodv->buffer);

      } /* while */
    } /* if no route */

    /* The packet contains a better route compared to the one already
      known */
    else if ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable) <
          rrepPkt->destSeq) ||
         ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable)
          == rrepPkt->destSeq) &&
          (RoutingAodvGetHopCount(rrepPkt->destAddr,
                        &aodv->routeTable) >
           rrepPkt->hopCount)))
    {
      RoutingAodvReplaceInsertRouteTable(
        rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount,
        srcAddr, simclock() + lifetime, TRUE,
        TRUE, &aodv->routeTable);

      /* Send any buffered packet to the destination */
      while (RoutingAodvLookupBuffer(
            rrepPkt->destAddr, &aodv->buffer))
      {
        newMsg = RoutingAodvGetBufferedPacket(
              rrepPkt->destAddr, &aodv->buffer);

        RoutingAodvTransmitData(node, newMsg, rrepPkt->destAddr);

        aodv->stats.numDataSent++;

        RoutingAodvDeleteBuffer(rrepPkt->destAddr, &aodv->buffer);

      } /* while */
    } /* else if */
    GLOMO_MsgFree(node, msg);
  } /* if source */

  /* Intermediate node of the route */
  else
  {
    /* the packet is the first reply received */
    if (!RoutingAodvCheckRouteExist(
          rrepPkt->destAddr, &aodv->routeTable))
    {
      RoutingAodvReplaceInsertRouteTable(
        rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount,
        srcAddr, simclock() + lifetime, TRUE, FALSE,
        &aodv->routeTable);
```

```
    RoutingAodvSetTimer(
      node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->destAddr,
      (clocktype)lifetime);

    RoutingAodvActivateRoute(rrepPkt->srcAddr, &aodv->routeTable);

    RoutingAodvSetTimer(
      node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->srcAddr,
      (clocktype)ACTIVE_ROUTE_TO);

    /* Forward the packet to the upstream of the route */
    RoutingAodvRelayRREP(node, msg, destAddr);

  } /* if new route */

  /* the packet carries a better route compared to the one already
    known */
  else if ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable) <
          rrepPkt->destSeq) ||
         ((RoutingAodvGetSeq(rrepPkt->destAddr, &aodv->routeTable)
          == rrepPkt->destSeq) &&
          (RoutingAodvGetHopCount(rrepPkt->destAddr,
                    &aodv->routeTable) >
          rrepPkt->hopCount)))
  {
    RoutingAodvReplaceInsertRouteTable(
      rrepPkt->destAddr, rrepPkt->destSeq, rrepPkt->hopCount,
      srcAddr, simclock() + lifetime, TRUE,
      FALSE, &aodv->routeTable);

    RoutingAodvSetTimer(
      node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->destAddr,
      (clocktype)lifetime);

    RoutingAodvActivateRoute(rrepPkt->srcAddr, &aodv->routeTable);

    RoutingAodvSetTimer(
      node, MSG_NETWORK_CheckRouteTimeout, rrepPkt->srcAddr,
      (clocktype)ACTIVE_ROUTE_TO);
    /* Forward the packet to the upstream of the route */
    RoutingAodvRelayRREP(node, msg, destAddr);

  } /* else if newer route or shorter route */

  else
  {
    GLOMO_MsgFree(node, msg);
  }//if//
}//if//
```

```
} /* RoutingAodvHandleReply */

//
// RoutingAodvHandleRouteError
//
// Processing procedure when RERR is received
//

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;

   newRerrPacket.pktType = (unsigned short)AODV_RERR;
   newRerrPacket.destinationCount = 0;

   for(I = 0; I < rerrPkt->destinationCount; I++) {
      // Mark the route inactive in the route table; Must not remove it
      // right away since the last hop count known is needed for future use
      // Remove destination from packet if it doesn't need to be forwarded
      // further.

      NODE_ADDR destination =
        rerrPkt->destinationPairArray[I].destinationAddress;
      int sequenceNum =
        rerrPkt->destinationPairArray[I].destinationSequenceNumber;

      BOOL mustRelay =
        RoutingAodvMarkRouteBroken(
          node,
          destination,
          &aodv->routeTable);

      RoutingAodvUpdateSeq(destination,
                  sequenceNum,
                  &aodv->routeTable);

      NetworkIpDeleteOutboundPacketsToANode(
        node, srcAddr, destination, FALSE);


      if (mustRelay) {
        newRerrPacket.destinationPairArray[newRerrPacket.destinationCount] =
          rerrPkt->destinationPairArray[I];
```

```
            newRerrPacket.destinationCount++;
        }//if//

    }//while//

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

    GLOMO_MsgFree(node, msg);
}//RoutingAodvHandleRouteError//

/*
 * RoutingAodvInitRouteTable
 *
 * Initialize the route table
 */
void RoutingAodvInitRouteTable(AODV_RT *routeTable)
{
    routeTable->head = NULL;
    routeTable->size = 0;

} /* RoutingAodvInitRouteTable */

/*
 * RoutingAodvInitNbrTable
 *
 * Initialize the neighbor table
 */
void RoutingAodvInitNbrTable(AODV_NT *nbrTable)
{
    nbrTable->head = NULL;
    nbrTable->size = 0;

} /* RoutingAodvInitNbrTable */

/*
 * RoutingAodvInitSeenTable
 *
 * Initialize the seen table
 */
void RoutingAodvInitSeenTable(AODV_RST *seenTable)
{
    seenTable->front = NULL;
    seenTable->rear = NULL;
    seenTable->size = 0;

} /* RoutingAodvInitSeenTable */
```

```
/*
 * RoutingAodvInitConfTable
 *
 * Initialize the confidence table
 */
void RoutingAodvInitConfTable(AODV_CFT *confTable)
{
   //printf("confTable gets initialized here");
   confTable->front = NULL;
   confTable->rear = NULL;
   confTable->size=0;

} /* RoutingAodvInitConfTable */

/*
 * RoutingAodvInitNbrRmvTable
 *
 * Initialize the neighbour remove table
 */
void RoutingAodvInitNbrRmvTable(AODV_NRT *NbrRmvTable)
{
   NbrRmvTable->front = NULL;
   NbrRmvTable->rear = NULL;
   NbrRmvTable->size = 0;

}/*RoutingAodvInitNbrRmvTable*/

/*
 * RoutingAodvInitBuffer
 *
 * Initialize the buffer
 */
void RoutingAodvInitBuffer(AODV_BUFFER *buffer)
{
   buffer->head = NULL;
   buffer->size = 0;

} /* RoutingAodvInitBuffer */

/*
 * RoutingAodvInitSent
 *
 * Initialize the sent table
 */
void RoutingAodvInitSent(AODV_SENT *sent)
{
   sent->head = NULL;
   sent->size = 0;
```

```
} /* RoutingAodvInitSent */

/*
 * RoutingAodvInitStats
 *
 * Initialize all the stat variables
 */
void RoutingAodvInitStats(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

   aodv->stats.numRequestSent = 0;
   aodv->stats.numReplySent = 0;
   aodv->stats.numRerrSent = 0;
   aodv->stats.numRwarnSent = 0;
   aodv->stats.numRerrResent = 0;
   aodv->stats.numDataSent = 0;
   aodv->stats.numDataTxed = 0;
   aodv->stats.numDataReceived = 0;
   aodv->stats.numRoutes = 0;
   aodv->stats.numHops = 0;
   aodv->stats.numPacketsDropped = 0;
   aodv->stats.numBrokenLinks = 0;
   aodv->stats.numBrokenLinkRetries = 0;
} /* RoutingAodvInitStats */

/*
 * RoutingAodvInitSeq
 *
 * Initialize the sequence number
 */
void RoutingAodvInitSeq(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

   aodv->seqNumber = 0;

} /* RoutingAodvInitSeq */

/*
 * RoutingAodvInitBcastId
 *
 * Initialize the broadcast id
 */
void RoutingAodvInitBcastId(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
```

```
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

    aodv->bcastId = 0;

} /* RoutingAodvInitBcastId */

/*
 * RoutingAodvGetNextHop
 *
 * Looks up the routing table to obtain next hop to the destinaton
 */
NODE_ADDR RoutingAodvGetNextHop(NODE_ADDR destAddr, AODV_RT *routeTable)
{
    AODV_RT_Node *current;

    for (current = routeTable->head;
        current != NULL && current->destAddr <= destAddr;
        current = current->next)
    {
       if (current->destAddr == destAddr && current->activated == TRUE)
       {
          return(current->nextHop);
       }
    }

    return (ANY_DEST);
} /* RoutingAodvGetNextHop */

/*
 * RoutingAodvGetBcastId
 *
 * Obtains the broadcast ID for the outgoing packet
 */
int RoutingAodvGetBcastId(GlomoNode *node)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    int bcast;

    bcast = aodv->bcastId;
    aodv->bcastId++;

    return (bcast);

} /* RoutingAodvGetBcastId */

/*
 * RoutingAodvGetSeq
 *
```

```
 * Obtains the sequence number of the destination node
 */
int RoutingAodvGetSeq(NODE_ADDR destAddr, AODV_RT *routeTable)
{
   AODV_RT_Node *current;

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

   return (-1);

} /* RoutingAodvGetSeq */

/*
 * RoutingAodvGetMySeq
 *
 * Obtains the node's seq number
 */
int RoutingAodvGetMySeq(GlomoNode *node)
{
   GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
   GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

   return (aodv->seqNumber);
} /* RoutingAodvGetMySeq */

/*
 * RoutingAodvGetHopCount
 *
 * Obtains the hop count to the destination node
 */
int RoutingAodvGetHopCount(NODE_ADDR destAddr, AODV_RT *routeTable)
{
   AODV_RT_Node *current;

   for (current = routeTable->head;
        current != NULL && current->destAddr <= destAddr;
        current = current->next)
   {
      if (current->destAddr == destAddr)
      {
         return(current->hopCount);
```

```
        }
    }

    return (-1);

} /* RoutingAodvGetHopCount */

/*
 * RoutingAodvGetLastHopCount
 *
 * Obtains the last hop count known to the destination node
 */
int RoutingAodvGetLastHopCount(NODE_ADDR destAddr, AODV_RT *routeTable)
{
    AODV_RT_Node *current;

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

    return (-1);

} /* RoutingAodvGetLastHopCount */

/*
 * RoutingAodvGetTtl
 *
 * Obtains the ttl value for the outgoing RREQ
 */
int RoutingAodvGetTtl(NODE_ADDR destAddr, AODV_SENT *sent)
{
    AODV_SENT_Node *current;

    for (current = sent->head;
         current != NULL && current->destAddr <= destAddr;
         current = current->next)
    {
        if (current->destAddr == destAddr)
        {
            return(current->ttl);
        }
    }
```

```
    return (TTL_START);

} /* RoutingAodvGetTtl */

/*
 * RoutingAodvGetTimes
 *
 * Obtains the number of times the RREQ was sent in TTL = NET_DIAMETER
 */
int RoutingAodvGetTimes(NODE_ADDR destAddr, AODV_SENT *sent)
{
    AODV_SENT_Node *current;

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

    return (0);

} /* RoutingAodvGetTimes */

/*
 * RoutingAodvGetLifetime
 *
 * Obtains the lifetime value of an entry in the route table
 */
clocktype RoutingAodvGetLifetime(NODE_ADDR destAddr, AODV_RT *routeTable)
{
    AODV_RT_Node *current;

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

    return (0);

} /* RoutingAodvGetLifetime */
```

```
/*
 * RoutingAodvGetBufferedPacket
 *
 * Extract the packet that was buffered
 */
Message *
RoutingAodvGetBufferedPacket(NODE_ADDR destAddr, AODV_BUFFER *buffer)
{
   AODV_BUFFER_Node *current;

   for (current = buffer->head;
        current != NULL && current->destAddr <= destAddr;
        current = current->next)
   {
      if (current->destAddr == destAddr)
      {
         return(current->msg);
      }
   }
   assert(FALSE); abort(); return NULL;

} /* RoutingAodvGetBufferedPacket */

/*
 * RoutingAodvCheckRouteExist
 *
 * Returns TRUE if any route to the destination is known
 */
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->hopCount != AODV_INFINITY) &&
          (current->lifetime > simclock()) &&
          (current->activated == TRUE))
      {
         return(TRUE);
      }
   }
```

```
  return (FALSE);

} /* RoutingAodvCheckRouteExist */

/*
 * RoutingAodvCheckNbrExist
 *
 * Returns TRUE if the node is already a neighbor
 */
BOOL RoutingAodvCheckNbrExist(NODE_ADDR destAddr, AODV_NT *nbrTable)
{
  AODV_NT_Node *current;

  if (nbrTable->size == 0)
  {
    return (FALSE);
  }

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

  return (FALSE);
} /* RoutingAodvCheckNbrExist */

/*
 * ORIGINAL RoutingAodvLookupSeenTable
 *
 * Returns TRUE if the broadcast packet is processed before
 */
/*
BOOL RoutingAodvLookupSeenTable(NODE_ADDR srcAddr,
                    int bcastId,
                    AODV_RST *seenTable)
{
  AODV_RST_Node *current;

  if (seenTable->size == 0)
  {
    return (FALSE);
  }

  for (current = seenTable->front;
```

```
    current != NULL;
    current = current->next)
  {
    if (current->srcAddr == srcAddr && current->bcastId == bcastId)
    {
      return (TRUE);
    }
  }

  return (FALSE);

}*/

/*

**********************************************************************************
******************
 * Edited RoutingAodvLookupSeenTable
 *
 * Returns TRUE if the broadcast packet is processed before
 */
BOOL RoutingAodvLookupSeenTable(GlomoNode *node,
                                NODE_ADDR srcAddr,
                    int bcastId,int hopCount,    /* parameter added */
                    NODE_ADDR lastAddr,
                    AODV_RST *seenTable,
                                AODV_CFT *confTable)
{

  GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
  GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
  AODV_RST_Node *current;

  int hasForwarded=0;

  if (seenTable->size == 0)
  {

    return (FALSE);
  }

  for (current = seenTable->front; current != NULL; current = current->next)
  {
    if (current->srcAddr == srcAddr && current->bcastId == bcastId)
    {
      if (current-> hopCount == hopCount - 1)
          {
                hasForwarded = hasForwarded + 1;/*total # packets forwarded*/
```

```
        }
        return(TRUE);
      }
    }


        RoutingAodvComputeConfTable(node,lastAddr,hasForwarded,requestToForward,&aodv
->confTable);

      //RoutingAodvComputeConfTable
      //(node,lastAddr,hasForwarded,requestToForward,&aodv->confTable);

 return (FALSE);

/**************************************************************************
***************/
} /* RoutingAodvLookupSeenTable */

/*
 *RoutingAodvComputeConfTable
 *
 *Updates the confidence table for the sending node
 *identified by the "lastAddr"
 *
 */
BOOL RoutingAodvComputeConfTable(GlomoNode *node,NODE_ADDR lastAddr, int
HFWD, int RTF,
AODV_CFT *confTable)
{
  GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
  GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
  AODV_CFT_Node *current;
  //current->nodeAddr=2;
  float conf_Ratio = HFWD / RTF ;
  float conf_Level = conf_Level + conf_Ratio;
  if (confTable->size == 0)
  {
    return (FALSE);
  }

  //malicious node falsely accusing
  if (node->nodeAddr == 5)
  {

        for(current=confTable->front; current != NULL; current=current->next)
        {
          if(current->nodeAddr == lastAddr)
        {
            current->conf_Ratio=0.0;
```

```
            current->conf_Level=0.0;
              }
              else
              {
                //RoutingAodvInsertConfTable(node,lastAddr,&aodv->confTable);

                confTable->rear->nodeAddr=lastAddr;
              confTable->rear->conf_Ratio=0.0;
                confTable->rear->conf_Level=0.0;
              }
        }

    }
    else // not malicious
    {
            //float conf_Ratio = HFWD / RTF ;
            //float conf_Level = conf_Level + conf_Ratio;

            for(current=confTable->front; current != NULL; current=current->next)
            {
              if(current->nodeAddr == lastAddr)
            {
                current->conf_Ratio=conf_Ratio;
              current->conf_Level=conf_Level;
              }
              else
              {
              //RoutingAodvInsertConfTable(node,lastAddr,&aodv->confTable);
                confTable->rear->nodeAddr=lastAddr;
              confTable->rear->conf_Ratio=conf_Ratio;
                confTable->rear->conf_Level=conf_Level;
              }
        }
     }

     return (TRUE);

}/*RoutingAodvComputeConfTable*/


void RoutingAodvInsertConfTable(GlomoNode *node,NODE_ADDR lastAddr,AODV_CFT
*confTable)
{

    if (confTable->size == 0)
    {
      confTable->rear = (AODV_CFT_Node *) pc_malloc(sizeof(AODV_CFT_Node));
      assert(confTable->rear != NULL);
      confTable->front = confTable->rear;
```

```
  }
  else
  {
    confTable->rear->next = (AODV_CFT_Node *) pc_malloc(sizeof(AODV_CFT_Node));
    assert(confTable->rear->next != NULL);
    confTable->rear = confTable->rear->next;
  }

  confTable->rear->nodeAddr = lastAddr;
  confTable->rear->conf_Ratio = 1.0;
  confTable->rear->conf_Level = 1.0;
  confTable->rear->OER = 1.0;
  confTable->rear->next = NULL;

  ++(confTable->size);

}

/*
 *RoutingAodvUpdateConfTable
 *Updates the node's OER using the confidence ratio and level
 */
BOOL RoutingAodvUpdateConfTable(NODE_ADDR lastAddr, float confRatio, float confLevel,
NODE_ADDR confNode,AODV_CFT* confTable)
{
  AODV_CFT_Node *current;

  float tempOER_Num=0.0;
  float tempOER_Den=0.0;

  if (confTable->size == 0)
  {
    return (FALSE);
  }

  for(current=confTable->front; current != NULL; current=current->next)
  {
        if(current->nodeAddr == confNode)
        {
       tempOER_Num = current->conf_Level * confLevel * confRatio;
          tempOER_Den = current->conf_Level * confLevel;

          current->OER = tempOER_Num/tempOER_Den;
     }
        else
     {
          confTable->rear->nodeAddr=lastAddr;
       confTable->rear->conf_Ratio=confRatio;
          confTable->rear->conf_Level=confLevel;
```

```
        }
    }

    return (TRUE);

    //loop thru the confTable to get the pointer to the wanted address
    //calculate the new OER
    //if the loop ended and no match was found in the table
    //insert values at the end of the table

}/*RoutingAodvUpdateConfTable*/


/*
 * RoutingAodvInsertNbrRmvTable
 */
void RoutingAodvInsertNbrRmvTable(NODE_ADDR destAddr, AODV_NRT *NbrRmvTable)
{
    /*
    AODV_NRT_Node* current;
    AODV_NRT_Node* previous;

    AODV_NRT_Node* newNode = (AODV_NRT_Node
*)checked_pc_malloc(sizeof(AODV_NRT_Node));

    newNode->nodeAddr = destAddr;
    newNode->TimeOfLeaving= time_of_leaving + 2;
    newNode->next = NULL; */

    if (NbrRmvTable->size == 0)
    {
        NbrRmvTable->rear = (AODV_NRT_Node *) pc_malloc(sizeof(AODV_NRT_Node));
        assert(NbrRmvTable->rear != NULL);
        NbrRmvTable->front = NbrRmvTable->rear;
    }
    else
    {
        NbrRmvTable->rear->next = (AODV_NRT_Node
*)pc_malloc(sizeof(AODV_NRT_Node));
        assert(NbrRmvTable->rear->next != NULL);
        NbrRmvTable->rear = NbrRmvTable->rear->next;
    }

    NbrRmvTable->rear->nodeAddr = destAddr;
    NbrRmvTable->rear->TimeOfLeaving = time_of_leaving + 2;
    NbrRmvTable->rear->next = NULL;

    ++(NbrRmvTable->size);
```

```
}/*RoutingAodvInsertNbrRmvTable*/


/*
 * RoutingAodvComputeMean
 * calculates the mean for the nodes' exit and entrance into network
 */
float RoutingAodvComputeMean(NODE_ADDR targetAddr, AODV_NRT *NbrRmvTable)
{
  AODV_NRT_Node *current;
  //AODV_NRT *NbrRmvTable;
  int temp=0;
  int time=0;
  int diff=0;
  int count=1;
  float mean=0.0;

  for(current = NbrRmvTable->rear; current != NULL; current = current->next)
  {
        if(current->nodeAddr == targetAddr)
        {
           time = current->TimeOfLeaving;
        diff = diff + (time - temp);
        count++;
        temp = time;
     }
  }

   mean = diff/count;
   //here compare mean with Threshold
   //if less increase warning statstic


}/*RoutingAodvComputeMean*/

/*
 * RoutingAodvLookupBuffer
 *
 * Returns TRUE if any packet is buffered to the destination
 *
 */
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);

} /* RoutingAodvLookupBuffer */

/*
 * RoutingAodvCheckSent
 *
 * Check if RREQ has been sent; return TRUE if sent
 */
BOOL RoutingAodvCheckSent(NODE_ADDR destAddr, AODV_SENT *sent)
{
    AODV_SENT_Node *current;

    if (sent->size == 0)
    {
      return (FALSE);
    }

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

    return (FALSE);

} /* RoutingAodvCheckSent */

/*
 * RoutingAodvHandleProtocolPacket
 *
 * Called when the packet is received from MAC
 */
void RoutingAodvHandleProtocolPacket(
    GlomoNode *node, Message *msg, NODE_ADDR srcAddr,
```

```
  NODE_ADDR destAddr, int ttl)
{
  AODV_PacketType *aodvHeader = (AODV_PacketType*)GLOMO_MsgReturnPacket(msg);

  switch (*aodvHeader)
  {
    case AODV_RREQ:
    {
      RoutingAodvHandleRequest(node, msg, ttl);

      break;
    } /* RREQ */

    case AODV_RREP:
    {
      RoutingAodvHandleReply(node, msg, srcAddr, destAddr);

      break;
    } /* RREP */
    case AODV_RERR:
    {
      assert(destAddr == ANY_DEST);
      RoutingAodvHandleRouteError(node, msg, srcAddr);

      break;
    } /* RERR */

    default:
      assert(FALSE); abort();
      break;
  } /* switch */
} /* RoutingAodvHandleProtocolPacket */

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

  switch (msg->eventType) {

    /* Remove an entry from the RREQ Seen Table */
    case MSG_NETWORK_FlushTables: {
      RoutingAodvDeleteSeenTable(&aodv->seenTable);
      GLOMO_MsgFree(node, msg);
```

```
    break;
}

/* Remove the route that has not been used for awhile */
case MSG_NETWORK_CheckRouteTimeout: {
    NODE_ADDR *destAddr = (NODE_ADDR *)GLOMO_MsgReturnInfo(msg);

    RoutingAodvDeleteRouteTable(*destAddr, &aodv->routeTable);
    GLOMO_MsgFree(node, msg);

    break;
}

/* Check if RREP is received after sending RREQ */
case MSG_NETWORK_CheckReplied: {
    NODE_ADDR *destAddr = (NODE_ADDR *)GLOMO_MsgReturnInfo(msg);

    /* Route has not been obtained */
    if (!RoutingAodvCheckRouteExist(*destAddr, &aodv->routeTable))
    {
        if (RoutingAodvGetTimes(*destAddr, &aodv->sent) < RREQ_RETRIES)
        {
            /* Retry with increased TTL */
            RoutingAodvRetryRREQ(node, *destAddr);
        } /* if under the retry limit */

        /* over the limit */
        else
        {
            while (RoutingAodvLookupBuffer(*destAddr, &aodv->buffer))
            {
                Message* messageToDelete =
                    RoutingAodvGetBufferedPacket(
                        *destAddr, &aodv->buffer);
                RoutingAodvDeleteBuffer(*destAddr, &aodv->buffer);

                GLOMO_MsgFree(node, messageToDelete);
                aodv->stats.numPacketsDropped++;
            }
        } /* else */
    } /* if no route */

    GLOMO_MsgFree(node, msg);

    break;
}

default:
    fprintf(stderr, "RoutingAodv: Unknown MSG type %d!\n",
```

```
            msg->eventType);
        abort();

    } /* switch */

} /* RoutingAodvHandleProtocolEvent */

/*
 * RoutingAodvRouterFunction
 *
 * Determine the routing action to take for a the given data packet
 * set the PacketWasRouted variable to TRUE if no further handling of
 * this packet by IP is necessary
 */
void RoutingAodvRouterFunction(
    GlomoNode *node,
    Message *msg,
    NODE_ADDR destAddr,
    BOOL *packetWasRouted)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    IpHeaderType *ipHeader = (IpHeaderType *) msg->packet;

    //printf("router function TOP");
    /* Control packets */
    if (ipHeader->ip_p  == IPPROTO_AODV)
    {
        return;
    }

    if (destAddr == node->nodeAddr)
    {
        *packetWasRouted = FALSE;
    }
    else
    {
        *packetWasRouted = TRUE;
    }

    /* intermediate node or destination of the route */
    if (ipHeader->ip_src != node->nodeAddr)
    {
        RoutingAodvHandleData(node, msg, destAddr);
    }

    /* source has a route to the destination */
    else if (RoutingAodvCheckRouteExist(destAddr, &aodv->routeTable))
    {
```

```
        RoutingAodvTransmitData(node, msg, destAddr);
        aodv->stats.numDataSent++;
    }

    /* There is no route to the destination and RREQ has not been sent */
    else if (!RoutingAodvLookupBuffer(destAddr, &aodv->buffer))
    {

        RoutingAodvInsertBuffer(msg, destAddr, &aodv->buffer);
        RoutingAodvInitiateRREQ(node, destAddr);
    }

    /* There is no route but RREQ has already been sent */
    else
    {
        RoutingAodvInsertBuffer(msg, destAddr, &aodv->buffer);
    }

} /* RoutingAodvRouterFunction */

/*
 * RoutingAodvMacLayerStatusHandler
 *
 * Reacts to the signal sent by the MAC protocol after link failure
 */
void RoutingAodvPacketDropNotificationHandler(
    GlomoNode *node, const Message* msg, const NODE_ADDR nextHopAddress)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;

    IpHeaderType* ipHeader;
    NODE_ADDR destAddr;
    int numberRouteDestinations;

    ipHeader = (IpHeaderType *) GLOMO_MsgReturnPacket(msg);

    if (ipHeader->ip_p  == IPPROTO_AODV)
    {
        return;
    }//if//


    destAddr = ipHeader->ip_dst;

    if (nextHopAddress == ANY_DEST) {
        aodv->stats.numBrokenLinkRetries++;
        return;
    }//if//
```

```
    NetworkIpDeleteOutboundPacketsToANode(
      node, nextHopAddress, ANY_DEST, FALSE);

    aodv->stats.numBrokenLinks++;

    RoutingAodvDeleteNbrTable(nextHopAddress, &aodv->nbrTable);
    RoutingAodvIncreaseSeq(node);

    do {
      AODV_RERR_Packet newRerrPacket;
      newRerrPacket.pktType = AODV_RERR;

      RoutingAodvInactivateRoutesAndGetDestinations(
        node,
        &aodv->routeTable,
        nextHopAddress,
        newRerrPacket.destinationPairArray,
        AODV_MAX_RERR_DESTINATIONS,
        &numberRouteDestinations);
      newRerrPacket.destinationCount = numberRouteDestinations;

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

    } while (numberRouteDestinations == AODV_MAX_RERR_DESTINATIONS);
}//RoutingAodvMaclayerStatusHandler//

/*
 *
RoutingAodvSetTimer
 *
 * Set timers for protocol events
 */
void RoutingAodvSetTimer(
    GlomoNode *node, long eventType, NODE_ADDR destAddr, clocktype delay)
{
    Message *newMsg;
    NODE_ADDR *info;

    newMsg = GLOMO_MsgAlloc(node,
                        GLOMO_NETWORK_LAYER,
                        ROUTING_PROTOCOL_AODV,
                        eventType);

    GLOMO_MsgInfoAlloc(node, newMsg, sizeof(NODE_ADDR));
    info = (NODE_ADDR *) GLOMO_MsgReturnInfo(newMsg);
    *info = destAddr;
```

```
    GLOMO_MsgSend(node, newMsg, delay);

} /* RoutingAodvSetTimer */

/*
 * RoutingAodvInitiateRREQ
 *
 * Initiate a Route Request packet when no route to destination is known
 */
void RoutingAodvInitiateRREQ(GlomoNode *node, NODE_ADDR destAddr)
{

    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    Message *newMsg;
    AODV_RREQ_Packet *rreqPkt;
    AODV_CFT_Node *CFT;
    char *pktPtr;
    int pktSize = sizeof(AODV_RREQ_Packet);
    int ttl;

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

    pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
    rreqPkt = (AODV_RREQ_Packet *) pktPtr;
    RoutingAodvInsertConfTable(node,rreqPkt->destAddr,&aodv->confTable);

    rreqPkt->pktType = AODV_RREQ;
    rreqPkt->bcastId = RoutingAodvGetBcastId(node);
    rreqPkt->destAddr = destAddr;
    rreqPkt->destSeq = RoutingAodvGetSeq(destAddr, &aodv->routeTable);
    rreqPkt->srcAddr = node->nodeAddr;
    rreqPkt->srcSeq = RoutingAodvGetMySeq(node);
    rreqPkt->lastAddr = node->nodeAddr;
    rreqPkt->hopCount = 1;
    /*************conf table info*****************************************/
    rreqPkt->conf_Ratio = CFT->conf_Ratio;
    rreqPkt->conf_Level = CFT->conf_Level;
    rreqPkt->conf_Node = CFT->nodeAddr;

    if (RoutingAodvCheckSent(destAddr, &aodv->sent))
    {
        ttl = RoutingAodvGetTtl(destAddr, &aodv->sent);
        RoutingAodvIncreaseTtl(destAddr, &aodv->sent);
    }
    else
    {
```

```
        ttl = RoutingAodvGetLastHopCount(destAddr, &aodv->routeTable);

        if (ttl == -1)
        {
          ttl = TTL_START;
        }

        RoutingAodvInsertSent(destAddr, ttl, &aodv->sent);

        RoutingAodvIncreaseTtl(destAddr, &aodv->sent);
    }

    NetworkIpSendRawGlomoMessage(
        node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, ttl);
    /*********************************************/
    requestToForward ++;
    /*********************************************/
    aodv->stats.numRequestSent++;

    RoutingAodvInsertSeenTable(
            node, node->nodeAddr, rreqPkt->bcastId,rreqPkt->hopCount,rreqPkt-
>lastAddr,&aodv->seenTable);

    RoutingAodvSetTimer(node, MSG_NETWORK_CheckReplied, destAddr,
                (clocktype)2 * ttl * NODE_TRAVERSAL_TIME);

} /* RoutingAodvInitiateRREQ */
/*****************************************************************************
*******************/
/*
 * RoutingAodvRetryRREQ
 *
 * Send RREQ again after not receiving any RREP
 */
void RoutingAodvRetryRREQ(GlomoNode *node, NODE_ADDR destAddr)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    Message *newMsg;
    AODV_RREQ_Packet *rreqPkt;
    AODV_CFT_Node *CFT;
    char *pktPtr;
    int pktSize = sizeof(AODV_RREQ_Packet);
    int ttl;

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

```
    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->srcAddr = node->nodeAddr;
    rreqPkt->srcSeq = RoutingAodvGetMySeq(node);
    rreqPkt->lastAddr = node->nodeAddr;
    rreqPkt->hopCount = 1;
    /*************conf table info*********************************************/
    rreqPkt->conf_Ratio = CFT->conf_Ratio;
    rreqPkt->conf_Level = CFT->conf_Level;
    rreqPkt->conf_Node = CFT->nodeAddr;

    ttl = RoutingAodvGetTtl(destAddr, &aodv->sent);
    NetworkIpSendRawGlomoMessage(
        node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, ttl);

    RoutingAodvIncreaseTtl(destAddr, &aodv->sent);
    /*********************************************/
    requestToForward ++;
    /*********************************************/
    aodv->stats.numRequestSent++;

    RoutingAodvInsertSeenTable(
            node, node->nodeAddr, rreqPkt->bcastId,rreqPkt->hopCount,rreqPkt-
>lastAddr,&aodv->seenTable);

    if (ttl == NET_DIAMETER)
    {
      RoutingAodvIncreaseTimes(destAddr, &aodv->sent);
    }

    RoutingAodvSetTimer(node, MSG_NETWORK_CheckReplied, destAddr,
                (clocktype)2 * ttl * NODE_TRAVERSAL_TIME);

} /* RoutingAodvRetryRREQ */

/*
 * RoutingAodvTransmitData
 *
 * Forward the data packet to the next hop
 */
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);

    NetworkIpSendPacketToMacLayer(node, msg, DEFAULT_INTERFACE, nextHop);
    aodv->stats.numDataTxed++;

    RoutingAodvUpdateLifetime(destAddr, &aodv->routeTable);

    RoutingAodvSetTimer(node, MSG_NETWORK_CheckRouteTimeout,
                        destAddr, (clocktype)ACTIVE_ROUTE_TO);

} /* RoutingAodvTransmitData */

/*
 * RoutingAodvRelayRREQ
 *
 * Forward (re-broadcast) the RREQ
 */
void RoutingAodvRelayRREQ(GlomoNode *node, Message *msg, int ttl)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    Message *newMsg;
    AODV_RREQ_Packet *oldRreq;
    AODV_RREQ_Packet *newRreq;
    char *pktPtr;
    int pktSize = sizeof(AODV_RREQ_Packet);
    clocktype delay;

    oldRreq = (AODV_RREQ_Packet *) 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 = (AODV_RREQ_Packet *) pktPtr;

    newRreq->pktType = oldRreq->pktType;
    newRreq->bcastId = oldRreq->bcastId;
    newRreq->destAddr = oldRreq->destAddr;
    newRreq->destSeq = oldRreq->destSeq;
    newRreq->srcAddr = oldRreq->srcAddr;
```

```
    newRreq->srcSeq = oldRreq->srcSeq;
    newRreq->lastAddr = node->nodeAddr;
    newRreq->hopCount = oldRreq->hopCount + 1;

    delay = pc_erand(node->seed) * BROADCAST_JITTER;

    NetworkIpSendRawGlomoMessageWithDelay(
            node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, ttl, delay);
    /*********************************************/
    requestToForward ++;
    /*********************************************/
    aodv->stats.numRequestSent++;

    GLOMO_MsgFree(node, msg);

} /* RoutingAodvRelayRREQ */

/*
 * RoutingAodvInitiateRREP
 *
 * Destination of the route sends RREP in reaction to RREQ
 */
void RoutingAodvInitiateRREP(GlomoNode *node, Message *msg)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *)ipLayer->routingProtocol;
    Message *newMsg;
    AODV_RREQ_Packet *rreqPkt;
    AODV_RREP_Packet *rrepPkt;
    char *pktPtr;
    int pktSize = sizeof(AODV_RREP_Packet);
    int seq;

    rreqPkt = (AODV_RREQ_Packet *) 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 = (AODV_RREP_Packet *) pktPtr;

    rrepPkt->pktType = AODV_RREP;
    rrepPkt->srcAddr = rreqPkt->srcAddr;
    rrepPkt->destAddr = node->nodeAddr;
    seq = RoutingAodvGetMySeq(node);
    if (seq >= rreqPkt->destSeq)
    {
        rrepPkt->destSeq = seq;
```

```
    }
    else
    {
      rrepPkt->destSeq = rreqPkt->destSeq;
      RoutingAodvIncreaseSeq(node);
    }
    rrepPkt->hopCount = 1;
    rrepPkt->lifetime = (clocktype)MY_ROUTE_TO;

    NetworkIpSendRawGlomoMessageToMacLayer(
      node, newMsg, rreqPkt->lastAddr, CONTROL, IPPROTO_AODV, 1,
      DEFAULT_INTERFACE, rreqPkt->lastAddr);

    aodv->stats.numReplySent++;

    GLOMO_MsgFree(node, msg);

} /* RoutingAodvInitiateRREP */

/*
 * RoutingAodvInitiateRREPbyIN
 *
 * An intermediate node that knows the route to the destination sends the RREP
 */
void RoutingAodvInitiateRREPbyIN(GlomoNode *node, Message *msg)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *)node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *)ipLayer->routingProtocol;
    Message *newMsg;
    AODV_RREQ_Packet *rreqPkt;
    AODV_RREP_Packet *rrepPkt;
    char *pktPtr;
    int pktSize = sizeof(AODV_RREP_Packet);
    int seq;

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

    pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
    rrepPkt = (AODV_RREP_Packet *) pktPtr;

    rreqPkt = (AODV_RREQ_Packet *) GLOMO_MsgReturnPacket(msg);

    rrepPkt->pktType = AODV_RREP;
    rrepPkt->srcAddr = rreqPkt->srcAddr;
    rrepPkt->destAddr = rreqPkt->destAddr;
    rrepPkt->destSeq = RoutingAodvGetSeq(rreqPkt->destAddr, &aodv->routeTable);
    rrepPkt->lifetime = RoutingAodvGetLifetime(
```

```
                    rreqPkt->destAddr, &aodv->routeTable) - simclock();
    rrepPkt->hopCount =  RoutingAodvGetHopCount(
                    rreqPkt->destAddr, &aodv->routeTable) + 1;

    NetworkIpSendRawGlomoMessageToMacLayer(
      node, newMsg, rreqPkt->lastAddr, CONTROL, IPPROTO_AODV, 1,
      DEFAULT_INTERFACE, rreqPkt->lastAddr);

    aodv->stats.numReplySent++;

    GLOMO_MsgFree(node, msg);
} /* RoutingAodvInitiateRREPbyIN */

/*
 * RoutingAodvRelayRREP
 *
 * Forward the RREP packet
 */
void RoutingAodvRelayRREP(GlomoNode *node, Message *msg, NODE_ADDR destAddr)
{
    GlomoNetworkIp* ipLayer = (GlomoNetworkIp *) node->networkData.networkVar;
    GlomoRoutingAodv* aodv = (GlomoRoutingAodv *) ipLayer->routingProtocol;
    Message *newMsg;
    AODV_RREP_Packet *oldRrep;
    AODV_RREP_Packet *newRrep;
    char *pktPtr;
    NODE_ADDR nextHop;
    clocktype lifetime;
    int pktSize = sizeof(AODV_RREP_Packet);

    oldRrep = (AODV_RREP_Packet *) GLOMO_MsgReturnPacket(msg);

    memmove(&lifetime, &oldRrep->lifetime, sizeof(clocktype));

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

    pktPtr = (char *) GLOMO_MsgReturnPacket(newMsg);
    newRrep = (AODV_RREP_Packet *) pktPtr;

    newRrep->pktType = oldRrep->pktType;
    newRrep->srcAddr = oldRrep->srcAddr;
    newRrep->destAddr = oldRrep->destAddr;
    newRrep->destSeq = oldRrep->destSeq;
    newRrep->hopCount = oldRrep->hopCount + 1;
    newRrep->lifetime = lifetime;

    if (destAddr == ANY_DEST)
```

```
    {
      NetworkIpSendRawGlomoMessage(
        node, newMsg, ANY_DEST, CONTROL, IPPROTO_AODV, 1);
    }
    else
    {
      nextHop = RoutingAodvGetNextHop(oldRrep->srcAddr, &aodv->routeTable);

      if (nextHop != ANY_DEST)
      {
        NetworkIpSendRawGlomoMessageToMacLayer(
          node, newMsg, nextHop, CONTROL, IPPROTO_AODV, 1,
          DEFAULT_INTERFACE, nextHop);
      }
    }

    aodv->stats.numReplySent++;
    GLOMO_MsgFree(node, msg);

} /* RoutingAodvRelayRREP */
```

VITA

TIRTHANKAR GHOSH

Place of birth Kolkata, India

Education

08/2002 -Present Doctoral Candidate, **Electrical Engineering**, Florida International University, Miami, Florida
Dissertation: *Secure Routing and Trust Modeling in Multihop Infrastructure-less Networks*

01/2001 - 7/2002 Master of Science, **Computer Engineering,** Florida International University, Miami, Florida
Thesis: *Design of a Fast and Resource-efficient Fault Management System in Optical Networks*

07/1990 - 06/1994 Bachelor of Engineering, **Electrical Engineering,** Jadavpur University, India

Work Experience

Telecommunications & Information Technology Institute, Florida International University, Miami, FL, USA (January 2001- present)
Position: Research Assistant
Job Functions and Projects:

- Implementation and maintenance of a wireless ad-hoc network testbed with IBM ThinkPads running NIST_AODV routing protocol on top of Red Hat Linux 9.
- Setting up simulation testbed for running wireless infrastructured and ad-hoc networks with Glomosim simulator having Parsec compiler running on top of Red Hat Linux.
- Developing secure solutions in infrastructure-less wireless networks with large-scale simulation and extending the results to carry out implementation with real-time traffic on the ad-hoc network testbed.
- Developing policy-based trust computational models based on the security vulnerabilities in an ad-hoc wireless network.
- Installation and maintenance of IBM xSeries and pSeries servers in the research labs.
- Implementation of Security algorithms – Implemented RSA and DES to evaluate their performance on laptops running Windows for carrying out research with the ad-hoc network testbed.
- Design and implementation of Protocol Conversion software - Designed and implemented a protocol conversion software at the Data Link layer with C on Windows platform.
- Design and implementation of an IPv4 to IPv6 conversion software at the Network layer – implemented with C on Windows platform.

CESC Ltd., Kolkata, India (September 1994 – December 2000)
Position: Executive, Materials Management Division
Job Functions and Projects:

- System design for a company-wide ERP implementation with Oracle Purchasing, Oracle Financial and Oracle Inventory packages.
- Design, analysis and fine-tuning of the inventory management system.
- Inventory modeling for the Budge Budge Generating Station.
- Implementation of ISO 9002 in the Materials Management division.
- Design and generation of MIS reports for inventory and consumption analysis for the Materials Management division.

Project Summaries:

- ISO 9002 was implemented in the Material Management division integrating four generating stations, ten distribution centers and fifteen warehouses with stringent quality control measures and documentation.
- Enterprise Resource Planning (ERP) was being planned to be implemented with Oracle Purchasing, Oracle Financial and Oracle Inventory packages. System design was carried out to integrate the packages and with customized needs and requirements.
- The inventory management system was designed for Budge Budge Generating Station with approximately 5000 items to cater to two units of 250 MW each.

Publications

Journal and Book Chapters

1. "Towards Designing a Trusted Routing Solution in Mobile Ad Hoc Networks", to appear in the ACM Journal "Mobile Networks and Applications (MONET)" Special issue on Non-Cooperative Wireless Networking and Computing, 2005.

2. "An Overview of Security Issues for Multihop Mobile Ad Hoc Networks", IEC Publications; Network Security: Technology Advances, Strategies, and Change Drivers, ISBN: 0-931695-25-3, 2004.

Technical Conferences

1. "Collaborative Trust-based Secure Routing Against Colluding Malicious Nodes in Multi-hop Ad Hoc Networks", in Proceedings of the 29th IEEE Annual Conference on Local Computer Networks (LCN), Nov 16-18, Tampa, USA, 2004.

2. "Collaborative Trust-based Secure Routing in Multihop Ad Hoc Networks", in Proceedings of The Third IFIP-TC6 Networking Conference (Networking '04): Springer Verlag, Series: Lecture Notes in Computer Science, Vol. 3042, pp. 1446 - 1451, Athens, Greece, May 9-14, 2004 (co-authored with Pissinou, N. and Makki, K.).

3. "Study of Network Performance in a Simulated Network for Optimized Node Degree and Network Cost", Internet Computing'03, Las Vegas, USA, June 2003 (co-authored with Makki, S., Pissinou, N. and Deshpande, A.).

4. "Economic Modeling and Analysis of the Evolution Path in Current and Projected IP-Backbone Networks", Optical Fiber Conference, March 23-28, Atlanta, Georgia, 2003 (co-authored with Wang, J., Pissinou, N. and Makki, K.).