ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

INTRUSION DETECTION SYSTEM FOR MOBILE AD-HOC NETWORKS (MANETs)

BY

KONJIT DESALEGN

APRIL, 2006

INTRUSION DETECTION SYSTEM FOR MOBILE AD-HOC NETWORKS (MANETs)

By

Konjit Desalegn

A thesis submitted to the school of Graduate Studies of Addis Ababa University in partial fulfillment for the Degree of Master of Science in Computer Engineering

Addis Ababa

April, 2006

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

INTRUSION DETECTION SYSTEM FOR MOBILE AD-HOC NETWORKS (MANETs)

BY

KONJIT DESALEGN

Approval by Board of Examiners

Dr. Getachew Biru Chairman, Department of Electrical and Computer Engineering	_____ Signature
Dr. Fisseha Mekuria Senior Expert & Head of Telecom Engineering Department Graduate School of Telecom and IT, GSTIT Advisor	_____ Signature
_____ Examiner	_____ Signature
_____ External Examiner	_____ Signature

Dedicated to my brother ***Mekonnen Dessalegn*** whom I lost two years a go. May God rest his soul beside Abraham.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

MANET	Mobile Ad-hoc Networks
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
LAN	Local Area Network
DSDV	Destination Sequenced Distance Vector
OLSR	Optimized Link State Routing
DSR	Dynamic Source Routing
AODV	Ad-hoc On-demand Distance Vector
FSR	Fisheye State Routing
ARP	Address Resolution Protocol
IDS	Intrusion Detection System
NS-2	Network Simulator Version 2
NAM	Network Animator
CBR	Constant Bit Rate
UDP	User Datagram Protocol
MACT	Multicast Activation
RREQ	Route Request
RREP	Route Reply
RERR	Route Error
WLANs	Wireless Local Area Networks
NIDS	Network based IDS
HIDS	Host based IDS
TTL	Time to Live

Acknowledgement

First and foremost I thank GOD, without him this wouldn't have been possible.

I express my deepest gratitude to my advisor Dr Fisseha Mekuria, for his advice, concern, encouragement, and formal guidance.

My special thanks go to Ato Abyot Asalefew for his encouragement, and for scarifying his precious time whenever I needed help.

Finally, my heartfelt appreciation and indebtedness goes to my family and friends for their love, support, patience and encouragement throughout my study.

Abstract

The term wireless networking refers to technology that enables two or more computing devices to communicate using standard wireless networking protocols. Strictly speaking, any communication technology that uses such a scheme could be called wireless networking. Based on the coverage area, wireless networking technology can be classified as wide area network (ex. Mobile cellular standards such as GSM, IS95) and local area networking (ex. WLAN such as the IEEE 802.11x). Furthemore wireless networks can be classified as infrastructure based, that is a wireless network with a central control device or access point (IEEE WLAN and GSM) , the second type of wireless networks is the so called infrastructure-less or Mobile Ad hoc networking (MANETs).

This thesis deals with the analysis and implementation of two types of intrusion detection techniques for mobile Ad-hoc networks. Since MANETs have no established infrastructure: they are also described as self-organized peer-to-peer wireless network composed of mobile nodes. MANETs represent a dynamic network with no centralized control (i.e., a node can enter or leave a MANET group without collapsing the network). This property allows MANETs to change the network topology of a peer collection of mobile nodes with in the coverage range of each other, dynamically. However, this nature of ad-hoc wireless networks also makes them very vulnerable to an adversary's malicious attacks. To detect those malicious attacks the network needs an intrusion detection system.

Various types of intrusion detection systems for wireless Ad hoc networks exist and are discussed in the next sections. In this thesis performance evaluation of the MANET routing protocol, Ad-hoc on demand distance vector (AODV) is done using the so called misuse IDS technique. Implementation aspect of the IDS for a particular type of attack, called resource consumption attack is done using the NS-

2 simulator, for the AODV routing protocol. A significant decrease in packet drop is obtained due to the application of the misuse intrusion detection algorithm. Other metrics such as energy consumption, delay are considered, and show improvement due to the application of the IDS. An in-depth discussion of the simulation results can be found in section 7.

1. INTRODUCTION

On wireless computer networks, ad-hoc mode is a method for wireless devices to directly communicate with each other. Ad-hoc network consists of a collection of peer mobile nodes that are capable of communicating with each other without help from a fixed infrastructure. The inter-connections between nodes are capable of changing on a continual and arbitrary basis. Applications of ad-hoc networks range from military tactical operations to civil rapid deployment such as emergency search-and-rescue missions, data collection/sensor networks, and instantaneous classroom/meeting room applications.

The nature of ad-hoc networks makes them very vulnerable to an adversary's malicious attacks. Attackers inside an ad hoc network can be of different strength and have different intentions with the attacks. Attacks on ad hoc networks can be categorized to active and passive attacks. In a passive attack the malicious entity only listens to the traffic, without modifying or disturbing it in any way. The main threat by such an attack is that some confidential information is leaked to the attacker. In an active attack the malignant node actively disturbs the normal operation of the network. This can be done e.g. by forging packets, disrupting normal routing or consuming network and node resources.

All the problems presented above need to be dealt with by the routing schemes. Implementing routing in an ad hoc network is a challenging task. The distinctive features of the network make normal routing protocols nearly useless. Protocols for secure routing usually apply cryptography which is one of the intrusion prevention measures and thus come with a significant increase in complexity and computational overhead. Besides that intrusion preventive measures such as encryption and authentication can reduce intrusion but not eliminate them.

Encryption and authentication cannot defend against compromised nodes and the fact that such nodes already carry private keys, which makes the network more vulnerable.

To build a highly secure wireless ad-hoc network, we need to deploy intrusion detection and response techniques. However, intrusion prevention alone is not sufficient because as the systems become ever more complex, while security is still often the after-thought, there are always exploitable weaknesses in the systems due to design and programming errors, or various "social engineering" penetration techniques. Intrusion detection can be used as a second wall to protect network systems.

It's the aim of this paper to study the existing intrusion detection systems and to add intrusion detection feature in one of the Ad-Hoc routing protocols AODV and to evaluate its performance.

1.1 Problem Description

Wireless Ad hoc network is vulnerable to various attacks which are either intentional or non intentional. These attacks target the packets which are forwarded form one node to the other. So to secure these packets which will be routed between nodes a lot of work hast to be done on the routing protocols.

The routing protocols have to detect the intrusion and make major action to stop the various attacks. In order to make the routing protocols secure one has to study the weak points concerning security of the protocols and develop a security system that protect the system. Making the routing protocols secure makes this technology (Ad hoc wireless network) highly reliable and ultimately highly applicable in various sensitive areas like military service.

1.2 Outline

The paper is organized as follows. The first chapter gives a basic introduction and about the aim of thesis. In the next chapter a brief introduction about mobile Ad Hoc networks and their property will be given. In the third chapter the discussion about the Ad Hoc routing protocol will be presented. In this chapter a detailed description of the on the ad hoc on demand distance vector routing protocol (AODV) is discussed. Following this chapter we get chapter which present about different types of intrusions and intrusion detection systems followed by possible attacks applied on routing protocol AODV. In the fifth chapter a discussion of simulation software NS2 will be presented. In the sixth chapter the simulation result will be presented. Following this chapter we will find discussion and future work.

2. MOBILE AD HOC NETWORKS

2.1 Introduction

Since their emergence in 1970's, wireless networks have become increasingly popular in the computing industry. These networks provide mobile users with ubiquitous computing capability and information access regardless of the location. There are currently two variations of mobile wireless networks- infrastructured and infrastructureless networks.

The ***infrastructured networks***, also known as **Cellular network**, have fixed and wired gateways. They have fixed base stations which are connected to other base stations through wires. The transmission range of a base station constitutes a cell. All the mobile nodes lying within this cell connects to and communicates with the nearest bridge (base station). A "handoff" occurs as mobile host travels out of range of one Base Station and into the range of another and thus, mobile host is able to continue communication seamlessly throughout the network.

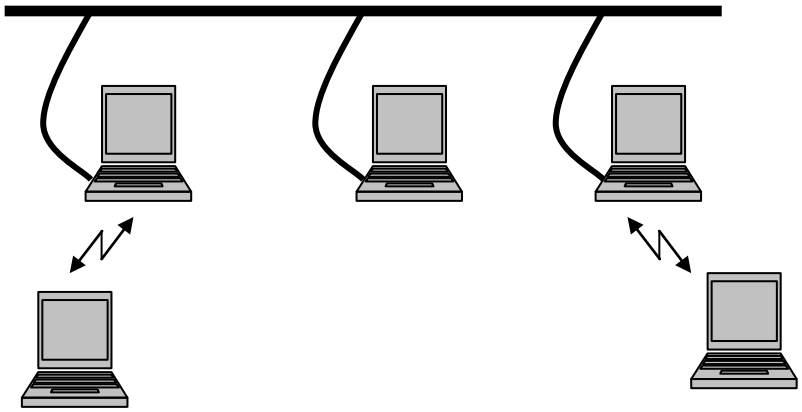

Fig 2.1 Infrastructure Network

The other type of network, ***infrastructureless network***, is known as **Mobile Ad NETwork (MANET).** These networks have no fixed routers. All nodes are capable of movement and can be connected dynamically in arbitrary manner. The responsibilities for organizing and controlling the network are distributed among the terminals themselves. The entire network is mobile, and the individual terminals are allowed to move at will relative to each other. In this type of network, some pairs of terminals may not be able to communicate directly to with each other and relaying of some messages is required so that they are delivered to their destinations.

Such networks are often refereed to as *multihop* or *store-and-forward* networks. The nodes of these networks function as routers, which discover and maintain routes to other nodes in the networks. The nodes may be located in or on airplanes, ships, trucks, cars, perhaps even on people or very small devices.

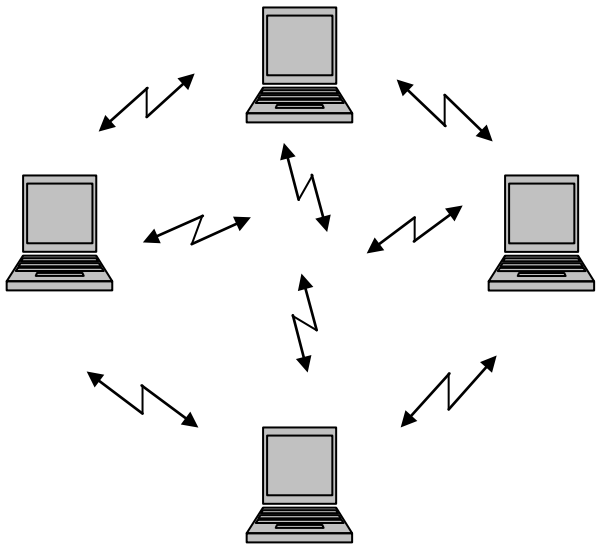

Fig 2.2 Ad Hoc Network

Based on these features, standard cellular networks and wireless fully connected networks do not qualify as ad hoc networks. Probably, the chief difference between ad hoc networks is the apparent lack of a centralized entity within an ad hoc network. There are no base stations or mobile switching centers in an ad hoc network. It is tried to compare the WLAN and Ad hoc networks using Kiviat graph.

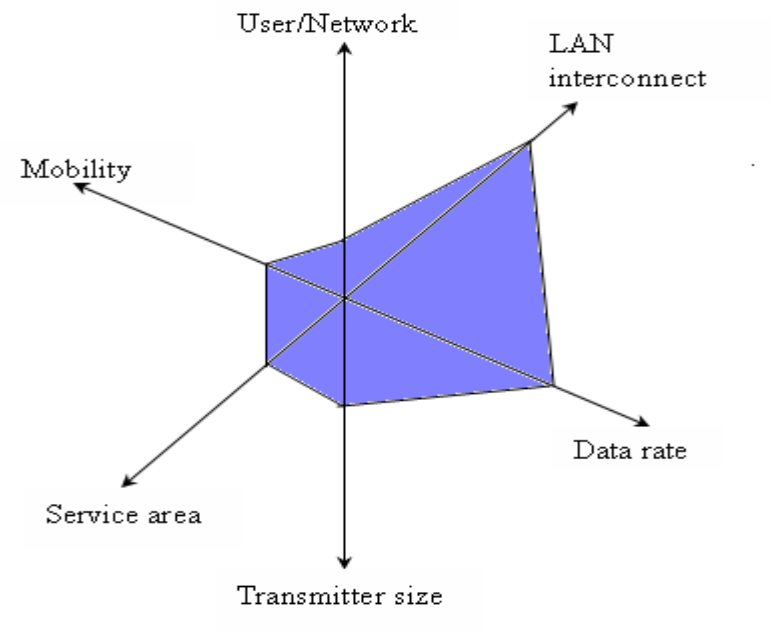

Fig 2.3 Kiviat Graph for WLAN

A kiviat graph provides a pictorial means of comparing systems along multiple variables. The variables are laid out at equal angular intervals. A given system is defined by one point on each variable; these points are connected to yield a shape that is characteristic of that system. The Kiviat graph for the ad hoc network is not based on the real value. It is based on the theoretical knowledge on the characteristics of the system.

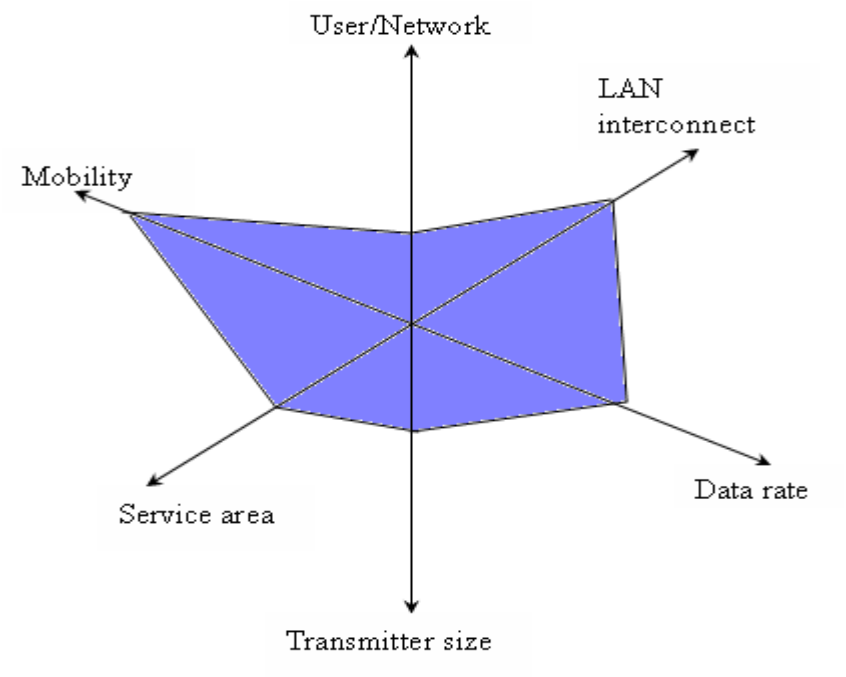

Fig 2.4 Kiviat Graph for Ad hoc Network

The interest in wireless ad hoc networks stems from of their well known advantages for certain types of applications. Since, there is no fixed infrastructure; a wireless ad hoc network can be deployed quickly. Thus, such networks can be used in situations where either there is no other wireless communication infrastructure present or where such infrastructure cannot be used because of security, cost, or safety reasons. Such a network is tolerant of the failure or departure of terminals, because the network does not rely on a few critical terminals for its organization or control. Similarly, terminals can be added easily to the network. The main applications lie in automated battlefield where tanks in a battalion may wish to establish an immediate communication in 'harsh' environments, where there is no time or resource for laying down the wired infrastructure. Applications such as rescue missions in times of natural disasters, law enforcement operation, commercial and educational use, and sensor networks are just few possible commercial examples.

Currently, there is no method available which enables mobile computers with wireless data communications equipment to freely roam about while still maintaining connections with the other, unless special assumptions are made about the way the computers are situated with respect to each other. One mobile computer may often be able to exchange data with two other mobile computers which cannot themselves directly exchange data. As a result, computer users in a conference room may be unable to predict which of their associates' computers could be relied upon to maintain network connection, especially as the users moved from place to place.

2.2 Properties of Ad hoc Networks

As already mentioned, ad hoc networks do not rely on pre-existing infrastructure and this may be their most distinguishing attribute. Instead ad hoc networks are formed by individual nodes when they come to close proximity and need to communicate with each other. This implies that there is no need for stationary components such as routers, bridges and cables and of course central administration is not required.

Due to the lack of stationary infrastructure, the participating nodes in the ad hoc network have to forward traffic on behalf of other nodes that are not in close proximity to the destination node. If they deny participating in the routing process, the connectivity between nodes may be lost and the network could be segmented. Therefore, the functionality of an ad hoc network heavily depends on the forwarding behavior of the participating nodes.

Ad hoc networks can be characterized as autonomous in the sense that most commonly they offer connectivity between the participating nodes and not connectivity to external LANs or internets. However in theory it is possible that

some of the ad hoc nodes are multi-homed with connections in both the ad hoc network and one or more external networks. Nothing prevents these nodes from acting as gateways between these networks but is not a common element.

Another very important property of ad hoc networks is their dynamic topology. Since the topology arbitrarily changes due to node mobility and changes of the surrounding environment, the utilized routing protocols have to be able to adapt to the dynamic topology. Traditional wired routing protocols like OSPF do not incorporate in their normal operation support for frequent network topology changes. Thus, the routing protocols that are currently utilized in ad hoc environments have specifically been designed to handle node mobility and rapidly changing topologies.

The devices that are usually employed in the ad hoc networks have their own limitations. Since, the only hardware component that is required to connect a device in an ad hoc network is a wireless interface, PDAs and mobile telephones can be utilized. Furthermore, differences in the radio transmission ranges and reception equipment sensitivities may lead to unidirectional links which could complicate routing in the ad hoc networks. Apart from the communication differences between the nodes, ad hoc networks suffer from limited hardware resources like limited battery, constrained CPUs and small memory capacity. The general Characteristics of MANETs is listed below:

Dynamic Topologies: Since nodes are free to move arbitrarily, the network topology may change randomly and rapidly at unpredictable times. The links may be unidirectional bi-directional.

Bandwidth constrained, variable capacity links: Wireless links have significantly lower capacity than their hardwired counterparts. Also, due to multiple access,

fading, noise, and interference conditions etc. the wireless links have low throughput.

Energy constrained operation: Some or all of the nodes in a MANET may rely on batteries. In this scenario, the most important system design criteria for optimization may be energy conservation.

Limited physical security: Mobile networks are generally more prone to physical security threats than are fixed cable networks. There is increased possibility of eavesdropping, spoofing and denial-of-service attacks in these networks.

3. ADHOC ROUTING PROTOCOLS

3.1 Routing Protocols

Routing support for mobile hosts is presently being formulated as "mobile IP" technology. When the mobile agent moves from its home network to a foreign (visited) network, the mobile agent tells a home agent on the home network to which foreign agent their packets should be forwarded. In addition, the mobile agent registers itself with that foreign agent on the foreign network. Thus, all packets intended for the mobile agent are forwarded by the home agent to the foreign agent which sends them to the mobile agent on the foreign network. When the mobile agent returns to its original network, it informs both agents (home and foreign) that the original configuration has been restored. No one on the outside networks need to know that the mobile agent moved. But in ad-hoc networks there is no concept of home agent as it itself may be "moving".

Supporting Mobile IP form of host mobility (or nomadicity) requires address management, protocol inter operability enhancements and the like, but core network functions such as hop-by hop routing still presently rely upon pre-existing routing protocols operating within the fixed network. In contrast, the goal of mobile ad hoc networking is to extend mobility into the realm of autonomous, mobile, wireless domains, where a set of nodes, which may be combined routers and hosts, themselves form the network routing infrastructure in an ad hoc fashion. Hence, the need to study special routing algorithms to support this dynamic topology environment.

In ad hoc networking environments an application packet from a specific node may have to travel several hops in order to reach its destination. The main function

of a routing protocol is to form and maintain a routing table with information relevant to which the next hop for this packet should be in order to reach its ultimate destination. All the nodes have their own routing tables that they consult to forward the routing traffic that it is not destined for them.

Although the problem of routing is not a new one in computer networks, routing in ad hoc networks due to its unique requirements cannot be successfully handled by utilizing existing routing schemes such as traditional link-state and distance vector routing protocols. One of the reasons that for example OSPF and RIP cannot be used in ad hoc networks is that these protocols where originally designed to operate in environments with relatively static topology. However, the nature of the ad hoc networks allows the participating nodes to move freely in and out of the network.

Another issue that contributes to the fact that the available routing protocols cannot operate in ad hoc mode is that they were designed with the assumption that all the links are bidirectional. In mobile ad hoc networks this is not always the case. The differences of the wireless networking hardware of the nodes or the radio signal fluctuations may result in some links becoming unidirectional.

Finally, both OSPF and RIP attempt to maintain routes to all the reachable destinations, but in ad hoc networks with high density this may lead in having very large numbers of routing entries imposing performance overhead. Therefore, there is a need for special routing protocols that will be able to cope with the unique attributes and limitations of mobile wireless ad hoc networks.

3.2 Properties of Ad hoc Routing Protocols

As it is clear from the previous analysis, there is a special need for routing protocols specifically designed to address the requirements of ad hoc networking. Some of the properties that ad hoc routing protocols should posses are suggested in and are analyzed below:

Distributed operation: Route computation must be distributed because centralized routing in a dynamic network is impossible even for fairly small networks.

Loop-freedom: Though, if not incorporated in the routing protocol, the TTL value could be used to prevent the packet from roaming in the network for arbitrarily long periods of time. But, still this property is desirable for efficient use of resources and better overall performance.

Demand-based operation: The routing algorithm should adapt to traffic on demand or need basis for efficient utilization of network energy and bandwidth resources. Though the obvious drawback would be increased delay.

Proactive operation: In some contexts additional latency, incurred due to demand based operation, may be unacceptable. So, if the resources and bandwidth permit, proactive operation must be used.

Security: While security concerns exist within wired infrastructures and routing protocols as well, maintaining the "physical" security of the transmission media is harder in practice with MANETs. Sufficient security protection to prohibit disruption of modification of protocol operation is desired.

"***Sleep" period operation:*** As a result of energy conservation or some other need to be inactive, nodes of a MANET may stop transmitting and/or receiving (even receiving requires power) for arbitrary time periods. A routing protocol should be able to accommodate such sleep periods without overly adverse consequences. This property may require close coupling with the link-layer protocol through a standardized interface.

Unidirectional link support: Many algorithms typically assume bi-directional links and thus, are incapable of functioning properly over unidirectional links. Nevertheless, unidirectional links can and do occur in wireless networks. Generally, a sufficient number of duplex links exist so that usage of unidirectional links is of limited added value. However, in situations where a pair of unidirectional links (in opposite directions) form the only bi-directional.

3.3 Types of Routing Schemes

There are two types of routing schemes, the table driven and on demand routing. Both routing schemes will be discussed in detail below.

3.3.1 Table-driven Ad hoc Routing Protocols

The table-driven operation, also known as proactive, requires that a node maintains a routing table that contains routing information regarding the connectivity to all other nodes that participate in the ad hoc network. When a node needs to forward a packet, the route is readily available; thus there is no delay in searching for a route. Any changes in the topology are periodically propagated by means of updates throughout to the entire network to ensure that all nodes share a consistent view of the network.

However, proactive behavior suffers from the disadvantage of additional control traffic that is required to continually update old route entries. Due to the mobility of the nodes the routes are likely to be broken frequently, thus two mobile nodes that had established a link between them will no longer be able to support that link and subsequently any other routes that were depended on that link. If the broken route has to be repaired even though no applications are using it the effort can be considered wasted. For a highly dynamic topology, the proactive schemes spend a significant amount of scarce wireless resource in keeping the complete routing information correct. Two well known protocols that follow the table-driven design approach are the Destination-Sequence Distance-Vector (DSDV) and the Optimised Link State Routing (OLSR) protocol.

3.3.2 On-demand Ad hoc Routing Protocols

An alternative approach to the one proposed by table-driven protocols is the on-demand approach, also known as a reactive behaviour. These types of routing protocols create routes only when they are required by the source node. Therefore, when a route is desired a route discovery process is initiated by the source node within the network. Therefore, the communication overhead is reduced at the expense of delay due to route search. The application packets transmitted while the route discovery process is in progress are buffered and sent after the route has been established. Once the route path has been established, the route maintenance process is not triggered until either the destination becomes inaccessible or until the route is no longer required.

These schemes are significant for the ad-hoc environment since battery power is conserved both by not sending the advertisements and by not needing to receive them (since a host could otherwise reduce its power usage by putting itself into the "sleep" or "standby" mode when not busy with other tasks). The Ad hoc On-

demand Distance Vector (AODV) protocol and the Dynamic Source Routing (DSR) protocol are examples of the on-demand routing protocols.

3.4 Ad-hoc On-demand Distance Vector (AODV) Routing Protocol

A large overhead, especially in large networks, is incurred by carrying source routes in each data packet. Thus instead of using source routing, AODV, relies on dynamically establishing route table entries at intermediate nodes. To solve the problem of stale cached routes, concept of sequence number is used.

The Ad-Hoc On Demand Distance Vector (AODV) routing algorithm is a routing protocol designed for ad-hoc mobile networks. It is capable of both unicast and multicast routing. It is an on demand algorithm, meaning that it builds routes between nodes only as desired by source nodes. It maintains these routes as long as they are needed by the sources. Additionally, AODV forms trees which connect multicast group members. The trees are composed of the group members and the nodes needed to connect the members.

AODV enables nodes to communicate with other nodes they are not in range of by routing packets through neighbouring nodes. The AODV protocol discovers these routes that packets may take between a source and destination. The protocol does this while ensuring that routing loops do not occur, and it also attempts to find the shortest route possible. It can handle changes in routes and discovers new routes when an old route no longer works.

AODV builds routes using a route request / route reply query cycle. When a source node desires a route to a destination for which it does not already have a route, it broadcasts a route request (RREQ) packet across the network. Nodes

receiving this packet update their information for the source node and set up *backwards pointers* to the source node in the route tables. In addition to the source node's IP address, current sequence number, and broadcast ID, the RREQ also contains the most recent sequence number for the destination of which the source node is aware.

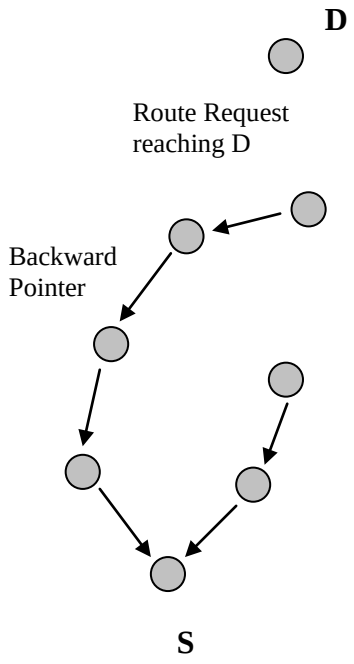

Fig 3.1 RREQ flow

The source sequence number is used to maintain freshness information about the reverse route to the source. This is to make sure that packets from the destination to the source follow a path, which was set up by the route request packet for this session. The destination sequence number specifies how fresh a route to the destination must be before it can be accepted by the source.

A node receiving the RREQ may send a route reply (RREP) if it is either the destination or if it has a route to the destination with corresponding sequence number greater than or equal to that contained in the RREQ. If this is the case, it

unicasts a RREP back to the source. Otherwise, it rebroadcasts the RREQ. Nodes keep track of the RREQ's source IP address and broadcast ID. If they receive a RREQ, which they have already processed, they discard the RREQ and do not forward it.

As the RREP propagates back to the source, nodes set up *forward pointers* to the destination. Once the source node receives the RREP, it may begin to forward data packets to the destination. If the source later receives a RREP containing a greater sequence number or contains the same sequence number with a smaller hop count, it may update its routing information for that destination and begin using the better route.

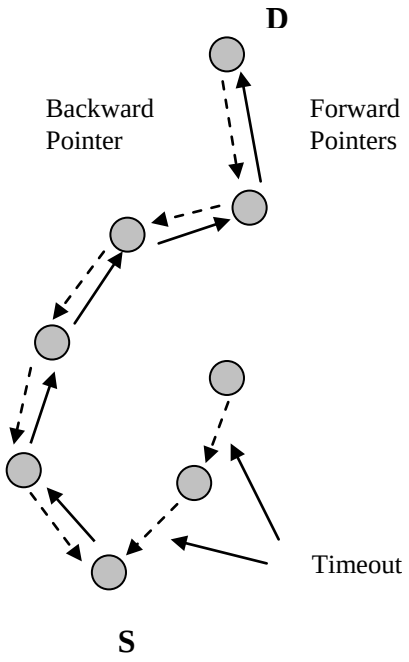

Fig 3.2 RREP flow

As long as the route remains active, it will continue to be maintained. A route is considered active as long as there are data packets periodically traveling from the source to the destination along that path. Once the source stops sending data

packets, the links will *time out* and eventually be deleted from the intermediate node routing tables. If a link break occurs while the route is active, the node upstream of the break propagates a route error (RERR) message to the source node to inform it of the now unreachable destination(s). After receiving the RERR, if the source node still desires the route, it can reinitiate route discovery.

Multicast routes are set up in a similar manner. A node wishing to join a multicast group broadcasts a RREQ with the destination IP address set to that of the multicast group and with the 'J'(join) flag set to indicate that it would like to join the group. Any node receiving this RREQ that is a member of the multicast tree that has a fresh enough sequence number for the multicast group may send a RREP. As the RREPs propagate back to the source, the nodes forwarding the message set up pointers in their multicast route tables. As the source node receives the RREPs, it keeps track of the route with the freshest sequence number, and beyond that the smallest hop count to the next multicast group member.

After the specified discovery period, the source node wills unicast a Multicast Activation (MACT) message to its selected next hop. This message serves the purpose of activating the route. A node that does not receive this message that had set up a multicast route pointer will timeout and delete the pointer. If the node receiving the MACT was not already a part of the multicast tree, it will also have been keeping track of the best route from the RREPs it received. Hence it must also unicast a MACT to its next hop, and so on until a node that was previously a member of the multicast tree is reached.

AODV maintains routes for as long as the route is active. This includes maintaining a multicast tree for the life of the multicast group.

Advantages: It uses bandwidth efficiently (by minimizing the network load for control and data traffic), is responsive to changes in topology, is scalable and ensures loop free routing.

The AODV protocol consists of a number of messages which it uses for route discovery, route maintenance and repair, and neighbour detection.

3.4.1 Route Request (RREQ) Messages

When a node needs to send a message to another node that is not its direct neighbour, it broadcasts a Route Request message to initiate the discovery of a route. The RREQ message contains several important bits of information: the source IP address, the destination IP address, the lifespan of the message and a sequence number which uniquely identifies messages from this source.

When the neighbours of the node who initiated the Route Request receive the message, they can do one of two things: if they know of a route to the destination or they themselves are the destination, they can unicast a Route Reply (RREP) message back to the source node; otherwise, they will rebroadcast the Route Request to their neighbours.

The lifespan of the Route Request is decremented by one at each hop, and the message is simply discarded when the lifespan reaches zero. In this manner, the protocol can implement an expanding ring search, in which the lifetime of an initial Route Request is set to a low number to limit the propagation of RREQ messages. If no reply is received within a specified amount of time, the source node issues a new Route Request with a new sequence number and higher lifetime. A number of different attempts can be made using successively larger lifetimes, or after a fixed number of retries, the lifetime is set to be greater than the

network diameter, so the Route Request will be broadcast to all nodes connected in the network.

All the nodes keep a list of the Route Requests, including sequence numbers, for a particular source that they have rebroadcast in a fixed interval, to ensure they do not rebroadcast the Route Request more than once.

Type	J	R	Reserved	Hop Count
Broadcast ID				
Destination IP Address				
Destination Sequence Number				
Source IP Addtess				
Source Sequence Number				

Fig 3.3 The format of the Route Request message is illustrated above

RREQ message contains the following fields:

Type xx

J Join flag; set when source node wants to join multicast group.

R Repair flag; set when a node wants to initiate a repair to connect two previously disconnected portions of the multicast tree.

Reserved Sent as 0; ignored on reception.

Hop Count

The number of hops from the Source IP Address to the node handling

the request.

Broadcast ID

A sequence number identifying the particular RREQ uniquely when taken in conjunction with the source node's IP address.

Destination IP Address

The IP address of the destination for which a route is desired.

Destination Sequence Number

The last sequence number received in the past by the source for any route towards the destination.

3.4.2 Route Reply (RREP) Messages

When a node contains an up-to-date route to a destination that is the target of a Route Request it receives, or is the destination itself, it unicasts a Route Reply (RREP) message back to the node it received the Route Request from. Each node along the path that the Route Request was propagated updates its routing table to mark the node from which it received the Route Reply as the next hop for the new route. As such, the Route Reply is propagated along the reverse path all the way to the source of the original Route Request and the routing table of each node along the way is updated to reflect the next hop along the route.

In case the node replying to the Route Request was not the destination but instead knew a valid route to the destination, then this node also sends a *gratuitous* Route Reply to the destination along the path it knows to that destination, such that the destination knows how to reply to the source when it receives data from it,

without having to explicitly send out another Route Request to search for the source.

Sequence Numbers: The protocol uses sequence number to ensure loop freedom in routes and to act as a kind of timestamp such that nodes may detect when they receive more up to date routing information. Each node maintains its own sequence number, which it increases any time it sends out any kind of message. Each node maintains a record of the sequence number of all the nodes it has routing information for. A higher sequence number indicates a fresher route. Thus it is possible for other nodes to determine which Route Reply message has more up-to-date information. Nodes may for example update their routes to a destination if they observe a Route Reply that contains a higher sequence number for the destination than the one stored in their routing tables.

Type	L	Reserved	Hop Count
Destination IP Address			
Destination Sequence Number			
Lifetime			

Fig 3.4 The format of the Route Reply message is illustrated above

RREP message contains the following fields:

Type xx

Reserved

Sent as 0; ignored on reception.

Hop Count

The number of hops from the Source IP Address to the Destination IP Address. For multicast route requests this indicates the number of hops to the multicast grouphead.

L If the 'L' bit is set, the message is a "hello" message and contains a list of the node's neighbors.

Destination IP Address

The IP address of the destination for which a route is supplied.

Destination Sequence Number

The destination sequence number associated to the route.

Lifetime

The time for which nodes receiving the RREP consider the route to be valid.

Route Maintenance Process: In addition to the route discovery process just outlined, AODV is responsible for maintaining active routes in the network. Routes are only kept for as long as they are in use. After a timeout period, stale routes will be removed from a node's routing table. The route maintenance process is also concerned with detecting route breakages. Each node in the network monitors its connectivity to neighbours that are being used as next hops for active routes. It can use link layer notification methods to detect route breakages. For example, in the 802.11 standard, the absence of a link layer ACK or failure to get a CTS after sending an RTS, even after the maximum number of retransmission attempts, indicates loss of the link to this active next hop.

In the absence of link layer information, a node uses passive acknowledgement to detect broken links. Receipt of packets from the next hop, including HELLO messages are usually used for this process, described next.

3.4.3 HELLO Messages

In order for nodes to remain aware of who their neighbours are, they may periodically broadcast HELLO messages. HELLO messages are simply Route Reply messages sent with a hop count of zero, so it is not propagated. A node keeps track of its neighbours by listening for the periodic messages. After an allowable HELLO message loss, a node will detect a broken link by the absence of a HELLO message, indicating that the nodes can no longer directly communicate. If this link was in use by any active routes, this broken-link detection mechanism will result in the sending of a Route Error (RERR) message, as described next.

Type	U	M	Reserved	Hop Count
Group Leader IP Addess				
Multicast Group IP Address				
Multicast Group Sequence Number				

Fig 3.5 The format of the Group Hello message is illustrated above

Hello message contains the following fields:

Type 5

U Update flag; set when there has been a change in group leader information.

M Off_Mtree flag; set by a node receiving the group hello that is not on the multicast tree.

Reserved Sent as 0; ignored on reception.

3.4.4 Route Error (RERR) Messages

Route Error (RERR) messages allow AODV to adjust routes when nodes move around or otherwise lose the ability to transmit to one or more of their neighbours. When a node receives a Route Error message, it removes all the routes from its routing tables that contain the invalid next hop. There are three circumstances in which a node will broadcast a Route Error message.

Type	Reserved	Hop Count
Unreachable Destination IP Addess(1)		
Additional Unreachable Destination IP Addess(if needed)		

Fig 3.6 The format of the Route Error message is illustrated above

RERR message contains the following fields:

Type 3

Length The number of unreachable destinations included in the message. Must be at least 1.

Unreachable Destination IP Address

The IP address of the destination which has become unreachable due to a link break.

If a node receives data from a neighbouring node for a destination to which it has no route, it will broadcast a Route Error message. In this case the neighbouring node had some stale or otherwise incorrect routing information. In the second scenario the node receives a Route Error message that causes at least one of its routes to become invalidated. If this happens, the node then sends out a Route Error message with all the new Nodes which are now unreachable. In the final scenario, if a node detects (through the absence of HELLO messages, or via other link level notification methods) that it can no longer communicate with one of its neighbours, it will check its routing table for all routes that use this neighbour as the next hop, and mark them as invalid. It then sends out a Route Error message for the neighbour and the invalid routes.

Local Repair Mechanism: When a link break occurs in an active route, a node upstream of the break may choose to attempt to repair the route locally if it is within a certain number of hops away from the destination. In this case it issues a Route Request for the destination, and defers sending the Route Error message unless the local repair mechanism is unsuccessful. Incoming packets at the node upstream of the break should be buffered by the node until the local repair is complete.

4. INTRUSION DETECTION SYSTEM

4.1 Attacks and threats on ad hoc networks

The nature of ad-hoc networks makes them very vulnerable to an adversary's malicious attacks. Attackers inside an ad hoc network can be of different strength and have different intentions with the attacks.

a) The use of wireless links renders a wireless ad-hoc network susceptible to attacks ranging from passive eavesdropping to active interfering. Unlike wired networks where an adversary must gain physical access to the network wires or pass through several lines of defense at firewalls and gate ways, attacks on a wireless ad-hoc network can come from all directions and target at any node. Damages can include leaking secret information, message contamination, and node impersonation. All these mean that an ad-hoc network will not have a clear line of defense, and every node must be prepared for encounters with an adversary directly or indirectly.

b) Mobile nodes are autonomous units that are capable of roaming independently. This means that nodes with inadequate physical protection are receptive to being captured, compromised, and hijacked. Since tracking down a particular mobile node in a large scale ad-hoc network cannot be done easily, attacks by a compromised node from within the network are far more damaging and much harder to detect. Therefore, any node in a wireless ad-hoc network must be prepared to operate in a mode that trusts no peer.

c) Decision-making in ad-hoc networks is usually decentralized and many ad-hoc network algorithms rely on the cooperative participation of all nodes. The lack of centralized authority means that the adversaries can exploit this vulnerability for new types of attacks designed to break the cooperative algorithms.

The main threats to an ad hoc network routing protocol are as follows. This list also provides the basis for a generic list of security requirements.

Confidentiality. The primary confidentiality threat in the context of routing protocols is to the privacy of the routing information itself, which leads to a secondary privacy threat to information such as the network topology, geographical location, etc.

Integrity. The integrity of a network depends on all nodes in the network following correct routing procedures so that every node has correct routing information. Therefore threats to integrity are those, which either introduce incorrect routing information or alter existing information.

Availability. This is defined as access to routing information at all times upon demand. If a route exists to a mobile node, then any node should be able to get that route when they require it. Also a routing operation should not take an excessive amount of time to perform, delaying a node from receiving up-to-date route information. Related to this, a node should be able to carry out normal operations without excessive interference caused by the routing protocol or security.

Authorization. An unauthorized node is one which is not allowed to have access to routing information, and is not authorized to participate in the ad hoc routing protocol. There is no assumption that there is an explicit and formal protocol,

simply an abstract notion of authorization. However, as discussed below, formal identity authentication is a very important security requirement, needed to provide access control services within the ad hoc network.

Dependability and reliability. One of the most common applications for ad hoc networks is in emergency situations when the use of wired infrastructure is infeasible. Hence, routing must be reliable, and emergency procedures may be required. For example, if a routing table becomes full due to memory constraints, a reactive protocol should still be able to find an emergency route to a given destination.

Accountability. This will be required so that any actions affecting security can be selectively logged and protected, allowing for appropriate reaction against attacks. As explained below, the misbehaviors demonstrated by different types of nodes will need to be detected, if not prevented. Event logging will also help provide non-repudiation, preventing a node from repudiating involvement in a security violation.

4.2 Internal and External Threats

Threats are classified as external and internal attacks. External attacks are performed by unauthorized nodes or entities. These threats are likely to be more easily detected than threats from internal nodes.

Internal attacks are posed by internal nodes, i.e. they are performed by authorized nodes within the ad hoc network. These threats are thus likely to be more difficult to detect as they arise from trusted sources.

In the text below, 'correct' data packets and 'correct' procedures are simply those that adhere strictly to the routing protocol being used. By contrast 'incorrect' data packets and 'incorrect' procedures are those, which are in any way different to the format and behavior as stated in the protocol. 'False' data packets are data packets that are of the correct protocol format, but contain false information.

4.2.1 External Threats

In the presence of an authentication protocol to protect the upper layers, external threats are directed at the physical and data link layers. Physical layer security is intrinsically difficult to provide due to the possibly mobile nature of ad hoc nodes.

We divide external threats into two major categories: passive eavesdropping, where the adversary simply listens to transmitted signals, and active interference, where the opponent sends signals or data designed to disrupt the network in some way.

4.2.1.1 Passive Eavesdropping

This can allow unauthorized principals to listen to and receive messages including routing updates. An unauthorized node will be able to gather data that can be used to infer the network topology, and other information such as the identities of the more heavily used nodes, which forward or receives data. Hence, techniques may be needed to hide such information. Eavesdropping is also a threat to location privacy. Note that passive eavesdropping also allows unauthorized nodes to discover that a network actually exists within a geographical location, by just detecting that there is a signal present. Traffic engineering techniques have been developed to combat this.

4.2.1.2 Active Interference

The major threat from active interference is a denial of service attack caused by blocking the wireless communication channel, or distorting communications. The effects of such attacks depend on their duration, and the routing protocol in use. With regard to the routing of data packets, reactive routing protocols may see a denial of service attack as a link break. Route maintenance operations will cause most protocols to report the link as broken so that participating nodes can find an alternative route. Proactive routing protocols do not react immediately to non-delivery of data packets. If the route is believed to be broken, it will eventually be timed out and deleted.

Probably the most serious type of denial of service attack is a sleep deprivation torture attack, where node energy is deliberately wasted. With limited power and resources, prevention of such attacks is of utmost importance. Security against such attacks has already been extensively studied and developed by the military for packet radio networks. Spread spectrum technology is designed to be resistant to noise, interference, jamming, and unauthorized intrusion. It is prudent to note that protection against sleep deprivation torture can not be achieved at the physical layer, even though power constraint is in deed a physical layer attribute. The fact that power levels affect all ad hoc network operations makes securing such networks particularly difficult.

There are also threats to integrity, e.g. where an external attacker can attempt to replay old messages, or change the order of messages. Old messages may be replayed to reintroduce out-of-date information. Out-of-date routing information could lead to further denial of service attacks as nodes try to use old but invalid routes, or delete current valid routes. If the routing protocol utilizes neighbor sensing by monitoring received data packets, replaying old packets may falsely

lead nodes in to believing that an 'old' link with a neighbor has become active and usable again.

4.2.2 Internal Threats

The threats posed by internal nodes are very serious, as internal nodes will have the necessary information to participate in distributed operations. Internal nodes can misbehave in a variety of different ways; we identify four categories of misbehavior failed nodes, badly failed nodes, selfish nodes and malicious nodes.
Note that two misbehaving nodes within the same category may exhibit different degrees of incorrect node behavior. For example, some nodes will be more selfish than others. Also, a node may demonstrate behaviors from more than one category indeed; this may even be the typical case.

4.2.2.1 Failed Nodes

Failed nodes are simply those unable to perform an operation; this could be for many reasons, including power failure and environmental events. The main issues for ad hoc routing are failing to update data structures, or the failure to send or forward data packets, including routing messages. This is important as those data packets may contain important information pertaining to security, such as authentication data and routing information. A failure to forward route error messages will mean that originator nodes will not learn of broken links and continue to try to use them, creating bottlenecks. The threat of having failed nodes is most serious if failed nodes are needed as part of an emergency route, or form part of a secure route.

4.2.2.2 Selfish Nodes

Selfish nodes exploit the routing protocol to their own advantage, e.g. to enhance performance or save resources. Selfish nodes are typified by their unwillingness to cooperate as the protocol requires whenever there is a personal cost involved, and will exhibit the same behaviors as failed nodes, depending on what operations they decide not to perform. Packet dropping is the main attack by selfish nodes, where most routing protocols have no mechanism to detect whether data packets have been forwarded, DSR being the only exception. Thus, another pattern of behavior to consider is partial dropping, which could be difficult to prevent and detect. It is important to emphasize that, in this model, selfish nodes do not perform any action to compromise network integrity by actively introducing incorrect information.

4.2.2.3 Malicious Nodes

Malicious nodes aim to deliberately disrupt the correct operation of the routing protocol, denying network services if possible. Hence, they may display any of the behaviors shown by the other types of failed nodes. The impact of a malicious node's actions is greatly increased if it is the only link between groups of neighboring nodes.

4.2.2.3.1 Denial of Service Attacks

The nodes in an ad hoc network are usually considered to be small handheld devices, like mobile phones or PDAs and that they operate on batteries. This means that minimizing power consumption is a high priority for the nodes. At the same time the nodes in the network are required to do extra work by acting as router for

the network. These two requirements can be seen as contradictory, because routing consumes batteries by utilizing processing power and the transmission channel.

Because of the nature of the nodes, different kinds of denial of service, or DoS attacks are possible. DoS can be achieved through consuming resources to such an extent that normal communication becomes impossible. A DoS attack can be targeted to exhaust the transmission channel or alternatively resources of the nodes.

DoS by disturbing the transmission channel is the easiest way to create DoS on a limited area. An attacker can set up a station with a powerful transmitter and flood the channel. To save energy he could also try to start transmitting random data every time a legitimate node tries to initiate contact with another node. The effect is the same; all the nodes within transmission range of the attackers' transmission device will be unable to initiate any proper transmission because of collisions. It should be noted that a determined attacker can use much higher transmission power than what the nodes are capable of. Likewise disturbing transmission is much easier than correctly transmitting data, so the attacker has the upper hand. Solving this problem would require some changes to the hardware of the nodes.

DoS attacks targeted at the nodes aim to consume the scarce resources of the nodes. DoS can either be momentary or persistent. The former can be achieved by momentarily exhausting e.g. the memory or the CPU to the extent that normal operation momentarily becomes impossible. The attacked node can however recover after a fairly short period of time. Persistent DoS can be achieved by the attacker by exhausting the batteries of honest nodes. This can be achieved by constantly sending data to a node over a long period of time. This is even more efficient if some kind of encryption is used, then the receiving node needs to do CPU-intensive decryption on packets that arrive to the node.

4.2.2.3.2 Misdirecting Traffic

As previously mentioned, a malicious node can usually masquerade by just using a false source address in the data packets it sends. In FSR, nodes examine the IP Header source address and use it as a neighbour address. If the malicious node uses a false address, which belongs to another node, then it can affect network integrity by getting all nodes in the network to point their routes to the malicious node, instead of the true owner of the source address. A malicious node can do this in a reactive protocol by replying to route requests before the original owner can, and the same effect can be achieved in a proactive protocol where the malicious node just advertises false routes in the hope they get accepted before the true routes. The malicious node will then receive any information, which was intended for the owner of the address. This attack has been named the 'black hole' attack, akin to the celestial structure, which sucks in all objects and matter. Its also possible for the malicious node to prevent a given node from communicating with any other node in the network, node isolation attack.

4.2.2.3.3 Routing disruption attack

In routing disruption the attacker uses the weaknesses in routing protocols to prevent the network from working in an optimal way. Routing loops can with some routing algorithms be inserted with carefully selected routing packets. A routing loop is a path that travels through the same node more than once. Routing loops cause packets to be sent by the same nodes over and over again until the TTL-field is exhausted. Routing loops can be used to create DoS, because sending packets consumes resources from all the nodes in the loop. The destination node or nodes can also be isolated from the rest of the network, if none of the packets sent to them reach their destination.

Routing loops can be created also by accident, because the nodes are allowed to move relative to each other. This is why most routing algorithms contain some kind of protection against routing loops.

All the problems presented above need to be dealt with by the routing schemes. Traditional ad hoc routing only protect against accidental modifications or replications of messages. An intelligent adversary can however manufacture messages the precise way that allows him to circumvent the security measures.

Ad-hoc network has inherent vulnerabilities that are easily preventable. To build a highly secure wireless ad-hoc network, we need to deploy intrusion detection and response techniques. However, intrusion prevention alone is not sufficient because as the systems become ever more complex, while security is still often the after-thought, there are always exploitable weaknesses in the systems due to design and programming errors, or various "social engineering" penetration techniques. Intrusion detection can be used as a second wall to protect network systems.

4.3 Intrusion Detections

Intrusion detection is defined as the method to identify "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource". It is pertaining to techniques that attempt to detect intrusion into a computer or network by observation of actions, security logs, or audit data. Hence in the context of wireless ad-hoc network, we need to identify any malicious nodes either from outside the network trying to break into or nodes that have turned bad. Bad nodes can easily disrupt or partition the network using the various forms of attacks as seen from the previous section.

Detection of break-ins or attempts is done either manually or via software expert systems that operate on logs or other information available on the network. Humans can detect much more types of intrusions manually but we are interested in using automated systems that can study the audit data via certain mechanisms or rules. When working on intrusion detection, there are some primary assumptions to be made. Firstly, user and program activities are observable, that is the information regarding the usage of a system by a user or program must be recordable and analyzable. Secondly and more importantly, normal and intrusive behaviors must have distinct characteristics. Why is there a need for intrusion detection in wireless ad-hoc network? Isn't intrusion prevention enough?

Intrusion preventive measures such as encryption and authentication can reduce intrusion but not eliminate them. Encryption and authentication cannot defend against compromised nodes and the fact that such nodes already carry private keys, which makes the network more vulnerable. The dynamic nature of the ad-hoc network also means that trust between nodes in the network is virtually non-existent. Without trust, preventive measures are unproductive and measures that rely on a certain level of trust between nodes are susceptible attacks themselves.

Another reason for not just having intrusion prevention is that it is often an after-thought during the design and development stages of computer systems. This makes room for loopholes in the system, which people can exploit. As systems grow more and more complex, they become increasingly difficult to design and develop as well as maintain. The intrusion preventive measures will be inadequate as there will be more programming errors or bugs. According to Evans' Law, security risk is the product of the vulnerabilities and the number of malicious users. This works out to be about a quadrillion times worse today than in a few decades ago in terms of security problems. Hence there is the need for intrusion detection as it provides a second line of defense.

As a wireless computing device is usually of limited electrical power and intensive processing drains any stored electrical power, we have to avoid the situation whereby the device has to do more routing than other devices on the network. Hence an optimal routing algorithm has to be employed. This is made even critical as power consumption increases tremendously when the wireless transceiver is active. We do not want a device to be exhaust of electrical power faster than it is necessary, especially when it is part of an optimum or even critical routing path where such a device is not operating results in the network needing route repairs or worse, segregated. Therefore, a good intrusion detection system should not only conduct intensive processing for detecting intrusion, it will be better if the system rides on an intelligent routing protocol.

4.3.1. Anomaly Detection vs. Misuse Detection

In order to detect an intrusion attack, one needs to make use of a model of intrusion. That is, we need to know what an Intrusion Detection System (IDS) should look out for. There are basically two types of models employed in current IDS: anomaly detection (fig 4.2) and misuse detection (fig 4.3).

The first model hypothesizes its detection upon the profile of a user's (or a group of users') normal behavior. It analyzes the user's current session and compares them to the profile representing the user's normal behavior statistically. It then reports any significant deviations to a designated system administrator. As it catches sessions which are not normal, this model is hence referred to as an "anomaly' detection model.

Anomaly detection bases its idea on statistical behavior modeling and anomaly detectors look for behavior that deviates from normal system use. A typical anomaly detection system takes in audit data for analysis.

The audit data is transformed to a format statistically comparable to the profile of a user. The user's profile is generated dynamically by the system (usually using a baseline rule laid by the system administrator) initially and subsequently updated based on the user's usage. Thresholds are normally always associated to all the profiles. If any comparison between the audit data and the user's profile resulted in deviation crossing a threshold set, an alarm of intrusion is declared. This type of detection systems is well suited to detect unknown or previously not encountered attacks.

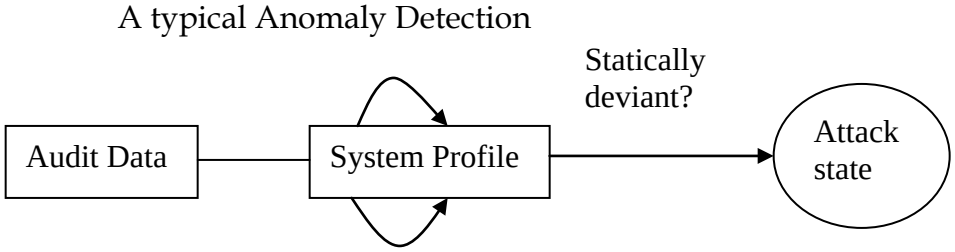

Fig 4.1 Example of an anomaly detection system

The second type of model bases its detection upon a comparison of parameters of the user's session and the user's commands to a rule base of techniques used by attackers to penetrate a system. Known attack methods are what this model looks for in a user's behavior. Since this model looks for patterns known to cause security problems, it is called a "misuse" detection model.

A typical Misuse Detection System

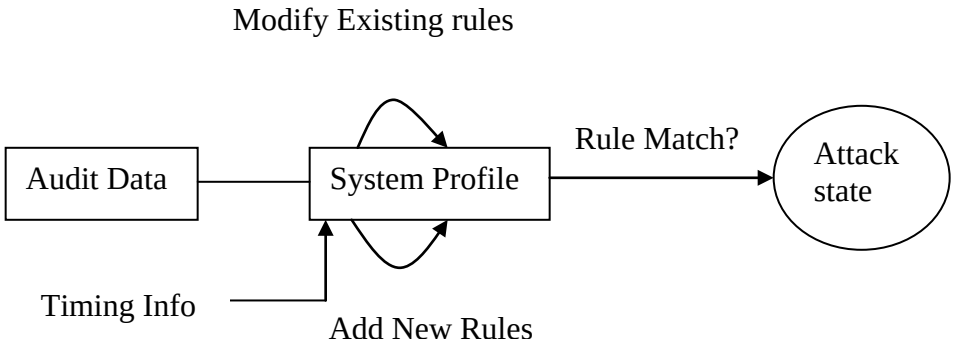

Fig 4.2 Example of a misuse detection system

Misuse detection bases its idea on precedence and rules and misuse detectors look for behavior that matches a known attack scenario. A typical misuse detection system takes in audit data for analysis and compares the data to large databases of attack signatures. The attack signatures are normally specified as rules with respect to timing information and are also referred to as known attack patterns. If any comparison between the audit data and the known attack patterns described resulted in a match, an alarm of intrusion is sounded. This type of detection systems is useful in networks with highly dynamic behavioral patterns but like a virus detection system, it is only as good as the database of attack signatures that it uses to compare with.

4.3.2. Host-Based vs. Network-Based Intrusion Detection

Most intrusion detection systems (IDS) take either a network-based or a host-based approach to recognizing and deflecting attacks. In either case, these products look for specific patterns that usually indicate malicious or suspicious intent. An IDS is network-based when it looks for these patterns in network traffic. It is host-based when it looks for patterns in log files.

Network-based systems (NIDS) listen on the network, and capture and examine individual packets flowing through a network. That is, they use raw network packets as the data source. They typically utilize a network adapter running in promiscuous mode to monitor and analyze all traffic in real-time as it travels across the network. They are able to look at the payload within a packet, to see which particular host application is being accessed, and to raise alerts when attacker tries to exploit a bug in such code. NIDS are typically host-independent but can also be a software package installed on dedicated workstation. A side effect of NIDS is that its active scanning can slow down the network considerably. Hence usage of it on an ad-hoc network needs to be evaluated.

Host-based systems (HIDS) are concerned with what is happening on each individual host. They are able to detect actions such as repeated failed access attempts or changes to critical system files, and normally operate by accessing log files or monitoring real-time system usage. In order for a HIDS to function, clients have to be installed on every host in the network. These clients reside on the hosts as processes and perform analysis on the audit data gathered locally, at the expense of the already limited resources of the hosts. Hence care has to be taken to ensure that the HIDS client running on a host in an ad-hoc network does not drain resources more than necessary.

4.3.3. Online Detection vs. Offline Detection

The classification of intrusion detection systems can be further segregated according to the timeliness of the audit data being gathered and processed. Audit data can be gathered and processed while the hosts is either online (connected to the network) or offline (disconnected from the network). When a system is performing intrusion detection in online mode, the audit data is processed real-time continuously. A host-based system will gather information about a host as

long as the host is connected to the network. A network-based system will monitor the network traffic of the hosts throughout the time they are connected. Any intrusion detected is immediately notified to other hosts.

When a system is performing intrusion detection in offline mode, the audit data is not processed real-time but periodically. A host-based system will gather information about a host even if it is not connected to the network. Even if the host is connected, detection is done as scheduled by the system. A network-based system will monitor the network traffic of the hosts periodically as can be in the case of polling. Any intrusion detected is still immediately notified to other hosts but a delay is expected. A typical technique of an offline intrusion detection system is data mining.

5. THE SIMULATION SOFTWARE

Network Simulator Version 2 (NS-2)

Network Simulator 2 (NS2) is a discrete event simulator targeted at networking research. NS2 provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks.

NS2 began as a variant of the REAL network simulator in 1989 and has evolved substantially over the past few years. NS2 has always included substantial contributions from researchers, like wireless code from the CMU Monarch projects and Sun Microsystems. Even though there is a considerable confidence in NS-2, it is not a polished and finished product yet and bugs are being discovered and corrected continuously.

NS2 is an object oriented simulator, written in C++, with an OTcl interpreter as a frontend. The simulator supports a class hierarchy in C++ (also called the compiled hierarchy), and a similar class hierarchy within the OTcl interpreter (also called the interpreted hierarchy). The C++ part which is fast to run but slower to change is used for detailed protocols implementation. The OTCL part, on the other hand, which runs much slower but can be changed very quickly, is used for simulation configuration. The two hierarchies are closely related to each other; from the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy. The root of this hierarchy is the class TclObject.

Users create new simulator objects through the interpreter; these objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. The interpreted class hierarchy is automatically

established through methods defined in the class TclClass. User instantiated objects are mirrored through methods defined in the class TclObject. There are other hierarchies in the C++ code and OTcl scripts; these other hierarchies are not mirrored in the manner of TclObject.

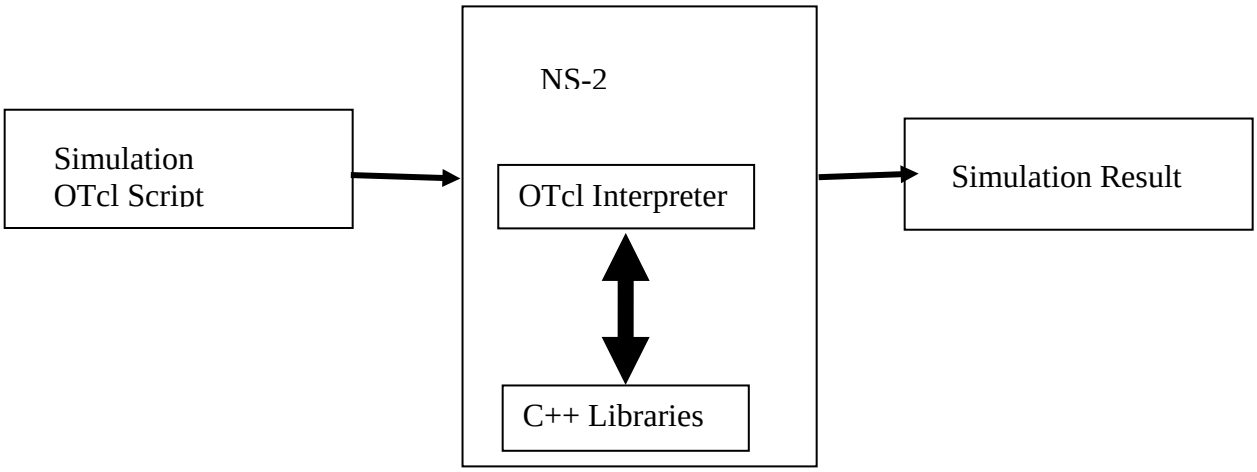

Fig 5.1 simplified user's view of NS-2

One of the advantages of split-language programming approach is that, it allows for fast generation of large scenarios. To simply use the simulator, it is sufficient to know OTCL. On the other hand, one disadvantage is that modifying and extending the simulator requires programming and debugging in both languages simultaneously.

Wireless networking in NS-2

Mobile Node

The wireless networking in NS-2 essentially consists of the Mobile Node at the core. Mobile Node is the basic ns Node object with added functionalities like

ability to move within a given topology, and to transmit and receive on a channel that allows it to be used to create mobile, wireless simulation environments.

A major difference between a node of wired network and mobile node is that a Mobile Node is not connected by means of Links to other nodes or mobile nodes. Moreover, routing in mobile networks especially in Ad-hoc networks is distributed and there is no centralized entity (as router in wired network). Therefore a mobile node acts as a router and as a node at the same time.

Node movement

The mobile node is designed to move in a three dimensional topology. However the third dimension (Z) is not used. That is, the mobile node is assumed to move always on a flat terrain with Z always equal to 0. Thus, the mobile node has X, Y, Z(=0) co-ordinates that is continually adjusted as the node moves.

There are two mechanisms to induce the movement in mobile nodes. In the first method, starting position of the node and its future destinations may be set explicitly. These directives are normally included in a separate movement scenario file. The starting position and future destinations for a mobile node may be set by using the following APIs:

```
$node set X_ \<x1\>
$node set Y_ \<y1\>
$node set Z_ \<z1\>
$ns at $time $node setdest \<x2\> \<y2\> \<speed\>
```

At $time sec, the node would start moving from its initial position of (x1,y1) towards a destination (x2,y2) at the defined speed. In this method the node-movement-updates are triggered whenever the position of the node at a given time is required to be known. This may be triggered by a query from a neighboring

node seeking to know the distance between them, or the setdest directive described above that changes the direction and speed of the node.
The second method employs random movement of the node. The primitive to be used is:

```
$mobilenode start
```

which starts the mobile node with a random position and have routined updates to change the direction and speed of the node. The destination and speed values are generated in a random fashion. The mobile node movement is implemented in C++.

Irrespective of the methods used to generate the node movement, the topography for mobile nodes needs to be defined. It should be defined before creating the mobile nodes. Normally, flat topology is created by specifying the length and width of the topography using the following primitive:

```
set topo        [new Topography]
$topo load_flatgrid $opt(x) $opt(y)
```

where opt(x) and opt(y) are the boundaries used in simulation.
The movement of the node is determined by setting an area and mobility scenarios.

Network Components in a mobile node

The network stack for a mobile node consists of a link layer (LL), an ARP module connected to LL, an interface priority queue (IFq), a MAC layer (MAC), a network interface (netIF), all connected to the channel. These network components are created and plumbed together in OTcl. Each component is discussed briefly as follows:

Link Layer

The LL object is responsible for simulating the data link protocols like in wired networks. The only difference being the link layer for mobile node has an ARP module connected to it which resolves all IP to hardware (Mac) address conversions. Normally for all outgoing (into the channel) packets, the packets are handed down to the LL by the Routing Agent. The LL hands down packets to the interface queue. For all incoming packets (out of the channel), the mac layer hands up packets to the LL which is then handed off at the node_entry_ point. [19]

ARP

The Address Resolution Protocol module receives queries from Link layer. If ARP has the hardware address for destination, it writes it into the Mac header of the packet. Otherwise it broadcasts an ARP query, and caches the packet temporarily. For each unknown destination hardware address, there is a buffer for a single packet. In case, additional packets to the same destination are sent to ARP, the earlier buffered packet is dropped. Once the hardware address of a packet's next hop is known, the packet is inserted into the interface queue. [19]

Interface Queue

The PriQueue../ns-2/priqueue.h [19] is implemented as a priority queue which gives priority to routing protocol packets, inserting them at the head of the queue. It supports running a filter over all packets in the queue and removes those with a specified destination address. [19]

Mac Layer

The IEEE 802.11 distributed coordination function (DCF) Mac protocol has been implemented by CMU. It uses a RTS/CTS/DATA/ACK pattern for all unicast

packets and simply sends out DATA for all broadcast packets. The implementation uses both physical and virtual carrier sense. [19]

Network Interfaces

The Network Interface layer serves as a hardware interface which is used by mobile node to access the channel. The wireless shared media interface is implemented as Phy/WirelessPhy../ns-2/wireless-phy.h [19]. This interface subject to collisions and the radio propagation model receives packets transmitted by other node interfaces to the channel. The interface, stamps each transmitted packet with the meta-data related to the transmitting interface like the transmission power, wavelength etc. This meta-data in packet header is used by the propagation model in receiving network interface to determine if the packet has minimum power to be received and/or captured and/or detected (carrier sense) by the receiving node. The model approximates the DSSS radio interface (Lucent WaveLan direct-sequence spread-spectrum). [19]

Radio Propagation Model

It uses Friss-space attenuation () at near distances and an approximation to Two ray Ground () at far distances. The approximation assumes specular reflection off a flat ground plane. See tworayground.{cc,h}in [19].

Antenna

An omni-directional antenna having unity gain is used by mobilenodes. See antenna.{cc,h} for implementation details in[19].

Routing Protocols or Agents

There are four different types of Ad-hoc routing agents defined by NS-2 currently. These are:

- AODV(Ad-hoc On-Demand Distance Vector)
- DSDV (Destination Sequenced Distance Vector)
- DSR (Dynamic Source Routing)
- TORA (Temporal Ordered Routing Algorithm)

NS-2 Trace Support

The NS-2 trace files are used for post processing the simulation that is carried on. The following are the supported formats of trace files:

- Trace files for wired networks that is wired
- Satellite
- Wireless (old and new trace)
- Wired-cum-wireless

The trace support for wireless simulations currently use cmu-trace objects. In the simulation future this shall be extended to merge with trace and monitoring support available in ns, which would also include nam support for wireless modules. The cmu-trace objects are of three types - CMUTrace/Drop, CMUTrace/Recv and CMUTrace/Send. These are used for tracing packets that are dropped, received and sent by agents, routers, mac layers or interface queues in ns.

An example of a trace for a tcp packet is as follows:

r 160.093884945 _6_ RTR --- 5 tcp 1492 [a2 4 6 800] ------- [65536:0 16777984:0 31 16777984] [1 0] 2 0

Here we see a TCP data packet being received by a node with id of 6. UID of this pkt is 5 with a cmn hdr size of 1492. The mac details shows an IP pkt (ETHERTYPE_IP is defined as 0x0800, ETHERTYPE_ARP is 0x0806), mac-id of this receiving node is 4. That of the sending node is 6 and expected time to send this data pkt over the wireless channel is a2 (hex2dec conversion: 160+2 sec).

Additionally, IP traces information about IP src and destination addresses. The src translates (using a 3 level hier-address of 8/8/8) to a address string of 0.1.0 with port of 0. The dest address is 1.0.3 with port address of 0. The TTL value is 31 and the destination was a hop away from the src. Additionally TCP format prints information about tcp seqno of 1, ackno of 0.

The Cmu trace format can be divided into the following fields:

Event type

In the traces above, the first field (as in the older trace format) describes the type of event taking place at the node and can be one of the four types:

S	Send
R	Receive
D	Drop
F	Forward

General tag

The second field starting with "-t" may stand for time or global setting

-t	Time
-t	Global setting

Node property tags

This field denotes the node properties like node-id, the level at which tracing is being done like agent, router or MAC. The tags start with a leading "-N" and are listed as below:

-Ni	Node id
-Nx	Node's x-coordinate
-Ny	Node's y-coordinate
-Nz	Node's z-coordinate
-Ne	Node's energy level
-Nl	Trace level, such as AGT,RTR,MAC
-Nw	Reason for event

Packet information at IP level

The tags for this field start with a leading "-I" and are listed along with their explanations as following:

-Is	Source address.source port number
-Id	Dest address.dest port number
-It	Packet type
-Il	Packet size
-If	Flow id
-Ii	Unique id
-Iv	TTL value

Next hop info

This field provides next hop info and the tag starts with a leading "-H".

-Hs	Id for this node
-Hd	Id for next hop toward the destination

Packet info at MAC level

This field gives MAC layer information and starts with a leading "-M" as shown below:

-Ma	Duration
-Md	Destination Ethernet address
-Ms	Source Ethernet address
-Mt	Ethernet type

Packet info at "Application level"

The packet information at application level consists of the type of application like ARP, TCP, the type of ad-hoc routing protocol like DSDV, DSR, AODV etc being traced. This field consists of a leading "-P" and list of tags for different applications are listed as below:

-P arp	Address Resolution Protocol
-Po	ARP Request/Reply
-Pm	Source MAC address
-Ps	Source address
-Pa	Destination MAC address
-Pd	Destination address
-P dsr	DSR routing protocol (an example DSR and the following details for DSR)
-Pn	How many nodes traversed
-Pq	Routing request flag
-Pi	Route request sequence number
-Pp	Route reply flag
-Pl	Reply length
-Pe	Source of source routing to destination
-Pw	Error report flag
-Pm	Number of errors

-Pc	Report to whom
-Pb	Link error form linka->linkb
-P cbr	Constant bit rate information and the following are details of CBR
-Pi	Sequence number
-Pf	How many times his packet was forwarded
-Po	Optimal number of forwards
-P tcp	Information about TCP flow and the following are details of TCP
-Ps	Sequence number
-Pa	Acknowledgment number
-Po	Optimal number of forwards
-Pf	How many times this packet was forwarded

The new trace format is also under improvement to support other detailed information.

NS-2 Network Animator (NAM)

Network Animator (NAM) is an animation tool for viewing network simulation traces and real world packet traces [19]. It supports topology layout, packet level simulation and various data inspection tools.

Before starting to use NAM, a trace file has to be created. This trace file is usually generated by NS-2 as discussed above. It contains topology information for example node and links as well as packet losses. During simulation, the user can produce topology configuration, layout information and packets traces using tracing events in NS-2.

Once the trace file is generated, NAM can be used to animate it. Upon starting, NAM will read the trace file, create the topology, pop up a window, do layout if necessary and pause at time 0. Through its user interface, NAM provides control over many aspects of animation. The following figure shows a NAM window.

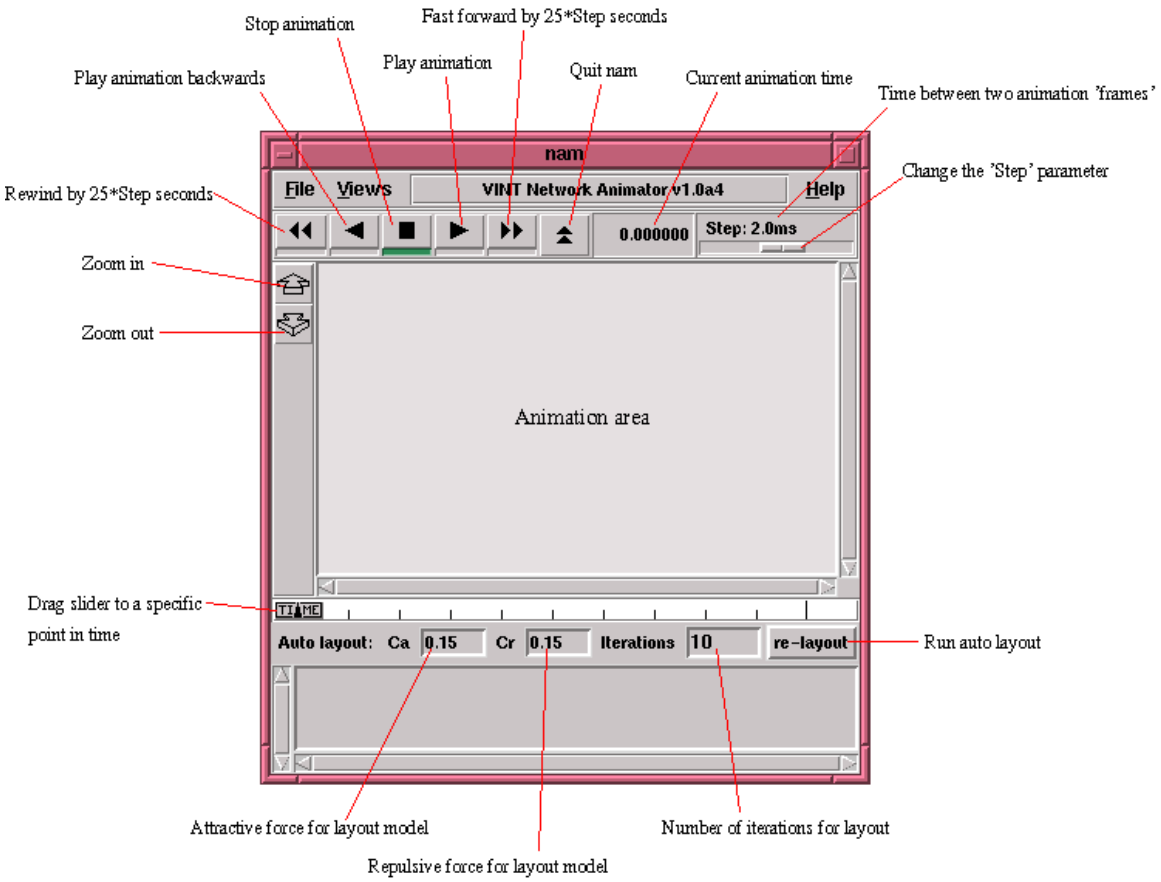

Fig 5.2 NS-2 Network Animator window

6. IMPLEMENTATION

6.1 Attacks implemented on AODV protocol

In general, AODV is efficient and scalable in terms of network performance, but it allows attackers to easily advertise falsified route information to redirect routes and to launch various kinds of attacks. In each AODV routing packet, some critical fields such as hop count, sequence numbers of source and destination, IP headers as well as IP addresses of AODV source and destination, and RREQ ID, are essential to the correct protocol execution. Any misuse of these fields can cause AODV to malfunction. Table 1 denotes several vulnerable fields in AODV routing messages and how possibly they are tampered.

Field	**Modification**
RREQ ID	**Increase to create a new RREQ request**
Hop Count	**If sequence number is the same, decrease it to update other nodes' forwarding table, or increase it to invalidate the update**
IP headers as well as AODV source and destination IP address	**Replace it with another or invalid IP address**
Sequence number of source and destination	**Increase it to update other nodes' forwarding route tables, or decrease it to suppress its update**

Table 6.1: Vulnerable Fields in AODV Packets

An attacker could launch a single (packet) attack (atomic attack) consisting of several carefully modified fields, or an aggregate attack(compound attack) consisting of multiple attack messages, which cause more damages and last longer than a single attack does.

An inside attacker has enough incentives to launch homogeneous compound attacks by repeating an atomic attack. First, the routing tables of mobile nodes may change from time to time. Thus, an attacker may have to repeat an atomic misuse periodically to sustain already achieved goals, such as resource consumption, route disruption, and node isolation. Secondly, most ad-hoc routing protocols such as AODV have built-in local repair mechanism, which is intended to allow mobile nodes to recover from failures. This implies that the atomic attacks targeted at disrupting services can only generate temporary impact. This recovery mechanism also forces an attacker to repeat atomic misuses to sustain the disruption of services. In this thesis resources consumption type of compound attack is implemented on RREQ message of AODV. We will briefly describe the attack below.

6.1.1 Resource Consumption Attack

The first type of attack implemented is resource consumption attack. Such kind of attack applied to consume all the available network bandwidth with irrelevant traffic and to consume energy and processing power from the nodes. The attack is applied on the route request message packets. In the attack, the attacking node floods the network with route request. The attacker generates a fake RREQ and broadcast it as a forward. It fakes the IP header, the source and destination IP address and also the RREQ ID. It uses non existing IP header as the request IPheader to hide itself and send the attack without being detected. It changes the RREQ ID every time it generates the request. So it will be taken as new request and

will not be discarded. In the simulation, the attacker repeatedly sends out faked RREQ messages in a fixed interval (20pkt/s) to disable the routes between two nodes. In AODV specification, a mobile node should not send more than 10 RREQ messages per second. However, the attacker can broadcast faked RREQ messages in a higher rate.

6.1.1 Node Isolation Attack

Node isolation refers to preventing a given node from communicating with any other node in the network. It differs from Route Disruption in that Route Disruption is targeting at a route with two given endpoints, while node isolation is aiming at all possible routes. The second implemented attack is node isolation. The attack is applied on the RREQ message. When a node broadcasts a route request, the attacking node receives and modifies the message before forwarding. It modifies the source IP in the IP header to its number, the destination IP address to non existing IP and increase the sequence number. Since it increment the sequence number the other nodes will take its request as a fresh request. Then they update the next hop to the source to the attackers IP address. When the source node gets the modified request it will drop it thinking that it gets its own request. Any packet send to the victim node will pass through the attacker. So the attacker will drop those packets.

6.2 IDS implemented on AODV protocol

Intrusion detection systems (IDSs) are mainly used to detect and call attention to odd and suspicious behaviour. Current approaches to intrusion detection can be broadly classified into two trends, *anomaly-detection*, also known as *behaviour-based* intrusion detection, and *misuse-detection*, also called *knowledge-based* intrusion detection. Behaviour-based intrusion detection systems monitor and build a reference profile of normal behaviour for the information system by using

statistical methods and try to detect activity that deviates from the normal behaviour profile. Knowledge-based IDSs accumulate knowledge about attacks, examine traffic and try to identify patterns indicating that a suspicious activity is occurring. This approach can be applied against known attack patterns only, and needs to update the knowledge base frequently. In this specific paper misuse type of detection mechanism is used.

6.2.1 Resource consumption attack Detection

As it was stated above this attack generates RREQ messages more than the threshold value. It counts the number of RREQ messages sent from a single node within a second. If the number of count is more than 10 which is the threshold value the detector will drop the packet and generate error message. Since it automatically drops the RREQ message from the attacker the resources that would have been used for processing the packet and either rebroadcasting or replying to the destination via the non existing node will be saved. The detail of the system performance will be discussed in the simulation and results chapter. For the naming convenience the security enhanced AODV is named as AODVIDS.

6.2.1 Node Isolation attack Detection

Node Isolation attack is a bit complicated attack as compared to resource consumption attack. The victim node is not in a position to recognize the attacker node unless it makes the sequence number check before dropping the modified request. In the AODV protocol if a node gets a request with the source id of its, thinking that it gets his, will drop it. So, the detection system checks whether the this request originated from it or not. To do this it takes the destination ip address and check in its table whether there is a request sent destined to that IP. If there is none it means the node sending these request is the attacker. The other

methodology is by ckecking the sequence number of the request. If the sequence number is unreasonably higher than the one given by the source, it means the node sending this request is attacker. In both cases the system will drop the request and generate error. Unfortunatly this detection system doesn't output the expected result so the result is not included in the simulations and results part of the thesis.

7. SIMULATIONS AND RESULTS

Simulations create scenarios that resemble the real event. In NS2 to simulate any network and analyze the output first one needs to create a scenario that describes the movement pattern of the node and a communication file which describes the traffic in the network.

These files are then used for the simulation and as a result a trace file is generated as an output. Prior to simulation, the parameters that are going to be traced during the simulation must be selected. The trace file can be scanned and analyzed for various parameters that we want to measure. The trace file can be used to visualize the simulation run with the Network Animator (NAM).

7.1 Performance Metrics

The performance of any routing protocol is measured by certain quantitative metrics. The RFC 2501[10] defines the following measures.

- End-to-end data throughput and delay.
  - analyze the efficiency of data routing
- Route Acquisition Time
  - measures the time required to establish routes
- Percentage Out-of-Order Delivery
  - measures the connectionless routing performance
- Efficiency
  - refers to internal effectiveness of a routing policy

The *networking context*, in which a protocol's performance is measured, should be considered. Essential parameters that should be varied include:

Network size: measured in terms of the number of nodes

Network connectivity: the average degree of a node (i.e. the average number of neighbors of a node)

Topological rate of change: the speed with which a network topology is changing

Link capacity*:* effective link speed measured in bits/second, after accounting for losses due to multiple access, coding, framing, etc.

Fraction of unidirectional links: how effectively does a protocol perform as a function of the presence of unidirectional links

***Traffic patterns*:** how effective is a protocol in adapting to non uniform or bursty traffic patterns

Mobility: when, and under what circumstances, is temporal and spatial topological correlation relevant to the performance of a routing protocol

7.2 Simulation

Security of a network is measured with the performance metrics of efficiency. In this paper it's tried to show how efficient is the enhanced AODV or the AODVIDS routing when compared with the old one. And the simulation is done based on the number of attack RREQ applied as measured by the duration of the attack. And the other simulation is based on Network connectivity.

7.2.1 Number of attack Simulation

In this simulation the number of attack will be varied. The number of attack is measured per the amount of time it is applied. Because the longer time we applied

the attack the more effect it has on the network. One thing not to forget is not only the amount of attacking packets but also the duration of the attack matters in such attacks.

The scenario that is used for this simulation is shown in the following table.

Number of Nodes	20
Simulation time	10,20,30,……100sec
Number of connections	8
Maximum Speed	0.5 m/s
Number of Inside Attacker	1
Network area	1000×600
Traffic type	CBR
Agent	UDP
Packet size	512bytes
Initial Energy	100Watt

Table 7.1 Scenario of the number of attack simulation

7.2.2 Result and Analysis of Number of attack Simulation

The graph below shows the number of packets dropped by both the AODV and AODVIDS. As it can be seen below the number of packets dropped is a lot in the AODV routing protocol as compared to AODVIDS. Packets are dropped after processing the request and trying to reply to the non existing node in AODV. On the other hand AODVIDS drops the packets at the request level. Because of that the number of dropped packets is much lesser. This number doesn't include the number of the attack packets dropped by AODVIDS. By dropping the attacking packets at the request level AODVIDS reduce the amount of resource consumed.

In the second graph it is tried to show the effect of the applied intrusion detection on the energy consumption reduction. Thus reduction of resource consumption.

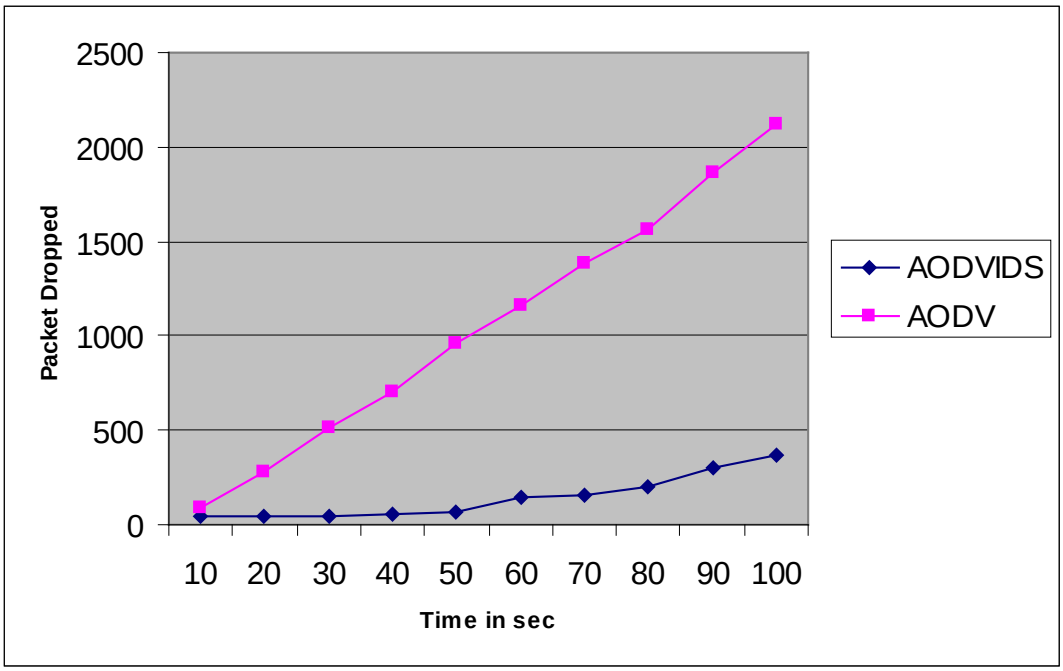

Fig 7.1 Number of attacks simulation result

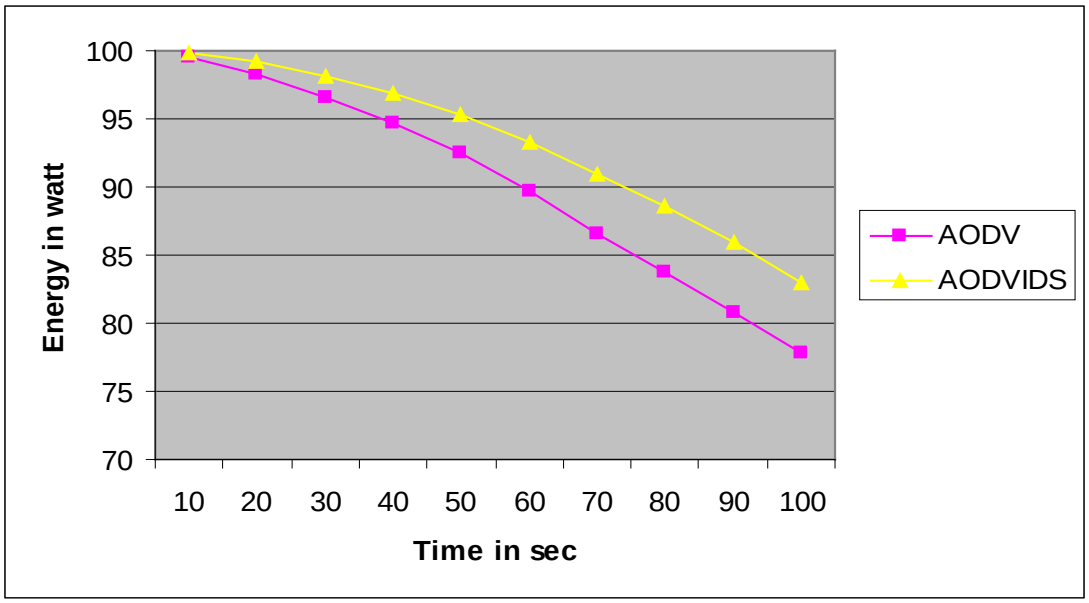

Fig 7.2 Energy consumption comparison graph

7.2.3 Network Connectivity Simulation

The effects of a malicious node differ depending on the position they have. That is depending on how many neighbours they have their effect will be high or low. In this simulation the number of neighbouring nodes will be varied.

The scenario that is used for this simulation is shown in the following table.

Number of Nodes	20
Simulation time	10,20,30,……100sec
Number of connections	2,4,6,7,8,9
Maximum Speed	0.5 - 3.5 m/s
Number of Inside Attacker	1
Network area	1000×600
Traffic type	CBR
Agent	UDP
Packet size	512bytes
Initial Energy	100Watt

Table 7.2 Scenario of the network connectivity simulation

7.2.4 Result and Analysis of network connectivity simulation

The following graph shows the result of the network connectivity simulation comparing the AODV and the AODVIDS. The number of packet dropped in the AODV routing protocol is relatively high as compared to AODVIDS. The effect of the attacker position was shown in the graph though doesn't seem to fully satisfy what it is expected, which is to increase in a certain fashion. This is because of the movement of the nodes. The number of neighbour varies even in one scenario

because of the nodes movement. But still it performs well as compared to the AODV when it comes to resource consumption that increases the efficiency.

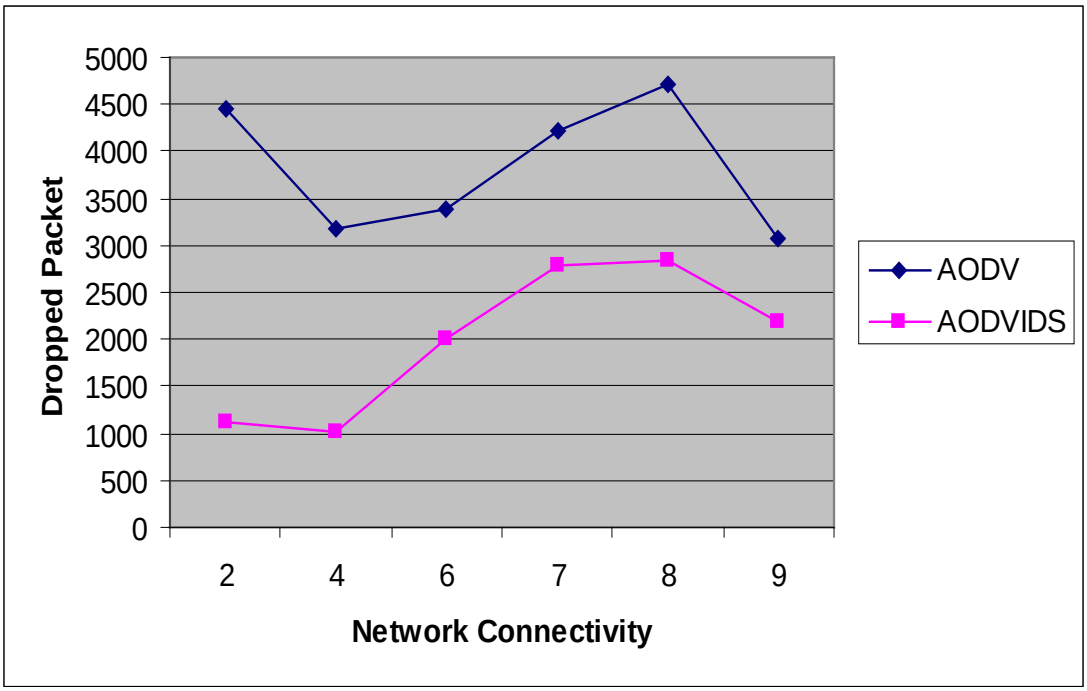

Fig 7.3 Network Connectivity simulation result

8. CONCLUSION AND RECOMMENDATION

8.1 Conclusion

In this thesis the Ad hoc wireless network routing protocols AODV has been considered for its better security features. By using the NS-2 network simulation environment, the misuse intrusion detection system is implemented on the AODV. Using this method the detection performance evaluation and comparison for the two types of attacks applied on AODV became more meaningful.

The first simulation result as shown in figure 7.1, is based on the number of attack simulation, and shows that the AODVIDS performs well in terms of energy efficiency by reducing the resource consumption applied on the system. AODVIDS was able to drop 88% of the attacking packet. In the case of AODV it doesn't seem to drop the attacking request packets without processing. It rather accepts the request and tries to process the packet for reply or forwarding. The packet is dropped very late and only after the AODV realizes that there is no destination node or route. This increases the power consumption of the MANET devices and results in wastage of bandwidth of the network. The energy consumption graph, from the simulations clearly shows the superiority of AODVIDS over AODV with respect to power consumption. It is also necessary to mention that the performance gain by AODVIDS increases as the number of attacks increases.

In the second simulation, figure 7.3, an attempt is made to confirm that the effect of the attack is based on the network connection variability. Most importantly this shows the QoS performance of AODVIDS versus AODV. It can be seen from the graph that AODV shows a higher number of packet drops while the AODVIDS seem to perform well.

The third simulation result, figure 7.4, shows increase in number of data packet delivered to the victim irrespective of the nodes mobility rate in the case of AODVIDS. It also shows AODVIDS performs better in this aspect then AODV. This implies a significant increase in network performance by the introduction of the IDS.

The first two simulation results are for the resource consumption attack and the third one is for node isolation attack. From their result it can be said that the implemented intrusion detection system increases the network performance.

8.2 Recommendation

The final output from these thesis paper shows AODVIDS performs well in securing the system for certain types of attacks. The attacks can be made to be more complex by involving different types of atomic attacks either simultaneously or one after the other. And also different types of attacks are coming everyday. So adding a security feature to routing protocols seems to be endless task.

This thesis finally recommends further enhancements to the AODV or other ad-hoc routing protocols in areas like considering

- Security for various types of intrusions
- Quality of Service.

Appendix I Misuse implementation Code

```
/* active-misuse.cc
   implement the active misuses of routing messages in AODV routing
protocol.

*/


#include <aodv/myaodv.h>
#include <aodv/aodv_packet.h>
#include <random.h>
#include <cmu-trace.h>

#define max(a,b)        a > b ? a : b
#define CURRENT_TIME    Scheduler::instance().clock()

#ifdef DEBUG
static int extra_route_reply = 0;
static int limit_route_request = 0;
static int route_request = 0;
#endif


/*
  TCL Hooks
*/


int hdr_aodv::offset_;
static class MYAODVHeaderClass : public PacketHeaderClass {
public:
        MYAODVHeaderClass() : PacketHeaderClass("PacketHeader/AODV",
                                                sizeof(hdr_all_aodv)) {
        bind_offset(&hdr_aodv::offset_);
      }
} class_rtProtoAODV_hdr;

static class MYAODVclass : public TclClass {
public:
        MYAODVclass() : TclClass("Agent/MYAODV") {}
        TclObject* create(int argc, const char*const* argv) {
          assert(argc == 5);
          return (new MYAODV((nsaddr_t) atoi(argv[4])));
        }
} class_rtProtoAODV;


int MYAODV::command(int argc, const char*const* argv) {
  if(argc == 2) {
  Tcl& tcl = Tcl::instance();

    if(strncasecmp(argv[1], "id", 2) == 0) {
      tcl.resultf("%d", index);
```

```
      return TCL_OK;
    }


    if(strncasecmp(argv[1], "start", 2) == 0) {
      btimer.handle((Event*) 0);


#ifndef AODV_LINK_LAYER_DETECTION
      htimer.handle((Event*) 0);
      ntimer.handle((Event*) 0);
#endif // LINK LAYER DETECTION

      rtimer.handle((Event*) 0);
      return TCL_OK;
     }
  }
  else if(argc == 3) {


    if(strcmp(argv[1], "index") == 0) {
      index = atoi(argv[2]);
      return TCL_OK;
    }

    else if(strcmp(argv[1], "log-target") == 0 || strcmp(argv[1],
"tracetarget") == 0) {
      logtarget = (Trace*) TclObject::lookup(argv[2]);
      if(logtarget == 0)
      return TCL_ERROR;
      return TCL_OK;
    }
    else if(strcmp(argv[1], "drop-target") == 0) {
      int stat = rqueue.command(argc,argv);
      if (stat != TCL_OK) return stat;
      return Agent::command(argc, argv);
    }
    else if(strcmp(argv[1], "if-queue") == 0) {
      ifqueue = (PriQueue*) TclObject::lookup(argv[2]);

      if(ifqueue == 0)
      return TCL_ERROR;
      return TCL_OK;
    }
  }



    }

  else if (argc ==6)
    {
      // send RREQ to disrupt routes  --sk 02/27/03
      if(strncasecmp(argv[1], "rreqs_rd", 8) == 0) {
      int src = atoi(argv[2]);
      int dst = atoi(argv[3]);
      double simtime = atof(argv[4]);
```

```
      double interval = atof(argv[5]);
      rreqRouteDisrupt(src, dst, simtime, interval);
      return TCL_OK;
      }

      }
      }

  return Agent::command(argc, argv);
}

/*
   Constructor
*/

MYAODV::MYAODV(nsaddr_t id) : Agent(PT_AODV),
                    btimer(this), htimer(this), ntimer(this),
                    rtimer(this), lrtimer(this), rqueue() {


  index = id;
  seqno = 2;
  bid = 1;

  LIST_INIT(&nbhead);
  LIST_INIT(&bihead);

  logtarget = 0;
  ifqueue = 0;

  //bind an viariable to be used by attacker. -sk
  bind("victimNode", &victim_);
}

/*
  Timers
*/

void MyBroadcastTimer::handle(Event*) {
  agent->id_purge();
  Scheduler::instance().schedule(this, &intr, BCAST_ID_SAVE);
}

void MyHelloTimer::handle(Event*) {
   agent->sendHello();
   double interval = MinHelloInterval +
                 ((MaxHelloInterval - MinHelloInterval) *
Random::uniform());
   assert(interval >= 0);
   Scheduler::instance().schedule(this, &intr, interval);
}

void MyNeighborTimer::handle(Event*) {
  agent->nb_purge();
  Scheduler::instance().schedule(this, &intr, HELLO_INTERVAL);
}
```

```
void MyRouteCacheTimer::handle(Event*) {
  agent->rt_purge();
#define FREQUENCY 0.5 // sec
  Scheduler::instance().schedule(this, &intr, FREQUENCY);
}

void MyLocalRepairTimer::handle(Event* p)  {  // SRD: 5/4/99
aodv_rt_entry *rt;
struct hdr_ip *ih = HDR_IP( (Packet *)p);

   /* you get here after the timeout in a local repair attempt */
   /* fprintf(stderr, "%s\n", __FUNCTION__); */


    rt = agent->rtable.rt_lookup(ih->daddr());

    if (rt && rt->rt_flags != RTF_UP) {
    // route is yet to be repaired
    // I will be conservative and bring down the route
    // and send route errors upstream.
    /* The following assert fails, not sure why */
    /* assert (rt->rt_flags == RTF_IN_REPAIR); */

      //rt->rt_seqno++;
      agent->rt_down(rt);
      // send RERR
#ifdef DEBUG
      fprintf(stderr,"Node %d: Dst - %d, failed local repair\n",index,
rt->rt_dst);
#endif
    }
    Packet::free((Packet *)p);
}


/*
   Broadcast ID Management  Functions
*/


void MYAODV::id_insert(nsaddr_t id, u_int32_t bid) {
BroadcastID *b = new BroadcastID(id, bid);

 assert(b);
 b->expire = CURRENT_TIME + BCAST_ID_SAVE;
 LIST_INSERT_HEAD(&bihead, b, link);
}

/* SRD */
bool MYAODV::id_lookup(nsaddr_t id, u_int32_t bid) {
BroadcastID *b = bihead.lh_first;

 // Search the list for a match of source and bid
 for( ; b; b = b->link.le_next) {
   if ((b->src == id) && (b->id == bid))
     return true;
 }
```

```
 return false;
}

void MYAODV::id_purge() {
BroadcastID *b = bihead.lh_first;
BroadcastID *bn;
double now = CURRENT_TIME;

 for(; b; b = bn) {
   bn = b->link.le_next;
   if(b->expire <= now) {
     LIST_REMOVE(b,link);
     delete b;
   }
 }
}

/*
  Helper Functions
*/

Double MYAODV::PerHopTime(aodv_rt_entry *rt) {
int num_non_zero = 0, i;
double total_latency = 0.0;

 if (!rt)
   return ((double) NODE_TRAVERSAL_TIME );

 for (i=0; i < MAX_HISTORY; i++) {
   if (rt->rt_disc_latency[i] > 0.0) {
      num_non_zero++;
      total_latency += rt->rt_disc_latency[i];
   }
 }
 if (num_non_zero > 0)
   return(total_latency / (double) num_non_zero);
 else
   return((double) NODE_TRAVERSAL_TIME);

}

/*
  Link Failure Management Functions
*/

static void
aodv_rt_failed_callback(Packet *p, void *arg) {
  ((MYAODV*) arg)->rt_ll_failed(p);
}

/*
 * This routine is invoked when the link-layer reports a route failed.
 */
Void MYAODV::rt_ll_failed(Packet *p) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
aodv_rt_entry *rt;
```

```
nsaddr_t broken_nbr = ch->next_hop_;

#ifndef AODV_LINK_LAYER_DETECTION
 drop(p, DROP_RTR_MAC_CALLBACK);
#else

 /*
  * Non-data packets and Broadcast Packets can be dropped.
  */
  if(! DATA_PACKET(ch->ptype()) ||
     (u_int32_t) ih->daddr() == IP_BROADCAST) {
    drop(p, DROP_RTR_MAC_CALLBACK);
    return;
  }
  log_link_broke(p);
      if((rt = rtable.rt_lookup(ih->daddr())) == 0) {
    drop(p, DROP_RTR_MAC_CALLBACK);
    return;
  }
  log_link_del(ch->next_hop_);

#ifdef AODV_LOCAL_REPAIR
  /* if the broken link is closer to the dest than source,
     attempt a local repair. Otherwise, bring down the route. */


  if (ch->num_forwards() > rt->rt_hops) {
    local_rt_repair(rt, p); // local repair
    // retrieve all the packets in the ifq using this link,
    // queue the packets for which local repair is done,
    return;
  }
  else
#endif // LOCAL REPAIR

  {
    drop(p, DROP_RTR_MAC_CALLBACK);
    // Do the same thing for other packets in the interface queue using
the
    // broken link -Mahesh
while((p = ifqueue->filter(broken_nbr))) {
     drop(p, DROP_RTR_MAC_CALLBACK);
    }
    nb_delete(broken_nbr);
  }

#endif // LINK LAYER DETECTION
}

Void MYAODV::handle_link_failure(nsaddr_t id) {
aodv_rt_entry *rt, *rtn;
Packet *rerr = Packet::alloc();
struct hdr_aodv_error *re = HDR_AODV_ERROR(rerr);

 re->DestCount = 0;
 for(rt = rtable.head(); rt; rt = rtn) {  // for each rt entry
   rtn = rt->rt_link.le_next;
```

```
   if ((rt->rt_hops != INFINITY2) && (rt->rt_nexthop == id) ) {
     assert (rt->rt_flags == RTF_UP);
     assert((rt->rt_seqno%2) == 0);
     rt->rt_seqno++;
     re->unreachable_dst[re->DestCount] = rt->rt_dst;
     re->unreachable_dst_seqno[re->DestCount] = rt->rt_seqno;
#ifdef DEBUG
     fprintf(stderr, "%s(%f): %d\t(%d\t%u\t%d)\n", __FUNCTION__,
CURRENT_TIME,
                 index, re->unreachable_dst[re->DestCount],
                 re->unreachable_dst_seqno[re->DestCount], rt-
>rt_nexthop);
#endif // DEBUG
     re->DestCount += 1;
     rt_down(rt);
   }
   // remove the lost neighbor from all the precursor lists
   rt->pc_delete(id);
 }

 if (re->DestCount > 0) {
#ifdef DEBUG
   fprintf(stderr, "%s(%f): %d\tsending RERR...\n", __FUNCTION__,
CURRENT_TIME, index);
#endif // DEBUG
   sendError(rerr, false);
 }
 else {
   Packet::free(rerr);
 }
}

Void MYAODV::local_rt_repair(aodv_rt_entry *rt, Packet *p) {
#ifdef DEBUG
  fprintf(stderr,"%s: Dst - %d\n", __FUNCTION__, rt->rt_dst);
#endif
  // Buffer the packet
  rqueue.enque(p);

  // mark the route as under repair
  rt->rt_flags = RTF_IN_REPAIR;

  sendRequest(rt->rt_dst);

  // set up a timer interrupt
  Scheduler::instance().schedule(&lrtimer, p->copy(), rt-
>rt_req_timeout);
}

Void MYAODV::rt_update(aodv_rt_entry *rt, u_int32_t seqnum, u_int16_t
metric,
                   nsaddr_t nexthop, double expire_time) {

     rt->rt_seqno = seqnum;
     rt->rt_hops = metric;
     rt->rt_flags = RTF_UP;
     rt->rt_nexthop = nexthop;
```

```
     rt->rt_expire = expire_time;
}

Void MYAODV::rt_down(aodv_rt_entry *rt) {
  /*
   *  Make sure that you don't "down" a route more than once.
   */

  if(rt->rt_flags == RTF_DOWN) {
    return;
  }

  // assert (rt->rt_seqno%2); // is the seqno odd?
  rt->rt_last_hop_count = rt->rt_hops;
  rt->rt_hops = INFINITY2;
  rt->rt_flags = RTF_DOWN;
  rt->rt_nexthop = 0;
  rt->rt_expire = 0;

} /* rt_down function */

/*
  Route Handling Functions
*/

Void MYAODV::rt_resolve(Packet *p) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
aodv_rt_entry *rt;

 /*
  *  Set the transmit failure callback.  That
  *  won't change.
  */
 ch->xmit_failure_ = aodv_rt_failed_callback;
 ch->xmit_failure_data_ = (void*) this;
      rt = rtable.rt_lookup(ih->daddr());
 if(rt == 0) {
        rt = rtable.rt_add(ih->daddr());
 }

 /*
  * If the route is up, forward the packet
  */

 if(rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   forward(rt, p, NO_DELAY);
 }
 /*
  *  if I am the source of the packet, then do a Route Request.
  */
      else if(ih->saddr() == index) {
   rqueue.enque(p);
   sendRequest(rt->rt_dst);
 }
 /*
```

```
  *   A local repair is in progress. Buffer the packet.
  */
 else if (rt->rt_flags == RTF_IN_REPAIR) {
   rqueue.enque(p);
 }

 /*
  * I am trying to forward a packet for someone else to which
  * I don't have a route.
  */
 else {
 Packet *rerr = Packet::alloc();
 struct hdr_aodv_error *re = HDR_AODV_ERROR(rerr);
 /*
  * For now, drop the packet and send error upstream.
  * Now the route errors are broadcast to upstream
  * neighbors - Mahesh 09/11/99
  */

   assert (rt->rt_flags == RTF_DOWN);
   re->DestCount = 0;
   re->unreachable_dst[re->DestCount] = rt->rt_dst;
   re->unreachable_dst_seqno[re->DestCount] = rt->rt_seqno;
   re->DestCount += 1;
#ifdef DEBUG
   fprintf(stderr, "%s: sending RERR...\n", __FUNCTION__);
#endif
   sendError(rerr, false);

   drop(p, DROP_RTR_NO_ROUTE);
 }

}

Void MYAODV::rt_purge() {
aodv_rt_entry *rt, *rtn;
double now = CURRENT_TIME;
double delay = 0.0;
Packet *p;

 for(rt = rtable.head(); rt; rt = rtn) {  // for each rt entry
   rtn = rt->rt_link.le_next;
   if ((rt->rt_flags == RTF_UP) && (rt->rt_expire < now)) {
   // if a valid route has expired, purge all packets from
   // send buffer and invalidate the route.
      assert(rt->rt_hops != INFINITY2);
     while((p = rqueue.deque(rt->rt_dst))) {
#ifdef DEBUG
       fprintf(stderr, "%s: calling drop()\n",
                        __FUNCTION__);
#endif // DEBUG
       drop(p, DROP_RTR_NO_ROUTE);
     }
     rt->rt_seqno++;
     assert (rt->rt_seqno%2);
     rt_down(rt);
   }
```

```
   else if (rt->rt_flags == RTF_UP) {
   // If the route is not expired,
   // and there are packets in the sendbuffer waiting,
   // forward them. This should not be needed, but this extra
   // check does no harm.
     assert(rt->rt_hops != INFINITY2);
     while((p = rqueue.deque(rt->rt_dst))) {
       forward (rt, p, delay);
       delay += ARP_DELAY;
     }
   }
   else if (rqueue.find(rt->rt_dst))
   // If the route is down and
   // if there is a packet for this destination waiting in
   // the sendbuffer, then send out route request. sendRequest
   // will check whether it is time to really send out request
   // or not.
   // This may not be crucial to do it here, as each generated
   // packet will do a sendRequest anyway.

     sendRequest(rt->rt_dst);
   }

}

/*
  Packet Reception Routines
*/

Void MYAODV::recv(Packet *p, Handler*) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);

 assert(initialized());
 //assert(p->incoming == 0);
 // XXXXX NOTE: use of incoming flag has been depracated; In order to
track direction of pkt flow, direction_ in hdr_cmn is used instead. see
packet.h for details.

 if(ch->ptype() == PT_AODV) {
   ih->ttl_ -= 1;
   recvAODV(p);
   return;
 }

//To isolate victim node 4 from receiving data packets, drops all the
packets destined
 //to the victim node.

 if (ih->daddr() == 4)
   {
     Packet::free(p);
     return;
   }


 /*
```

```
  *  Must be a packet I'm originating...
  */
if((ih->saddr() == index) && (ch->num_forwards() == 0)) {
 /*
  * Add the IP Header
  */
   ch->size() += IP_HDR_LEN;
   ih->ttl_ = NETWORK_DIAMETER;
 }
 /*
  *  I received a packet that I sent.  Probably
  *  a routing loop.
  */
else if(ih->saddr() == index) {
   drop(p, DROP_RTR_ROUTE_LOOP);
   return;
 }
 /*
  *  Packet I'm forwarding...
  */
 else {
 /*
  *  Check the TTL.  If it is zero, then discard.
  */
   if(--ih->ttl_ == 0) {
     drop(p, DROP_RTR_TTL);
     return;
   }
 }

 rt_resolve(p);

}


Void MYAODV::recvAODV(Packet *p) {
struct hdr_aodv *ah = HDR_AODV(p);
struct hdr_ip *ih = HDR_IP(p);

 assert(ih->sport() == RT_PORT);
 assert(ih->dport() == RT_PORT);

 /*
  * Incoming Packets.
  */
 switch(ah->ah_type) {

 case AODVTYPE_RREQ:
   recvRequest(p);
   break;

 case AODVTYPE_RREP:
   recvReply(p);
   break;

 case AODVTYPE_RERR:
   recvError(p);
```

```
   break;

 case AODVTYPE_HELLO:
   recvHello(p);
   break;

 default:
   fprintf(stderr, "Invalid AODV type (%x)\n", ah->ah_type);
   exit(1);
 }

}



/*void MYAODV::recvRequest(Packet *p) {
  Packet::free(p);
  return;
}  */


Void MYAODV::recvRequest(Packet *p) {
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
aodv_rt_entry *rt;

  //
  // Drop if:
  //      - I'm the source
  //      - I recently heard this request.
  //

  if(rq->rq_src == index) {
#ifdef DEBUG
    fprintf(stderr, "%s: got my own REQUEST\n", __FUNCTION__);
#endif // DEBUG
    Packet::free(p);
    return;
  }

 if (id_lookup(rq->rq_src, rq->rq_bcast_id)) {

#ifdef DEBUG
   fprintf(stderr, "%s: discarding request\n", __FUNCTION__);
#endif // DEBUG

   Packet::free(p);
   return;
 }


  // Cache the broadcast ID

 id_insert(rq->rq_src, rq->rq_bcast_id);
```

```
  //* We are either going to forward the REQUEST or generate a
  //* REPLY. Before we do anything, we make sure that the REVERSE
  //* route is in the route table.
  //
 aodv_rt_entry *rt0; // rt0 is the reverse route

   rt0 = rtable.rt_lookup(rq->rq_src);
   if(rt0 == 0) { // if not in the route table
   // create an entry for the reverse route.
     rt0 = rtable.rt_add(rq->rq_src);
   }

   rt0->rt_expire = max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE));

   if ( (rq->rq_src_seqno > rt0->rt_seqno ) ||
      ((rq->rq_src_seqno == rt0->rt_seqno) &&
       (rq->rq_hop_count < rt0->rt_hops)) ) {
   // If we have a fresher seq no. or lesser #hops for the
   // same seq no., update the rt entry. Else don't bother.
rt_update(rt0, rq->rq_src_seqno, rq->rq_hop_count, ih->saddr(),
             max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE)) );
     if (rt0->rt_req_timeout > 0.0) {
     // Reset the soft state and
     // Set expiry time to CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT
     // This is because route is used in the forward direction,
     // but only sources get benefited by this change
       rt0->rt_req_cnt = 0;
       rt0->rt_req_timeout = 0.0;
       rt0->rt_req_last_ttl = rq->rq_hop_count;
       rt0->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
     }

     //* Find out whether any buffered packet can benefit from the
      //* reverse route.
      //* May need some change in the following code - Mahesh 09/11/99
      //
     assert (rt0->rt_flags == RTF_UP);
     Packet *buffered_pkt;
     while ((buffered_pkt = rqueue.deque(rt0->rt_dst))) {
       if (rt0 && (rt0->rt_flags == RTF_UP)) {
      assert(rt0->rt_hops != INFINITY2);
         forward(rt0, buffered_pkt, NO_DELAY);
       }
     }
   }
   // End for putting reverse route in rt table


 //*
 // * We have taken care of the reverse route stuff.
 // * Now see whether we can send a route reply.
  //

 rt = rtable.rt_lookup(rq->rq_dst);

 // First check if I am the destination ..
```

```
 if(rq->rq_dst == index) {

#ifdef DEBUG
   fprintf(stderr, "%d - %s: destination sending reply\n",
                   index, __FUNCTION__);
#endif // DEBUG


   // Just to be safe, I use the max. Somebody may have
   // incremented the dst seqno.
   seqno = max(seqno, rq->rq_dst_seqno)+1;
   if (seqno%2) seqno++;

   sendReply(rq->rq_src,           // IP Destination
             1,                    // Hop Count
             index,                // Dest IP Address
             seqno,                // Dest Sequence Num
             MY_ROUTE_TIMEOUT,     // Lifetime
             rq->rq_timestamp);    // timestamp

   Packet::free(p);
 }

 // I am not the destination, but I may have a fresh enough route.

 else if (rt && (rt->rt_hops != INFINITY2) &&
            (rt->rt_seqno >= rq->rq_dst_seqno) ) {

   //assert (rt->rt_flags == RTF_UP);
   assert(rq->rq_dst == rt->rt_dst);
   //assert ((rt->rt_seqno%2) == 0);      // is the seqno even?
   sendReply(rq->rq_src,
             rt->rt_hops + 1,
             rq->rq_dst,
             rt->rt_seqno,
           (u_int32_t) (rt->rt_expire - CURRENT_TIME),
           //             rt->rt_expire - CURRENT_TIME,
             rq->rq_timestamp);
   // Insert nexthops to RREQ source and RREQ destination in the
   // precursor lists of destination and source respectively
   rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
   rt0->pc_insert(rt->rt_nexthop); // nexthop to RREQ destination
   // TODO: send grat RREP to dst if G flag set in RREQ
   Packet::free(p);
 }
 /*
  * Can't reply. So forward the  Route Request
  */
 else {
   ih->saddr() = index;
   ih->daddr() = IP_BROADCAST;
   rq->rq_hop_count += 1;
   // Maximum sequence number seen en route
   if (rt) rq->rq_dst_seqno = max(rt->rt_seqno, rq->rq_dst_seqno);
   forward((aodv_rt_entry*) 0, p, DELAY);
 }
```

```
/*
  * disrupt the route between node 4 and other nodes
  */
 else if ( (rq->rq_src==4) || (rq->rq_dst ==4))
   {

     if ( rq->rq_src == 4)
       {
       rq->rq_src_seqno = rq->rq_src_seqno + 6;
       }
     else if ( rq->rq_dst == 4 )
       {
       rq->rq_src_seqno =  rq->rq_dst_seqno +6;
       }

     ih->saddr() = index;
     ih->daddr() = IP_BROADCAST;
     //rq->rq_hop_count += 1;
     // Maximum sequence number seen en route
     rq->rq_dst_seqno = 10;

     rq->rq_type = AODVTYPE_RREQ;
     rq->rq_hop_count = 2;
     rq->rq_bcast_id = 200+seqno;
     rq->rq_src = 4;
     rq->rq_dst = 66;
     seqno +=2;

     forward((aodv_rt_entry*) 0, p, DELAY);
     //   drop(p, DROP_RTR_NO_ROUTE);
     // Packet::free(p);
   }
 else
   Packet::free(p);

}
Void MYAODV::recvReply(Packet *p) {
//struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
aodv_rt_entry *rt;
char suppress_reply = 0;
double delay = 0.0;

#ifdef DEBUG
 fprintf(stderr, "%d - %s: received a REPLY\n", index, __FUNCTION__);
#endif // DEBUG


 /*
  *  Got a reply. So reset the "soft state" maintained for
  *  route requests in the request table. We don't really have
  *  have a separate request table. It is just a part of the
  *  routing table itself.
  */
 // Note that rp_dst is the dest of the data packets, not the
 // the dest of the reply, which is the src of the data packets.
```

```
 rt = rtable.rt_lookup(rp->rp_dst);

 /*
  *  If I don't have a rt entry to this host... adding
  */
 if(rt == 0) {
   rt = rtable.rt_add(rp->rp_dst);
 }

 /*
  * Add a forward route table entry... here I am following
  * Perkins-Royer AODV paper almost literally - SRD 5/99
  */

 if ( (rt->rt_seqno < rp->rp_dst_seqno) ||   // newer route,  the default
value of rt_seqno=0
      ((rt->rt_seqno == rp->rp_dst_seqno) &&
       (rt->rt_hops > rp->rp_hop_count)) ) { // shorter or better route,
the default value of rt_hops=INFINITY2

  // Update the rt entry
  rt_update(rt, rp->rp_dst_seqno, rp->rp_hop_count,
            rp->rp_src, CURRENT_TIME + rp->rp_lifetime);

  // reset the soft state
  rt->rt_req_cnt = 0;
  rt->rt_req_timeout = 0.0;
  rt->rt_req_last_ttl = rp->rp_hop_count;

if (ih->daddr() == index) { // If I am the original source
  // Update the route discovery latency statistics
  // rp->rp_timestamp is the time of request origination

    rt->rt_disc_latency[rt->hist_indx] = (CURRENT_TIME - rp-
>rp_timestamp)
                                         / (double) rp->rp_hop_count;
    // increment indx for next time
    rt->hist_indx = (rt->hist_indx + 1) % MAX_HISTORY;
  }

  /*
   * Send all packets queued in the sendbuffer destined for
   * this destination.
   * XXX - observe the "second" use of p.
   */
  Packet *buf_pkt;
  while((buf_pkt = rqueue.deque(rt->rt_dst))) {
    if(rt->rt_hops != INFINITY2) {
          assert (rt->rt_flags == RTF_UP);
    // Delay them a little to help ARP. Otherwise ARP
    // may drop packets. -SRD 5/23/99
      forward(rt, buf_pkt, delay);
      delay += ARP_DELAY;
    }
  }
 }
```

```
 else {
  suppress_reply = 1;
 }

 /*
  * If reply is for me, discard it.
  */

if(ih->daddr() == index || suppress_reply) {
   Packet::free(p);
 }
 /*
  * Otherwise, forward the Route Reply.
  */
 else {
 // Find the rt entry
aodv_rt_entry *rt0 = rtable.rt_lookup(ih->daddr());
   // If the rt is up, forward
   if(rt0 && (rt0->rt_hops != INFINITY2)) {
        assert (rt0->rt_flags == RTF_UP);
     rp->rp_hop_count += 1;
     rp->rp_src = index;
     forward(rt0, p, NO_DELAY);
     // Insert the nexthop towards the RREQ source to
     // the precursor list of the RREQ destination
     rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source

   }
   else {
   // I don't know how to forward .. drop the reply.
#ifdef DEBUG
     fprintf(stderr, "%s: dropping Route Reply\n", __FUNCTION__);
#endif // DEBUG
     drop(p, DROP_RTR_NO_ROUTE);
   }
 }
}

Void MYAODV::recvError(Packet *p) {
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_error *re = HDR_AODV_ERROR(p);
aodv_rt_entry *rt;
u_int8_t i;
Packet *rerr = Packet::alloc();
struct hdr_aodv_error *nre = HDR_AODV_ERROR(rerr);

 nre->DestCount = 0;

 for (i=0; i<re->DestCount; i++) {
 // For each unreachable destination
   rt = rtable.rt_lookup(re->unreachable_dst[i]);
   if ( rt && (rt->rt_hops != INFINITY2) &&
      (rt->rt_nexthop == ih->saddr()) &&
      (rt->rt_seqno <= re->unreachable_dst_seqno[i]) ) {
      assert(rt->rt_flags == RTF_UP);
      assert((rt->rt_seqno%2) == 0); // is the seqno even?
#ifdef DEBUG
```

```
     fprintf(stderr, "%s(%f): %d\t(%d\t%u\t%d)\t(%d\t%u\t%d)\n",
__FUNCTION__,CURRENT_TIME,
                 index, rt->rt_dst, rt->rt_seqno, rt->rt_nexthop,
                 re->unreachable_dst[i],re->unreachable_dst_seqno[i],
                   ih->saddr());
#endif // DEBUG
      rt->rt_seqno = re->unreachable_dst_seqno[i];
      rt_down(rt);

   // Not sure whether this is the right thing to do
   Packet *pkt;
      while((pkt = ifqueue->filter(ih->saddr()))) {
            drop(pkt, DROP_RTR_MAC_CALLBACK);
      }

     // if precursor list non-empty add to RERR and delete the precursor
list
      if (!rt->pc_empty()) {
            nre->unreachable_dst[nre->DestCount] = rt->rt_dst;
            nre->unreachable_dst_seqno[nre->DestCount] = rt->rt_seqno;
            nre->DestCount += 1;
            rt->pc_delete();
      }
   }
 }

 if (nre->DestCount > 0) {
#ifdef DEBUG
   fprintf(stderr, "%s(%f): %d\t sending RERR...\n", __FUNCTION__,
CURRENT_TIME, index);
#endif // DEBUG
   sendError(rerr);
 }
 else {
   Packet::free(rerr);
 }

 Packet::free(p);
}


/*
   Packet Transmission Routines
*/

Void MYAODV::forward(aodv_rt_entry *rt, Packet *p, double delay) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);

 if(ih->ttl_ == 0) {

#ifdef DEBUG
  fprintf(stderr, "%s: calling drop()\n", __PRETTY_FUNCTION__);
#endif // DEBUG

  drop(p, DROP_RTR_TTL);
  return;
```

```
 }

 if (rt) {
   assert(rt->rt_flags == RTF_UP);
   rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
   ch->next_hop_ = rt->rt_nexthop;
   ch->addr_type() = NS_AF_INET;
   ch->direction() = hdr_cmn::DOWN;       //important: change the
packet's direction
 }
 else { // if it is a broadcast packet
   assert(ch->ptype() == PT_AODV);
   assert(ih->daddr() == (nsaddr_t) IP_BROADCAST);
   ch->addr_type() = NS_AF_NONE;
   ch->direction() = hdr_cmn::DOWN;       //important: change the
packet's direction
 }

if (ih->daddr() == (nsaddr_t) IP_BROADCAST) {
 // If it is a broadcast packet
   assert(rt == 0);
   /*
    *  Jitter the sending of broadcast packets by 10ms
    */
   Scheduler::instance().schedule(target_, p,
                                  0.01 * Random::uniform());
 }
 else { // Not a broadcast packet
   if(delay > 0.0) {
     Scheduler::instance().schedule(target_, p, delay);
   }
   else {
   // Not a broadcast packet, no delay, send immediately
     Scheduler::instance().schedule(target_, p, 0.);
   }
 }

}

Void MYAODV::sendRequest(nsaddr_t dst) {
// Allocate a RREQ packet
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
aodv_rt_entry *rt = rtable.rt_lookup(dst);

 assert(rt);

 /*
  *  Rate limit sending of Route Requests. We are very conservative
  *  about sending out route requests.
  */

 if (rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   Packet::free((Packet *)p);
```

```
  return;
 }

 if (rt->rt_req_timeout > CURRENT_TIME) {
   Packet::free((Packet *)p);
   return;
 }

 // rt_req_cnt is the no. of times we did network-wide broadcast
 // RREQ_RETRIES is the maximum number we will allow before
 // going to a long timeout.

 if (rt->rt_req_cnt > RREQ_RETRIES) {
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
   rt->rt_req_cnt = 0;
 Packet *buf_pkt;
   while ((buf_pkt = rqueue.deque(rt->rt_dst))) {
       drop(buf_pkt, DROP_RTR_NO_ROUTE);
   }
   Packet::free((Packet *)p);
   return;
 }

#ifdef DEBUG
   fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
                   ++route_request, index, rt->rt_dst);
#endif // DEBUG

 // Determine the TTL to be used this time.
 // Dynamic TTL evaluation - SRD

 rt->rt_req_last_ttl = max(rt->rt_req_last_ttl,rt->rt_last_hop_count);

 if (0 == rt->rt_req_last_ttl) {
 // first time query broadcast
   ih->ttl_ = TTL_START;
 }
 else {
 // Expanding ring search.
   if (rt->rt_req_last_ttl < TTL_THRESHOLD)
     ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
   else {
   // network-wide broadcast
     ih->ttl_ = NETWORK_DIAMETER;
     rt->rt_req_cnt += 1;
   }
 }

 // remember the TTL used  for the next time
 rt->rt_req_last_ttl = ih->ttl_;

 // PerHopTime is the roundtrip time per hop for route requests.
 // The factor 2.0 is just to be safe .. SRD 5/22/99
 // Also note that we are making timeouts to be larger if we have
 // done network wide broadcast before.

 rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime(rt);
```

```
 if (rt->rt_req_cnt > 0)
   rt->rt_req_timeout *= rt->rt_req_cnt;
 rt->rt_req_timeout += CURRENT_TIME;

 // Don't let the timeout to be too large, however .. SRD 6/8/99
 if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
 rt->rt_expire = 0;

#ifdef DEBUG
 fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f
ms\n",
              ++route_request,
             index, rt->rt_dst,
             rt->rt_req_timeout - CURRENT_TIME);
#endif      // DEBUG


 // Fill out the RREQ packet
 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rq->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->prev_hop_ = index;          // AODV hack

 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;

 // Fill up some more fields.
 rq->rq_type = AODVTYPE_RREQ;
 rq->rq_hop_count = 1;
 rq->rq_bcast_id = bid++;
 rq->rq_dst = dst;

 rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
 rq->rq_src = index;
 seqno += 2;
 assert ((seqno%2) == 0);
 rq->rq_src_seqno = seqno;
 rq->rq_timestamp = CURRENT_TIME;

 Scheduler::instance().schedule(target_, p, 0.);

}


Void MYAODV::sendReply(nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t
rpdst,
                u_int32_t rpseq, u_int32_t lifetime, double timestamp) {
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
```

```
aodv_rt_entry *rt = rtable.rt_lookup(ipdst);

#ifdef DEBUG
fprintf(stderr, "sending Reply from %d at %.2f\n", index,
Scheduler::instance().clock());
#endif // DEBUG
 assert(rt);

 rp->rp_type = AODVTYPE_RREP;
 //rp->rp_flags = 0x00;
 rp->rp_hop_count = hop_count;
 rp->rp_dst = rpdst;
 rp->rp_dst_seqno = rpseq;
 rp->rp_src = index;
 rp->rp_lifetime = lifetime;
 rp->rp_timestamp = timestamp;

 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rp->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_INET;
 ch->next_hop_ = rt->rt_nexthop;
 ch->prev_hop_ = index;          // AODV hack
 ch->direction() = hdr_cmn::DOWN;

 ih->saddr() = index;
 ih->daddr() = ipdst;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = NETWORK_DIAMETER;

 Scheduler::instance().schedule(target_, p, 0.);

}

Void MYAODV::rreqResourceConsumption(nsaddr_t srcNode, nsaddr_t dstNode,
double simtime, double interval) {
  double inter;
  double attackDelay=0;
  int  count=0;
  int total;
  inter = interval;
  total = (int)(simtime/inter);

  while ( count <total )
    {

      Packet *p = Packet::alloc();
      struct hdr_cmn *ch = HDR_CMN(p);
      struct hdr_ip *ih = HDR_IP(p);
      struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);


      // Fill out the RREQ packet
      // ch->uid() = 0;
```

```
      ch->ptype() = PT_AODV;
      ch->size() = IP_HDR_LEN + rq->size();
      ch->iface() = -2;
      ch->error() = 0;
      ch->addr_type() = NS_AF_NONE;
      //      ch->prev_hop_ = index;          // AODV hack
      ch->prev_hop_ = 88;          // AODV hack

      //      ih->saddr() = index;
      ih->saddr() = 88;
      ih->daddr() = IP_BROADCAST;
      ih->sport() = RT_PORT;
      ih->dport() = RT_PORT;

      // Fill up some more fields.
      rq->rq_type = AODVTYPE_RREQ;
      rq->rq_hop_count = 1;
      rq->rq_bcast_id = 100+bid++;  //to be accepted by source node

      //      rq->rq_dst = dst;
      rq->rq_dst = srcNode;
      rq->rq_dst_seqno = count;
      //  rq->rq_src = index;
      rq->rq_src = dstNode;
      seqno += 2;
      assert ((seqno%2) == 0);
      rq->rq_src_seqno = seqno+6;
      rq->rq_timestamp = CURRENT_TIME;

      // while(count<3)
      // {
      // Scheduler::instance().schedule(target_, p, 0.);
      p->uid_ = 0;
      attackDelay = count*inter;
      Scheduler::instance().schedule(target_, p, attackDelay);
      count += 1;
    }

}


Void MYAODV::sendError(Packet *p, bool jitter = true) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_error *re = HDR_AODV_ERROR(p);

#ifdef ERROR
fprintf(stderr, "sending Error from %d at %.2f\n", index,
Scheduler::instance().clock());
#endif // DEBUG

 re->re_type = AODVTYPE_RERR;
 //re->reserved[0] = 0x00; re->reserved[1] = 0x00;
 // DestCount and list of unreachable destinations are already filled

 // ch->uid() = 0;
```

```
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + re->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->next_hop_ = 0;
 ch->prev_hop_ = index;          // AODV hack
 ch->direction() = hdr_cmn::DOWN;       //important: change the packet's
direction

 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = 1;

 // Do we need any jitter? Yes
 if (jitter)
      Scheduler::instance().schedule(target_, p, 0.01*Random::uniform());
 else
      Scheduler::instance().schedule(target_, p, 0.0);

}


/*
   Neighbor Management Functions
*/

Void MYAODV::sendHello() {
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rh = HDR_AODV_REPLY(p);

#ifdef DEBUG
fprintf(stderr, "sending Hello from %d at %.2f\n", index,
Scheduler::instance().clock());
#endif // DEBUG

 rh->rp_type = AODVTYPE_HELLO;
 //rh->rp_flags = 0x00;
 rh->rp_hop_count = 1;
 rh->rp_dst = index;
 rh->rp_dst_seqno = seqno;
 rh->rp_lifetime = (1 + ALLOWED_HELLO_LOSS) * HELLO_INTERVAL;

 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rh->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->prev_hop_ = index;          // AODV hack

 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
```

```
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = 1;

 Scheduler::instance().schedule(target_, p, 0.0);
}

Void MYAODV::recvHello(Packet *p) {
//struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
AODV_Neighbor *nb;

 nb = nb_lookup(rp->rp_dst);
 if(nb == 0) {
   nb_insert(rp->rp_dst);
 }
 else {
   nb->nb_expire = CURRENT_TIME +
                   (1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
 }

 Packet::free(p);
}

Void MYAODV::nb_insert(nsaddr_t id) {
AODV_Neighbor *nb = new AODV_Neighbor(id);

 assert(nb);
 nb->nb_expire = CURRENT_TIME +
                 (1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
 LIST_INSERT_HEAD(&nbhead, nb, nb_link);
 seqno += 2;             // set of neighbors changed
 assert ((seqno%2) == 0);
}


AODV_Neighbor*
MYAODV::nb_lookup(nsaddr_t id) {
AODV_Neighbor *nb = nbhead.lh_first;

 for(; nb; nb = nb->nb_link.le_next) {
   if(nb->nb_addr == id) break;
 }
 return nb;
}
/*
 * Called when we receive *explicit* notification that a Neighbor
 * is no longer reachable.
 */
Void MYAODV::nb_delete(nsaddr_t id) {
AODV_Neighbor *nb = nbhead.lh_first;

 log_link_del(id);
 seqno += 2;     // Set of neighbors changed
 assert ((seqno%2) == 0);

 for(; nb; nb = nb->nb_link.le_next) {
```

```
   if(nb->nb_addr == id) {
     LIST_REMOVE(nb,nb_link);
     delete nb;
     break;
   }
 }
 handle_link_failure(id);

}
/*
 * Purges all timed-out Neighbor Entries - runs every
 * HELLO_INTERVAL * 1.5 seconds.
 */
Void MYAODV::nb_purge() {
AODV_Neighbor *nb = nbhead.lh_first;
AODV_Neighbor *nbn;
double now = CURRENT_TIME;

 for(; nb; nb = nbn) {
   nbn = nb->nb_link.le_next;
   if(nb->nb_expire <= now) {
     //nb_delete(nb->nb_addr);
   }
 }

}
```

Appendix II Implementation code of IDS for AODV

```
*/

//#include <ip.h>

#include <aodv/aodvids.h>
#include <aodv/aodv_packet.h>
#include <random.h>
#include <cmu-trace.h>
//#include <energy-model.h>

#define max(a,b)        a > b ? a : b
#define CURRENT_TIME    Scheduler::instance().clock()

//#define DEBUG
//#define ERROR

#ifdef DEBUG
static int extra_route_reply = 0;
static int limit_route_request = 0;
static int route_request = 0;
#endif


/*
  TCL Hooks
*/


//int hdr_aodv::offset_;
static class AODVIDSHeaderClass : public PacketHeaderClass {
public:
        AODVIDSHeaderClass() : PacketHeaderClass("PacketHeader/AODV",
                                                 sizeof(hdr_all_aodv)) {
        bind_offset(&hdr_aodv::offset_);
      }
} class_rtProtoAODV_hdr;

static class AODVIDSclass : public TclClass {
public:
        AODVIDSclass() : TclClass("Agent/AODVIDS") {}
        TclObject* create(int argc, const char*const* argv) {
          assert(argc == 5);
          //return (new AODV((nsaddr_t) atoi(argv[4])));
        return (new AODVIDS((nsaddr_t)
Address::instance().str2addr(argv[4])));
        }
} class_rtProtoAODV;


int
```

```
AODVIDS::command(int argc, const char*const* argv) {
  if(argc == 2) {
  Tcl& tcl = Tcl::instance();

    if(strncasecmp(argv[1], "id", 2) == 0) {
      tcl.resultf("%d", index);
      return TCL_OK;
    }

    if(strncasecmp(argv[1], "start", 2) == 0) {
      btimer.handle((Event*) 0);

#ifndef AODV_LINK_LAYER_DETECTION
      htimer.handle((Event*) 0);
      ntimer.handle((Event*) 0);
#endif // LINK LAYER DETECTION

      rtimer.handle((Event*) 0);
      return TCL_OK;
     }
  }
  else if(argc == 3) {
    if(strcmp(argv[1], "index") == 0) {
      index = atoi(argv[2]);
      return TCL_OK;
    }

    else if(strcmp(argv[1], "log-target") == 0 || strcmp(argv[1],
"tracetarget") == 0) {
      logtarget = (Trace*) TclObject::lookup(argv[2]);
      if(logtarget == 0)
      return TCL_ERROR;
      return TCL_OK;
    }
    else if(strcmp(argv[1], "drop-target") == 0) {
    int stat = rqueue.command(argc,argv);
      if (stat != TCL_OK) return stat;
      return Agent::command(argc, argv);
    }
    else if(strcmp(argv[1], "if-queue") == 0) {
    ifqueue = (PriQueue*) TclObject::lookup(argv[2]);

      if(ifqueue == 0)
      return TCL_ERROR;
      return TCL_OK;
    }
  }
  return Agent::command(argc, argv);
}

/*
   Constructor
*/

AODVIDS::AODVIDS(nsaddr_t id) : Agent(PT_AODV),
                    btimer(this), htimer(this), ntimer(this),
                    rtimer(this), lrtimer(this), rqueue() {
```

```
  index = id;
  seqno = 2;
  bid = 1;
  head = NULL;

  LIST_INIT(&nbhead);
  LIST_INIT(&bihead);

  logtarget = 0;
  ifqueue = 0;
}
/*
* Koni Addition
*/
/*struct SusListNode {
      nsaddr_t    nodeid;
      int         counter;
      float       intime;
      SusListNode*      next;
};

struct SusListNode* head=NULL;*/
/*
  Timers
*/
Void IDSBroadcastTimer::handle(Event*) {
  agent->id_purge();
  Scheduler::instance().schedule(this, &intr, BCAST_ID_SAVE);
}

Void IDSHelloTimer::handle(Event*) {
   agent->sendHello();
   double interval = MinHelloInterval +
                ((MaxHelloInterval - MinHelloInterval) *
Random::uniform());
   assert(interval >= 0);
   Scheduler::instance().schedule(this, &intr, interval);
}

Void IDSNeighborTimer::handle(Event*) {
  agent->nb_purge();
  Scheduler::instance().schedule(this, &intr, HELLO_INTERVAL);
}

Void IDSRouteCacheTimer::handle(Event*) {
  agent->rt_purge();
#define FREQUENCY 0.5 // sec
  Scheduler::instance().schedule(this, &intr, FREQUENCY);
}

Void IDSLocalRepairTimer::handle(Event* p)  {  // SRD: 5/4/99
aodv_rt_entry *rt;
struct hdr_ip *ih = HDR_IP( (Packet *)p);

   /* you get here after the timeout in a local repair attempt */
```

```
   /* fprintf(stderr, "%s\n", __FUNCTION__); */


    rt = agent->rtable.rt_lookup(ih->daddr());

    if (rt && rt->rt_flags != RTF_UP) {
    // route is yet to be repaired
    // I will be conservative and bring down the route
    // and send route errors upstream.
    /* The following assert fails, not sure why */
    /* assert (rt->rt_flags == RTF_IN_REPAIR); */

      //rt->rt_seqno++;
      agent->rt_down(rt);
      // send RERR
#ifdef DEBUG
      fprintf(stderr,"Node %d: Dst - %d, failed local repair\n",index,
rt->rt_dst);
#endif
    }
    Packet::free((Packet *)p);
}

/*
   Broadcast ID Management  Functions
*/


Void AODVIDS::id_insert(nsaddr_t id, u_int32_t bid) {
BroadcastID *b = new BroadcastID(id, bid);

 assert(b);
 b->expire = CURRENT_TIME + BCAST_ID_SAVE;
 LIST_INSERT_HEAD(&bihead, b, link);
}

/* SRD */
Bool AODVIDS::id_lookup(nsaddr_t id, u_int32_t bid) {
BroadcastID *b = bihead.lh_first;

 // Search the list for a match of source and bid
 for( ; b; b = b->link.le_next) {
   if ((b->src == id) && (b->id == bid))
     return true;
 }
 return false;
}

Void AODVIDS::id_purge() {
BroadcastID *b = bihead.lh_first;
BroadcastID *bn;
double now = CURRENT_TIME;

 for(; b; b = bn) {
   bn = b->link.le_next;
   if(b->expire <= now) {
     LIST_REMOVE(b,link);
```

```
     delete b;
   }
 }
}

/*
  Helper Functions
*/

Double AODVIDS::PerHopTime(aodv_rt_entry *rt) {
int num_non_zero = 0, i;
double total_latency = 0.0;

 if (!rt)
   return ((double) NODE_TRAVERSAL_TIME );

 for (i=0; i < MAX_HISTORY; i++) {
   if (rt->rt_disc_latency[i] > 0.0) {
      num_non_zero++;
      total_latency += rt->rt_disc_latency[i];
   }
 }
 if (num_non_zero > 0)
   return(total_latency / (double) num_non_zero);
 else
   return((double) NODE_TRAVERSAL_TIME);

}

/*
  Link Failure Management Functions
*/

static void
aodv_rt_failed_callback(Packet *p, void *arg) {
  ((AODVIDS*) arg)->rt_ll_failed(p);
}

/*
 * This routine is invoked when the link-layer reports a route failed.
 */
Void AODVIDS::rt_ll_failed(Packet *p) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
aodv_rt_entry *rt;
nsaddr_t broken_nbr = ch->next_hop_;

#ifndef AODV_LINK_LAYER_DETECTION
 drop(p, DROP_RTR_MAC_CALLBACK);
#else

 /*
  * Non-data packets and Broadcast Packets can be dropped.
  */
  if(! DATA_PACKET(ch->ptype()) ||
     (u_int32_t) ih->daddr() == IP_BROADCAST) {
    drop(p, DROP_RTR_MAC_CALLBACK);
```

```
    return;
  }
  log_link_broke(p);
      if((rt = rtable.rt_lookup(ih->daddr())) == 0) {
    drop(p, DROP_RTR_MAC_CALLBACK);
    return;
  }
  log_link_del(ch->next_hop_);

#ifdef AODV_LOCAL_REPAIR
  /* if the broken link is closer to the dest than source,
     attempt a local repair. Otherwise, bring down the route. */


  if (ch->num_forwards() > rt->rt_hops) {
    local_rt_repair(rt, p); // local repair
    // retrieve all the packets in the ifq using this link,
    // queue the packets for which local repair is done,
    return;
  }
  else
#endif // LOCAL REPAIR

  {
    drop(p, DROP_RTR_MAC_CALLBACK);
    // Do the same thing for other packets in the interface queue using
the
    // broken link -Mahesh
while((p = ifqueue->filter(broken_nbr))) {
     drop(p, DROP_RTR_MAC_CALLBACK);
    }
    nb_delete(broken_nbr);
  }

#endif // LINK LAYER DETECTION
}

Void AODVIDS::handle_link_failure(nsaddr_t id) {
aodv_rt_entry *rt, *rtn;
Packet *rerr = Packet::alloc();
struct hdr_aodv_error *re = HDR_AODV_ERROR(rerr);

 re->DestCount = 0;
 for(rt = rtable.head(); rt; rt = rtn) {  // for each rt entry
   rtn = rt->rt_link.le_next;
   if ((rt->rt_hops != INFINITY2) && (rt->rt_nexthop == id) ) {
     assert (rt->rt_flags == RTF_UP);
     assert((rt->rt_seqno%2) == 0);
     rt->rt_seqno++;
     re->unreachable_dst[re->DestCount] = rt->rt_dst;
     re->unreachable_dst_seqno[re->DestCount] = rt->rt_seqno;
#ifdef DEBUG
     fprintf(stderr, "%s(%f): %d\t(%d\t%u\t%d)\n", __FUNCTION__,
CURRENT_TIME,
                  index, re->unreachable_dst[re->DestCount],
                  re->unreachable_dst_seqno[re->DestCount], rt-
>rt_nexthop);
```

```
#endif // DEBUG
     re->DestCount += 1;
     rt_down(rt);
   }
   // remove the lost neighbor from all the precursor lists
   rt->pc_delete(id);
 }

 if (re->DestCount > 0) {
#ifdef DEBUG
   fprintf(stderr, "%s(%f): %d\tsending RERR...\n", __FUNCTION__,
CURRENT_TIME, index);
#endif // DEBUG
   sendError(rerr, false);
 }
 else {
   Packet::free(rerr);
 }
}

Void AODVIDS::local_rt_repair(aodv_rt_entry *rt, Packet *p) {
#ifdef DEBUG
  fprintf(stderr,"%s: Dst - %d\n", __FUNCTION__, rt->rt_dst);
#endif
  // Buffer the packet
  rqueue.enque(p);

  // mark the route as under repair
  rt->rt_flags = RTF_IN_REPAIR;

  sendRequest(rt->rt_dst);

  // set up a timer interrupt
  Scheduler::instance().schedule(&lrtimer, p->copy(), rt-
>rt_req_timeout);
}

Void AODVIDS::rt_update(aodv_rt_entry *rt, u_int32_t seqnum, u_int16_t
metric,
                  nsaddr_t nexthop, double expire_time) {

     rt->rt_seqno = seqnum;
     rt->rt_hops = metric;
     rt->rt_flags = RTF_UP;
     rt->rt_nexthop = nexthop;
     rt->rt_expire = expire_time;
}

Void AODVIDS::rt_down(aodv_rt_entry *rt) {
  /*
   *  Make sure that you don't "down" a route more than once.
   */

  if(rt->rt_flags == RTF_DOWN) {
    return;
  }
```

```
  // assert (rt->rt_seqno%2); // is the seqno odd?
  rt->rt_last_hop_count = rt->rt_hops;
  rt->rt_hops = INFINITY2;
  rt->rt_flags = RTF_DOWN;
  rt->rt_nexthop = 0;
  rt->rt_expire = 0;

} /* rt_down function */

/*
  Route Handling Functions
*/

Void AODVIDS::rt_resolve(Packet *p) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
aodv_rt_entry *rt;

 /*
  *  Set the transmit failure callback.  That
  *  won't change.
  */
 ch->xmit_failure_ = aodv_rt_failed_callback;
 ch->xmit_failure_data_ = (void*) this;
      rt = rtable.rt_lookup(ih->daddr());
 if(rt == 0) {
        rt = rtable.rt_add(ih->daddr());
 }

 /*
  * If the route is up, forward the packet
  */

 if(rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   forward(rt, p, NO_DELAY);
 }
 /*
  *  if I am the source of the packet, then do a Route Request.
  */
      else if(ih->saddr() == index) {
   rqueue.enque(p);
   sendRequest(rt->rt_dst);
 }
 /*
  *   A local repair is in progress. Buffer the packet.
  */
 else if (rt->rt_flags == RTF_IN_REPAIR) {
   rqueue.enque(p);
 }

 /*
  * I am trying to forward a packet for someone else to which
  * I don't have a route.
  */
 else {
 Packet *rerr = Packet::alloc();
```

```
 struct hdr_aodv_error *re = HDR_AODV_ERROR(rerr);
 /*
  * For now, drop the packet and send error upstream.
  * Now the route errors are broadcast to upstream
  * neighbors - Mahesh 09/11/99
  */

   assert (rt->rt_flags == RTF_DOWN);
   re->DestCount = 0;
   re->unreachable_dst[re->DestCount] = rt->rt_dst;
   re->unreachable_dst_seqno[re->DestCount] = rt->rt_seqno;
   re->DestCount += 1;
#ifdef DEBUG
   fprintf(stderr, "%s: sending RERR...\n", __FUNCTION__);
#endif
   sendError(rerr, false);

   drop(p, DROP_RTR_NO_ROUTE);
 }

}

Void AODVIDS::rt_purge() {
aodv_rt_entry *rt, *rtn;
double now = CURRENT_TIME;
double delay = 0.0;
Packet *p;

 for(rt = rtable.head(); rt; rt = rtn) {  // for each rt entry
   rtn = rt->rt_link.le_next;
   if ((rt->rt_flags == RTF_UP) && (rt->rt_expire < now)) {
   // if a valid route has expired, purge all packets from
   // send buffer and invalidate the route.
      assert(rt->rt_hops != INFINITY2);
     while((p = rqueue.deque(rt->rt_dst))) {
#ifdef DEBUG
       fprintf(stderr, "%s: calling drop()\n",
                        __FUNCTION__);
#endif // DEBUG
       drop(p, DROP_RTR_NO_ROUTE);
     }
     rt->rt_seqno++;
     assert (rt->rt_seqno%2);
     rt_down(rt);
   }
   else if (rt->rt_flags == RTF_UP) {
   // If the route is not expired,
   // and there are packets in the sendbuffer waiting,
   // forward them. This should not be needed, but this extra
   // check does no harm.
     assert(rt->rt_hops != INFINITY2);
     while((p = rqueue.deque(rt->rt_dst))) {
       forward (rt, p, delay);
       delay += ARP_DELAY;
     }
   }
   else if (rqueue.find(rt->rt_dst))
```

```
   // If the route is down and
   // if there is a packet for this destination waiting in
   // the sendbuffer, then send out route request. sendRequest
   // will check whether it is time to really send out request
   // or not.
   // This may not be crucial to do it here, as each generated
   // packet will do a sendRequest anyway.

     sendRequest(rt->rt_dst);
   }

}

/*
  Packet Reception Routines
*/

Void AODVIDS::recv(Packet *p, Handler*) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);

 assert(initialized());
 //assert(p->incoming == 0);
 // XXXXX NOTE: use of incoming flag has been depracated; In order to
track direction of pkt flow, direction_ in hdr_cmn is used instead. see
packet.h for details.

 if(ch->ptype() == PT_AODV) {
   ih->ttl_ -= 1;
   recvAODV(p);
   return;
 }

 /*
  *  Must be a packet I'm originating...
  */
if((ih->saddr() == index) && (ch->num_forwards() == 0)) {
 /*
  * Add the IP Header
  */
   ch->size() += IP_HDR_LEN;
   // Added by Parag Dadhania && John Novatnack to handle broadcasting
   if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
     ih->ttl_ = NETWORK_DIAMETER;
}
 /*
  *  I received a packet that I sent.  Probably
  *  a routing loop.
  */
else if(ih->saddr() == index) {
   drop(p, DROP_RTR_ROUTE_LOOP);
   return;
 }
 /*
  *  Packet I'm forwarding...
  */
 else {
```

```
 /*
  *  Check the TTL.  If it is zero, then discard.
  */
   if(--ih->ttl_ == 0) {
     drop(p, DROP_RTR_TTL);
     return;
   }
 }
// Added by Parag Dadhania && John Novatnack to handle broadcasting
 if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
   rt_resolve(p);
 else
   forward((aodv_rt_entry*) 0, p, NO_DELAY);
}


Void AODVIDS::recvAODV(Packet *p) {
struct hdr_aodv *ah = HDR_AODV(p);
struct hdr_ip *ih = HDR_IP(p);

 assert(ih->sport() == RT_PORT);
 assert(ih->dport() == RT_PORT);

 /*
  * Incoming Packets.
  */
 switch(ah->ah_type) {

 case AODVTYPE_RREQ:
   recvRequest(p);
   break;

 case AODVTYPE_RREP:
   recvReply(p);
   break;

 case AODVTYPE_RERR:
   recvError(p);
   break;

 case AODVTYPE_HELLO:
   recvHello(p);
   break;

 default:
   fprintf(stderr, "Invalid AODV type (%x)\n", ah->ah_type);
   exit(1);
 }

}
```

```
/* Check if I got to much request from this node
 *
 *
 *    Koni Addition starts
 *
 *
 */

/*

struct SusListNode{
      nsaddr_t    nodeid;
      int         counter;
      float       intime;
      SusListNode*      next;
};

static SusListNode* head=NULL;*/

int AODVIDS::checkTooMuchRequest(Packet *p, int myID) {

      int nodeCount = 0;
      struct hdr_ip *ih = HDR_IP(p);
      struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
      nsaddr_t nodeid = ih->saddr();
      struct SusListNode* newNode = new SusListNode;
        newNode->nodeid = ih->saddr();
        newNode->counter = 1;
        newNode->intime = CURRENT_TIME;


      struct SusListNode* current=head;

      while (current != NULL){
            if( current->nodeid == newNode->nodeid ){
                  current->counter++;
                  //printf("Current time:%f node time:%f Receiver:%d
sender:%d pkt count:%d\n",CURRENT_TIME,current->intime,myID, current-
>nodeid, current->counter);

                  if(current->counter > 10 ){

                        if(CURRENT_TIME - current->intime <= 1){
                              printf("Receiver:%d Sender:%d
initialtime:%f time diff:%f counter:%d\n",myID, current->nodeid, current-
>intime,CURRENT_TIME-current->intime, current->counter);
                              delete newNode;
                  //          current->counter = 0;
                              current->intime = CURRENT_TIME;
                              return 1;
                        }
//                      current->counter = 0;
//                      current->intime = CURRENT_TIME;
                  }
                  //printf("%d,%d,%d\n", current->nodeid,current-
>intime,current->counter);
                  delete newNode;
```

```
                return 0;
            }
            current = current->next;
      }
      newNode->next = head;
      head = newNode;
      return 0;
}

 /* Check if I got to much request from this node
 *
 *
 *    Koni Addition ends
 *
 *
 */

Void AODVIDS::recvRequest(Packet *p) {
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
aodv_rt_entry *rt;
int misuser;

/*
   * Drop if:
   *      - I'm the source
   *      - I recently heard this request.
   *    - I got to much request from this node (Koni Addition)
   */

  if(rq->rq_src == index) {

    aodv_rt_entry *rt0; // rt0 is the reverse route
    if  (rq->rq_src_seqno > rt0->rt_seqno )
      {
      printf("get you");
      }
#ifdef DEBUG
    fprintf(stderr, "%s: got my own REQUEST\n", __FUNCTION__);
#endif // DEBUG
    Packet::free(p);
    return;
  }

 if (id_lookup(rq->rq_src, rq->rq_bcast_id)) {

#ifdef DEBUG
   fprintf(stderr, "%s: discarding request\n", __FUNCTION__);
#endif // DEBUG

   Packet::free(p);
   return;
 }
 /* Check if I got to much request from this node
 *
 *
 *    Koni Addition
```

```
 *
 *
 */
Packet *rerr = Packet::alloc();
misuser = checkTooMuchRequest(p, index);
//printf("Function called and returned: %d\n", misuser);

  if( misuser == 1 ) {
            //fprintf(stderr, "%s: Node - %d - with timestamp - %f time
in list - %f\n", __FUNCTION__, current->nodeid, current->intime,
(CURRENT_TIME-current->intime));
//sendError(rerr,false);
drop(p,DROP_RTR_NO_ROUTE);
//    Packet::free(p);
      return;
}


 /*
  * Cache the broadcast ID
  */
 id_insert(rq->rq_src, rq->rq_bcast_id);



 /*
  * We are either going to forward the REQUEST or generate a
  * REPLY. Before we do anything, we make sure that the REVERSE
  * route is in the route table.
  */
 aodv_rt_entry *rt0; // rt0 is the reverse route

   rt0 = rtable.rt_lookup(rq->rq_src);
   if(rt0 == 0) { /* if not in the route table */
   // create an entry for the reverse route.
     rt0 = rtable.rt_add(rq->rq_src);
   }

   rt0->rt_expire = max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE));

   if ( (rq->rq_src_seqno > rt0->rt_seqno ) ||
      ((rq->rq_src_seqno == rt0->rt_seqno) &&
       (rq->rq_hop_count < rt0->rt_hops)) ) {
   // If we have a fresher seq no. or lesser #hops for the
   // same seq no., update the rt entry. Else don't bother.
rt_update(rt0, rq->rq_src_seqno, rq->rq_hop_count, ih->saddr(),
             max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE)) );
     if (rt0->rt_req_timeout > 0.0) {
     // Reset the soft state and
     // Set expiry time to CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT
     // This is because route is used in the forward direction,
     // but only sources get benefited by this change
       rt0->rt_req_cnt = 0;
       rt0->rt_req_timeout = 0.0;
       rt0->rt_req_last_ttl = rq->rq_hop_count;
       rt0->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
     }
```

```
    /* Find out whether any buffered packet can benefit from the
     * reverse route.
     * May need some change in the following code - Mahesh 09/11/99
     */
    assert (rt0->rt_flags == RTF_UP);
    Packet *buffered_pkt;
    while ((buffered_pkt = rqueue.deque(rt0->rt_dst))) {
      if (rt0 && (rt0->rt_flags == RTF_UP)) {
     assert(rt0->rt_hops != INFINITY2);
        forward(rt0, buffered_pkt, NO_DELAY);
      }
    }
  }
  // End for putting reverse route in rt table


 /*
  * We have taken care of the reverse route stuff.
  * Now see whether we can send a route reply.
  */

 rt = rtable.rt_lookup(rq->rq_dst);

 // First check if I am the destination ..

 if(rq->rq_dst == index) {

#ifdef DEBUG
   fprintf(stderr, "%d - %s: destination sending reply\n",
                   index, __FUNCTION__);
#endif // DEBUG


   // Just to be safe, I use the max. Somebody may have
   // incremented the dst seqno.
   seqno = max(seqno, rq->rq_dst_seqno)+1;
   if (seqno%2) seqno++;

   sendReply(rq->rq_src,           // IP Destination
             1,                    // Hop Count
             index,                // Dest IP Address
             seqno,                // Dest Sequence Num
             MY_ROUTE_TIMEOUT,     // Lifetime
             rq->rq_timestamp);    // timestamp

   Packet::free(p);
 }

 // I am not the destination, but I may have a fresh enough route.

 else if (rt && (rt->rt_hops != INFINITY2) &&
           (rt->rt_seqno >= rq->rq_dst_seqno) ) {

   //assert (rt->rt_flags == RTF_UP);
   assert(rq->rq_dst == rt->rt_dst);
   //assert ((rt->rt_seqno%2) == 0);      // is the seqno even?
```

```
   sendReply(rq->rq_src,
             rt->rt_hops + 1,
             rq->rq_dst,
             rt->rt_seqno,
            (u_int32_t) (rt->rt_expire - CURRENT_TIME),
            //             rt->rt_expire - CURRENT_TIME,
             rq->rq_timestamp);
   // Insert nexthops to RREQ source and RREQ destination in the
   // precursor lists of destination and source respectively
   rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
   rt0->pc_insert(rt->rt_nexthop); // nexthop to RREQ destination

#ifdef RREQ_GRAT_RREP

   sendReply(rq->rq_dst,
             rq->rq_hop_count,
             rq->rq_src,
             rq->rq_src_seqno,
            (u_int32_t) (rt->rt_expire - CURRENT_TIME),
            //             rt->rt_expire - CURRENT_TIME,
             rq->rq_timestamp);
#endif

// TODO: send grat RREP to dst if G flag set in RREQ using rq-
>rq_src_seqno, rq->rq_hop_counT

// DONE: Included gratuitous replies to be sent as per IETF aodv draft
specification. As of now, G flag has not been dynamically used and is
always set or reset in aodv-packet.h --- Anant Utgikar, 09/16/02.

      Packet::free(p);
 }
 /*
  * Can't reply. So forward the  Route Request
  */
 else {
   ih->saddr() = index;
   ih->daddr() = IP_BROADCAST;
   rq->rq_hop_count += 1;
   // Maximum sequence number seen en route
   if (rt) rq->rq_dst_seqno = max(rt->rt_seqno, rq->rq_dst_seqno);
   forward((aodv_rt_entry*) 0, p, DELAY);
 }

}


Void AODVIDS::recvReply(Packet *p) {
//struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
aodv_rt_entry *rt;
char suppress_reply = 0;
double delay = 0.0;

#ifdef DEBUG
 fprintf(stderr, "%d - %s: received a REPLY\n", index, __FUNCTION__);
```

```
#endif // DEBUG


 /*
  *  Got a reply. So reset the "soft state" maintained for
  *  route requests in the request table. We don't really have
  *  have a separate request table. It is just a part of the
  *  routing table itself.
  */
 // Note that rp_dst is the dest of the data packets, not the
 // the dest of the reply, which is the src of the data packets.

 rt = rtable.rt_lookup(rp->rp_dst);

 /*
  *  If I don't have a rt entry to this host... adding
  */
 if(rt == 0) {
   rt = rtable.rt_add(rp->rp_dst);
 }

 /*
  * Add a forward route table entry... here I am following
  * Perkins-Royer AODV paper almost literally - SRD 5/99
  */

 if ( (rt->rt_seqno < rp->rp_dst_seqno) ||   // newer route
      ((rt->rt_seqno == rp->rp_dst_seqno) &&
       (rt->rt_hops > rp->rp_hop_count)) ) { // shorter or better route

  // Update the rt entry
  rt_update(rt, rp->rp_dst_seqno, rp->rp_hop_count,
            rp->rp_src, CURRENT_TIME + rp->rp_lifetime);

  // reset the soft state
  rt->rt_req_cnt = 0;
  rt->rt_req_timeout = 0.0;
  rt->rt_req_last_ttl = rp->rp_hop_count;

if (ih->daddr() == index) { // If I am the original source
  // Update the route discovery latency statistics
  // rp->rp_timestamp is the time of request origination

    rt->rt_disc_latency[rt->hist_indx] = (CURRENT_TIME - rp-
>rp_timestamp)
                                         / (double) rp->rp_hop_count;
    // increment indx for next time
    rt->hist_indx = (rt->hist_indx + 1) % MAX_HISTORY;
  }

  /*
   * Send all packets queued in the sendbuffer destined for
   * this destination.
   * XXX - observe the "second" use of p.
   */
  Packet *buf_pkt;
  while((buf_pkt = rqueue.deque(rt->rt_dst))) {
```

```
    if(rt->rt_hops != INFINITY2) {
          assert (rt->rt_flags == RTF_UP);
    // Delay them a little to help ARP. Otherwise ARP
    // may drop packets. -SRD 5/23/99
      forward(rt, buf_pkt, delay);
      delay += ARP_DELAY;
    }
  }
 }
 else {
  suppress_reply = 1;
 }

 /*
  * If reply is for me, discard it.
  */

if(ih->daddr() == index || suppress_reply) {
   Packet::free(p);
 }
 /*
  * Otherwise, forward the Route Reply.
  */
 else {
 // Find the rt entry
aodv_rt_entry *rt0 = rtable.rt_lookup(ih->daddr());
   // If the rt is up, forward
   if(rt0 && (rt0->rt_hops != INFINITY2)) {
        assert (rt0->rt_flags == RTF_UP);
     rp->rp_hop_count += 1;
     rp->rp_src = index;
     forward(rt0, p, NO_DELAY);
     // Insert the nexthop towards the RREQ source to
     // the precursor list of the RREQ destination
     rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source

   }
   else {
   // I don't know how to forward .. drop the reply.
#ifdef DEBUG
     fprintf(stderr, "%s: dropping Route Reply\n", __FUNCTION__);
#endif // DEBUG
     drop(p, DROP_RTR_NO_ROUTE);
   }
 }
}


Void AODVIDS::recvError(Packet *p) {
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_error *re = HDR_AODV_ERROR(p);
aodv_rt_entry *rt;
u_int8_t i;
Packet *rerr = Packet::alloc();
struct hdr_aodv_error *nre = HDR_AODV_ERROR(rerr);

 nre->DestCount = 0;
```

```
 for (i=0; i<re->DestCount; i++) {
 // For each unreachable destination
   rt = rtable.rt_lookup(re->unreachable_dst[i]);
   if ( rt && (rt->rt_hops != INFINITY2) &&
      (rt->rt_nexthop == ih->saddr()) &&
      (rt->rt_seqno <= re->unreachable_dst_seqno[i]) ) {
      assert(rt->rt_flags == RTF_UP);
      assert((rt->rt_seqno%2) == 0); // is the seqno even?
#ifdef DEBUG
     fprintf(stderr, "%s(%f): %d\t(%d\t%u\t%d)\t(%d\t%u\t%d)\n",
__FUNCTION__,CURRENT_TIME,
                 index, rt->rt_dst, rt->rt_seqno, rt->rt_nexthop,
                 re->unreachable_dst[i],re->unreachable_dst_seqno[i],
                   ih->saddr());
#endif // DEBUG
      rt->rt_seqno = re->unreachable_dst_seqno[i];
      rt_down(rt);

   // Not sure whether this is the right thing to do
   Packet *pkt;
      while((pkt = ifqueue->filter(ih->saddr()))) {
            drop(pkt, DROP_RTR_MAC_CALLBACK);
      }

     // if precursor list non-empty add to RERR and delete the precursor
list
      if (!rt->pc_empty()) {
            nre->unreachable_dst[nre->DestCount] = rt->rt_dst;
            nre->unreachable_dst_seqno[nre->DestCount] = rt->rt_seqno;
            nre->DestCount += 1;
            rt->pc_delete();
      }
   }
 }

 if (nre->DestCount > 0) {
#ifdef DEBUG
   fprintf(stderr, "%s(%f): %d\t sending RERR...\n", __FUNCTION__,
CURRENT_TIME, index);
#endif // DEBUG
   sendError(rerr);
 }
 else {
   Packet::free(rerr);
 }

 Packet::free(p);
}


/*
   Packet Transmission Routines
*/

Void AODVIDS::forward(aodv_rt_entry *rt, Packet *p, double delay) {
struct hdr_cmn *ch = HDR_CMN(p);
```

```
struct hdr_ip *ih = HDR_IP(p);

 if(ih->ttl_ == 0) {

#ifdef DEBUG
  fprintf(stderr, "%s: calling drop()\n", __PRETTY_FUNCTION__);
#endif // DEBUG

  drop(p, DROP_RTR_TTL);
  return;
 }

 if (rt) {
   assert(rt->rt_flags == RTF_UP);
   rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
   ch->next_hop_ = rt->rt_nexthop;
   ch->addr_type() = NS_AF_INET;
   ch->direction() = hdr_cmn::DOWN;       //important: change the
packet's direction
 }
 else { // if it is a broadcast packet
   assert(ch->ptype() == PT_AODV);
   assert(ih->daddr() == (nsaddr_t) IP_BROADCAST);
   ch->addr_type() = NS_AF_NONE;
   ch->direction() = hdr_cmn::DOWN;       //important: change the
packet's direction
 }

if (ih->daddr() == (nsaddr_t) IP_BROADCAST) {
 // If it is a broadcast packet
   assert(rt == 0);
   /*
    *  Jitter the sending of broadcast packets by 10ms
    */
   Scheduler::instance().schedule(target_, p,
                                  0.01 * Random::uniform());
 }
 else { // Not a broadcast packet
   if(delay > 0.0) {
     Scheduler::instance().schedule(target_, p, delay);
   }
   else {
   // Not a broadcast packet, no delay, send immediately
     Scheduler::instance().schedule(target_, p, 0.);
   }
 }

}


Void AODVIDS::sendRequest(nsaddr_t dst) {
// Allocate a RREQ packet
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
aodv_rt_entry *rt = rtable.rt_lookup(dst);
```

```
 assert(rt);

 /*
  *  Rate limit sending of Route Requests. We are very conservative
  *  about sending out route requests.
  */

 if (rt->rt_flags == RTF_UP) {
   assert(rt->rt_hops != INFINITY2);
   Packet::free((Packet *)p);
   return;
 }

 if (rt->rt_req_timeout > CURRENT_TIME) {
   Packet::free((Packet *)p);
   return;
 }

 // rt_req_cnt is the no. of times we did network-wide broadcast
 // RREQ_RETRIES is the maximum number we will allow before
 // going to a long timeout.

 if (rt->rt_req_cnt > RREQ_RETRIES) {
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
   rt->rt_req_cnt = 0;
 Packet *buf_pkt;
   while ((buf_pkt = rqueue.deque(rt->rt_dst))) {
       drop(buf_pkt, DROP_RTR_NO_ROUTE);
   }
   Packet::free((Packet *)p);
   return;
 }
#ifdef DEBUG
   fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
                   ++route_request, index, rt->rt_dst);
#endif // DEBUG

 // Determine the TTL to be used this time.
 // Dynamic TTL evaluation - SRD

 rt->rt_req_last_ttl = max(rt->rt_req_last_ttl,rt->rt_last_hop_count);

 if (0 == rt->rt_req_last_ttl) {
 // first time query broadcast
   ih->ttl_ = TTL_START;
 }
 else {
 // Expanding ring search.
   if (rt->rt_req_last_ttl < TTL_THRESHOLD)
     ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
   else {
   // network-wide broadcast
     ih->ttl_ = NETWORK_DIAMETER;
     rt->rt_req_cnt += 1;
   }
 }
```

```
 // remember the TTL used  for the next time
 rt->rt_req_last_ttl = ih->ttl_;

 // PerHopTime is the roundtrip time per hop for route requests.
 // The factor 2.0 is just to be safe .. SRD 5/22/99
 // Also note that we are making timeouts to be larger if we have
 // done network wide broadcast before.

 rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime(rt);
 if (rt->rt_req_cnt > 0)
   rt->rt_req_timeout *= rt->rt_req_cnt;
 rt->rt_req_timeout += CURRENT_TIME;

 // Don't let the timeout to be too large, however .. SRD 6/8/99
 if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
   rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
 rt->rt_expire = 0;

#ifdef DEBUG
 fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f
ms\n",
              ++route_request,
             index, rt->rt_dst,
             rt->rt_req_timeout - CURRENT_TIME);
#endif      // DEBUG


 // Fill out the RREQ packet
 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rq->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->prev_hop_ = index;          // AODV hack

 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;

 // Fill up some more fields.
 rq->rq_type = AODVTYPE_RREQ;
 rq->rq_hop_count = 1;
 rq->rq_bcast_id = bid++;
 rq->rq_dst = dst;
 rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
 rq->rq_src = index;
 seqno += 2;
 assert ((seqno%2) == 0);
 rq->rq_src_seqno = seqno;
 rq->rq_timestamp = CURRENT_TIME;

 Scheduler::instance().schedule(target_, p, 0.);

}
```

```
Void AODVIDS::sendReply(nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t
rpdst,
                u_int32_t rpseq, u_int32_t lifetime, double timestamp) {
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
aodv_rt_entry *rt = rtable.rt_lookup(ipdst);

#ifdef DEBUG
fprintf(stderr, "sending Reply from %d at %.2f\n", index,
Scheduler::instance().clock());
#endif // DEBUG
 assert(rt);

 rp->rp_type = AODVTYPE_RREP;
 //rp->rp_flags = 0x00;
 rp->rp_hop_count = hop_count;
 rp->rp_dst = rpdst;
 rp->rp_dst_seqno = rpseq;
 rp->rp_src = index;
 rp->rp_lifetime = lifetime;
 rp->rp_timestamp = timestamp;

 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rp->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_INET;
 ch->next_hop_ = rt->rt_nexthop;
 ch->prev_hop_ = index;          // AODV hack
 ch->direction() = hdr_cmn::DOWN;

 ih->saddr() = index;
 ih->daddr() = ipdst;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = NETWORK_DIAMETER;

 Scheduler::instance().schedule(target_, p, 0.);

}

Void AODVIDS::sendError(Packet *p, bool jitter) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_error *re = HDR_AODV_ERROR(p);

#ifdef ERROR
fprintf(stderr, "sending Error from %d at %.2f\n", index,
Scheduler::instance().clock());
#endif // DEBUG

 re->re_type = AODVTYPE_RERR;
 //re->reserved[0] = 0x00; re->reserved[1] = 0x00;
```

```
 // DestCount and list of unreachable destinations are already filled

 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + re->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->next_hop_ = 0;
 ch->prev_hop_ = index;          // AODV hack
 ch->direction() = hdr_cmn::DOWN;       //important: change the packet's
direction

 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = 1;

 // Do we need any jitter? Yes
 if (jitter)
      Scheduler::instance().schedule(target_, p, 0.01*Random::uniform());
 else
      Scheduler::instance().schedule(target_, p, 0.0);

}


/*
   Neighbor Management Functions
*/

Void AODVIDS::sendHello() {
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rh = HDR_AODV_REPLY(p);

#ifdef DEBUG
fprintf(stderr, "sending Hello from %d at %.2f\n", index,
Scheduler::instance().clock());
#endif // DEBUG

 rh->rp_type = AODVTYPE_HELLO;
 //rh->rp_flags = 0x00;
 rh->rp_hop_count = 1;
 rh->rp_dst = index;
 rh->rp_dst_seqno = seqno;
 rh->rp_lifetime = (1 + ALLOWED_HELLO_LOSS) * HELLO_INTERVAL;

 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + rh->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->prev_hop_ = index;          // AODV hack
```

```
 ih->saddr() = index;
 ih->daddr() = IP_BROADCAST;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = 1;

 Scheduler::instance().schedule(target_, p, 0.0);
}


Void AODVIDS::recvHello(Packet *p) {
//struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
AODV_Neighbor *nb;

 nb = nb_lookup(rp->rp_dst);
 if(nb == 0) {
   nb_insert(rp->rp_dst);
 }
 else {
   nb->nb_expire = CURRENT_TIME +
                   (1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
 }

 Packet::free(p);
}

Void AODVIDS::nb_insert(nsaddr_t id) {
AODV_Neighbor *nb = new AODV_Neighbor(id);

 assert(nb);
 nb->nb_expire = CURRENT_TIME +
                 (1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
 LIST_INSERT_HEAD(&nbhead, nb, nb_link);
 seqno += 2;             // set of neighbors changed
 assert ((seqno%2) == 0);
}

AODV_Neighbor*
AODVIDS::nb_lookup(nsaddr_t id) {
AODV_Neighbor *nb = nbhead.lh_first;

 for(; nb; nb = nb->nb_link.le_next) {
   if(nb->nb_addr == id) break;
 }
 return nb;
}

/*
 * Called when we receive *explicit* notification that a Neighbor
 * is no longer reachable.
 */
Void AODVIDS::nb_delete(nsaddr_t id) {
AODV_Neighbor *nb = nbhead.lh_first;

 log_link_del(id);
```

```
 seqno += 2;     // Set of neighbors changed
 assert ((seqno%2) == 0);

 for(; nb; nb = nb->nb_link.le_next) {
   if(nb->nb_addr == id) {
     LIST_REMOVE(nb,nb_link);
     delete nb;
     break;
   }
 }

 handle_link_failure(id);

}
/*
 * Purges all timed-out Neighbor Entries - runs every
 * HELLO_INTERVAL * 1.5 seconds.
 */
Void AODVIDS::nb_purge() {
AODV_Neighbor *nb = nbhead.lh_first;
AODV_Neighbor *nbn;
double now = CURRENT_TIME;

 for(; nb; nb = nbn) {
   nbn = nb->nb_link.le_next;
   if(nb->nb_expire <= now) {
     nb_delete(nb->nb_addr);
   }
 }

}
```

Appendix III NS-2 Simulation Scripts

```
# resource_consumption_attack.tcl
#
# simulation of wireless AODV MANET Routing Protocol
# with 20 nodes,
#
#
# ======================================================================
# Define options
# ======================================================================

set val(chan)       Channel/WirelessChannel
set val(prop)       Propagation/TwoRayGround
set val(netif)      Phy/WirelessPhy
set val(mac)        Mac/802_11
set val(ifq)        Queue/DropTail/PriQueue
set val(ll)         LL
set val(ant)        Antenna/OmniAntenna
set val(x)              1000   ;# X dimension of the topography
set val(y)              1000   ;# Y dimension of the topography
set val(ifqlen)         50            ;# max packet in ifq
set val(seed)           1.0
set val(adhocRouting)   AODVIDS
set val(adhocRouting2)   MYAODV
set val(nn)             20            ;# how many nodes are simulated
set val(cp)             "cbr-20-1.0-20-10"
set val(sc)             "scen-20-2.0-0.0"
set val(stop)           100.0           ;# simulation time

# ======================================================================
# Main Program
# ======================================================================

#
# Initialize Global Variables
#

# create simulator instance

set ns_           [new Simulator]

# setup topography object

set topo    [new Topography]

# create trace object for ns and nam

set tracefd [open 20.tr w]
set namtrace    [open 20.nam w]

$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $val(x) $val(y)

# define topology
```

```
$topo load_flatgrid $val(x) $val(y)

#
# Create God
#
set god_ [create-god $val(nn)]

#
# define how node should be created
#

#global node setting

$ns_ node-config -adhocRouting $val(adhocRouting) \
                 -llType $val(ll) \
                 -macType $val(mac) \
                 -ifqType $val(ifq) \
                 -ifqLen $val(ifqlen) \
                 -antType $val(ant) \
                 -propType $val(prop) \
                 -phyType $val(netif) \
                 -channelType $val(chan) \
             -topoInstance $topo \
#            -energyModel EnergyModel \
#            -initialEnergy 100.0 \
#            -rxPower 0.3 \
#            -txPower 0.6 \
             -agentTrace ON \
                 -routerTrace ON \
                 -macTrace OFF

#
#  Create the specified number of nodes [$val(nn)] and "attach" them
#  to the channel.

set node_(0) [$ns_ node]
set node_(1) [$ns_ node]

#$ns_ node-config -adhocRouting $val(adhocRouting2)
for {set i 2} {$i < 3 } {incr i} {
      set node_($i) [$ns_ node]
      $node_($i) random-motion 0          ;# disable random motion
}

$ns_ node-config -adhocRouting $val(adhocRouting)
for {set i 3} {$i < $val(nn) } {incr i} {
      set node_($i) [$ns_ node]
      $node_($i) random-motion 0          ;# disable random motion
}

#
# Define node movement model
#
puts "Loading connection pattern..."
source $val(cp)

#
```

```
# Define traffic model
#
puts "Loading scenario file..."
source $val(sc)

#node 2 attached to a attack agent
set ragent_(2) [new Agent/MYAODV 2]
set ragent $ragent_(2)
$node_(2) attach $ragent 255

#Send fake RREQ messages to disrupt the
# route from node 4 to node 5 in time interval 0.05 second.
$ns_ at 6.00 "$ragent rreqs_rd 4 5 $val(stop) 0.05"



# Define node initial position in nam

for {set i 0} {$i < $val(nn) } {incr i} {

    # 20 defines the node size in nam, must adjust it according to your
scenario
    # The function must be called after mobility model is defined

    $ns_ initial_node_pos $node_($i) 10
}


#
# Tell nodes when the simulation ends
#
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at $val(stop).0 "$node_($i) reset";
}

$ns_ at  $val(stop).0002 "puts \"NS EXITING...\" ; $ns_ halt"

puts $tracefd "M 0.0 nn $val(nn) x $val(x) y $val(y) rp
$val(adhocRouting)"
puts $tracefd "M 0.0 sc $val(sc) cp $val(cp) seed $val(seed)"
puts $tracefd "M 0.0 prop $val(prop) ant $val(ant)"

puts "Starting Simulation..."
$ns_ run
```

Bibliography

[1] Ioanna Stamouli, Real-time Intrusion Detection for Ad hoc Networks, Proceedings of the Sixth IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks, 2005

[2] Zhang, Y. and Lee, W., "Intrusion Detection in Wireless Ad-Hoc Networks," In Proceedings of theSixth Annual International Conference on Mobile Communication and Networking, 2000.

[3] C. Perkins, E. Royer and S. Das, "AODV routing", Internet Draft draftietf-manet-aodv-13.txt, Feb 2003.

[4] Kimaya Sanzgiri, Bridget Dahill, Brian Neil Levine, Clay Shields, Elizabeth M. Belding-Royer. A Secure Routing Protocol for Ad Hoc Networks, In Proceedings of 2002 IEEE International Conference on Network Protocols (ICNP). November 2002.

[5] P. Ning and K. Sun, \How to misuse AODV: A case study of insider attacks against mobile ad-hoc routing protocols," Tech. Rep. TR-2003-07, CS Department, NC State University, 2003.

[6] Kwan-Wu Chin, John Judge, Aidan Williams and Roger Kermode, "Implementation Experience with MANET Routing Protocols," *ACM SIGCOMM Computer Communication Review*, Volume 32, Issue 5, November 2002

[7] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Speci_cation-based anomaly detection: A new approach for detecting network intrusions. In Proceedings of the ACM Computer and Communication Security Conference (CCS'02), 2002.

[8] Manel Guerrero Zapata, Secure Ad hoc On-Demand Distance Vector Routing, ACM Mobile Computing and Communications Review (MC2R), Vol 6. No. 3, July 2002.

[9] Yi-an Huang and Wenke Lee, Attack Analysis and Detection for Ad Hoc Routing Protocols, http://www.ece.cmu.edu/~adrian/731-sp05/readings/Huang-Lee-Ad-Hoc-Net-IDS.pdf

[10] C. E. Perkins, E. M. Royer, S. R. Das, and M. K. Marina. Performance comparison of two on-demand routing protocols for ad hoc networks. Feb, 2001. ieeeinfocom.org/2000/papers/673.ps

[11] Huang, Yi-an and Wenke Lee. Attack Analysis and Detection for Ad Hoc Routing Protocols. In Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID'04), French Riviera, France. September 2004.

[12] C. Perkins, E. Royer and S. Das, "Ad hoc On-demand Distance Vector (AODV)", *RFC 3561*, 2003.

[13] Tseng, Chin-Yang, et al. A Specification-based Intrusion Detection System for AODV., In Proceedings of the 1st ACM Workshop on Security of Ad hoc and Sensor Networks (SASN'03). Fairfax, VA. 2003.

[14] Giovanni Vigna Sumit Gwalani Kavitha Srinivasan Elizabeth M. Belding-Royer Richard A. Kemmerer, An Intrusion Detection Tool forAODV-based Ad hoc Wireless Networks, 20th Annual Computer Security Applications Conference, December 2004

[15] Zhou, L. and Haas Z.,"Securing Ad Hoc Networks," *IEEE Network Magazine*, vol. 13, no. 6, November/December 1999.

[16] Zhang, Y., Lee, W. and Huang Y. A., Intrusion detection techniques for mobile wireless networks. Wireless Networks, Volume 9 Issue 5, September 2003

[17] Amit Jain, ROUTING PROTOCOLS FOR MOBILE AD-HOC NETWORKS, cse.iitk.ac.in/users/dhee...manet.ps.gz

[18] PoWah Yau and Chris J. Mitchell, Security Vulnerabilities in Ad Hoc Networks, http://www.cs.ucsb.edu/~ravenben/papers/sensors/security-adhoc-ucl.pdf

[19] The network simulator NS-2, NS-2 Manual, http://www.isi.edu./nsnam/ns

[20] Brutch, P., Ko, C. "Challenges in Intrusion Detection for Wireless Ad-hoc Networks". Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on , Jan. 2003

DECLARATION

I, the undersigned, hereby declare that this thesis is my original work carried out under the supervision of Dr Fisseha Mekuria, has not been presented as a thesis for a degree program in any other university and that all sources of materials used for the thesis are duly acknowledged.

Konjit Desalegn