



Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num. SP06095 Call Num:

General Guidelines:

- ◆ Library books should be used with great care.
- ◆ Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- ◆ Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- ◆ The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT



SP06095

TO IMPLEMENT CLUSTERING IN MOBILE AD-HOC NETWORKS

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

in

Information Technology

By

Pranshu Gupta (061439)

Nishant Shekhar(061461)

under the supervision of

Mr. Amol Vasudeva

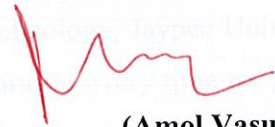


Jaypee University of Information Technology
Waknaghat, Solan - 173215, Himachal Pradesh

Certificate

This is to certify that the project entitled "To implement clustering in Mobile ad-hoc networks", submitted by Pranshu Gupta(061439) and Nishant Shekhar(061461) in partial fulfillment for the award of degree of Bachelor of Technology in Information Technology to Jaypee University of Information Technology, Wagnaghat, Solan has been carried out under my supervision.


Date: 13 May 2010



(Amol Vasudeva)

Lecturer(CSE Deptt.)

Certified that this work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.



1. Pranshu Gupta

(061439)



2. Nishant Shekhar

(061461)

Acknowledgement

Apart from the efforts by us, the success of this project depends largely on encouragement and guidelines of many others. We take this opportunity to express our gratitude to the people who have been instrumental in the successful completion of this project.

We would like to show our greatest appreciation to our supervisor Mr. Amol Vasudeva, Lecturer, Deptt. Of Computer Science and Technology, Jaypee University of Information Technology. We feel motivated and encouraged every time we get his encouragement. For his coherent guidance throughout the tenure of the project, we feel fortunate to be taught by him, who gave us his unwavering support. Besides being our mentor, he has taught us that there's no substitute for hard work. Being a dynamic personality himself, he has a practical approach towards a profession.

Finally, thanks to all our family members who supported us in our every grim phase. Our work will remain unaccomplished if we don't pay our gratitude to our parents for their constant encouragement and support. We thank them for the sacrifices they made so that we could grow up in a learning environment. They have always stood by us in everything fruitful we have done, providing constant support, encouragement and love.

13/05/2010

Pranshu Gupta-061439

Nishant Shekhar-061461

ABSTRACT

Mobile ad hoc network (MANET) is an autonomous system of mobile nodes connected by wireless links. Each node operates not only as an end system, but also as a router to forward packets. The nodes are free to move about and organize themselves into a network. Due to mobility and other constraints such as restricted power and processing capacity, nodes cannot run heavy applications to detect intrusions. If every node starts monitoring intrusions separately, processing overhead at each node will consume a large portion of their battery and power. The clustering approach can be taken as an additional advantage in these processing constrained networks to collaboratively detect intrusions with less power usage and minimal overhead. If the clusters are regularly changed due to routes, the intrusion detection will not prove to be effective.

As far as the simulation of a network is concerned ns-2 is one of the best discrete event simulators available. It covers a very large number of applications, protocols, network types, network elements and traffic models. NS is based on two languages: an object oriented simulator, written in C++, and a OTcl(an object oriented extension of Tcl) interpreter, used to execute user's command scripts.

In this work an attempt has been made to learn the working of different features present in ns-2, to study mobile ad-hoc networks, clustering in mobile ad-hoc networks, Sybil attack and to compare the performance of some prominent routing protocols for MANETs in ns-2 and to implement cluster based routing protocol under different scenarios.

TABLE OF CONTENTS

CERTIFICATE.....	i
ACKNOWLEDGEMENT.....	ii
ABSTRACT.....	iii
TABLE OF CONTENTS.....	iv

S. No.	Chapter Name	Page
1	Introduction and Statement of the Problem	1
	1.1 Introduction.....	1
	1.2 Statement of the Problem.....	2
	1.3 Organization of the Report.....	3
	1.4 Abbreviations.....	4
2	Background and Literature Review	6
	2.1 Introduction to MANET.....	6
	2.2 Characteristics of Mobile Ad-hoc Networks.....	8
	2.2.1 Constraints in MANET.....	9
	2.3 Advantages of Mobile Ad-hoc Networks.....	12
	2.4 Disadvantages of MANET.....	15
	2.5 Applications of MANET.....	15
	2.6 Concerns related to MANET.....	16
3	Introduction to NS-2	19
	3.1 The Basics.....	19
	3.2 Writing TCL Scripts.....	23
	3.2.1 2 Nodes Topology.....	23
	3.2.2 7 Nodes Scenario.....	25
	3.3 XGRAPH.....	28
	3.4 Running Wireless Simulations.....	30
	3.4.1 Scenario File.....	30
	3.4.2 Communication Pattern File.....	31
	3.4.3 Router Configuration File.....	32
4	Routing	35

4.1	Definition.....	35
4.2	Routing in MANET.....	35
4.3	Problems with routing in MANET.....	36
5	Clustering Approach	39
5.1	Clustering.....	40
5.2	Applications of Clustering.....	41
5.2.1	Limitations.....	41
5.3	Cluster Based Routing Protocol (CBRP).....	43
5.3.1	Election Process.....	43
5.3.1.1	Node Status.....	44
5.3.1.2	Data Structure.....	45
5.3.1.3	HELLO Messages.....	46
6	Conclusions & Future Work	49
6.1	Conclusion.....	49
6.2	Suggestions for Future Work.....	51
	References	118

CHAPTER 1

INTRODUCTION AND STATEMENT OF THE PROBLEM

1.1 Introduction

Network Security is of paramount interest in active research field of mobile ad hoc networks. Ad hoc network is a collection of mobile nodes that has no fixed infrastructure to support communications between them, rather is characterized by wireless multi-hop communications. The ad hoc network is vulnerable to attack due to wireless communication link, dynamic change in topology, cooperative nature for ad hoc routing algorithms, lack of centralized authority and lack of proper line of defense. Therefore, traditional security schemes cannot protect the nodes from all types of attacks in mobile ad hoc networks. In many cases, the protocols devised to defend against the attacks have been either extended or modified, even new protocols also have been proposed. These security measures of protocols are often referred to as the network's first line of defense. Different types of intrusion prevention techniques used in ad hoc networks, such as encryption and authentication, may reduce intrusion, but cannot eliminate them completely.

Literatures in the field of network security reveal the fact that whatever may be intrusion prevention measure adopted, there is some probability of intrusion exploiting some weak link. Hence, a second line of defense is required to make the network survived which may be offered by the intrusion detection techniques.

Most of the secured ad hoc routing protocols assume that each node represents unique address and mainly based on threshold security schemes or reputation schemes, which yield good result in case limited number of attackers in the network. However, the broadcast nature of this network allows a single node to pretend to be many nodes simultaneously by using different addresses while transmitting. This

attack is known as Sybil attack and most of the existing protocols fail to defend against this type of attack.

According to Douceur, the Sybil attack is an attack by which a single entity can control a fraction of the system by presenting multiple identities. Through Sybil attack a malicious node may present one or more fake identities to other nodes in the network and undermine the operation of collaborative tasks on peer-to-peer systems and other distributed systems.

1.2 Statement of the Problem

This report presents techniques adopted to defend against Sybil Attack in Mobile Ad Hoc Networks and implementing Cluster Based Routing Protocol(CBRP). Mobility is often a problem for providing security services in ad hoc networks. In this scheme, we have used mobility as a feature to enhance security. In the mobile environment a single entity impersonating multiple identities has an important constraint that can be detected. Because all identities are part of the same physical device, they must move in unison, while independent nodes are free to move at will. As nodes move geographically, all the Sybil identities will appear or disappear simultaneously as the attacker moves in and out of range.

Due to mobility and other constraints such as restricted power and processing capacity, nodes cannot run heavy applications to detect intrusions. If every node starts monitoring intrusions separately, processing overhead at each node will consume a large portion of their battery and power. Therefore, a scalable and fault tolerant IDS is required to govern these on-secure wireless ad-hoc networks against attacks. The efficient solution is to defend against intrusion co-operatively, rather than each mobile node performing full analysis of traffic passing through it. In order to cooperate, the nodes must trust on each other so that they should not audit all the data, and hence, they can save a lot more processing and memory overhead.

The clustering approach can be taken as an additional advantage in these processing constrained networks to collaboratively detect intrusions with less power usage and minimal overhead. Existing clustering protocols are not suitable for

intrusion detection purposes because they are linked with the routes. The route establishment and route renewal affects the clusters and as a consequence, the processing and traffic overhead increases due to instability of clusters. The ad-hoc networks are battery and power constraint, and therefore a trusted monitoring node should be available to detect and respond against intrusions in time. This can be achieved only if the clusters are stable for a long period of time. If the clusters are regularly changed due to routes, the intrusion detection will not prove to be effective. Therefore, a generalized clustering algorithm has been proposed that can run on top of any routing protocol and can monitor the intrusions constantly irrespective of the routes. However, this clustered approach can be applied to any type of Intrusion, but we have given emphasis only on the detection of Sybil attack, with higher accuracy and low memory and processing overheads.

1.3 Organization of the Report

The report is divided into six chapters including this introductory chapter. The rest of this thesis is organized as follows:

Chapter 2 gives an overview of the Mobile Ad hoc networks, its characteristics, advantages and disadvantages, applications, design issues and constraints and concerns.

Chapter 3 describes the software platform or the Simulator used, i.e., The Network Simulator also known as NS-2. It describes the basics of NS-2, how to write simple wired TCL scripts and how to simulate them and view the output in NAM(Network Animator). It also gives overview about the graph plotting tool present in NS-2, the XGRAPH, needed to plot different types of graphs to analyze different scenarios and then finally running wireless simulations.

Chapter 4 It describes Routing, its definition, various scenarios, what exactly is routing and what are the complications related to routing. Then routing in mobile ad-hoc networks is discussed and how it is different from general routing or routing in other networks. Finally problems related to routing in mobile ad-hoc networks.

Chapter 5 gives the overview of Clustering, Clustered Approach in the detection of intrusions in an ad-hoc network and cluster based routing protocol.

Chapter 6 concludes the dissertation and gives some suggestions for the future work.

1.4 Abbreviations Used

AODV	Ad-hoc On-Demand Distance Vector
CA	Certificate Authority
CAT	Cluster Adjacency Table
CBRP	Cluster Based Routing Protocol
DoS	Denial of Service
DSDV	Destination Sequenced Distance Vector
DSR	Dynamic Source Routing
ED	Election Done
EH	Election Head
ES	Election Start
EV	Election Vote
GPS	Global Positioning System
GW	Gateway Node
HD	Head Node
ID	Identification
IDS	Intrusion Detection System
IP	Internet Protocol
LANs	Local Area Networks
MAC	Medium Access Channel
MANET	Mobile Ad hoc Network
MB	Member Node
MT	Member Table
PAN	Personal Area Network
PDA	Personal Digital Assistant
PKI	Public Key Infrastructure

PRB	Probabilistic Routing Protocol
RREQ	Route Request
QoS	Quality of Service
TCP	Transmission Control Protocol
UD	Undecided Node
WLAN	Wireless Local Area Network

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

2.1 Introduction to Mobile Ad-hoc Networks (MANET)

Wireless cellular systems have been in use since 1980s. We have seen their evolutions to first, second and third generation's wireless systems. These systems work with the support of a centralized supporting structure such as an access point. The wireless users can be connected with the wireless system by the help of these access points, when they roam from one place to the other.

The adaptability of wireless systems is limited by the presence of a fixed supporting coordinate. It means that the technology cannot work efficiently in that places where there is no permanent infrastructure. Easy and fast deployment of wireless networks will be expected by the future generation wireless systems. This fast network deployment is not possible with the existing structure of present wireless systems

Recent advancements such as Bluetooth introduced a fresh type of wireless systems which is frequently known as mobile ad-hoc networks. Mobile ad-hoc networks or "short live" networks control in the nonexistence of permanent infrastructure. Mobile ad hoc network offers quick and horizontal network deployment in conditions where it is not possible otherwise. Ad-hoc is a Latin word, which means "for this or for this only." Mobile ad hoc network is an autonomous system of mobile nodes connected by wireless links; each node operates as an end system and a router for all other nodes in the network.

A wireless network is a growing new technology that will allow users to access services and information electronically, irrespective of their geographic position. Wireless networks can be classified in two types: - infrastructured network and infrastructure less (ad hoc) networks. Infrastructured network consists of a

network with fixed and wired gateways. A mobile host interacts with a bridge in the network (called base station) within its communication radius. The mobile unit can move geographically while it is communicating. When it goes out of range of one base station, it connects with new base station and starts communicating through it. This is called handoff. In this approach the base stations are fixed.

A Mobile ad hoc network is a group of wireless mobile computers (or nodes); in which nodes collaborate by forwarding packets for each other to allow them to communicate outside range of direct wireless transmission. Ad hoc networks require no centralized administration or fixed network infrastructure such as base stations or access points, and can be quickly and inexpensively set up as needed

A MANET is an autonomous group of mobile users that communicate over reasonably slow wireless links. The network topology may vary rapidly and unpredictably over time, because the nodes are mobile. The network is decentralized, where all network activity, including discovering the topology and delivering messages must be executed by the nodes themselves. Hence routing functionality will have to be incorporated into the mobile nodes.

MANET is a kind of wireless ad-hoc network and it is a self-configuring network of mobile routers (and associated hosts) connected by wireless links – the union of which forms an arbitrary topology. The routers, the participating nodes act as router, are free to move randomly and manage themselves arbitrarily; thus, the network's wireless topology may change rapidly and unpredictably. Such a network may operate in a standalone fashion, or may be connected to the larger Internet.

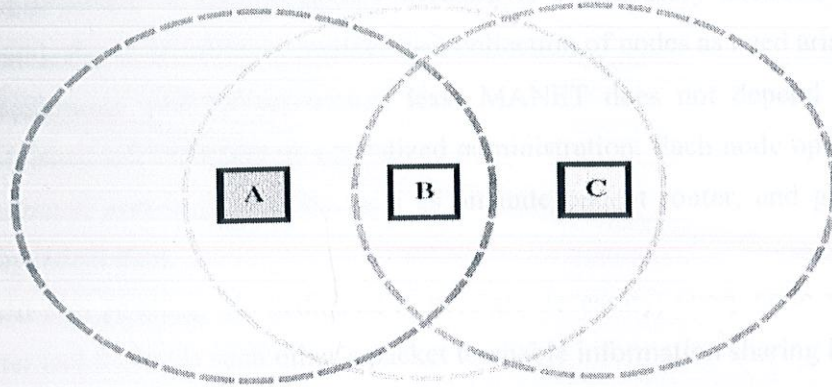


Figure 2.1 Example of a simple ad-hoc network with three participating nodes

Mobile ad hoc network is a collection of independent mobile nodes that can communicate to each other via radio waves. The mobile nodes can directly communicate to those nodes that are in radio range of each other, whereas other nodes need the help of intermediate nodes to route their packets. These networks are fully distributed, and can work at any place without the aid of any infrastructure. This property makes these networks highly robust.

2.2 Characteristics of Ad hoc networks

Mobile ad hoc network nodes are furnished with wireless transmitters and receivers using antennas, which may be highly directional (point-to-point), omnidirectional (broadcast), probably steerable, or some combination thereof. At a given point in time, depending on positions of nodes, their transmitter and receiver coverage patterns, communication power levels and co-channel interference levels, a wireless connectivity in the form of a random, multi-hop graph or "ad hoc" network exists among the nodes. This ad hoc topology may modify with time as the nodes move or adjust their transmission and reception parameters.

MANETs inherit common characteristics found in wireless networks in general, and add characteristics specific to ad hoc networking:

- **Wireless:** Nodes communicate wirelessly and share the same media (radio, infrared, etc.).

- **Ad-hoc-based:** A mobile ad hoc network is a temporary network formed dynamically in an arbitrary manner by a collection of nodes as need arises.
- **Autonomous and infrastructure less:** MANET does not depend on any established infrastructure or centralized administration. Each node operates in distributed peer-to-peer mode, acts as an independent router, and generates independent data.
- **Multi-hop routing:** No dedicated routers are necessary; every node acts as a router and forwards each other's packet to enable information sharing between mobile hosts.
- **Mobility:** Each node is free to move about while communicating with other nodes. The topology of such an ad hoc network is dynamic in nature due to constant movement of the participating nodes, causing the intercommunication patterns among nodes to change continuously.

2.2.1 Constraints in Mobile Ad-hoc Networks

As described in the previous section, the ad hoc architecture has many benefits, such as self-reconfiguration, ease of deployment, and so on. However, this flexibility and convenience come at a price. Ad hoc wireless networks inherit the traditional problems of wireless communications, such as bandwidth optimization, power control, and transmission quality enhancement, while, in addition, their mobility, multi-hop nature, and the lack of fixed infrastructure create a number of complexities and design constraints that are new to mobile ad hoc networks, as discussed in the following subsections:

- **They are infrastructure less:** Mobile ad hoc networks are multi-hop infrastructure less wireless networks. This lack of fixed infrastructure in addition to being wireless, generate new design issues compared with fixed networks. Also, lack of a centralized entity means network management has to be distributed across different nodes, which brings added difficulty in fault detection and management.

- **Dynamically Changing Network Topologies:** In mobile ad hoc networks, since nodes can move arbitrarily, the network topology, which is typically multi-hop, can change frequently and unpredictably, resulting in route changes, frequent network partitions, and possibly, packet losses.
- **Physical Layer Limitation:** The radio interface at each node uses broadcasting for transmitting traffic and usually has limited wireless transmission range, resulting in specific mobile ad hoc network problems like hidden terminal problems, exposed terminal problem, and so on. Collisions are inherent to the medium, and there is a higher probability of packet losses due to transmission errors compared to wired networks.
- **Limited Link Bandwidth and Quality:** Since mobile nodes communicate with each other via bandwidth-constrained, variable capacity, error-prone, and insecure wireless channels, wireless links will continue to have significantly lower capacity than wired links and, hence, congestion is more problematic.
- **Variation in Link and Node Capabilities:** Each node may be equipped with one or more radio interfaces that have varying transmission/receiving capabilities and operate across different frequency bands. This heterogeneity in node radio capabilities can result in possible asymmetric links. In addition, each mobile node might have different software/hardware configuration, resulting in variability in processing capabilities. Designing network protocols and algorithms for this heterogeneous network can be complex, requiring dynamic adaptation to the changing power and channel conditions, traffic load/distribution variations, load balancing, congestion, and service environments.
- **Energy Constrained Operation:** Because batteries carried by each mobile node have limited power, processing power is limited, which in turn limits services and applications that can be supported by each node. This becomes a bigger issue in mobile ad hoc networks because as each node is acting as both an end system and a router at the same time, additional energy is required to forward packets from other nodes.

- **Network Robustness and Reliability:** In MANET, network connectivity is obtained by routing and forwarding among nodes. Although this replaces the constraints of fixed infrastructure connectivity, it also brings design challenges. Due to various conditions like overload, acting selfishly, or having broken links, a node may fail to forward the packet. Misbehaving nodes and unreliable links can have a severe impact on overall network performance. Lack of centralized monitoring and management points means these types of misbehaviors cannot be detected and isolated quickly and easily, adding significant complexity to protocol design.
- **Network Scalability:** Current popular network management algorithms were mostly designed to work on fixed or relatively small wireless networks. Many mobile ad hoc network applications involve large networks with tens of thousands of nodes, as found, for example, in sensor networks and tactical networks. Scalability is critical to the successful deployment of such networks. The evolution toward a large network consisting of nodes with limited resources is not straightforward and presents many challenges that are still to be solved in areas such as addressing, routing, location management, configuration management, interoperability, security, high-capacity wireless technologies, and so on.
- **Quality of Service:** A quality of service (QoS) guarantee is essential for successful delivery of multimedia network traffic. QoS requirements typically refer to a wide set of metrics including throughput, packet loss, delay, jitter, error rate, and so on. Wireless and mobile ad hoc specific network characteristics and constraints described above, such as dynamically changing network topologies, limited link bandwidth and quality, variation in link and node capabilities, pose extra difficulty in achieving the required QoS guarantee in a mobile ad hoc network.
- **Network Security:** Mobile wireless networks are generally more vulnerable to information and physical security threats than fixed-wire line networks. The use of open and shared broadcast wireless channels means nodes with inadequate physical protection are prone to security threats. In addition,

because a mobile ad hoc network is a distributed infrastructure less network, it mainly relies on individual security solutions from each mobile node, as centralized security control is hard to implement. Some key security requirements in ad hoc networking include:

Confidentiality: preventing passive eavesdropping.

Access control: protecting access to wireless network infrastructure.

Data integrity: preventing tampering with traffic (i.e., accessing, modifying or injecting traffic).

2.3 Advantages of MANET

Ad-hoc networks have several advantages compared to traditional cellular systems. The advantages include:

- **On demand setup:** These networks can be setup at any time and at any place.
- **Unconstrained connectivity:** They provide access to information and services regardless of geographic positions.

A wireless ad-hoc network is a collection of mobile/semi-mobile nodes with no pre-established infrastructure forming a temporary network. Each of the nodes has a wireless interface and communicates with each other over either radio or infrared. Laptop computers and personal digital assistants that communicate directly with each other are some examples of nodes in an ad-hoc network. Nodes in the ad-hoc network are often mobile, but can also consist of stationary nodes, such as access points to the Internet. Semi mobile nodes can be used to deploy relay points in areas where relay points might be needed temporarily.

In general, mobile ad hoc networks are formed dynamically by an autonomous system of mobile nodes that are connected via wireless links without using an existing network infrastructure or centralized administration. The nodes are free to move randomly and organize themselves arbitrarily; thus, the network's wireless topology may change rapidly and unpredictably. Such a network may operate in a standalone fashion, or may be connected to the larger Internet. Mobile ad hoc networks are

infrastructure less networks since they do not require any fixed infrastructure such as a base station for their operation. In general, routes between nodes in an ad hoc network may include multiple hops and, hence, it is appropriate to call such networks “multi-hop wireless ad hoc networks”.

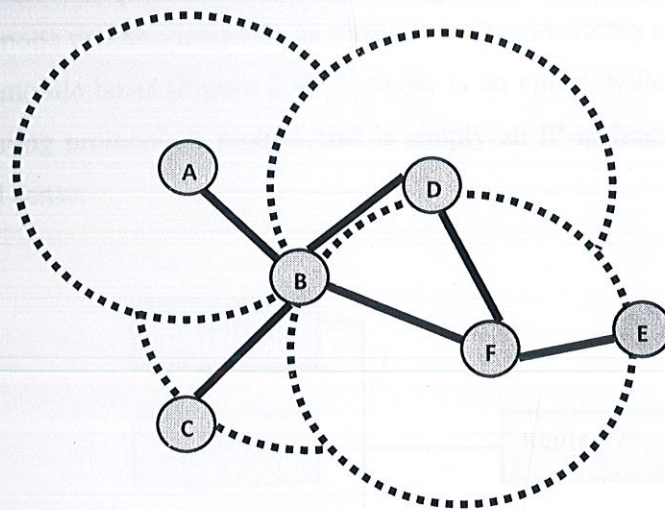


Figure 2.2: Mobile Ad-hoc network

As shown in Figure 2.2, an ad hoc network might consist of several home-computing devices, including notebooks, handheld PCs, and so on. Each node will be able to communicate directly with other nodes that reside within its transmission range. For communicating with nodes that reside beyond this range, the node needs to use intermediate nodes to relay messages hop by hop. For example the following Figure 2.3 shows a simple ad-hoc network with three nodes. The outermost nodes are not within transmission range of each other. However the middle node can be used to forward packets between the outermost nodes. The middle node is acting as a router and the three nodes have formed an ad-hoc network.

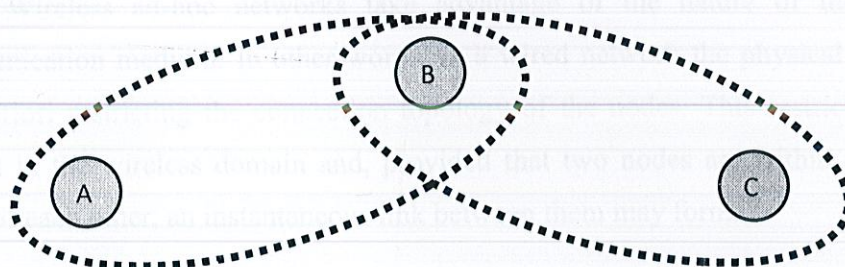


Figure: 2. 3 Example of simple ad-hoc network with three participating nodes

An ad-hoc network uses no centralized administration. This is to be sure that the network won't collapse just because one of the mobile nodes moves out of transmitter range of the other. Nodes should be able to enter/leave the network as they wish. Because of the limited transmitter range of the nodes, multiple hops may be needed to reach other nodes. Every node wishing to participate in an ad-hoc network must be willing to forward packets for other nodes. Thus every node acts as a host and as a router. A node can be viewed as an abstract entity consisting of a router and a set of affiliated mobile hosts (Figure 2.4). A router is an entity, which, among other things runs a routing protocol. A mobile host is simply an IP-addressable host/entity in the traditional sense.

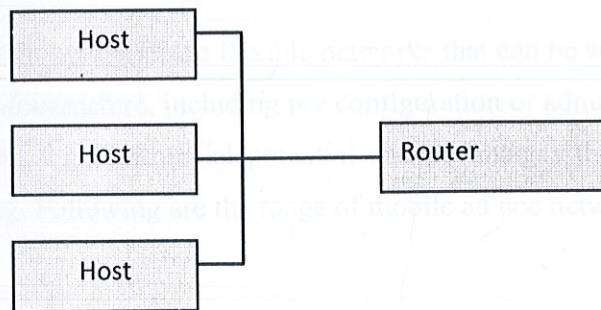


Figure 2.4: Block diagram of mobile node acting both as host and as router

Ad-hoc networks are also capable of handling topology changes and malfunctions in nodes. It is fixed through network reconfiguration. For instance, if a node leaves the network and causes link breakages, affected nodes can easily request new routes and the problem will be solved. This will slightly increase the delay, but the network will still be operational.

Wireless ad-hoc networks take advantage of the nature of the wireless communication medium. In other words, in a wired network the physical cabling is done priori restricting the connection topology of the nodes. This restriction is not present in the wireless domain and, provided that two nodes are within transmitter range of each other, an instantaneous link between them may form.

2.4 Disadvantages of MANET

Some of the disadvantages of MANETs are as follows:

- **Limited resources and physical security.**
- **Intrinsic mutual trust vulnerable to attacks.**
- **Lack of authorization facilities.**
- **Volatile network topology makes it hard to detect malicious nodes.**
- **Security protocols for wired networks cannot work for ad hoc networks.**

2.5 Applications of Mobile Ad-hoc Network

➤ Because ad hoc networks are flexible networks that can be set up anywhere at any time, without infrastructure, including pre configuration or administration, people have come to realize the commercial potential and advantages that mobile ad hoc networking can bring. Following are the range of mobile ad hoc network applications.

Emergency services

- Search-and-rescue operations as well as disaster recovery; e.g., early retrieval and transmission of patient data (record, status, diagnosis) from/to the hospital.
- Replacement of a fixed infrastructure in case of earthquakes, hurricanes, fire, etc.

Commercial environments

- *E-Commerce*, e.g., electronic payments from anywhere (i.e., in a taxi).
- *Business*:
 - Dynamic access to customer files stored in a central location on the fly provide consistent databases for all agents mobile office
- *Vehicular Services*:
 - Transmission of news, road conditions, weather, music
 - Local ad-hoc network with nearby vehicles for road/accident guidance.

Home and enterprise networking

- Home/office wireless networking (WLAN), e.g., shared whiteboard applications, use PDA to print anywhere, trade shows
- Personal area network (PAN).

Educational applications

- Set up virtual classrooms or conference rooms.
- Set up ad hoc communication during conferences, meetings, or lectures

Entertainment

- Multi-user games
- Robotic pets
- Outdoor Internet access

Location-aware services

- Follow-on services, e.g., automatic call forwarding, transmission of the actual workspace to the current location.
- *Information services:*
 - Push, e.g., advertise location-specific service, like gas stations
 - Pull, e.g., location-dependent travel guide; services (printer, fax, phone, server, gas stations) availability information; caches, intermediate results, state information, etc.

Tactical networks

- Military communication, operations
- Automated Battlefields

2.6 Concerns related to MANET

The wireless and mobile ad hoc nature of MANET brings new security challenges to network design. Because nodes in mobile ad hoc network generally communicate with each other via open and shared broadcast wireless channels, they are more vulnerable to security attacks. In addition, their distributed and infrastructure

less nature means that centralized security control is hard to implement and the network has to rely on individual security solutions from each mobile node. Furthermore, as ad hoc networks are often designed for specific environments and may have to operate with full availability even in adverse conditions, security solutions applied in more traditional networks may not be directly suitable.

Understanding the possible form of attacks is the first step toward developing good security solutions. In mobile ad hoc networks, the broadcasting wireless medium inherently signifies that an attack may come from any direction and from different layers (network or application transport such as TCP flooding and SYN flooding). The principle of ad hoc networks sounds like a great idea. A dynamic connection between devices that can be used from anywhere and offers limitless business, recreational and educational opportunities appears to be a promising technological advancement towards making our lives easier. However, as with conventional networks, security and safety considerations have to be taken into account.

Ad hoc networks are by nature very open to anyone. Their biggest advantage is also one of their biggest disadvantages: basically anyone with proper hardware and knowledge of network topology and protocols can connect to the network. This allows potential attackers to infiltrate the network and carry out attacks on its participants with the purpose of stealing or altering information.

Also, depending on the application, certain nodes or network components may be exposed to physical attacks which can disrupt the functionality. In contrary to conventional networks, ad hoc network hosts are more often than not part of an environment that is not maintained professionally. Wireless nodes might be scattered over a large area, where it may pose difficult to supervise all of them.

Another specialty of ad hoc networks is their heavy reliance on inter-node communication. Due to the dynamic nature of the link between the single nodes, it may happen that a certain node B is not in range of node A. In these cases, the information can be routed through intermittent nodes. Even though this is of course not a new concept since it is heavily utilized in the infrastructure of the Internet, the fact that ad hoc network nodes are usually mobile and can disappear at any time (both

from within the range of a particular node as well as from the entire network), the possibility that a certain data route becomes unavailable is significantly higher than in fixed-location networks. This makes easier for attackers to disrupt the network than in conventional networks.

3.1 The Basics

NS2 is a discrete event simulator (DES) which covers a large number of applications. It is a network system of network elements and of traffic models. These are called "simulated objects". NS2 is discrete event simulator where events are scheduled at discrete intervals of time. The advance of time depends on the timing of events which are handled by a scheduler. An event is executed in the C++ library with an unique ID, a schedule time and the pointer to an object or that handles the event. The scheduler keeps an ordered data structure with the events to be executed and that data structure is called the Event Queue. Objects or Events are maintained which are ordered by time. In this time variable changes at discrete values in time, for example, a packet is a packet which has a certain buffer capacity. NS2 is currently UNIX based. It is also supported in windows operating systems. Since most of the library files of ns2 is supported in UNIX based environment so to simulate our project we have used NS2 in UNIX.

NS2 simulator is used on two languages:-

1. Tcl interpreter: It is object oriented extension of Tel (Tool Command Language). It is used to execute user's command scripts like to define specific topology, specific protocols and application that has to be simulated.

2. C++ Programming language: Used as backend (library files) in NS2 which adds to achieving efficiency in simulation and faster execution time. Since NS2 is discrete event simulator, so backend files must be supported over there.

NS2 has a rich library of network and protocol objects. There are two class hierarchies, the compiled C++ hierarchy and the interpreted Tcl one, with one to one correspondence between them. The compiled C++ hierarchy

CHAPTER 3

Introduction to Ns-2

3.1 The Basics

NS2 is Network Simulator (version 2) covers a large number of applications, of protocols, of network types, of network elements and of traffic models. These are called “simulated objects”. NS2 is Discrete Event Simulator where events are to be maintained at discrete interval of time. The advance of time depends on the timing of events which are maintained by a scheduler. An event is an object in the C++ hierarchy with an unique ID, a schedule time and the pointer to an object in that handles the event. The scheduler keeps an ordered data structure with the events to be executed and fires them one by one, invoking the handler of the event. Queue of Events are maintained which are ordered by time. In this state variable changes at discrete points in time, for e.g. length of a queue in a packet switching network or buffer occupancy. NS2 is primarily UNIX based. It is also supported in windows operating environment, since most of the library files of ns2 is supported in UNIX based environment so to simulate our project we have used Ns2 in UNIX.

NS simulator is based on two languages:-

OTCL interpreter: It is Object oriented extension of Tcl (Tool Command Language). It is used to execute user’s command scripts like to define specific topology, specific protocols and application that has to be simulated.

C++ Programming language: Used as backhand (library files) in NS2 which allow in achieving efficiency in simulation and faster execution time, Since NS2 is discrete event simulator so backhand files must be supported over there.

NS has a rich library of network and protocol objects. There are two class hierarchies: the compiled C++ hierarchy and the interpreted OTcl one, with one to one correspondence between them. The compiled C++ hierarchy

allows us to achieve in the simulation and faster execution times. This is in particular useful for the detailed definition and operation of protocols.

TCL (Tool Command Language) is used by NS2. It is a language with a very simple syntax and it allows a very easy integration with other languages. The characteristics of these languages are:

- It Allows a fast development.
- It provides a graphique interface.
- It is compatible with many platforms.
- It is easy to use and free.

Ns2 supports for:-

- TCP
- Routing
- Multicast-Protocols
- Wired
- wireless (WLAN and satellite) networks
- Energy and movement model

Elements of a simulation

Simulator

- Main class
- Configuration of the simulation
- Creating objects
- Creating events and their scheduling

Nodes

- Nodes in a network (end or intermediate)
- Attached list of agents (~protocols)
- Attached list of neighbors (=> Links)
- Unique ID (~address)

Links

- connecting nodes („physically“)
- simplex-, duplex links
- multiple access LANs, including wireless
- bandwidth
- delay
- queue object
 - enqueue, dequeue,
 - drop
 - receive (implemented in next node)

Queues

- “part of” link
- store, drop packets if necessary
- Decide which packet is dropped
- Drop-tail (FIFO)
- Random Early Detect (RED)
- Class Based Queuing (CBQ, priority + Round Robin)
- Several Fair Queuing mechanisms (SFQ, DRR,...)
- „Drop Destination“
- Object all dropped packets are forwarded to

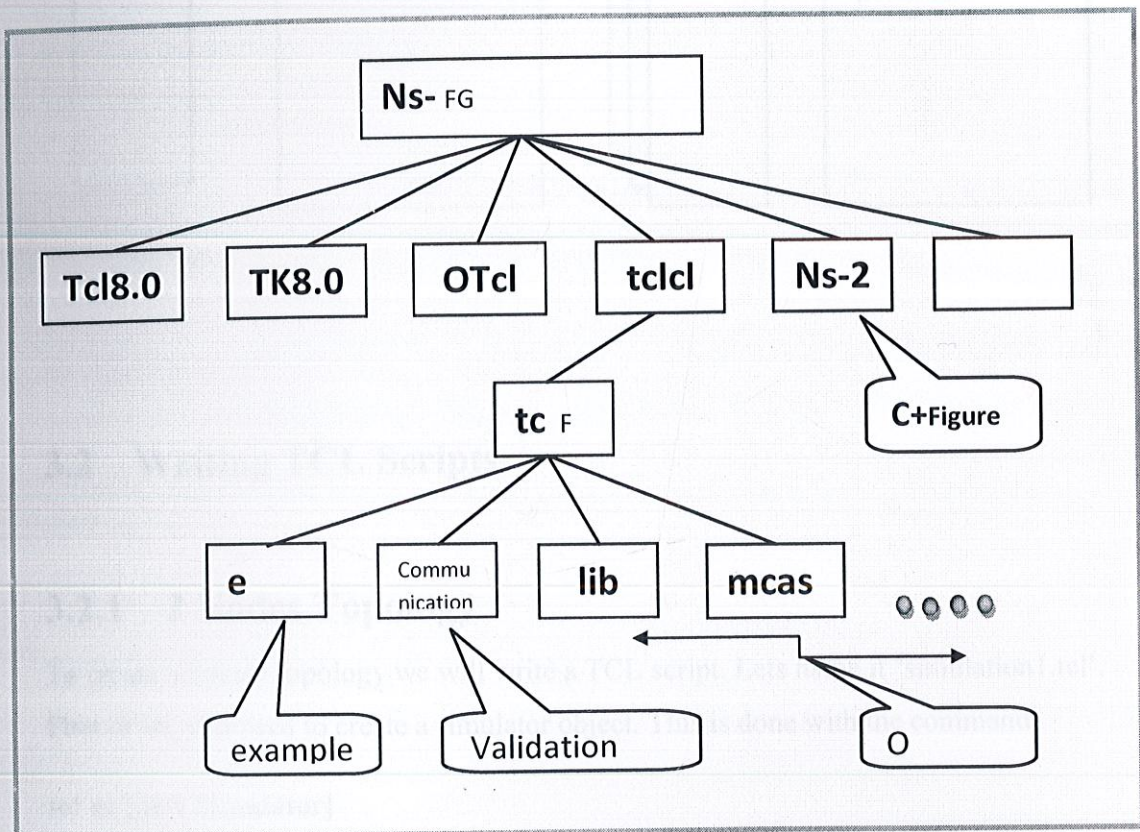
Agents

- Endpoint of (logical) connections
- ~ OSI level 3 (network)
- Create and receive packets
- implement protocols

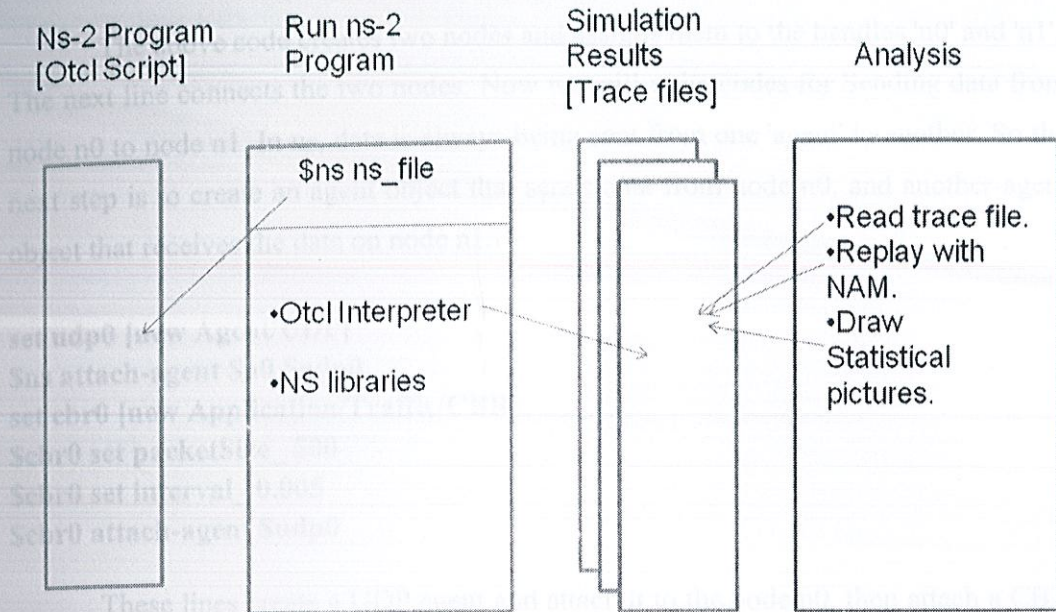


- Sometimes additional sender and receiver
- necessary
- TCP, TCPSink in div. „flavors“
- UDP
- RTP, RTCP

The NS2 Directory Structure:-



Running NS2 Program:-



3.2 Writing TCL Scripts

3.2.1 2 Nodes Topology

To create a 2 node topology we will write a TCL script. Lets name it 'simulation1.tcl'.

First of all, you need to create a simulator object. This is done with the command

```
set ns [new Simulator]
```

Now we open a file for writing that is going to be used for the nam trace data.

```
set nf [open out.nam w]
```

```
$ns namtrace-all $nf
```

The first line opens the file 'out.nam' for writing and gives it the file handle 'nf'. In the second line we tell the simulator. Object that we created above to write all simulation data that is going to be relevant for nam into this file.

```
set n0 [$ns node]
```

```
set n1 [$ns node]
```

```
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
```


The above code creates two nodes and assigns them to the handles 'n0' and 'n1'. The next line connects the two nodes. Now we will write codes for Sending data from node n0 to node n1. In ns, data is always being sent from one 'agent' to another. So the next step is to create an agent object that sends data from node n0, and another agent object that receives the data on node n1.

```
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

These lines create a UDP agent and attach it to the node n0, then attach a CBR traffic generator to the UDP agent. CBR stands for 'constant bit rate'. Line 7 and 8 should be self-explaining. The packet Size is being set to 500 bytes and a packet will be sent every 0.005 seconds (i.e. 200 packets per second). You can find the relevant parameters for each agent type. The next lines create a Null agent which acts as traffic sink and attach it to node n1. After that the two agents have to be connected with each other. Finally to tell the CBR agent when to send data and when to stop sending write last two lines.

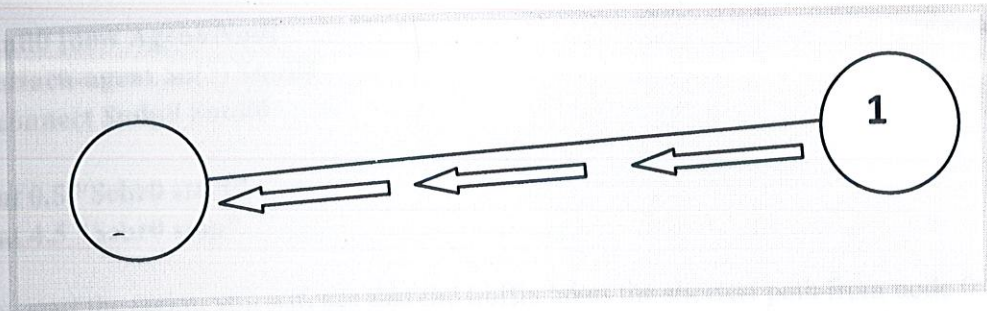
```
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0
$ns connect $udp0 $null0
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

In next step add a 'finish' procedure that closes the trace file and starts nam.

```
proc finish {} {
global ns nf
$ns flush-trace
close $nf
exec nam out.nam &
exit 0
}
```


The next line tells the simulator object to execute the 'finish' procedure after 5.0 seconds simulation time. The last line finally starts the simulation.

```
$ns at 5.0 "finish"  
$ns run
```



3.2.2 7 Nodes Scenario

As always, the topology has to be created first, though this time we take a different approach which you will find more comfortable when you want to create larger topologies.

The following code creates seven nodes and stores them in the array `n()`. To connect the nodes to create a circular topology last line are written.

```
for {set i 0} {$i < 7} {incr i} {  
  set n($i) [$ns node]  
}  
  
for {set i 0} {$i < 7} {incr i} {  
  $ns duplex-link n($i) n([expr ($i+1)%7]) 1Mb 10ms DropTail  
}
```

The next step is to send some data from node `n(0)` to node `n(3)`. Create a UDP agent and attach it to node `n(0)` and then Create a CBR traffic source and attach it to `udp0`.

```
set udp0 [new Agent/UDP]
```



```
$ns attach-agent $n(0) $udp0
```

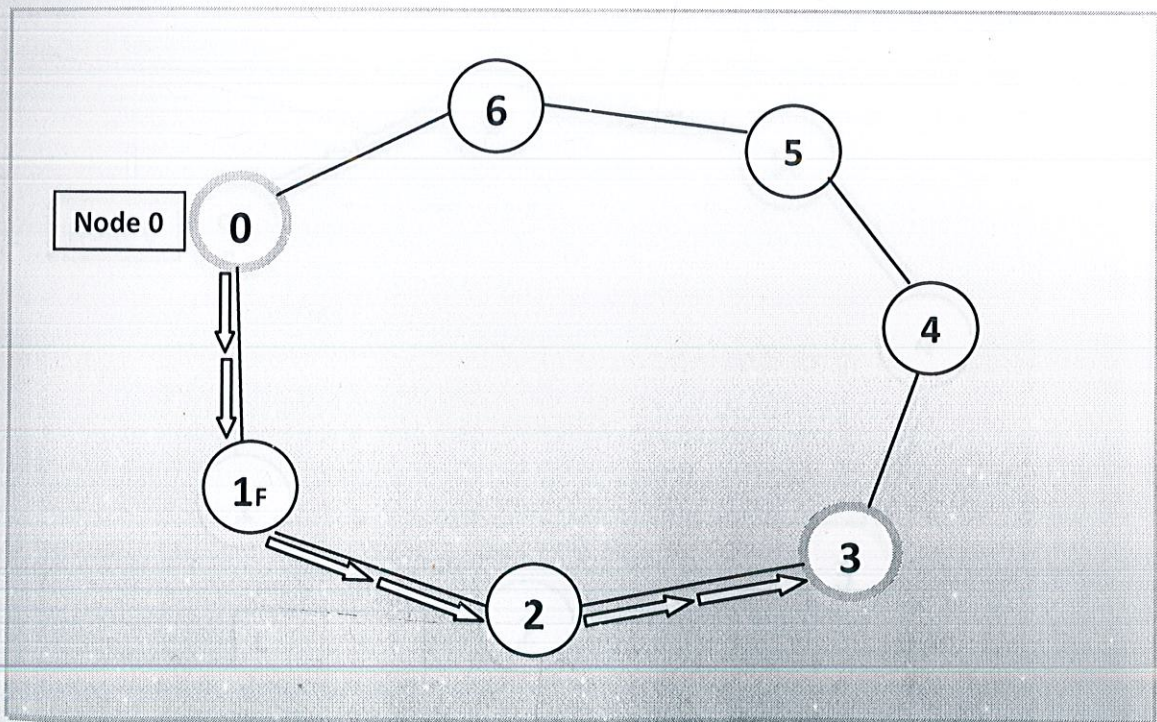
```
set cbr0 [new Application/Traffic/CBR]  
$cbr0 set packetSize_ 500  
$cbr0 set interval_ 0.005  
$cbr0 attach-agent $udp0
```

```
set null0 [new Agent/Null]  
$ns attach-agent $n(3) $null0  
$ns connect $udp0 $null0
```

```
$ns at 0.5 "$cbr0 start"
```

```
$ns at 4.5 "$cbr0 stop"
```

If you start the script, you will see that the traffic takes the shortest path from node 0 to node 3 through nodes 1 and 2, as could be expected. Here we complete 7 node scenarios. Only one difference we got over here is use of array to create large number of nodes.



Now we add another interesting feature. We let the link between node 1 and 2 (which

is being used by the traffic) go down for a second.

```
$ns rtmodel-at 1.0 down $n(1) $n(2)
```

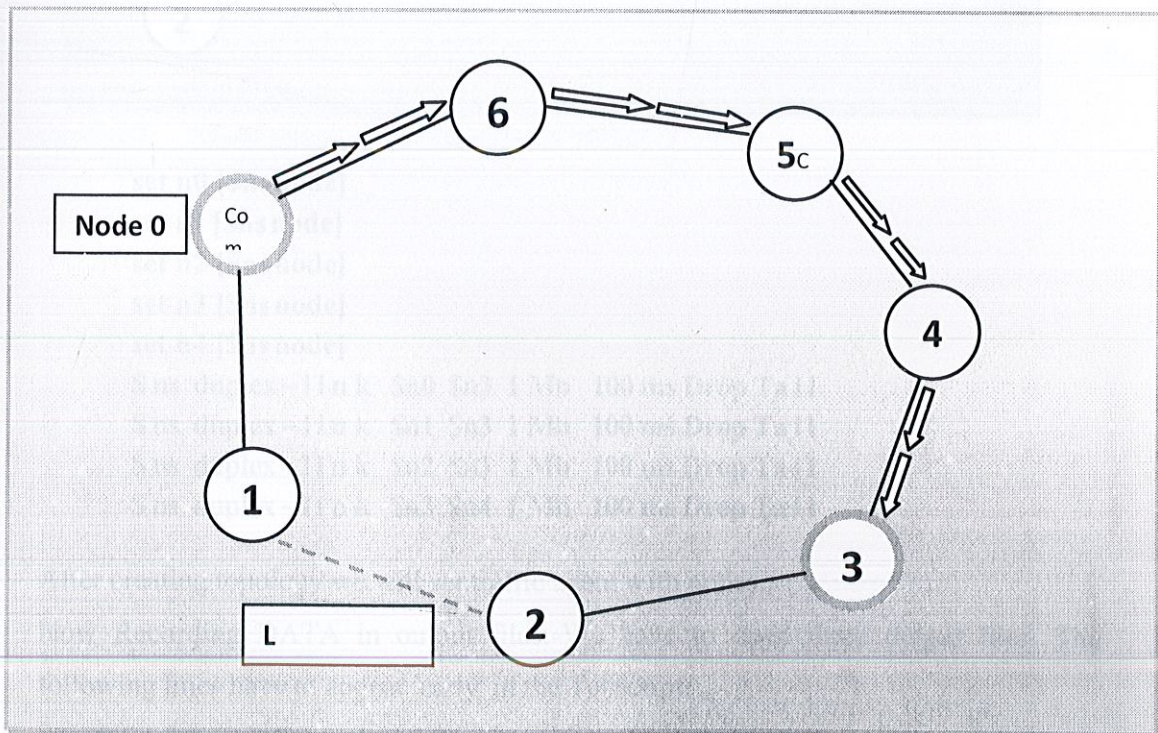
```
$ns rtmodel-at 2.0 up $n(1) $n(2)
```

Since link is broken between nodes we will use dynamic routing to solve that 'problem' by adding following line at the beginning of Tcl script, after the simulator object has been created.

```
$ns rtproto DV
```

Start the simulation again, and you will see how at first a lot of small packets run through the network. If you slow nam down enough to click on one of them, you will see that they are 'rtProtoDV' packets which are being used to exchange routing information between the nodes.

When the link goes down again at 1.0 seconds, the routing will be updated and the traffic will be re-routed through the nodes 6, 5 and 4.



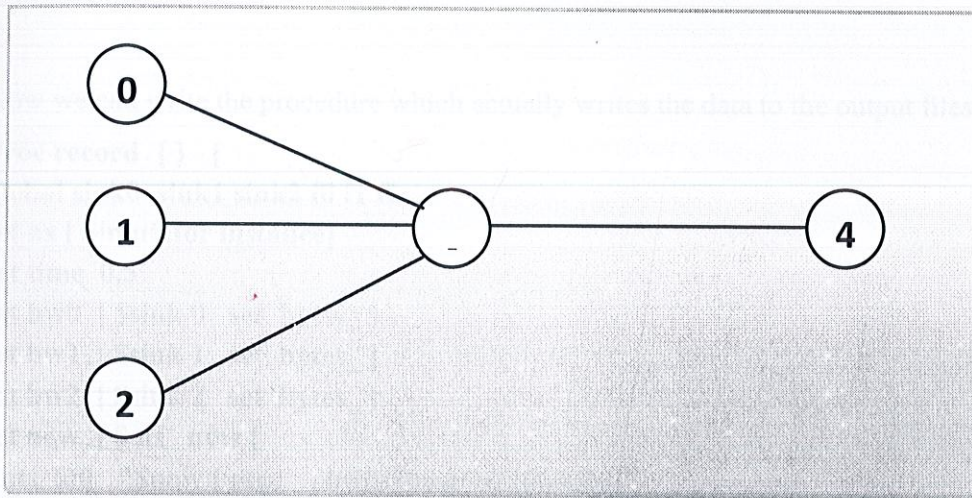
When link between node1 and node2 is broken then new dynamic route has been discovered and data is being transferred through next route.

In this way we have many routing protocol used for wired and wireless topology according to demand of specific situation.

3.3 XGRAPH

Creating Output Files for Xgraph:-

1. Creating topology and traffic sources: In below codes a 5 node scenario is being created to analyze traffic by generating XGRAPH.



```
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
$ns duplex-link $n0 $n3 1 Mb 100 ms Drop Tail
$ns duplex-link $n1 $n3 1 Mb 100 ms Drop Tail
$ns duplex-link $n2 $n3 1 Mb 100 ms Drop Tail
$ns duplex-link $n3 $n4 1 Mb 100 ms Drop Tail
```

After creating topology we will set traffic agent with nodes.

Now Recording DATA in output files:-We have to open three output files. The following lines have to appear 'early' in the Tcl script.

```
set f0 [open out0.tr w]
set f1 [open out1.tr w]
set f2 [open out2.tr w]
```


These files have to be closed at some point. We use a modified 'finish' procedure to do that. It not only closes the output files, but also calls xgraph to display the results. You may want to adapt the window size (800 x400) to your screen size.

```
Proc finish {} {  
  global f0 f1 f2  
  close $f0  
  close $f1  
  close $f2  
  exec xgraph out0.tr out1.tr out2.tr - geometry 800x400 &  
  exit 0  
}
```

Now we can write the procedure which actually writes the data to the output files.

```
Proc record {} {  
  global sink0 sink1 sink2 f0 f1 f2  
  set ns [ Simulator instance]  
  set time 0.5  
  set bw0 [ $sink 0 set bytes_  
  set bw1 [ $sink 1 set bytes_  
  set bw2 [ $sink 2 set bytes_  
  set now [ $ns now ]  
  puts $f0 "$now [ expr $bw0/$time* 8/1000000]"  
  puts $f1 "$now [ expr $bw1/$time* 8/1000000]"  
  puts $f2 "$now [ expr $bw2/$time* 8/1000000]"  
  $sink0 set bytes_ 0  
  $sink1 set bytes_ 0  
  $sink2 set bytes_ 0  
  $ns at [ expr $now +$time] "record"  
}
```

This procedure reads the number of bytes received from the three traffic sinks. Then it calculates the bandwidth (in M Bit/s) and writes it to the three output files together with the current time before it resets the bytes_ values on the traffic sinks. Then it re-schedules itself.

Now running the Simulation:-

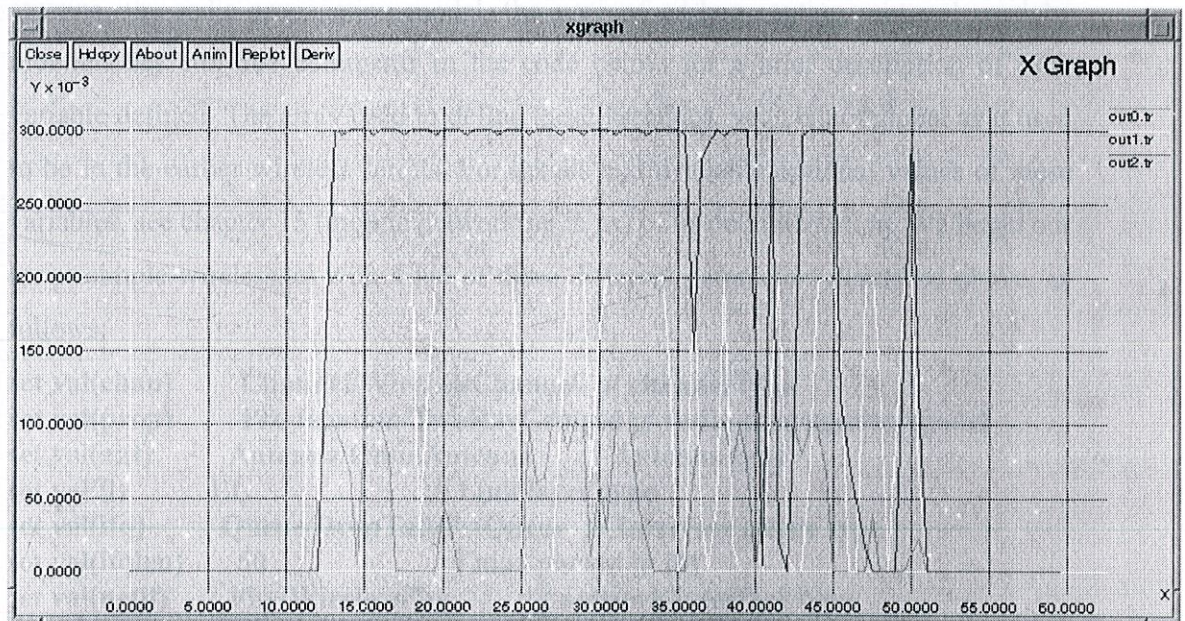
We can now schedule the following events:


```

$ns at 0.0 "record"
$ns at 10.0 "$ source0 start"
$ns at 10.0 "$ source1 start"
$ns at 10.0 "$ source2 start"
$ns at 50.0 "$ source0 stop"
$ns at 50.0 "$ source1 stop"
$ns at 50.0 "$ source2 stop"
$ns at 60.0 "finish"
$ns run

```

First, the 'record' procedure is called, and afterwards it will re-schedule itself periodically every 0.5 seconds. Then the three traffic sources are started at 10 seconds and stopped at 50 seconds. At 60 seconds, the 'finish' procedure is called. When you run the simulation, an xgraph window should open after some time which should look similar to this one:



3.4 Running Wireless Simulations

3.4.1 Scenario File

We are going to simulate a very simple 2-node wireless scenario. The topology consists of two mobilenodes, node₍₀₎ and node₍₁₎. The mobilenodes move about within an area whose boundary is defined in this example as 500mX500m. The nodes

start out initially at two opposite ends of the boundary. Then they move towards each other in the first half of the simulation and again move away for the second half. A TCP connection is setup between the two mobilenodes. Packets are exchanged between the nodes as they come within hearing range of one another. As they move away, packets start getting dropped.

Just as with any other ns simulation, we begin by creating a tcl script for the wireless simulation. We will call this file simple-wireless.tcl.

A mobilenode consists of network components like Link Layer (LL), Interface Queue (IfQ), MAC layer, the wireless channel nodes transmit and receive signals from etc. For details about these network components see section 1 of chapter 15 of At the beginning of a wireless simulation, we need to define the type for each of these network components. Additionally, we need to define other parameters like the type of antenna, the radio-propagation model, the type of ad-hoc routing protocol used by mobilenodes etc. See comments in the code below for a brief description of each variable defined. The array used to define these variables, val() is not global as it used to be in the earlier wireless scripts. For details and available optional values of these variables, see chapter 15 (mobile networking in ns) of ns documentation. We begin our script simple-wireless.tcl with a list of these different parameters described above, as follows:

```

set val(chan)      Channel/WirelessChannel ;# channel type
set val(prop)      Propagation/TwoRayGround ;# radio-propagation model
set val(ant)        Antenna/OmniAntenna   ;# Antenna type
set val(ll)         LL                     ;# Link layer type
set val(ifq)        Queue/DropTail/PriQueue ;# Interface queue type
set val(ifqlen)     50                     ;# max packet in ifq
set val(netif)      Phy/WirelessPhy       ;# network interface type
set val(mac)        Mac/802_11            ;# MAC type
set val(rp)         DSDV                   ;# ad-hoc routing protocol
set val(nn)         2                     ;# number of mobilenodes

```

Next we go to the main part of the program and start by creating an instance of the simulator,

```
set ns_ [new Simulator]
```

Then setup trace support by opening file simple.tr and call the procedure trace-all {} as follows:

```

set tracefd [open simple.tr w]
$ns_ trace-all $tracefd

```


Next create a topology object that keeps track of movements of mobilenodes within the topological boundary.

```
set topo [new Topography]
```

We had earlier mentioned that mobilenodes move within a topology of 500mX500m. We provide the topography object with x and y co-ordinates of the boundary, (x=500, y=500):

```
$topo load_flatgrid 500 500
```

The topography is broken up into grids and the default value of grid resolution is 1. A different value can be passed as a third parameter to `load_flatgrid {}` above. Next we create the object God, as follows:

```
create-god $val(nn)
```

Quoted from CMU document on god, "God (General Operations Director) is the object that is used to store global information about the state of the environment, network or nodes that an omniscient observer would have, but that should not be made known to any participant in the simulation." Currently, God object stores the total number of mobilenodes and a table of shortest number of hops required to reach from one node to another.

First, we need to configure nodes before we can create them. Node configuration API may consist of defining the type of addressing (flat/hierarchical etc), the type of adhoc routing protocol, Link Layer, MAC layer, IfQ etc.

The configuration API for creating mobilenodes looks as follows:

```
$ns_ node-config -adhocRouting $val(rp) \  
-llType $val(ll) \  
-macType $val(mac) \  
-ifqType $val(ifq) \  
-ifqLen $val(ifqlen) \  
-antType $val(ant) \  
-propType $val(prop) \  
-phyType $val(netif) \  
-topoInstance $topo \  

```



```

-channelType $val(chan) \
-agentTrace ON \
-routerTrace ON \
-macTrace OFF \
-movementTrace OFF

```

Next we create the 2 mobilenodes as follows:

```

for {set i 0} {$i < $val(nn)} {incr i} {
set node_($i) [$ns_node ]
$node_($i) random-motion 0    ;# disable random motion
}

```

The random-motion for nodes is disabled here, as we are going to provide node position and movement (speed & direction) directives next.

Now that we have created mobilenodes, we need to give them a position to start with,

```

$node_(0) set X_ 5.0
$node_(0) set Y_ 2.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 390.0
$node_(1) set Y_ 385.0
$node_(1) set Z_ 0.0

```

Node0 has a starting position of (5,2) while Node1 starts off at location (390,385).

Next produce some node movements,

```

$ns_ at 50.0 "$node_(1) setdest 25.0 20.0 15.0"
$ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"
$ns_ at 100.0 "$node_(1) setdest 490.0 480.0 15.0"

```

\$ns_ at 50.0 "\$node_(1) setdest 25.0 20.0 15.0" means at time 50.0s, node1 starts to move towards the destination (x=25,y=20) at a speed of 15m/s. This API is used to change direction and speed of movement of the mobilenodes.

Next setup traffic flow between the two nodes as follows:

```

set tcp [new Agent/TCP]

```



```

Step set class_2
set sink [new Agent/TCPSink]
$ns_ attach-agent $node_(0) $step
$ns_ attach-agent $node_(1) $sink
$ns_ connect $step $sink
set ftp [new Application/FTP]
$ftp attach-agent $step
$ns_ at 10.0 "$ftp start"

```

This sets up a TCP connection between the two nodes with a TCP source on node0. Then we need to define stop time when the simulation ends and tell mobilenodes to reset which actually resets their internal network components,

```

for {set i 0} {$i < $val(nn)} {incr i} {
$ns_ at 150.0 "$node_($i) reset";
}

$ns_ at 150.0001 "stop"
$ns_ at 150.0002 "puts \"NS EXITING...\" ; $ns_ halt"
proc stop {} {
global ns_ tracefd
close $tracefd
}

```

At time 150.0s, the simulation shall stop. The nodes are reset at that time and the "\$ns_ halt" is called at 150.0002s, a little later after resetting the nodes. The procedure stop {} is called to flush out traces and close the trace file. And finally the command to start the simulation,

```

puts "Starting Simulation..."
$ns_ run

```


CHAPTER 4

ROUTING

4.1 Definition

Routing is the act of moving information from a source to a destination in an internetwork. Routing is usually performed by a dedicated device called router. It is a key feature of the internet because it enables messages to pass from one computer to another and eventually reach the target machine. Each intermediary computer performs routing by passing along the message to the next computer.

Routing is often confused with bridging, which performs a similar function. The principal difference between the two is that bridging occurs at a lower level and is therefore more of a hardware function whereas routing occurs at a higher level where the software component is more important. And because routing occurs at a higher level, it can perform more complex analysis to determine the optimal path for the packet. The transferring of packets through an internetwork is called as packet switching which is straight forward, but the path determination could be very complex.

Routing protocols use several metrics as a standard measurement to calculate the best path for routing the packets to its destination. The process of path determination is that, routing algorithms find out and maintain routing tables, which contain the total route information for the packet. The information of route varies from one routing algorithm to another.

Routing is mainly classified into static routing and dynamic routing. Static routing refers to the routing strategy being stated manually or statically in the router. Static routing maintains a routing table usually written by the network administrator.

Dynamic routing refers to the routing strategy that is being learnt by an interior or exterior routing protocol. This routing primarily depends on the state of the network i.e., the routing table is affected by the activeness of the destination.

4.2 Routing in mobile Ad-hoc networks

Mobile Ad-hoc networks are self-organizing and self-configuring multihop wireless networks, where the structure of the network changes dynamically. This is mainly due to the mobility of the nodes. Nodes in these networks utilize the same random access wireless channel, cooperating in an intimate manner to engaging themselves in multihop forwarding. The node in the network not only acts as hosts but also as routers that route data to/from other nodes in network. In mobile ad-hoc networks there is no infrastructure support as is the case with wireless networks, and since a destination node might be out of range of a source node transferring packets; so there is need of a different routing procedure.

Each device in a MANET is free to move independently in any direction, and will therefore change its links to other devices frequently. Each must forward traffic unrelated to its own use, and therefore be a router. The primary challenge in building a MANET is equipping each device to continuously maintain the information required to properly route traffic.

4.3 Problems in routing with Mobile Ad-hoc Networks

- **Asymmetric Links:** Most of the wired networks rely on the symmetric links which are always fixed. But this is not a case with ad-hoc networks as the nodes are mobile and constantly changing their position within network.
- **Routing Overhead:** In wireless ad-hoc networks, nodes often change their location within network. So, some stale routes are generated in the routing table which leads to unnecessary routing overhead.
- **Interference:** This is the major problem with mobile ad-hoc networks as the links come and go depending on the transmission characteristics, one transmission might interfere with another and node might overhear transmissions of other nodes and can corrupt the total transmission.

- **Dynamic Topology:** Since the topology is not constant, so the mobile node might move or medium characteristics might change. In ad-hoc networks, routing tables must somehow reflect these changes in topology and routing algorithms have to be adapted. For example, in a fixed network routing table updating takes place at every fixed interval. This updating frequency must be very high in case of ad-hoc networks.

CHAPTER 5

CLUSTERING

Besides being advantageous of having low deployment cost, ad-hoc networks are battery and power constrained. The nodes in an ad hoc network can vary from a Laptop to a Cell Phone. These devices have limited power and processing capabilities. Therefore, the more the network grows, the more they are required to forward packets for other nodes; devoting a significant amount of processing power.

Intrusion Detection System can be deployed in these self-organizing multi-hop wireless networks to protect them against a number of attacks by offering auditing and monitoring capabilities to a node. However, normal intrusion detection approaches cannot be used in this environment. Since these networks lack infrastructure, we need to monitor intrusions at all nodes in the network. But, due to mobility and other constraints such as restricted power and processing capacity, nodes cannot run heavy applications to detect intrusions. If every node starts monitoring intrusions separately, processing overhead at each node will consume a large portion of their battery and power. Therefore, a scalable and fault tolerant IDS is required to govern these on-secure wireless ad-hoc networks against attacks.

The efficient solution is to defend against intrusion co-operatively, rather than each mobile node performing full analysis of traffic passing through it. In order to cooperate, the nodes must trust on each other so that they should not audit all the data, and hence, they can save a lot more processing and memory overhead.

The clustering approach can be taken as an additional advantage in these processing constrained networks to collaboratively detect intrusions with less power usage and minimal overhead. Existing clustering protocols are not suitable for intrusion detection purposes because they are linked with the routes. The route establishment and route renewal affects the clusters and as a consequence, the

processing and traffic overhead increases due to instability of clusters. The ad-hoc networks are battery and power constraint, and therefore a trusted monitoring node should be available to detect and respond against intrusions in time. This can be achieved only if the clusters are stable for a long period of time. If the clusters are regularly changed due to routes, the intrusion detection will not prove to be effective. Therefore, a generalized clustering algorithm is needed that can run on top of any routing protocol and can monitor the intrusions constantly irrespective of the routes. Clustering is also useful to detect intrusions collaboratively since an individual node can neither detect the malicious node alone nor it can take action against that node on its own.

5.1 Clustering

One of the key questions in cooperative intrusion architecture is related to the organization of nodes. If every node is required to cooperate with every other node (as in a peer-to-peer architecture) without a structured organization, then overheads will grow proportionally to the square of the number of nodes. Rather than assuming a completely flat structure, it may be possible to organize nodes in a hierarchy by forming groups efficiently, leading to reduced communication overheads.

It is critical to organize the groups appropriately in order to achieve a significant reduction in overhead. This problem of designing groups efficiently is a fundamental problem that has been the focus of research in MANET used which has been considered especially in relation to routing. The same concept can be adapted for the intrusion detection function. When designing the clustering schemes for detection, researchers have tried to exploit several characteristics of the network in order to minimize overheads.

An intuitive approach for organizing into clusters is according to node mobility. Clusters are not static but change as nodes move around. As the nodes move, they may drop out from their existing cluster and join a different cluster that is closer to them. Clusters in some cases may join together or a single cluster may split

into two or more clusters. Typically, a node might be elected in the cluster as a leader and such a node is typically called the cluster-head.

There are several algorithms that have been proposed in the literature for organizing nodes in the clusters based on a variety of criteria, including how the clusters adapt to connectivity changes due to mobility and how cluster-heads for each cluster are elected.

5.2 Applications of Clustering

The clustering algorithms have been studied extensively for MANET and have been shown to improve the performance of routing and other functions in MANET. So a natural question here is whether it is possible to use the clustering concept for organizing the intrusion detection functions efficiently in a MANET. The clustering algorithms used for routing may not be quite applicable directly and might have to be modified because of the unique characteristics of the intrusion detection process.

One potential use of clustering in intrusion detection is for creating a dynamic hierarchy for collecting and consolidating intrusion detection information. This would be in lieu of the peer-to-peer architecture. Another potential application of clustering is in distributing the function of collecting evidence in the network. For example, the formation of cluster in which a cluster head is one hop away from every member of the cluster. In that case, a cluster-head can observe all communications in the cluster through promiscuous monitoring and therefore become the natural chokepoint in the neighborhood for placing the intrusion detection function. The cluster-head in this case can then be the single node in the cluster that needs to run the intrusion detection function. Such a cluster-head would focus on malicious behavior that can be observed from the network traffic. Multiple nodes in the cluster do not need to perform the same function, reducing the power consumption on those nodes.

5.2.1 Limitations

The use of clustering algorithms for intrusion detection also has significant limitations specific to intrusion detection. The biggest limitation, as remarked earlier,

is that these algorithms are not resistant to attacks such as cluster-heads being taken over by an adversary in order to bypass the intrusion detection mechanism. A cluster-head that becomes malicious can introduce malicious intrusion detection reports and share these reports with the rest of the hierarchy. This could result in a well-behaved node being declared malicious and being expelled from the network. The higher the cluster-head is in the hierarchy, the more severe is the damage caused by a malicious cluster-head. The root of the hierarchy can potentially make the whole network inoperable if it becomes malicious.

Another limitation of the use of clustering for intrusion detection purposes is that algorithms may be manipulated by a malicious node to cause that node to be elected as a cluster-head. The election of a malicious node as a cluster-head can have significant impact in bypassing the overall intrusion detection architecture.

Due to mobility and other constraints such as restricted power and processing capacity, nodes cannot run heavy applications to detect intrusions. Therefore, the architecture should be simple yet effective to provide security against different type of attacks. The more efficient solution is to defend against intrusion co-operatively, rather than each mobile node performing full analysis of traffic passing through it. In order to co-operate, the nodes must trust on each other so that they should not audit all the data, and hence they can save a lot more processing and memory overhead.

The clustering schemes like Cluster-Based Routing Protocol, Cluster Switch Gateway Routing, and One-Demand Clustering result in a monitoring node, called "Head Node", which can be trusted in this regard if the selection of head node is based upon fair and secure criteria.

We aimed to use ad-hoc clustering protocols such as Cluster-Based Routing Protocol or Cluster Switch Gateway Routing, since only the cluster head node forwards the data packets on behalf of all the cluster members; thus increasing route efficiency by reducing network traffic. However, the existing clustering protocols are not widely used in ad hoc networks. Another important drawback with the clustering protocol is that the clusters need to be changed whenever the route is changed.

Therefore, a generalized clustering algorithm is required that can be used specifically for intrusion detection purpose irrespective of the routes. The cluster members are ordinary nodes that parse the data for intrusion during the communication and may seek help from the cluster head to detect intrusion if they find some suspicious activity but their gathered information is still incomplete.

5.3 The Cluster Based Routing Protocol

Cluster Based Routing Protocol (CBRP) is a distributed, efficient and scalable protocol that uses clustering approach to minimize on-demand route discovery traffic. The cluster head is elected for each cluster to maintain cluster membership information. The nodes periodically exchange HELLO packets to maintain a neighbor table and to maintain a 2-hop topology link state table. To discover 3 hop away cluster-heads, cluster adjacency table (CAT) is exchanged in HELLO message. It does not allow two cluster heads to have a direct bi-directional link to each other. Cluster Based Routing Protocol also opposes frequent elections. The overhead, however, in this scheme is to maintain neighbor table and CAT at every node, thus involving bulky traffic and processing load. Also, two heads in a radio range are not a problem in our architecture and they are automatically adjusted after election timeout.

5.3.1 Election Process

The clusters are formed to obtain the head node to monitor traffic within its cluster. The cluster head not only manages its own cluster, but also provides a way for communication to other clusters. It maintains information of every member node and neighbor clusters (forwarded by the gateway node). The neighbor information is useful for network-wide communication. The cluster management responsibility is rotated among the capable members of the cluster for the load balancing and fault tolerance and must be fair and secure. This can be achieved by conducting regular elections. The proposed election process is simple. The cluster head keeps an election interval timer for managing the elections. Every node in the cluster must participate in the election process by casting their vote showing their willingness to become the

cluster head. The node showing the highest willingness becomes the cluster head until the next timeout period.

5.3.1.1 Node Status

The feature of node status has been taken from Cluster Based Routing Protocol and Passive Clustering. UNDECIDED (UD), HEAD (HD), MEMBER (MB), and GATEWAY (GW). Every node comes up in UD state. It switches to MB state if it finds any HD node in its neighbor, otherwise it switches to HD state (if it succeeds in election). If the connection of MB nodes with their HD node breaks, they fall back to UD state. However, if MB node finds another HD node among its neighbor (due to mobility or election process), it becomes GW. Both the MB or GW nodes can move to HD state after election. Simultaneously, HD node upon failing in the election process becomes MB or GW (depending upon number of HD nodes, it is linked to). The GW node upon losing all its HD nodes except one goes back to MB State. The transmission node states discussed is shown in the figure below:

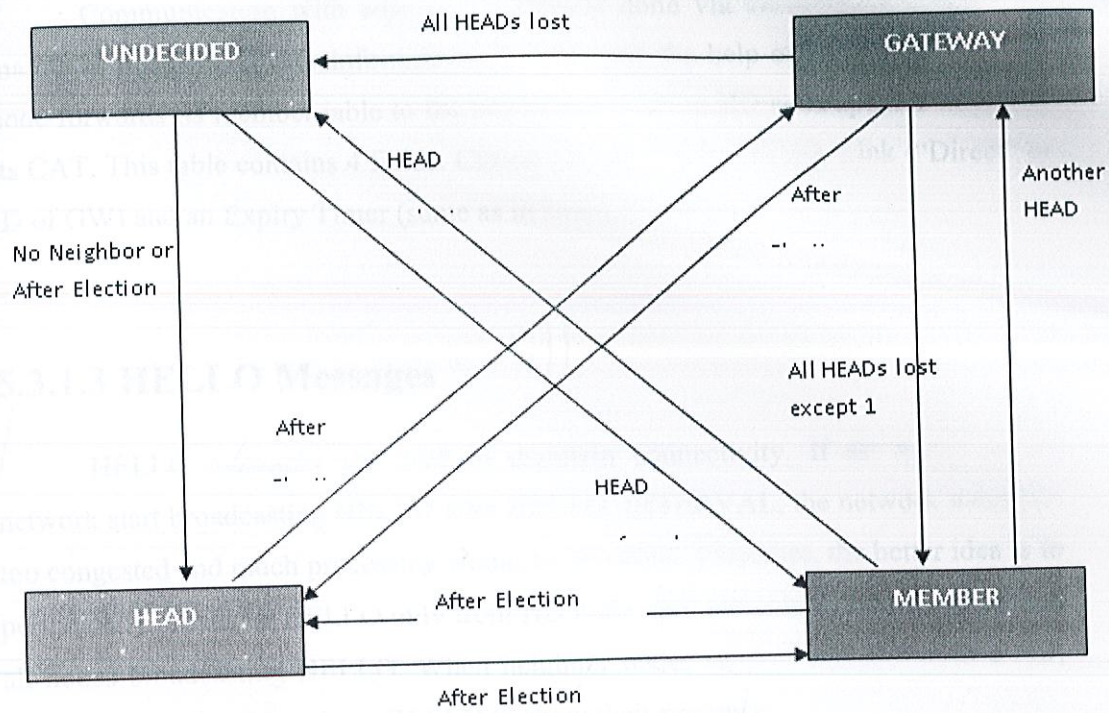


Figure 5.1: Transition diagram of nodes taking part in the cluster formation

5.3.1.2 Data Structure

Two types of data structures have been used: "Member Table" and "Cluster Adjacency Table" (CAT) as in. Since the proposed architecture does not perform routing, the processing overhead and storage space at each node are reduced by keeping all neighbors information only at the head node. The member nodes do not keep the information about their neighbors (except their respective head node) as communication among the member nodes is not required. The cluster-head maintains connections with all its neighbors (MB or GW) in member table using HELLO messages. The fields in member table are Neighbor ID, Status (MB or GW) and an Expiry Timer. The timers associated with each entry are used to delete stale entries. MB and GW nodes only keep information of their respective head nodes in their member tables.

Communication with adjacent clusters is done via cluster-heads. HD node maintains neighbor cluster information in CAT with the help of its GW nodes. GW node forwards its member table to the HD node by which HD adds/updates entries in its CAT. This table contains 4 fields: Cluster ID, Cluster HEAD ID, Link ("Direct" or ID of GW) and an Expiry Timer (same as in Member table).

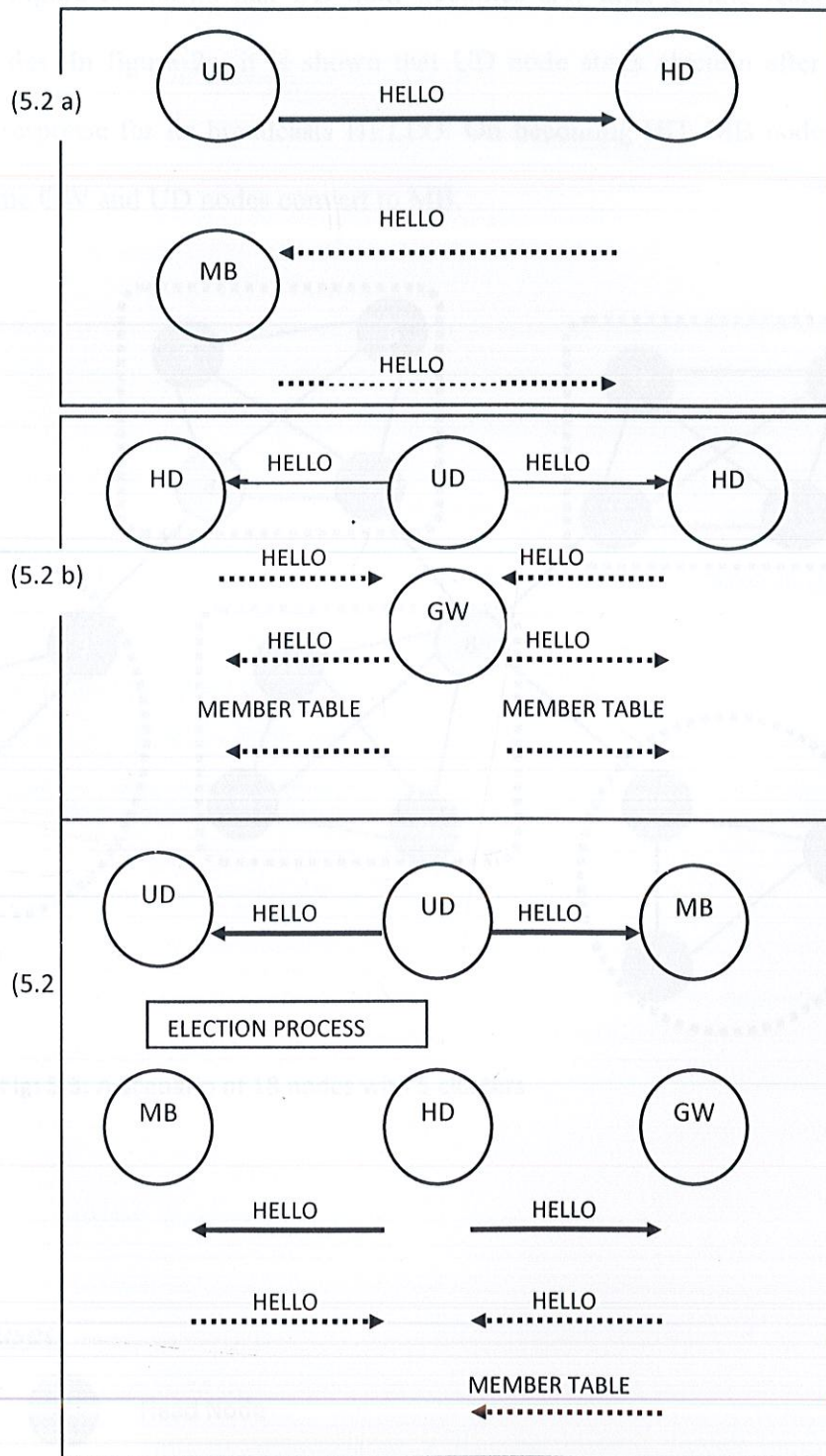
5.3.1.3 HELLO Messages

HELLO messages are used to maintain connectivity. If all nodes in the network start broadcasting HELLO after HELLO_INTERVAL, the network would be too congested and much processing would be involved. Therefore, the better idea is to periodically broadcast HELLO only from HD node after every hello timer, rather than all nodes broadcasting HELLO. When neighbor nodes receive HELLO from a HD, they simply reply with unicast HELLO to show their presence.

Besides HD, a UD node can also broadcast the HELLO message. This is done to find any HD node, and starts HELLO_INTERVAL timer. If some HD node replies, the UD node changes its status to MB. Nodes other than HD simply drop the packet. If timer expires and no HD node replies, the UD node starts election. It is mandatory for a UD node to broadcast at least 1 HELLO packet before starting the election. The following figure 5.2 demonstrates the process of a UD node broadcasting HELLO.



Figure 5.2: Different scenarios for reply to broadcast HELLO from UD node



Notations Used:



Shows Broadcast message

In figure 2a, UD node broadcasts HELLO, gets response from HD and becomes MB. Figure 2b shows that UD node becomes GW after getting response from 2 HD nodes. In figure 2c, it is shown that UD node starts election after not receiving any response for its broadcasts HELLO. On becoming HD, MB nodes in neighbor become GW and UD nodes convert to MB.

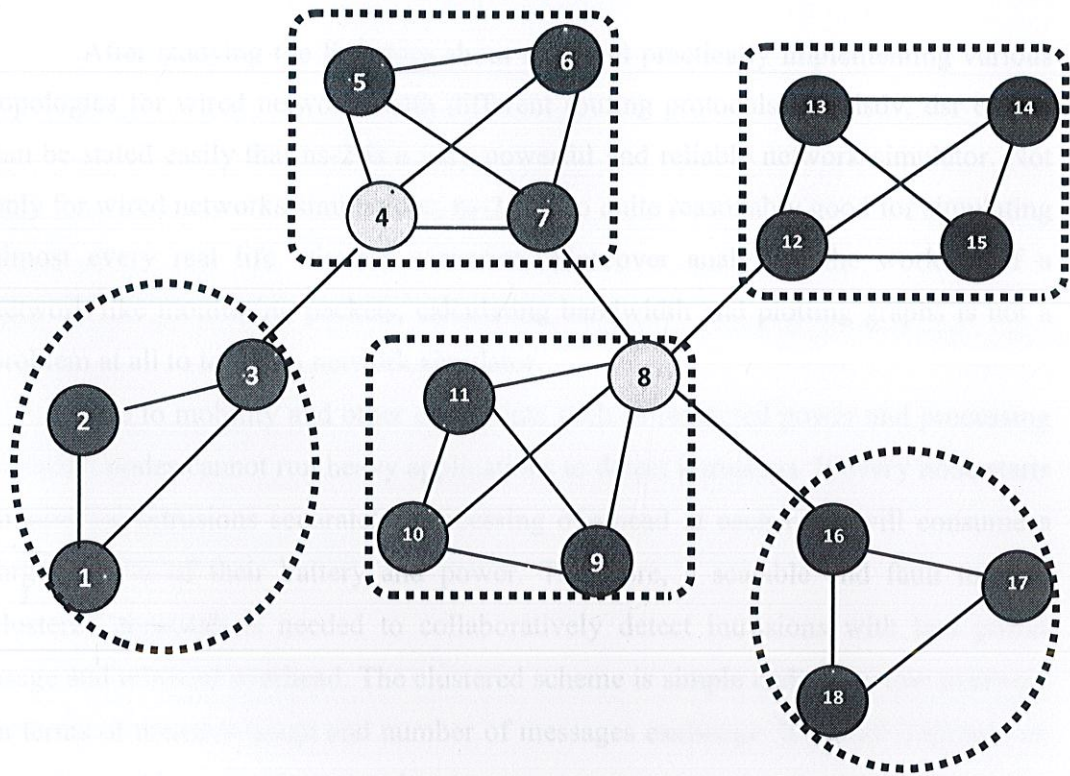





Fig: 5.3: A Scenario of 18 nodes with 5 clusters

Notations used

-  Head Node
-  Member Node
-  Gateway Node

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

After studying the literature about ns-2 and practically implementing various topologies for wired networks with different routing protocols like dsdv, dsr etc., it can be stated easily that ns-2 is a very powerful and reliable network simulator. Not only for wired networks simulations, ns-2 is also quite reasonably good for simulating almost every real life wireless scenarios. Moreover analyzing the working of a network like monitoring packets, calculating bandwidth and plotting graphs is not a problem at all to tackle in network simulator.

Due to mobility and other constraints such as restricted power and processing capacity, nodes cannot run heavy applications to detect intrusions. If every node starts monitoring intrusions separately, processing overhead at each node will consume a large portion of their battery and power. Therefore, a scalable and fault tolerant clustered approach is needed to collaboratively detect intrusions with less power usage and minimal overhead. The clustered scheme is simple and offers low overhead in terms of memory usage and number of messages exchange. It can be deployed on top of any ad hoc routing protocol.

6.2 Suggestions for Future Work

As far as working in ns-2 is concerned more complex scenarios of wired as well as wireless networks could be simulated and analyzed. Xgraph feature of ns-2 for plotting graphs could be used to compare cbrp with other in-built routing protocols like dsdv, dsr etc.

Clustering could be implemented for an ad-hoc network with dynamic nodes i.e. nodes are free to move completely including the head node and can disappear at

any time like in real life situations. Sybil attack detection and detection of other type of intrusions could also be studied in more detail and implemented using clustering formation or using some different approach like: If mobile nodes cooperating in detecting a Sybil node had more accurate and closely synchronized clocks then it is possible for the trusted nodes to establish their relative positions using the localization scheme. Once the relative position was determined, then each node could record the time that every individual message was received. By then comparing these times, each node could determine the distance from each sender. This could quickly show that different nodes were not close together, ruling out the possibility that they were a Sybil attacker. As the hardware costs decrease, it is likely that many mobile nodes will contain Global Positioning System (GPS) receivers. This will let nodes know their positions down to within a meter. Given just this absolute position information, nodes could collaborate to determine the position of an ad hoc Sybil attacker. In this case, each node could record its position when it received a message. Later comparison of the locations in which the same messages were received by different nodes would quickly indicate if different identities were heard in different locations that were far enough apart to rule out movement between them.

SAMPLE CODES

cbrp.tcl

```
# dsr.tcl
# $Id: dsr.tcl,v 1.10 1998/08/11 14:46:54 dmaltz Exp $

#
=====
# Default Script Options
#
=====

set opt(rt_port) 255
set opt(cc)      "off"           ;# have god check the caches
for bad links?

Agent/CBRP set sport_ 255
Agent/CBRP set dport_ 255

Agent/CBRP set no_of_clusters_ 0
#Agent/CBRP set sport_          0
#Agent/CBRP set dport_          0
#Agent/CBRP set wst0_           6      ; As specified by
Pravin
#Agent/CBRP set perup_          15     ; As given in the paper
(update period)
Agent/CBRP set use_mac_         0      ;# Performance suffers
with this on
Agent/CBRP set be_random_       1      ;# Flavor the
performance numbers :)
#Agent/CBRP set alpha_          0.875  ; 7/8, as in RIP(?)
#Agent/CBRP set min_update_periods_ 1  ; #Missing perups
before linkbreak
Agent/CBRP set myaddr_          0      ;# My address
Agent/CBRP set verbose_         1      ;#
Agent/CBRP set trace_wst_       0      ;#

#
=====
# god cache monitoring

source tcl/ex/timer.tcl
Class CacheTimer -superclass Timer
CacheTimer instproc timeout {} {
    global opt node_
    $self instvar agent;
    $agent check-cache
    $self sched 1.0
}
}
```



```

proc checkcache {a} {
    global cachetimer ns_ ns

    set ns $ns_
    set cachetimer [new CacheTimer]
    $cachetimer set agent $a
    $cachetimer sched 1.0
}

#
=====
=====
Class SRNode -superclass MobileNode

SRNode instproc init {args} {
    global opt ns_ tracefd RouterTrace
    $self instvar cbrp_agent_ dmux_ entry_point_

    eval $self next $args ;# parent class constructor

    set cbrp_agent_ [new Agent/CBRP]
    $cbrp_agent_ ip-addr [$self id]
    $cbrp_agent_ set myaddr_ [$self id]

    puts "Creating node [$self id]";

    if { $RouterTrace == "ON" } {
        # Recv Target
        set rcvT [cmu-trace Recv "RTR" $self]
        $rcvT target $cbrp_agent_
        set entry_point_ $rcvT
    } else {
        # Recv Target
        set entry_point_ $cbrp_agent_
    }

    #
    # Drop Target (always on regardless of other tracing)
    #
    set drpT [cmu-trace Drop "RTR" $self]
    $cbrp_agent_ drop-target $drpT

    #
    # Log Target
    #
    set T [new Trace/Generic]
    $T target [$ns_ set nullAgent_]
    $T attach $tracefd
    $T set src_ [$self id]
    $cbrp_agent_ log-target $T

    $cbrp_agent_ target $dmux_

    # packets to the DSR port should be dropped, since we've
    # already handled them in the DSRAgent at the entry.

```



```

set nullAgent_ [$ns_ set nullAgent_]
$dmux_ install $opt(rt_port) $nullAgent_
#$dmux_ install $opt(rt_port) $cbrp_agent_

# SRNodes don't use the IP addr classifier. The DSRAgent
should
# be the entry point
$self instvar classifier_
set classifier_ "srnode made illegal use of classifier_"
}

SRNode instproc start-cbrp {} {
    $self instvar cbrp_agent_
    global opt;

    $cbrp_agent_ startcbrp
    if {$opt(cc) == "on"} {checkcache $cbrp_agent_}
}

SRNode instproc entry {} {
    $self instvar entry_point_
    return $entry_point_
}

SRNode instproc add-interface {args} {
    # args are expected to be of the form
    # $chan $prop $tracefd $opt(ll) $opt(mac)
    global ns_ opt RouterTrace

    eval $self next $args

    $self instvar cbrp_agent_ ll_ mac_ ifq_

    $cbrp_agent_ mac-addr [$mac_(0) id]

    if { $RouterTrace == "ON" } {
        # Send Target
        set sndT [cmu-trace Send "RTR" $self]
        $sndT target $ll_(0)
        $cbrp_agent_ add-ll $sndT $ifq_(0)
    } else {
        # Send Target
        $cbrp_agent_ add-ll $ll_(0) $ifq_(0)
    }

    # setup promiscuous tap into mac layer
    #$cbrp_agent_ install-tap $mac_(0)
}

SRNode instproc reset args {
    $self instvar cbrp_agent_

```



```

eval $self next $args

$cbrp_agent_ reset
}

#
=====

proc create-mobile-node { id } {
    global ns_ chan prop topo tracefd opt node_
    global chan prop tracefd topo opt

    set node_($id) [new SRNode]

    set node $node_($id)
    $node random-motion 0           ;# disable random motion
    $node topography $topo

    # connect up the channel
    $node add-interface $chan $prop $opt(ll) $opt(mac) \
        $opt(ifq) $opt(ifqlen) $opt(netif) $opt(ant)

    #
    # This Trace Target is used to log changes in direction
    # and velocity for the mobile node and log actions of the
DSR agent
    #
    set T [new Trace/Generic]
    $T target [$ns_ set nullAgent_]
    $T attach $tracefd
    $T set src_ $id
    $node log-target $T

    $ns_ at 0.0 "$node start-cbrp"
}

```

cbrp_Packet.h

```

/*
    cbrp_packet.h
    Jinyang 6/9/99 modified from dmaltz' srpacket.h from DSR
*/

#ifndef _cbrp_packet_h_
#define _cbrp_packet_h_

#include <packet.h>
#include "hdr_cbrp.h"
#include "path.h"

struct CBRP_Packet {

```



```

ID dest;
ID src;
Packet *pkt; /* the inner NS packet */
Path route;
CBRP_Packet(Packet *p, struct hdr_cbrp *cbrph) : pkt(p) {
    if (cbrph->valid() && (cbrph->num_addrs() > 0)) {
        route.setLength(cbrph->num_addrs());
        for (int i=0;i<cbrph->num_addrs();i++) {
            route[i] = ID(cbrph->addrs[i]);
        }
        if (cbrph->cur_addr()>=0) {
            route.setIterator(cbrph->cur_addr());
        }
    }else {
        route.reset();
    }
}
CBRP_Packet() : pkt(NULL) {}
};

#endif _cbrp_packet_h_

```

cbrpagent.h

```

/* cbrp.h
*/
#ifndef cbrp_h_
#define cbrp_h_

#include <agent.h>
#include <ip.h>
#include <delay.h>
#include <scheduler.h>
#include <queue.h>
#include <trace.h>
#include "config.h"
#include "scheduler.h"
#include "queue.h"
#include <cmu/arp.h>
#include <cmu/ll.h>
#include <cmu/mac.h>
#include <cmu/priqueue.h>

//cbrp's request table, route cache implementation are all
from DSR.
//:) thanks david!
#include <cmu/dsr/path.h>
#include <cmu/dsr/routecache.h>
#include <cmu/dsr/requesttable.h>

#include "cbrp_packet.h"
#include "ntable.h"

```



```

typedef double Time;
typedef unsigned int uint;

#define MAX_QUEUE_LENGTH 5
#define ROUTER_PORT      0xff

/* The below set of constants are taken from DSR's
implementation, I think they work fine */
#define BUFFER_CHECK 0.03      // seconds between buffer
checks
#define RREQ_JITTER 0.010     // seconds to jitter broadcast
route requests
#define SEND_TIMEOUT 30.0     // # seconds a packet can live
in sendbuf
#define SEND_BUF_SIZE 64
#define RTREP_HOLDOFF_SIZE 10

#define GRAT_ROUTE_ERROR 0

class CBRP_Agent;
class NeighborTablePeriodicHandler;
class NeighborTableTimeoutHandler;
class NeighborTableCFormationHandler;
class ntable_ent;
class adjtable_ent;
class nexthop_ent;

struct CBRP_RtRepHoldoff {      // Taken from DSR's
implementation
    ID requestor;
    ID requested_dest;
    int best_length;
    int our_length;
};

struct CBRP_SendBufEntry {
    Time t;                    // insertion time
    CBRP_Packet p;
};

class CBRP_SendBufferTimer : public TimerHandler { //Taken
from DSR's implementation
public:
    CBRP_SendBufferTimer(CBRP_Agent *a) : TimerHandler() { a_ =
a;}
    void expire(Event_*e);
protected:
    CBRP_Agent *a_;
};

class CBRP_Agent : public Tap, public Agent {

friend class NeighborTable;
friend class NeighborTablePeriodicHandler;

```



```

friend class NeighborTableTimeoutHandler;
friend class NeighborTableCFormationHandler;
friend class NeighborTableCContentionHandler;
friend class CBRP_SendBufferTimer;

public:
    CBRP_Agent();
    virtual int command(int argc, const char * const * argv);
    virtual void recv(Packet *, Handler* callback = 0);
    void Terminate(void);
    void lost_link(Packet *p);
    static int no_of_clusters_ ; // Jinyang It prints out
    how many clusters that have been formed in total

    void tap(const Packet *p);

protected:
    void process_cluster_update(Packet * p);
    void forwardSRPacket(Packet *);
    void startUp();

    void trace(char* fmt, ...);
    void tracepkt(Packet *, double, int, const char *);

    Trace *logtarget; // Trace Target
    PriorityQueue *ll_queue; // link level output queue

    int myaddr; // My address...
    int be_random;

    NeighborTable *ntable;

    // Randomness/MAC/logging parameters used in DSR's
    implementation
    int use_mac;
    int verbose;
    int trace_wst;

private:
    int off_mac;
    int off_ll;
    int off_ip;
    int off_cbrp;

    ID net_id, MAC_id; // our IP addr and MAC addr
    NsObject *ll; // our link layer output
    PriorityQueue *ifq; // output interface queue

    /***** internal state *****/
    CBRP_SendBufferTimer send_buf_timer;
    CBRP_SendBufEntry send_buf[SEND_BUF_SIZE];
    RequestTable request_table;

```



```

RouteCache *route_cache;

int route_request_num;           // number for our next
route_request

//below codes r inherited from DSR for dealing with route
errors
bool route_error_held; // are we holding a rt err to
propagate?
ID err_from, err_to; // data from the last route err
sent to us
Time route_error_data_time; // time err data was filled in

void handlePktWithoutSR(CBRP_Packet& p, bool retry);
void handlePacketReceipt(CBRP_Packet& p);
void handleForwarding(CBRP_Packet& p);
void handleRREQ(CBRP_Packet &p);
int handleRREP(CBRP_Packet &p);
bool ignoreRouteRequestp(CBRP_Packet& p);
void sendOutPacketWithRoute(CBRP_Packet& p, bool fresh,
Time delay = 0.0);
void sendOutRtReq(CBRP_Packet &p, int max_prop =
MAX_SR_LEN);
void getRouteForPacket(CBRP_Packet &p, bool retry);
void acceptRREP(CBRP_Packet &p);
void returnSrcRouteToRequestor(CBRP_Packet &p);
bool replyFromRouteCache(CBRP_Packet &p);
void processBrokenRouteError(CBRP_Packet& p);
void xmitFailed(Packet *pkt);
void undeliverablePkt(Packet *p, int mine);
void dropSendBuff(CBRP_Packet &p);
void stickPacketInSendBuffer(CBRP_Packet& p);
void sendBufferCheck();
void sendRouteShortening(CBRP_Packet &p, int heard_at, int
xmit_at);
void BroadcastRREQ(CBRP_Packet &p);
int UnicastRREQ(CBRP_Packet &p, int nextcluster);
void fillCBRPPath(Path &path, hdr_cbrp *&cbrph);

void testinit();
friend void CBRP_XmitFailureCallback(Packet *pkt, void
*data);
friend int CBRP_FilterFailure(Packet *p, void *data);

};

#endif

```

cbrpagent.cc


```

/* cbrp.cc
   */

extern "C" {
#include <stdarg.h>
#include <float.h>
};

#include "cbrpagent.h"
#include "priqueue.h"

#include <random.h>

#include <cmu/cmu-trace.h>
#include <cmu/marshall.h>
#include <packet.h>
#include <cmu/mac-802_11.h>
#define CBRP_ALMOST_NOW      0.1 // jitter used for events that
should be effectively
prevent                      // instantaneous but are jittered to
                              // synchronization
#define CBRP_MIN_TUP_PERIOD 1.0 // minimum time between
triggered updates
#define CBRP_IP_DEF_TTL    32 // default TTL
#define MAX_LOCAL_REPAIR_TIMES 5

static const int verbose_srr = 0;

//the following codes are from DSR's implementation -Jinyang
Time cbrp_arp_timeout = 30.0e-3; // (sec) arp request
timeout
Time cbrp_rt_rq_period = 0.5; // (sec) length of one
backoff period
Time cbrp_rt_rq_max_period = 10.0; // (sec) maximum time
between rt reqs
Time cbrp_max_err_hold = 1.0; // (sec)

/***** selectors *****/

bool cbrp_shortening_route_if_possible = true;
bool cbrp_local_repair = true;
bool cbrp_snoop_forwarded_errors = true; //same as DSR's snoop
error

//with CBRP, you do get much benefit from snooping other
nodes' source
//routes as DSR, netherless, we include this for people to try
out themselves
bool cbrp_snoop_source_routes = false;

// should we only respond to the first route request we
receive from a host?
bool cbrp_reply_only_to_first_rtreq = false;

```



```

// should we take the data from the last route error msg sent
to us
// and propagate it around on the next propagating route
request we do?
// this is aka grat route error propagation DSR uses this
// to check invalid entries in route cache. As most of route
replies in DSR
// comes from route cache, keeping route cache as updated as
possible is
// crucial to DSR's performance. In CBRP, most route replies
comes from
//destination node itself and hence does not depend too much
on route cache.
bool cbrp_propagate_last_error = false;

// Should CBRP save the packet if the Local Repair procedure
fails?
bool cbrp_salvage_with_cache = true;

// should we listen to a promiscuous tap?
// Unlike DSR, most of CBRP's RREP comes from destination node
instead of
// cache and hence doing eavesdropping does no significant
improvements.
bool cbrp_use_tap = false;

// CBRP does not use cache to reply when propagating RREQ.
bool cbrp_reply_from_cache_on_propagating = false;

// Used only when a node is eavesdropping.
bool cbrpagent_reply_from_cache_uninvited = false;

// Returns a random number between 0 and max, from dmaltz
static inline double
jitter (double max, int be_random_)
{
    return (be_random_ ? Random::uniform(max) : 0);
}

#ifdef CBRP_DEBUG
static char*
return_reply_path(Packet *p, int off_cbrp_)
{
    hdr_cbrp *cbrph = (hdr_cbrp *)p->access(off_cbrp_);
    static char buf[100];
    char *ptr = buf;
    char *rtn_buf = ptr;
    int len = cbrph->route_reply_len();

    if (len== 0)
    {
        sprintf(rtn_buf, "[<empty path>]");
        return rtn_buf;
    }
    *ptr++ = '[';

```



```

for (int c = 0 ; c < len ; c ++ )
{
    ptr += sprintf(ptr, "%d ", cbrph->reply_addrs()[c].addr);
}
*ptr++ = ']';
*ptr++ = '\\0';
return rtn_buf;
}

```

```

static char*
return_RREQ_forward_nodes(Packet *p, int off_cbrp_)
{
    hdr_cbrp *cbrph = (hdr_cbrp *)p->access(off_cbrp_);
    static char buf[100];
    char *ptr = buf;
    char *rtn_buf = ptr;
    int len = cbrph->num_forwarders();

    if (len== 0)
    {
        sprintf(rtn_buf, "[<empty forwarders>]");
        return rtn_buf;
    }
    *ptr++ = '[';
    for (int c = 0 ; c < len ; c+=2)
    {
        ptr += sprintf(ptr, "%d(%d) ", cbrph-
>forwarders[c].which_head, cbrph->forwarders[c].which_gateway);
    }
    *ptr++ = ']';
    *ptr++ = '\\0';
    return rtn_buf;
}
#endif //CBRP_DEBUG

```

//the below two functions are from DSR's implementation -
Jinyang :)

```

void CBRP_Agent:: trace (char *fmt, ...)
{
    va_list ap;

    if (!logtarget)
        return;

    va_start (ap, fmt);
    vsprintf (logtarget->buffer (), fmt, ap);
    logtarget->dump ();
    va_end (ap);
}

```

```

void
CBRP_Agent::tracepkt (Packet * p, double now, int me, const
char *type)
{
    char buf[1024];

```



```

unsigned char *walk = p->accessdata ();

int ct = *(walk++);
int seq, dst, met;

snprintf (buf, 1024, "V%s %.5f _%d_ [%d]:", type, now, me,
ct);
while (ct--)
{
    dst = *(walk++);
    met = *(walk++);
    seq = *(walk++);
    seq = seq << 8 | *(walk++);
    seq = seq << 8 | *(walk++);
    seq = seq << 8 | *(walk++);
    snprintf (buf, 1024, "%s (%d,%d,%d)", buf, dst, met,
seq);
}
// Now do trigger handling.
//trace("VTU %.5f %d", now, me);
if (verbose_)
    trace ("%s", buf);
}

```

```

void
CBRP_Agent::recv(Packet * p, Handler *)
{
    hdr_ip *iph = (hdr_ip *) p->access (off_ip_);
    hdr_cmn *cmh = (hdr_cmn *) p->access (off_cmn_);
    hdr_cbrp *cbrph = (hdr_cbrp*)p->access(off_cbrp_);

    /*
     * Must be a packet I'm originating...
     */
    if(iph->src_ == myaddr_ && cmh->num_forwards() == 0) {
        iph->ttl_ = CBRP_IP_DEF_TTL;
    }
    /*
     * Packet I'm forwarding...
     */
    else {
        /*
         * Check the TTL. If it is zero, then discard.
         */
        if(--iph->ttl_ == 0) {
            drop(p, DROP_RTR_TTL);
            return;
        }
    }
}

```

```

if ((iph->src_ != myaddr_) && (!cbrph->valid()))
{

```



```

        // process a BROADCAST pkt..
        ntable->processUpdate(p);
    }
else
    {
        // process a CBRP source route pkt, RREP, RREQ, ERR pkt.
        forwardSRPacket(p);
    }
}

void
CBRP_Agent::tap(const Packet *packet)
/* CBRP does not need promiscuous mode, this is just for
interested
party to test out
The implementation is mostly taken from DSR -Jinyang*/
{
    hdr_cbrp *cbrph = (hdr_cbrp*) ((Packet *)packet)-
>access(off_cbrp_);
    hdr_ip *iph = (hdr_ip*) ((Packet *)packet)->access(off_ip_);
    hdr_cmh *cmh = (hdr_cmh*) ((Packet *)packet)-
>access(off_cmh_);

    if (!cbrp_use_tap) return;

    if (!cbrph->valid()) return;    // can't do anything with it

    // don't trouble me with packets I'm about to receive anyway
    /* this change added 5/13/98 -dam */
    ID next_hop(cbrph->addrs[cbrph->cur_addr()].addr, ::IP);
    if (next_hop == net_id || next_hop == MAC_id) return;

    CBRP_Packet p((Packet *) packet, cbrph);
    p.dest = ID(iph->dst(), ::IP);
    p.src = ID(iph->src(), ::IP);

    if (p.src == net_id) return;

    /* snoop on the errori */
    if (cbrph->route_error())
    {
        if (verbose_)
            trace("Sdebug _%s_ tap saw error %d", net_id.dump(),
cmh->uid());
        processBrokenRouteError(p);
    }

    if (cbrph->route_reply())
    {
        Path reply_path(cbrph->reply_addrs(), cbrph-
>route_reply_len());
        reply_path.reverseInPlace();
    }
}

```



```

        if(verbose_)
            trace("Sdebug _%s_ tap saw route reply %d %s",
                  net_id.dump(), cmh->uid(), reply_path.dump());
        route_cache->noticeRouteUsed(reply_path,
Scheduler::instance().clock(),
                                      p.src);

        return;
    }

    if (cbrph->route_request() && ((unsigned long)iph->dst()) !=
IP_BROADCAST ) {
        if (cbrpagent_reply_from_cache_uninvited) {
            Packet *p_copy = p.pkt->copy();
            p.pkt = p_copy;
            //this copying is because replyFromRouteCache will free
the pkt which we dont want
            replyFromRouteCache(p);
        }
        }else if (cbrph->route_request() && ((unsigned long)iph-
>dst()) == IP_BROADCAST) {
            return;
        }

        // the logic is wrong for shortening rtreq's anyway,
cur_addr always = 0

        if (cbrp_snoop_source_routes)
            {
                if (verbose_)
                    trace("Sdebug _%s_ tap saw route use %d %s",
net_id.dump(),
                        cmh->uid(), p.route.dump());
                route_cache->noticeRouteUsed(p.route,
Scheduler::instance().clock(), net_id);
            }
    }

static class CBRPClass:public TclClass
{
public:
    CBRPClass ():TclClass ("Agent/CBRP")
    {
    }
    TclObject *create (int, const char *const *)
    {
        return (new CBRP_Agent ());
    }
} class_cbrp;

CBRP_Agent::CBRP_Agent (): Agent (PT_CBRP), ll_queue (0),
myaddr_ (0), be_random_ (1),
use_mac_ (0), verbose_ (1), trace_wst_ (0),
send_buf_timer(this),request_table(128),route_cache(NULL)
{

```



```

ntable = new NeighborTable(this);
route_cache = makeRouteCache();

route_request_num = 1;

target_ = 0;
logtarget = 0;

bind("off_CBRP_", &off_cbrp_);
bind("off_ll_", &off_ll_);
bind("off_mac_", &off_mac_);
bind("off_ip_", &off_ip_);

route_error_held = false;

bind ("use_mac_", &use_mac_);
bind ("be_random_", &be_random_);
bind ("verbose_", &verbose_);
bind ("trace_wst_", &trace_wst_);
bind ("no_of_clusters_", &no_of_clusters_);
}

void
CBRP_Agent::startUp()
{
    // kick off periodic advertisements
    ntable->startUp();
}

/* This command function is mainly taken from DSR
implementation */
int
CBRP_Agent::command (int argc, const char *const *argv)
{
    TclObject *obj;

    if (argc == 2)
    {
        if (strcmp (argv[1], "startcbrp") == 0)
        {
            startUp();
            send_buf_timer.sched(BUFFER_CHECK
                                + BUFFER_CHECK *
Random::uniform(1.0));
            return (TCL_OK);
        }
        else if (strcmp (argv[1], "testinit") == 0)
        {
            testinit();
            return (TCL_OK);
        }
        else if (strcmp (argv[1], "reset") == 0)
        {
            Terminate();
        }
    }
}

```



```

        return Agent::command(argc, argv);
    }
    else if (strcasecmp(argv[1], "check-cache") == 0)
    {
        return route_cache->command(argc, argv);
    }
    else if (strcasecmp (argv[1], "ll-queue") == 0)
    {
        if (!(ll_queue = (PriQueue *) TclObject::lookup
failed\n", argv[2]))
        {
            fprintf (stderr, "CBRP_Agent: ll-queue lookup of %s
failed\n", argv[2]);
            return TCL_ERROR;
        }

        return TCL_OK;
    }
}
else if (argc == 3)
{
    if (strcasecmp(argv[1], "ip-addr") == 0)
    {
        net_id = ID(atoi(argv[2]), ::IP);
        myaddr_ = net_id.addr; //myadd_ is the IP addr
shortform
        route_cache->net_id = net_id;
        return TCL_OK;
    }
    else if(strcasecmp(argv[1], "mac-addr") == 0)
    {
        MAC_id = ID(atoi(argv[2]), ::MAC);
        route_cache->MAC_id = MAC_id;
        return TCL_OK;
    }

    if( (obj = TclObject::lookup(argv[2])) == 0)
    {
        fprintf(stderr, "CBRP_Agent: %s lookup of %s
failed\n", argv[1],
            argv[2]);
        return TCL_ERROR;
    }
    if (strcasecmp (argv[1], "log-target") == 0)
    {
        logtarget = (Trace *)obj;
        return route_cache->command(argc, argv);
    }
    else if (strcasecmp(argv[1], "install-tap") == 0)
    {
        Mac *m = (Mac*) obj;
        m->installTap(this);
        return TCL_OK;
    }
}

```



```

    }

else if (argc == 4)
    {
        if (strcasecmp(argv[1], "add-ll") == 0)
            {
                if( (obj = TclObject::lookup(argv[2])) == 0) {
                    fprintf(stderr, "CBRP_Agent: %s lookup of %s
failed\n", argv[1],
                        argv[2]);
                    return TCL_ERROR;
                }
                ll = (NsObject*) obj;
                if( (obj = TclObject::lookup(argv[3])) == 0) {
                    fprintf(stderr, "CBRP_Agent: %s lookup of %s
failed\n", argv[1],
                        argv[3]);
                    return TCL_ERROR;
                }
                ifq = (PriQueue *) obj;
                return TCL_OK;
            }
    }

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

/*=====
SendBuf management and helpers -- from cmu/dsr/cbrp.cc by
dmaltz
-----*/

void
CBRP_SendBufferTimer::expire(Event *e)
{
    a_ ->sendBufferCheck();
    resched(BUFFER_CHECK + BUFFER_CHECK * (double) ((int) e>>5 &
0xff) / 256.0);
}

void
CBRP_Agent::dropSendBuff(CBRP_Packet &p)
    // log p as being dropped by the sendbuffer in CBRP agent
{
    trace("Ssb %.5f %s_ dropped %s -> %s",
Scheduler::instance().clock(),
        net_id.dump(), p.src.dump(), p.dest.dump());
    drop(p.pkt, DROP_RTR_QTIMEOUT);
    p.pkt = 0;
    p.route.reset();
}

void
CBRP_Agent::stickPacketInSendBuffer(CBRP_Packet& p)

```



```

{
    Time min = DBL_MAX;
    int min_index = 0;
    int c;

    if (verbose_)
        trace("Sdebug %.5f %s_ stuck into send buff %s -> %s",
              Scheduler::instance().clock(),
              net_id.dump(), p.src.dump(), p.dest.dump());

    for (c = 0 ; c < SEND_BUF_SIZE ; c ++)
        if (send_buf[c].p.pkt == NULL)
            {
                send_buf[c].t = Scheduler::instance().clock();
                send_buf[c].p = p;
                return;
            }
        else if (send_buf[c].t < min)
            {
                min = send_buf[c].t;
                min_index = c;
            }

    // kill somebody
    dropSendBuff(send_buf[min_index].p);
    send_buf[min_index].t = Scheduler::instance().clock();
    send_buf[min_index].p = p;
}

void
CBRP_Agent::sendBufferCheck()
    // see if any packets in send buffer need route requests
sent out
    // for them, or need to be expired
{ // this is called about once a second.  run everybody
through the
    // get route for pkt routine to see if it's time to do
another
    // route request or what not
    int c;

    for (c = 0 ; c < SEND_BUF_SIZE ; c++)
        {
            if (send_buf[c].p.pkt == NULL) continue;
            if (Scheduler::instance().clock() - send_buf[c].t >
SEND_TIMEOUT)
                {
                    dropSendBuff(send_buf[c].p);
                    send_buf[c].p.pkt = 0;
                    continue;
                }
        }
#ifdef DEBUG
    trace("Sdebug %.5f %s_ checking for route for dst %s",
          Scheduler::instance().clock(), net_id.dump(),
          send_buf[c].p.dest.dump());
#endif
}

```



```

#endif
    handlePktWithoutSR(send_buf[c].p, true);
#ifdef DEBUG
    if (send_buf[c].p.pkt == NULL)
        trace("Sdebug %.5f _%s_ sendbuf pkt to %s liberated by
handlePktWOSR",
            Scheduler::instance().clock(), net_id.dump(),
            send_buf[c].p.dest.dump());
#endif
    }
}

/*=====
Route Request backoff -dmaltz
-----*/
static bool
BackOffTest(Entry *e, Time time)
// look at the entry and decide if we can send another route
// request or not.  update entry as well
{
    Time next = ((Time) (0x1<<(e->rt_reqs_outstanding*2))) *
cbrp_rt_rq_period;
    if (next > cbrp_rt_rq_max_period) next =
cbrp_rt_rq_max_period;
    if (next + e->last_rt_req > time) return false;
    // don't let rt_reqs_outstanding overflow next on the
LogicalShiftsLeft's
    if (e->rt_reqs_outstanding < 15) e->rt_reqs_outstanding++;
    e->last_rt_req = time;
    return true;
}

void
CBRP_Agent::Terminate()
{
    int c;
    for (c = 0 ; c < SEND_BUF_SIZE ; c++) {
        if (send_buf[c].p.pkt) {
            drop(send_buf[c].p.pkt,
DROP_END_OF_SIMULATION);
            send_buf[c].p.pkt = 0;
        }
    }
}

void
CBRP_Agent::testinit()
{
    struct hdr_cbrp hsr;

    if (net_id == ID(1,::IP))
    {
        printf("adding route to 1\n");
        hsr.init();
    }
}

```



```

    hsr.append_addr( 1, AF_INET );
    hsr.append_addr( 2, AF_INET );
    hsr.append_addr( 3, AF_INET );
    hsr.append_addr( 4, AF_INET );

    route_cache->addRoute(Path(hsr.addrs, hsr.num_addrs()),
0.0, ID(1,::IP));
}

if (net_id == ID(3,::IP))
{
    printf("adding route to 3\n");
    hsr.init();
    hsr.append_addr( 3, AF_INET );
    hsr.append_addr( 2, AF_INET );

    route_cache->addRoute(Path(hsr.addrs, hsr.num_addrs()),
0.0, ID(3,::IP));
}
}

/*-----from here onwards, we deal with more CBRP
specific stuff---*/

void
CBRP_Agent::forwardSRPacket(Packet* packet)
{
    hdr_cbrp *cbrph = (hdr_cbrp*)packet->access(off_cbrp_);
    hdr_ip *iph = (hdr_ip*)packet->access(off_ip_);
    hdr_cmn *cmh = (hdr_cmn*)packet->access(off_cmn_);

    assert(cmh->size() >= 0);

    CBRP_Packet p(packet, cbrph);
    p.dest = ID(iph->dst(),::IP);
    p.src = ID(iph->src(),::IP);

    assert(logtarget != 0);

    if (!cbrph->valid())
    {
        // this must be an outgoing packet, it doesn't have a SR
header on it
        cbrph->init(); // give packet an SR header
now
        if (verbose_)
            trace("S %.9f %s_ originating %s -> %s",
Scheduler::instance().clock(), net_id.dump(),
p.src.dump(),
p.dest.dump());
        cmh->size() += IP_HDR_LEN;
        cmh->ptype() = PT_CBR;
        handlePktWithoutSR(p, false);
        goto done;
    }
}

```



```

    }

    if ( ((p.dest == net_id) && (!cbrph->route_request()))
        || ((p.dest == IP_broadcast) && (!cbrph-
>route_request()))
        || (cbrph->route_request() && ((unsigned
long)cbrph->request_destination() == net_id.addr) )
    {
        //this is either a route_reply for my previously sent-
out RREQ or a data packet for me
        //or a route_request searching for me
        handlePacketReceipt(p);
        goto done;
    }

    // should we check to see if it's an error packet we're
handling
    // and if so call processBrokenRouteError to snoop
    if (cbrp_snoop_forwarded_errors && cbrph->route_error())
    {
        processBrokenRouteError(p);
    }

    if (cbrph->route_request())
    {
        // propagate a route_request that's not for us
        //if I am not a cluster head, forward the RREQ as
indicated to p.dest
        //if I am cluster head, broadcast to other un-visited
neighboring clusters
        handleRREQ(p);
    }
    else if (!cbrph->route_repaired() && !cbrph-
>route_shortened() && cbrph->route_reply())
    {
        //if I am cluster head, i have to calculate part of the
RREP
        //if I am ordinary node, record myself in RREP and
forward it to the next cluster head
        handleRREP(p);
    }
    else
    { // we're not the intended final receipt, but we're a
hop along the route
        // since repaired route is hop by hop, we will use
normal data packet forwarding
        // function to handle it
        handleForwarding(p);
    }
}

done:
    assert(p.pkt == 0);
    return;
}

```



```

/*-----
-----
different packet handling functions,
-----
-----*/
void
CBRP_Agent::handlePktWithoutSR(CBRP_Packet& p, bool retry)
/* obtain a source route to p's destination and send it off.
this should be a retry if the packet is already in the
sendbuffer */
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    hdr_ip *iph = (hdr_ip*)p.pkt->access(off_ip_);
    hdr_cmn *ch = (hdr_cmn*)p.pkt->access(off_cmn_);

    assert(cbrph->valid());
    assert(iph->src() == myaddr_);
    if (p.dest == net_id)
    {
        // it doesn't need a source route, 'cause it's for us
        handlePacketReceipt(p);
        return;
    }

    //firstly, try our route cache to see if the route has been
    discovered
    if (route_cache->findRoute(p.dest, p.route, 1))
    {
        // lucky! we've got a route...
        if (verbose_)
            trace("S$hit %.5f _%s_ %s -> %s %s",
                Scheduler::instance().clock(), net_id.dump(),
                p.src.dump(), p.dest.dump(), p.route.dump());
        sendOutPacketWithRoute(p, true);
        return;
    }

    //secondly, examine neighborhood table to see if destination
    node is our neighbor
    if (ntable->isNeighbor(p.dest.addr))
    {
        p.route.reset();
        p.route.appendToPath(net_id);
        p.route.appendToPath(p.dest);
        if (verbose_) {
            trace("S$direct hit %.5f _%s_ %d -> %s %s",
                Scheduler::instance().clock(), net_id.dump(),
                iph->src(), p.dest.dump(), p.route.dump());
        }
        sendOutPacketWithRoute(p, true);
        return;
    }
}

```



```

//thirdly, we'll check using two-hop-neighbor table if we
could reach
// the destination using only one intermediate node
if (ntable->GetQuickNextNode(p.dest.addr))
{
    p.route.reset();
    p.route.appendToPath(net_id);
    p.route.appendToPath(ID(ntable-
>GetQuickNextNode(p.dest.addr), ::IP));
    p.route.appendToPath(p.dest);

    if (verbose_) {
        trace("S$direct hit %.5f _%s_ %d -> %s %s",
            Scheduler::instance().clock(), net_id.dump(),
            iph->src(), p.dest.dump(), p.route.dump());
    }
    sendOutPacketWithRoute(p, true);
    return;
}

// We have to send out a RREQ to discover the destination

if (verbose_)
    trace("S$miss %.5f _%s_ %s -> %s retry %d",
        Scheduler::instance().clock(), net_id.dump(),
        net_id.dump(), p.dest.dump(), retry?1:0);
getRouteForPacket(p, retry);
}

void
CBRP_Agent::handlePacketReceipt(CBRP_Packet& p)
/* Handle a packet destined to us */
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);

    if (cbrph->route_reply())
    {
        //we've got a route_reply!
        acceptRREP(p);
    }

    if (cbrph->route_request())
    {
        assert((unsigned long)cbrph->request_destination() ==
net_id.addr || net_id == p.dest);
        if (cbrp_reply_only_to_first_rtreq &&
ignoreRouteRequestp(p))
        { //we only respond to the first route request
            // we receive from a host
            Packet::free(p.pkt); // drop silently
            p.pkt = 0;
            return;
        }
        else

```



```

        { // we're going to process this request now, so
record the req_num
        request_table.insert(p.src, p.src, cbrph-
>rtreq_seq());
        //initiate a RREP in reply to the RREQ
        returnSrcRouteToRequestor(p);
    }
}

if (cbrph->route_error())
{ // register the dead route
    processBrokenRouteError(p);
}

if (cbrph->route_shortening()) {
    //the previous route shortening attempt is successful,
shorten the route
    for (int c = (cbrph->cur_addr()-1); c < cbrph-
>num_addrs()-1 ; c++)
    {
        p.route[c].addr = p.route[c+1].addr;
    }
    p.route.setLength(cbrph->num_addrs()-1);
    cbrph->route_shortened() = 1;
    cbrph->route_shortening() = 0;
}

if (!cbrph->route_reply() && (unsigned long)cbrph-
>addrs[0].addr == p.src.addr
    && (cbrph->route_shortened() || cbrph-
>route_repaired())) {

    CBRP_Packet p_copy = p;
    p_copy.pkt = allocpkt();
    p_copy.dest = p.src;
    p_copy.src = net_id;

    hdr_ip *new_iph = (hdr_ip*)p_copy.pkt->access(off_ip_);
    hdr_cbrp *new_cbrph = (hdr_cbrp*)p_copy.pkt-
>access(off_cbrp_);

    new_cbrph->init();
    new_cbrph->route_repaired() = cbrph->route_repaired();
    new_cbrph->route_shortened() = cbrph->route_shortened();
    new_cbrph->route_reply() = 1;

    new_iph->src() = net_id.addr;
    new_iph->dst() = p.src.addr;
    new_iph->dport() = RT_PORT;
    new_iph->sport() = RT_PORT;
    new_iph->ttl() = 255;

    p_copy.route.reverseInPlace();
}

```



```

    p_copy.route.resetIterator();
    fillCBRPPath(p_copy.route, new_cbrph);

    hdr_cmn *new_cmnh = (hdr_cmn*)p_copy.pkt-
>access(off_cmn_);
    new_cmnh->ptype() = PT_CBRP;
    sendOutPacketWithRoute(p_copy, 1);

}

target_->recv(p.pkt, (Handler*)0);
p.pkt = 0;
}

void
CBRP_Agent::handleForwarding(CBRP_Packet &p)
/* forward packet on to next host in source route,
snooping as appropriate */
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    hdr_cmn *cmh = (hdr_cmn*)p.pkt->access(off_cmn_);

    trace("cbrp %.9f _%s_ --- %d [%s -> %s] %s",
Scheduler::instance().clock(), net_id.dump(), cmh-
>uid(),
        p.src.dump(), p.dest.dump(), cbrph->dump());

    // first make sure we are the ``current'' host along the
source route.
    // if we're not, the previous node set up the source route
incorrectly.
    assert(p.route[p.route.index()] == net_id ||
p.route[p.route.index()] == MAC_id);

    if (p.route.index() >= p.route.length())
    {
        fprintf(stderr, "dfu: ran off the end of a source
route\n");
        trace("SDFU: ran off the end of a source route\n");
        drop(p.pkt, DROP_RTR_ROUTE_LOOP);
        p.pkt = 0;
        // maybe we should send this packet back as an error...
        return;
    }

    // if there's a source route, maybe we should snoop it too,
does not do it by default
    if (cbrp_snoop_source_routes)
        route_cache->noticeRouteUsed(p.route,
Scheduler::instance().clock(),
                                net_id);

    // sendOutPacketWithRoute will add in the size of the src
hdr, so
    // we have to subtract it out here

```



```

struct hdr_cmn *ch = HDR_CMN(p.pkt);

if (cbrph->route_shortening()) {
    //the previous node has successfully shortened one node
    for (int c = (cbrph->cur_addr()-1); c < cbrph-
>num_addrs()-1 ; c++)
    {
        cbrph->addrs[c].addr = cbrph->addrs[c+1].addr;
        p.route[c].addr = p.route[c+1].addr;
    }
    cbrph->cur_addr() -= 1;
    p.route.setIterator(cbrph->cur_addr());
    cbrph->num_addrs() -= 1;
    p.route.setLength(cbrph->num_addrs());
    cbrph->route_shortening() = 0;
    cbrph->route_shortened() = 1;
    trace("CBRP %.5f %d success-shorten %s",
Scheduler::instance().clock(),myaddr_, p.route.dump());
}
ch->size() -= cbrph->size();
sendOutPacketWithRoute(p, false);
}

int
CBRP_Agent::handleRREP(CBRP_Packet &p)
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    hdr_cmn *cmnh = (hdr_cmn*)p.pkt->access(off_cmn_);

    cmnh->addr_type() = AF_INET;

    cmnh->xmit_failure_ = CBRP_XmitFailureCallback;
    cmnh->xmit_failure_data_ = (void *) this;

    //just a cautionary step...
    if (p.route.length()<=0) {

#ifdef CBRP_DEBUG
        printf("ERROR route len %d\n", p.route.length());
        printf("p.src %s, p.dest %s, me %s, route-reply %s\n",
p.src.dump(),p.dest.dump(),net_id.dump(),return_reply_path(p.p
kt, off_cbrp_));
#endif
        abort();
    }

    if ((cbrph->cur_addr() >= 0) && (p.route[p.route.index()] ==
net_id) && (ntable->my_status == CLUSTER_HEAD)) {
        // I am the cluster head and have to calculate the route
        cbrph->cur_addr()--;

        if (cbrph->cur_addr()<0) {

```



```

src
    //i am the first cluster head, should be able to reach
    if (ntable->isNeighbor(p.dest.getNSAddr_t())) {
        cmnh->next_hop() = p.dest.getNSAddr_t();
    }else {
        if (!(cmnh->next_hop()==ntable-
>GetQuickNextNode(p.dest.getNSAddr_t())) {
            trace("CBRP %.5f _%d_ drop-reply-no-route unable
return route to %s", Scheduler::instance().clock(),
myaddr_, p.dest.dump());
            Packet::free(p.pkt);
            p.pkt = NULL;
            return 0;
        }
    }

    //if there is a link between next hop and the previous
    calculated hop, i am bypassed.
    //otherwise include me in the calculated route.
    //Right now, the head won't search in its two-hop-
    topology table
    //for a non-cluster-head guy to take off its work as the
    current 2-hop topology table
    //implementation is not complete. A more efficient data
    structure
    //has to be implemented to speed up the search. :)
    if (!ntable->existsLink(cmnh->next_hop(),
        (cbrph->reply_addrs())[cbrph-
>route_reply_len()-1]).addr) {
        (cbrph->reply_addrs())[cbrph->route_reply_len()].addr
= net_id.addr;
        (cbrph->reply_addrs())[cbrph-
>route_reply_len()].addr_type = AF_INET;
        cbrph->route_reply_len()++;
    }
    cmnh->prev_hop_ = net_id.getNSAddr_t();

    cmnh->size() = cbrph->size();
    Scheduler::instance().schedule(11, p.pkt, 0);
    goto done;
}

    // I am not the first cluster head, I'll try to reach the
    prev. cluster head
    nsaddr_t prev_cluster_ = (cbrph->addrs)[cbrph-
>cur_addr_].addr;
    nsaddr_t next_node_ = (cbrph->reply_addrs())[cbrph-
>route_reply_len()-1].addr;

    // firstly try to reach the prev_cluster by 2 hops
    adjtable_ent *ent = ntable->GetAdjEntry(prev_cluster,1);

    if (ent == NULL) {

```



```

    //if i could not reach prev_cluster by 2 hops, try 3-hop
    gateway.
    ent = ntable->GetAdjEntry(prev_cluster, 0);
    }

    if (ent == NULL) {
        //I failed to reach prev_cluster
        trace("CBRP %.5f _%s_ drop-reply-no-route #%d (route
reply %s -> %s cannot reach prev cluster %d on route %s)
previous hop %d", Scheduler::instance().clock(),
net_id.dump(), cbrph->rtreq_seq(), p.src.dump(),
p.dest.dump(), prev_cluster, p.route.dump(), cmnh->prev_hop_);
        Packet::free(p.pkt);
        p.pkt = NULL;
        return 0;
    }

    nexthop_ent *next_hop = ent->next_hop;
    assert(next_hop != NULL);

    while (next_hop) {
        //try to search for a gateway node that could be
        directly reached by route's next node
        //therefore, bypassing me.
        if (ntable->existsLink(next_hop->next_node, next_node)) {
            cmnh->next_hop() = next_hop->next_node;
            cmnh->addr_type() = AF_INET;
            break;
        }
        next_hop = next_hop->next;
    }

    if (next_hop == NULL) {
        cmnh->next_hop() = ent->next_hop->next_node;
        assert(cmnh->next_hop()>0);
        // it has to go pass me. -Jinyang
        (cbrph->reply_addrs()[cbrph->route_reply_len()]).addr =
net_id.addr;
        (cbrph->reply_addrs()[cbrph-
>route_reply_len()]).addr_type = AF_INET;
        cbrph->route_reply_len()++;
    }
    cmnh->size() = cbrph->size();
    cmnh->prev_hop_ = net_id.getNSAddr_t();
    Scheduler::instance().schedule(11, p.pkt, 0);

    } else {

        //I am an ordinary node non-CLUSTER_HEAD

        nsaddr_t next_cluster;

        //if I am just an ordinary node, I just forward it to the
        next cluster head
        cmnh->addr_type() = AF_INET;
    }
}

```



```

    if (cbrph->cur_addr_<0) {
        //I am supposed to be two hop away from the destination
node
        next_cluster = p.dest.addr;

        if (ntable->isNeighbor(next_cluster)) {
            cmnh->next_hop() = next_cluster;
        }else {
            cmnh->next_hop() = ntable-
>GetQuickNextNode(next_cluster);
            if (!cmnh->next_hop()) {
                trace("CBRP %.5f _%d_ drop-reply-no-route %s->%s #%d
(route reply cannot reach prev cluster
%d)", Scheduler::instance().clock(), myaddr_, p.src.dump(),
p.dest.dump(), cbrph->rtreq_seq(), next_cluster);
                Packet::free(p.pkt);
                p.pkt = NULL;
                return 0;
            }
        }
    }else {
        //try to reach the next_cluster head.
        next_cluster = cbrph->addrs[cbrph->cur_addr()].addr;
        //check if the neighbor cluster is just one hop away
        if (ntable->isNeighbor(next_cluster)) {
            cmnh->next_hop() = next_cluster;
        } else {
            cmnh->next_hop() = ntable-
>GetQuickNextNode(next_cluster);
            if (!cmnh->next_hop()) {
                trace("CBRP %.5f _%d_ drop-reply-no-route %s->%s #%d
(route reply cannot reach prev cluster %d)",
Scheduler::instance().clock(), myaddr_, p.src.dump(),
p.dest.dump(), cbrph->rtreq_seq(), next_cluster);
                Packet::free(p.pkt);
                p.pkt = NULL;
                return 0;
            }
        }
    }

    //checking for wrong route reply calculation which could
cause looping.
    int i;
    for (i=cbrph->route_reply_len();i>0;i--) {
        if (cbrph->reply_addrs()[i-1].addr == cmnh->next_hop())
    {
        break;
    }
    }
    if (i>0) {
        trace("CBRP %.5f _%d_ drop-reply-looped %s->%s #%d
existing route:%s reaching next cluster %d via

```



```

%d", Scheduler::instance().clock(), myaddr_, p.src.dump(),
p.dest.dump(), cbrph-
>rtreq_seq(), p.route.dump(), next_cluster, cmnh->next_hop());
    Packet::free(p.pkt);
    p.pkt = NULL;
    return 0;
}

//add myself to the route reply
(cbrph->reply_addrs()[cbrph->route_reply_len()-1]).addr =
net_id.addr;
(cbrph->reply_addrs()[cbrph->route_reply_len()-1]).addr_type
= AF_INET;
cbrph->route_reply_len()++;

cmnh->prev_hop_ = net_id.getNSAddr_t();
cmnh->size() = cbrph->size();

//Scheduler::instance().schedule(11, p.pkt,
Random::uniform(RREQ_JITTER));
Scheduler::instance().schedule(11, p.pkt, 0 );
}

done:
#ifdef CBPR_DEBUG
    trace("cbrp_%d_route-reply-sent %s #%d -> %s , next hop %d
to reach %d, rest of recorded cluster heads %s, constructed
return path %s", myaddr_, p.src.dump(), cbrph->rtreq_seq(),
p.dest.dump(), cmnh->next_hop(), (cbrph->cur_addr()-1)?cbrph-
>addrs[cbrph->cur_addr()-1].addr:iph-
>dst(), p.route.dump(), return_reply_path(p.pkt, off_cbrp_));
#endif
    p.pkt = NULL;
    return 1;
}

void
CBRP_Agent::handleRREQ(CBRP_Packet &p)
/* process a route request that isn't targeted at us
cluster heads will broadcast RREQ to all chosen gateways
Chosen gateways will unicast RREQ to neighbor cluster
head*/
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    hdr_ip *iph = (hdr_ip*)p.pkt->access(off_ip_);

    assert (cbrph->route_request());
    if ((u_int32_t)iph->dst() != IP_BROADCAST) {
        //I am the neighborcluster head, i should broadcast this
RREQ to my neighbors
        if (ntable->my_status == CLUSTER_HEAD || (iph->dst() ==
myaddr_)) {
            BroadcastRREQ(p);
        }else {

```



```

    //i should forward packet to the clusterhead as specified
    in ip destination
    //this could happen when the neighbor cluster head is 3
    hops away and I am the second gateway
        UnicastRREQ(p,iph->dst());
    }

} else {

    Packet *p_ori = p.pkt;

    for (int i=0;i<cbrph->num_forwarders();i++) {

        if (cbrph->forwarders[i].which_gateway == myaddr_) {
            p.pkt = p_ori->copy();

            if ((cbrph->forwarders[i].which_head == myaddr_) &&
                (ntable->my_status == CLUSTER_HEAD)) {
                //broadcast the pkt to other cluster heads
                BroadcastRREQ(p);
            } else {
                // if I am just an ordinary node, then just
                forward the RREQ as indicated in header
                UnicastRREQ(p,cbrph->forwarders[i].which_head);
            }
        }
    }

    // discard un-used RREQ packet
    if (p_ori) Packet::free(p_ori);

}

p.pkt = NULL;

}

void
CBRP_Agent::BroadcastRREQ(CBRP_Packet &p)
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    hdr_cmn *cmnh = (hdr_cmn*)p.pkt->access(off_cmn_);
    hdr_ip *iph = (hdr_ip*)p.pkt->access(off_ip_);

    assert(cbrph->route_request());

    if (ignoreRouteRequestp(p))
    {
        //-Jinyang only cluster heads records RREQs seen and
        drop RREQ if duplicate etc.
        if (verbose_srr)
            trace("cbrp %.5f _%s_ dropped %s #%d (ignored)",
                Scheduler::instance().clock(), net_id.dump(),
                p.src.dump(),

```



```

        cbrph->rtreq_seq());
    Packet::free(p.pkt); // pkt is a route request we've
already processed
    return; // drop silently
}

request_table.insert(p.src, p.src, cbrph->rtreq_seq());

if (p.route.length() > (cbrph->max_propagation()/2))
// the chain of cluster heads are roughly twice as long as
the hop by hop route
{
    if (verbose_srr)
        trace("cbrp %.5f %s_ dropped %s #d (prop limit
exceeded)",
            Scheduler::instance().clock(), net_id.dump(),
p.src.dump(),
            cbrph->rtreq_seq());
    Packet::free(p.pkt); // pkt isn't for us, and isn't
data carrying
    p.pkt = 0;
    return;
}

if ((2*p.route.length()) > MAX_SR_LEN)
{
    // no propagation
    trace("cbrp %.5f %s_ dropped %s #d (SR full)",
        Scheduler::instance().clock(), net_id.dump(),
p.src.dump(),
        cbrph->rtreq_seq());
    Packet::free(p.pkt);
    p.pkt = 0;
    return;
}

p.route.appendToPath(net_id);
fillCBRPPath(p.route, cbrph);

cmnh->ptype() = PT_CBRP;
cmnh->addr_type() = AF_INET;

if (ntable->isNeighbor(cbrph->request_destination())) {

//directly unicast to the destination
cmnh->next_hop() = cbrph->request_destination();
iph->dst() = cbrph->request_destination();
cbrph->num_forwarders() = 0;
cmnh->size() = cbrph->size();

Scheduler &s = Scheduler::instance();
s.schedule(ll, p.pkt, 0.0);
}

```



```

    return;
}

#ifdef CBRP_DEBUG
    trace("cbrp RREQ %d: %s",myaddr_,ntable->printNeighbors());
#endif

    if (ntable->GetQuickNextNode(cbrph->request_destination()))
    {
        iph->dst() = cbrph->request_destination();
        cmnh->next_hop() = ntable->GetQuickNextNode(cbrph->
>request_destination());

        cbrph->num_forwarders() = 0;
        cmnh->size() = cbrph->size();

        Scheduler::instance().schedule(11,p.pkt,0.0);
        p.pkt = 0;
        return;
    }

    if (cbrp_reply_from_cache_on_propagating &&
replyFromRouteCache(p)) {
        p.pkt = NULL;
        return;
    }

    cmnh->next_hop() = MAC_BROADCAST;
    iph->dst() = IP_BROADCAST;

    int total = cbrph->num_forwarders();

    Packet *p_copy = p.pkt->copy();

    hdr_cbrp *new_cbrph = (hdr_cbrp*)p_copy->access(off_cbrp_);

    adjtable_ent *p_adj = ntable->adjtable_1;

    int i=0;
    int j=0;

    while (p_adj && i<MAX_NEIGHBORS) {
        //check if a neighbor cluster head has been visited b4 by
this RREQ by checking its recorded route so far
        if (p.route.member(ID(p_adj->neighbor_cluster, ::IP))) {

#ifdef CBRP_DEBUG
            trace("cbrp RREQ%d: ignored %d, duplicate in
route(%s)",myaddr_,p_adj->neighbor_cluster,p.route.dump());
#endif
            p_adj = p_adj->next;
            continue;

```



```

    }

    //check if a neighbor cluster will have been visited by
    prev. cluster head by checking the forwarders' list
    for (j=0;j<total;j++) {

        if (p_adj->neighbor_cluster == cbrph-
>forwarders[j].which_head) {

#ifdef CBRP_DEBUG
            trace("cbrp RREQ%d: ignored %d, duplicate in previous
broadcast",myaddr_,p_adj->neighbor_cluster);
#endif
            break;
        }
    }

    if (j>= total) { //the next cluster head has NOT been
visited to the best of our knowledge. :)

#ifdef CBRP_DEBUG
        trace("cbrp %.5f _%s_ rebroadcast %s #d ->%d %s to 2
hop head %d via %d",Scheduler::instance().clock(),
net_id.dump(),
            p.src.dump(), cbrph->rtreq_seq(), cbrph-
>request_destination(),
            p.route.dump(), p_adj->neighbor_cluster,
            p_adj->next_hop->next_node);
#endif

        /*new_cbrph->reply_addrs()[i].which_gateway= p_adj-
>next_hop->next_node; */
        new_cbrph->forwarders[i].which_gateway= ntable-
>GetQuickNextNode(p_adj->neighbor_cluster);
        new_cbrph->forwarders[i++].which_head= p_adj-
>neighbor_cluster;
    }
    p_adj = p_adj->next;
}

//if I am not a cluster head, i don't send RREQ to 3-hop
neighbor cluster head

if (ntable->my_status != CLUSTER_HEAD) {
    goto DONE;
}

p_adj.= ntable->adjtable_2;

while (p_adj && i<MAX_NEIGHBORS) {

    if (p.route.member(ID(p_adj->neighbor_cluster, ::IP)) {

```



```

#ifdef CBRP_DEBUG
    trace("cbrp RREQ%d: ignored %d,
route(%s)",myaddr_,p_adj->neighbor_cluster,p.route.dump());
#endif
    p_adj= p_adj->next;
    continue;
}
// i don't send to 3-hop neighbor head in cluster
adjacency table if it is within 2-hop range
if (ntable->GetAdjEntry(p_adj->neighbor_cluster,1)) {
#ifdef CBRP_DEBUG
    trace("cbrp RREQ%d: ignored (sent it to 2-hop gw) %d,
route(%s)",myaddr_,p_adj->neighbor_cluster,p.route.dump());
#endif
    p_adj= p_adj->next;
    continue;
}

for (j=0;j<total;j++) {
    if (p_adj->neighbor_cluster == cbrph-
>forwarders[j].which_head) {
#ifdef CBRP_DEBUG
        trace("cbrp RREQ%d: ignored %d, duplicate in previous
broadcast",myaddr_,p_adj->neighbor_cluster);
#endif
        break;
    }
}
if (j >= total ) {
    new_cbrph->forwarders[i].which_gateway= p_adj->next_hop-
>next_node;
    new_cbrph->forwarders[i++].which_head= p_adj-
>neighbor_cluster;
}
p_adj = p_adj->next;
}

DONE:
new_cbrph->num_forwarders() = i;

if (i != 0) {

    hdr_cmn *new_cmnh=(hdr_cmn*)p_copy->access(off_cmn_);
    new_cmnh->size() =new_cbrph->size();
#ifdef CBRP_DEBUG
    trace("cbrp %.5f _%d_ my route addres size %d forward path
%s",Scheduler::instance().clock(), myaddr_, new_cbrph-
>num_addrs(),return_RREQ_forward_nodes(p_copy,off_cbrp_));
#endif
    trace("CBRP %.5f _%d_ broadcast-request %d->%d",
Scheduler::instance().clock(),myaddr_, iph->src(),cbrph-
>request_destination());

    Scheduler::instance().schedule(11,p_copy,0.0);
}

```



```

Packet::free(p.pkt);
p.pkt = NULL;
}

int
CBRP_Agent::UnicastRREQ(CBRP_Packet &p, nsaddr_t nextcluster)
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    hdr_cmn *cmnh = (hdr_cmn*)p.pkt->access(off_cmn_);
    hdr_ip *iph = (hdr_ip*)p.pkt->access(off_ip_);

    cmnh->addr_type() = AF_INET;

    if (ntable->isNeighbor(cbrph->request_destination())) {

        //directly unicast to the destination
        cmnh->next_hop() = cbrph->request_destination();
        iph->dst() = cbrph->request_destination();

        cbrph->num_forwarders() = 0;
        cmnh->size() = cbrph->size();

        Scheduler &s = Scheduler::instance();
        s.schedule(11,p.pkt,0.0);
        return 1;

    }

    if (cbrp_reply_from_cache_on_propagating &&
        replyFromRouteCache(p)) {
        p.pkt = NULL;
        return 1;
    }

    if (ntable->isNeighbor(nextcluster)) {

        cmnh->next_hop() = nextcluster;
        iph->dst() = nextcluster;

    }else if ((cmnh->next_hop() = ntable->
        GetQuickNextNode(nextcluster))) {

        iph->dst() = nextcluster;

    }else {

        trace("CBRP %.5f %d_ drop-request : dropped(does not know
        how to reach next cluster %d), route(%s) cluster head?
        %s",Scheduler::instance().clock(),myaddr_,nextcluster,p.route.
        dump(), (ntable->my_status == CLUSTER_HEAD )? "yes":"no");

        Packet::free(p.pkt);
    }
}

```



```

    p.pkt = NULL;
    return 0;
}

cmnh->xmit_failure_ = CBRP_XmitFailureCallback;
cmnh->xmit_failure_data_ = (void *) this;

    trace("cbrp %.5f _%s_ relay %s #%d ->%d %s to designated
head %d via %d", Scheduler::instance().clock(), net_id.dump(),
p.src.dump(),
    cbrph->rtreq_seq(), cbrph->request_destination(),
p.route.dump(),
    nextcluster, cmnh->next_hop());

cmnh->size() = cbrph->size();

Scheduler::instance().schedule(11,p.pkt, 0.0);
p.pkt = NULL;
return 1;
}

/*=====
=====
    Helpers
-----*/
bool
CBRP_Agent::ignoreRouteRequestp(CBRP_Packet &p)
// should we ignore this route request? by dmaltz from DSR
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);

    if (request_table.get(p.src) >= cbrph->rtreq_seq())
        { // we've already processed a copy of this request so
          // we should drop the request silently
            return true;
        }
    if (p.route.member(net_id,MAC_id))
        { // we're already on the route, drop silently
            return true;
        }

    if (p.route.full())
        {
            return true;
        }
    return false;
}

bool
CBRP_Agent::replyFromRouteCache(CBRP_Packet &p)
/* - see if can reply to this route request from our cache
if so, do it and return true, otherwise, return false

```



```

    - frees or hands off p iff returns true by dmaltz from
DSR,modified for CBRP
    - CBRP does not reply from route-cache by default*/
{
    Path rest_of_route;

    // do we have a cached route the target?
    /* XXX what if we have more than 1? (and one is legal for
reply from
    cache and one isn't?) 1/28/97 -dam */
    hdr_cbrp *old_cbrph = (hdr_cbrp *)p.pkt->access(off_cbrp_);
    p.dest = ID(old_cbrph->request_destination(),::IP);
    if (!route_cache->findRoute(p.dest, rest_of_route, 0))
        { // no route => we're done
            return false;
        }

    /* but we should be on on the remainder of the route (and
should be at
    the start of the route */
    assert(rest_of_route[0] == net_id || rest_of_route[0] ==
MAC_id);

    if (rest_of_route.length() + 2 * p.route.length() >
MAX_SR_LEN)
        return false; // too long to work with...

    // if there is any other information piggybacked into the
// route request pkt, we need to forward it on to the dst
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    int request_seqnum = cbrph->rtreq_seq();

    // make up and send out a route reply
    Packet* rrp = allocpkt();

    hdr_ip *iph = (hdr_ip*)rrp->access(off_ip_);
    iph->src() = myaddr_;
    iph->sport() = RT_PORT;
    iph->dst() = p.src.addr;
    iph->dport() = RT_PORT;
    iph->ttl() = 255;

    cbrph = (hdr_cbrp*)rrp->access(off_cbrp_);
    cbrph->init();
    cbrph->route_reply() = 1;

    int len = rest_of_route.length();
    for (int i = 0; i < len; i++)
        rest_of_route[len - i - 1].fillSRAddr(cbrph-
>reply_addrs()[i]);
    cbrph->route_reply_len() = len;

    // propagate the request sequence number in the reply for
analysis purposes

```



```

cbrph->rtreq_seq() = request_seqnum;

hdr_cmn *cmnh = (hdr_cmn*)rrp->access(off_cmn_);
cmnh->ptype() = PT_CBRP;
cmnh->size() = cbrph->size();

// copy the rest of cluster head by cluster head route over
fillCBRPPath(p.route, cbrph);
cbrph->cur_addr() = cbrph->num_addrs() - 1;

nsaddr_t next_cluster;
if (cbrph->cur_addr() >= 0) {
    next_cluster = cbrph->addrs[cbrph->cur_addr()].addr;
} else {
    next_cluster = iph->dst();
}

adjtable_ent *adj_ent;

if ((cmnh->next_hop() = ntable-
>GetQuickNextNode(next_cluster))) {
    goto DONE;
} else if ((adj_ent = ntable->GetAdjEntry(next_cluster, 0)))
{
    cmnh->next_hop() = adj_ent->next_hop->next_node;
    goto DONE;
} else {
    Packet::free(rrp);
    return false;
}

DONE:
Scheduler::instance().schedule(11, rrp, 0);
trace("CBRP %.9f %s_ cache-reply-sent %d -> %d #%d (len %d)
%s",
        Scheduler::instance().clock(), net_id.dump(),
        iph->src(), iph->dst(), request_seqnum,
rest_of_route.length(),
        rest_of_route.dump());
Packet::free(p.pkt);
p.pkt = NULL;
return true;
}

void
CBRP_Agent::sendOutPacketWithRoute(CBRP_Packet& p, bool fresh,
Time delay = 0.0)
    // take packet and send it out, packet must a have a
route in it
    // This packet should not not be a RREQ packet
    // if fresh is true then reset the path before using it,
if fresh
    // is false then our caller wants us use a path with the
index
    // set as it currently is

```



```

{

hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
hdr_cmn *cmnh = (hdr_cmn*)p.pkt->access(off_cmn_);

assert(cbrph->valid());
assert(!cbrph->route_request());

if ((p.dest == net_id) //Jinyang
    { // it doesn't need to go on the wire, 'cause it's for us
      rcv(p.pkt, (Handler *) 0);
      p.pkt = 0;
      return;
    }

if (fresh)
  {
    p.route.resetIterator();
    if (verbose_)
      {
        trace("SO %.9f %s_ originating %s, RREP? %s",
              Scheduler::instance().clock(),
              net_id.dump(), p.route.dump(), cbrph-
>route_reply()?"yes":"no" );
      }
  }

fillCBRPPath(p.route, cbrph);
assert((unsigned long)cbrph->addrs[cbrph->cur_addr()].addr
== net_id.addr);

// check if we could shorten the route by looking at our
two-hop-topology table
// we don't shorten the route for route errors and RREP
(route reply)
if ((!cbrph->route_error()) && (!cbrph->route_reply()) &&
    cbrp_shortening_route_if_possible && (cbrph-
>cur_addr() < cbrph->num_addrs()-2)) {

    //we only check to see if the node after next is
    reachable or not.
    //cause that's the most likely case. :)
    nsaddr_t next_next_node = cbrph->addrs[cbrph-
>cur_addr()+2].addr;

    if (ntable->isNeighbor(next_next_node)) {

        //I'll send the packet directly to the next next
node. :)
        cmnh->next_hop() = next_next_node;
        cbrph->cur_addr() += 2;
        //set the flag to indicate I'm temporarily
shortening the route,
        //no changes are made to route at this point of
time.
    }
}

```



```

        cbrph->route_shortening() = 1;
        trace("cbrp %.5f _%d_ trying to shorten the route
%d->%d", Scheduler::instance().clock(), myaddr_, myaddr_, cbrph-
>addr[ cbrph->cur_addr() ].addr);

        }else {
            cmnh->next_hop() = cbrph->get_next_addr();
            cbrph->cur_addr() = cbrph->cur_addr() + 1;
        }
    } else {
        cmnh->next_hop() = cbrph->get_next_addr();
        cbrph->cur_addr() = cbrph->cur_addr() + 1;
    }
    cmnh->xmit_failure_ = CBRP_XmitFailureCallback;
    cmnh->xmit_failure_data_ = (void *) this;
    cmnh->prev_hop_ = net_id.getNSAddr_t();
    cmnh->addr_type() = AF_INET;

    assert(p.pkt->incoming == 0); // this is an outgoing packet

    if (ifq->length() > 25)
        trace("SIFQ %.5f _%s_ len %d",
            Scheduler::instance().clock(), net_id.dump(), ifq-
>length());

    //calculate the packet size
    if (cbrph->route_reply() || cbrph->route_error()) {
        cmnh->size() = cbrph->size();
    }else {
        cmnh->size() += cbrph->size();
    }

    // off it goes!!!!
    //assert(cmnh->ptype() == PT_CBR || cbrph->route_reply() ||
cbrph->route_error());
    Scheduler::instance().schedule(11, p.pkt, delay);
    p.pkt = NULL;

}

void
CBRP_Agent::getRouteForPacket(CBRP_Packet &p, bool retry)
/* try to obtain a route for packet
   pkt is freed or handed off as needed, unless retry ==
true
   in which case it is not touched */
{
    Entry *e = request_table.getEntry(p.dest);
    Time time = Scheduler::instance().clock();

    if (!retry)
    {
        stickPacketInSendBuffer(p);
    }
}

```



```

    p.pkt = 0; // pkt is handled for now (it's in
sendbuffer)
}

/* make the route request packet */
CBRP_Packet rrq;

rrq.dest = ID(0, ::IP); // the ip destination of RREQ is
not necessarily same the destinate node
rrq.src = net_id;

rrq.pkt = allocpkt();
hdr_cbrp *cbrph = (hdr_cbrp*)rrq.pkt->access(off_cbrp_);
hdr_ip *iph = (hdr_ip*)rrq.pkt->access(off_ip_);
hdr_cmn *cmnh = (hdr_cmn*)rrq.pkt->access(off_cmn_);

iph->dport() = RT_PORT;
iph->src() = net_id.getNSAddr_t();
iph->sport() = RT_PORT;

cmnh->ptype() = PT_CBRP;
cmnh->num_forwards() = 0;
cmnh->addr_type() = AF_INET;

cbrph->init();

//set the requested destination node
cbrph->route_request() = 1;
cbrph->request_destination() = p.dest.getNSAddr_t();

if (BackOffTest(e, time))
{ // it's time to start another route request cycle
// CBRP does not have ring_zero search, but we still set
the appropriate
// e->last_type for reusing this part of DSR's code
e->last_type = UNLIMIT;
sendOutRtReq(rrq, MAX_SR_LEN);
e->last_arp = time;
}
else if (LIMIT0 == e->last_type && (time - e->last_arp) >
cbrp_arp_timeout)
{
// try propagating rt req since we haven't heard back
from limited one
// Actually this part of the code will never be executed
by CBRP - Li Jinyang
e->last_type = UNLIMIT;
sendOutRtReq(rrq, MAX_SR_LEN);
}
else
{
// it's not time to send another route request...
if (!retry && verbose_srr) {

```



```

        trace("SRR %.5f %s RR-not-sent %s -> %s",
              Scheduler::instance().clock(),
              net_id.dump(), rrq.src.dump(), p.dest.dump());
    }
    trace("cbrp %s -> %s, request entry rt_reqs_outstanding
%d, last_rt_req %.9f", rrq.src.dump(), p.dest.dump(), e-
>rt_reqs_outstanding, e->last_rt_req);

    Packet::free(rrq.pkt); // dump the route request packet
we made up
    rrq.pkt = 0;
}

}

void
CBRP_Agent::sendOutRtReq(CBRP_Packet &p, int max_prop)
// Send out the Route request packet we have made
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    hdr_ip *iph = (hdr_ip*)p.pkt->access(off_ip_);
    hdr_cmn *cmnh = (hdr_cmn*)p.pkt->access(off_cmn_);
    int i=0;

    assert(cbrph->valid());

    cbrph->rtreq_seq() = route_request_num++;
    cbrph->max_propagation() = max_prop;

    p.route.reset();

    // CBRP does not propagate last error seen on Route Request,
if you are
    // interested to see how this might help as it does for DSR,
turn it on and try
    if (cbrp_propagate_last_error && route_error_held
        && Scheduler::instance().clock() - route_error_data_time
    < cbrp_max_err_hold)
    {
        assert(cbrph->num_route_errors() < MAX_ROUTE_ERRORS);
        cbrph->route_error() = 1;
        link_down_cbrp *deadlink = &(cbrph->down_links()[cbrph-
>num_route_errors()]);
        deadlink->addr_type = AF_INET;
        deadlink->from_addr = err_from.getNSAddr_t();
        deadlink->to_addr = err_to.getNSAddr_t();
        deadlink->tell_addr = GRAT_ROUTE_ERROR;
        cbrph->num_route_errors() += 1;
        /*
        * Make sure that the Route Error gets on a propagating
request.
        */
        if(max_prop > 0) route_error_held = false;
    }
}

```



```

    }

    trace("CBRP %.5f _%s_ new-request %d %s #%d -> %d",
          Scheduler::instance().clock(), net_id.dump(),
          max_prop, p.src.dump(), cbrph->rtreq_seq(), cbrph-
>request_destination());

    //Jinyang - if i am the cluster head, append myself to the
    path before sending RREQ
    if (ntable->my_status == CLUSTER_HEAD) {

        p.route.appendToPath(net_id);
        fillCBRPPath(p.route, cbrph);

        //I will include all my 2-hop neighbor cluster head on
        forwarders list
        adjtable_ent *p_ent = ntable->adjtable_1;
        while (p_ent && (i<MAX_NEIGHBORS)) {
            cbrph->forwarders[i].which_gateway= p_ent->next_hop-
>next_node;
            cbrph->forwarders[i].which_head = p_ent-
>neighbor_cluster;

#ifdef CBRP_DEBUG
            trace("cbrp %.5f _%s_ (head) RREQ(%d->%d) broadcasted to
            2-hop head %d via %d",
            Scheduler::instance().clock(), net_id.dump(),
            cbrph->rtreq_seq(), cbrph-
>request_destination(), p_ent->neighbor_cluster,
            p_ent->next_hop->next_node);
#endif
            i++;
            p_ent = p_ent->next;
        }

        // I will include all my 3-hop neighbor cluster head on
        forwarders list
        p_ent = ntable->adjtable_2;

        while (p_ent && (i<MAX_NEIGHBORS)) {
            cbrph->forwarders[i].which_gateway= p_ent->next_hop-
>next_node;
            cbrph->forwarders[i].which_head= p_ent-
>neighbor_cluster;

#ifdef CBRP_DEBUG
            trace("cbrp %.5f _%s_ (head) RREQ(%d->%d) broadcasted to
            3-hop head %d via %d",
            Scheduler::instance().clock(), net_id.dump(),
            cbrph->rtreq_seq(), cbrph-
>request_destination(), p_ent->neighbor_cluster,
            p_ent->next_hop->next_node);
#endif
            i++;
            p_ent = p_ent->next;
        }
    }

```



```

    }
} else {

    //I have to send the route request to one of my cluster
head
    ntable_ent *ent = ntable->head;

    while (ent && (i < MAX_NEIGHBORS)) {
        if (ent->neighbor_status == CLUSTER_HEAD) {
            cbrph->forwarders[i].which_gateway= ent->neighbor;
            cbrph->forwarders[i].which_head = ent->neighbor;
#ifdef CBRP_DEBUG
            trace("cbrp %.5f _%s_ RREQ(%d->%d) broadcasted to 1-
hop head %d",

Scheduler::instance().clock(), net_id.dump(), cbrph-
>rtreq_seq(),
                cbrph->request_destination(), ent->neighbor);
#endif
            i++;
        }
        ent = ent->next;
    }

    // I will include all 2-hop cluster head in forwarders
list
    adjtable_ent *adjent=ntable->adjtable_1;

    while (adjent && (i < MAX_NEIGHBORS)) {

        // if the 2-hop cluster head is "really" not directly
reachable
        if (!ntable->GetEntry(adjent->neighbor_cluster)) {

            cbrph->forwarders[i].which_gateway= adjent->next_hop-
>next_node;
            cbrph->forwarders[i].which_head= adjent-
>neighbor_cluster;

#ifdef CBRP_DEBUG
            trace("cbrp %.5f _%s_ RREQ(%d->%d) broadcasted to 2-
hop head %d via %d",

Scheduler::instance().clock(), net_id.dump(), cbrph-
>rtreq_seq(),
                cbrph->request_destination(), adjent-
>neighbor_cluster, adjent->next_hop->next_node);
#endif
            i++;
        }
        adjent = adjent->next;
    }

}
cbrph->num_forwarders() = i;

```



```

cmnh->next_hop() = MAC_BROADCAST;
iph->dst() = IP_BROADCAST;
iph->src() = myaddr_;

cmnh->size() = cbrph->size();
Scheduler::instance().schedule(11,p.pkt,0.0);
}

void
CBRP_Agent::returnSrcRouteToRequestor(CBRP_Packet &p)
// take the route in p, add us to the end of it and return
the
// RREP to the sender of p
// doesn't free p.pkt
{
    hdr_cbrp *old_cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);

    if (p.route.full())
        return; // alas, the route would be to long once we add
ourselves

    CBRP_Packet p_copy = p;
    p_copy.pkt = allocpkt();
    p_copy.dest = p.src;

    hdr_ip *new_iph = (hdr_ip*)p_copy.pkt->access(off_ip_);
    hdr_cbrp *new_cbrph = (hdr_cbrp*)p_copy.pkt-
>access(off_cbrp_);
    hdr_cmn *new_cmnh = (hdr_cmn*)p_copy.pkt->access(off_cmn_);

    new_cbrph->init();
    new_cbrph->route_reply() = 1;

    trace("CBRP %.5f %s_reply-sent %s->%d %s",
        Scheduler::instance().clock(),net_id.dump(),
        p.src.dump(),old_cbrph-
>request_destination(),p.route.dump());

    assert(old_cbrph->request_destination() ==
net_id.getNSAddr_t());

    p_copy.src = net_id;

    //if i am the cluster head, i have to complete the recorded
route first
    if (ntable->my_status == CLUSTER_HEAD) {
        new_cbrph->reply_addrs()[0].addr = net_id.addr;
        new_cbrph->reply_addrs()[0].addr_type = AF_INET;
        new_cbrph->route_reply_len() = 1;
        p_copy.route.appendToPath(net_id);
    }
    fillCBRPPath(p_copy.route,new_cbrph);
}

```



```

// just in case the route is empty
if (!old_cbrph->num_addrs()) {
    //try to reach the sender directly!
    if ((new_cmnh->next_hop() = ntable-
>GetQuickNextNode(p_copy.dest.addr)) {

        new_cbrph->reply_addrs()[0].addr = p_copy.src.addr;
        new_cbrph->reply_addrs()[0].addr_type = AF_INET;
        new_cbrph->route_reply_len() = 1;
        new_iph->dst() = p_copy.dest.addr;
        new_iph->src() = myaddr_;
        new_cmnh->size() = new_cbrph->size();
        Scheduler::instance().schedule(11,p_copy.pkt,0.0);

    }else {
        Packet::free(p_copy.pkt);
    }
    return;
}

//initialize route index pointer
new_cbrph->cur_addr() = p_copy.route.length()-1;
p_copy.route.setIterator(new_cbrph->cur_addr());

new_iph->dst() = p_copy.dest.addr;
new_iph->dport() = RT_PORT;
new_iph->src() = p_copy.src.addr;
new_iph->sport() = RT_PORT;
new_iph->ttl() = 255;

// propagate the request sequence number in the reply for
analysis purposes
new_cbrph->rtreq_seq() = old_cbrph->rtreq_seq();
new_cmnh->ptype() = PT_CBRP;
handlerREP(p_copy);
}

void
CBRP_Agent::acceptRREP(CBRP_Packet &p)
/* - enter the packet's source route into our cache
   - see if any packets are waiting to be sent out with this
source route
   - doesn't free the pkt */
{
    Path reply_route;
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);

    if (cbrph->route_repaired()) {
        trace("CBRP %.5f %d_ grat-reply-repair from %d, route
%s", Scheduler::instance().clock(),myaddr_, p.src.addr,
p.route.dump());
    }else if (cbrph->route_shortened()){

```



```

    trace("CBRP %.5f _%d_ grat-reply-shorten from %d, route
%s", Scheduler::instance().clock(),myaddr_, p.src.addr,
p.route.dump());
    }else {
        trace("CBRP %.5f _%d_ reply-received from %d, route %s",
Scheduler::instance().clock(),myaddr_,p.src.addr,p.route.dump(
));
    }

    if (!cbrph->route_reply())
    { // somethings wrong...
        trace("SDFU non route containing packet given to
acceptRREP");
        fprintf(stderr, "dfu: non route containing packet given
to acceptRRP\n");
    }

    if ((!cbrph->route_repaired()) && (!cbrph-
>route_shortened())) {
        Path reverse_reply_route(cbrph->reply_addrs(), cbrph-
>route_reply_len());
        //Jinyang - in CBRP, the returned source route is in
reverse order
        reverse_reply_route.appendToPath(net_id);
        reply_route = reverse_reply_route.reverse();
    }else {
        Path reverse_reply_route(cbrph->addrs, cbrph-
>num_addrs());
        reply_route = reverse_reply_route.reverse();
    }

    /* check to see if this reply is valid or not using god info
*/
    int i;
    bool good_reply = true;
    for (i = 0; i < reply_route.length()-1 ; i++)
        if (God::instance()->hops(reply_route[i].getNSAddr_t(),
reply_route[i+1].getNSAddr_t())
!= 1)
        {
            good_reply = false;
            break;
        }

#ifdef CBRP_DEBUG
    trace("cbrp %.9f _%s_ reply-received %d from %s %s #%d ->
%s %s",
Scheduler::instance().clock(), net_id.dump(),
good_reply ? 1 : 0,
p.src.dump(), reply_route[0].dump(), cbrph-
>rtreq_seq(),
reply_route[reply_route.length()-1].dump(),
reply_route.dump());
#endif
#endif

```



```

// add the new route into our cache
route_cache->addRoute(reply_route,
Scheduler::instance().clock(), p.src);

// add the new loose source route to our loose source route
cache
Path source_route(cbrph->addrs, cbrph->num_addrs());

// back down the route request counters
Entry *e =
request_table.getEntry(reply_route[reply_route.length()-1]);
e->rt_reqs_outstanding = 0;
e->last_rt_req = 0.0;

// see if the addition of this route allows us to send out
// any of the packets we have waiting
Time delay = 0.0;
for (int c = 0; c < SEND_BUF_SIZE; c++)
{
    if (send_buf[c].p.pkt == NULL) continue;
    if (route_cache->findRoute(send_buf[c].p.dest,
send_buf[c].p.route, 1))
        { // we have a route!
#ifdef CBRP_DEBUG
            struct hdr_cmh *ch = HDR_CMH(send_buf[c].p.pkt);
            if(ch->size() < 0) {
                drop(send_buf[c].p.pkt, "XXX");
                abort();
            }
#endif
            if (verbose_)
                trace("Sdebug %.9f _%s_ liberated from sendbuf %s-
>%s %s",
Scheduler::instance().clock(),
net_id.dump(),
send_buf[c].p.src.dump(),
send_buf[c].p.dest.dump(),
send_buf[c].p.route.dump());

            sendOutPacketWithRoute(send_buf[c].p, true, delay);
            /* DSR spread out the rate in order to arp to
complete, since
we've already optimized ARP to take in MAC
address of neighbors
in advance, we don't really need to delay any
more */
            //delay += cbrp_arp_timeout;
            send_buf[c].p.pkt = NULL;
        }
    }
}

void
CBRP_Agent::processBrokenRouteError(CBRP_Packet& p)

```



```

// take the error packet and process our part of it.
// if needed, send the remainder of the errors to the next
person
// doesn't free p.pkt, mostly taken from DSR's implementation
by dmaltz
{
    hdr_cbrp *cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);

    if (!cbrph->route_error())
        return; // what happened??

    /* if we hear A->B is dead, should we also run the link B->A
through the
        cache as being dead, since 802.11 requires bidirectional
links
        XXX -dam 4/23/98 */

    // since CPU time is cheaper than network time, we'll
process
    // all the dead links in the error packet
    assert(cbrph->num_route_errors() > 0);
    int count = cbrph->num_route_errors();
    for (int c = 0 ; c < count ; c++)
    {
        assert(cbrph->down_links()[c].addr_type == AF_INET);
        route_cache->noticeDeadLink(ID(cbrph-
>down_links()[c].from_addr,::IP),
                                ID(cbrph-
>down_links()[c].to_addr,::IP),
Scheduler::instance().clock());
        // I'll assume everything's of type AF_INET for the
printout... XXX
        if (verbose_srr)
            trace("SRR %.9f %s_ dead-link tell %d %d -> %d",
Scheduler::instance().clock(), net_id.dump(),
cbrph->down_links()[c].tell_addr,
cbrph->down_links()[c].from_addr,
cbrph->down_links()[c].to_addr);
    }

    ID who = ID(cbrph->down_links()[cbrph->num_route_errors()-
1].tell_addr, ::IP);
    if (who != net_id && who != MAC_id)
    { // this error packet wasn't meant for us to deal with
        // since the outer entry doesn't list our name
        return;
    }

    // record this route error data for possible propagation on
our next
    // route request
    route_error_held = true;
    err_from = ID(cbrph->down_links()[cbrph->num_route_errors()-
1].from_addr,::IP);

```



```

err_to = ID(cbrph->down_links()[cbrph->num_route_errors()-
1].to_addr, ::IP);
route_error_data_time = Scheduler::instance().clock();

if (1 == cbrph->num_route_errors())
    { // this error packet has done its job
      // it's either for us, in which case we've done what it
sez
      // or it's not for us, in which case we still don't have
to forward
      // it to whoever it is for
      return;
    }

/* make a copy of the packet and send it to the next
tell_addr on the
error list. the copy is needed in case there is other
data in the
packet (such as nested route errors) that need to be
delivered */
if (verbose_)
    trace("Sdebug %.5f _%s_ unwrapping nested route error",
Scheduler::instance().clock(), net_id.dump());

CBRP_Packet p_copy = p;
p_copy.pkt = p.pkt->copy();

hdr_cbrp *new_cbrph = (hdr_cbrp*)p_copy.pkt-
>access(off_cbrp_);
hdr_ip *new_iph = (hdr_ip*)p_copy.pkt->access(off_ip_);

// remove us from the list of errors
new_cbrph->num_route_errors() -= 1;
new_cbrph->route_request() = 0;
//new_cbrph->route_error() = 1;

// send the packet to the person listed in what's now the
last entry
p_copy.dest = ID(new_cbrph->down_links()[new_cbrph-
>num_route_errors()-1].tell_addr, ::IP);
p_copy.src = net_id;

new_iph->dst() = p_copy.dest.addr;
new_iph->dport() = RT_PORT;
new_iph->src() = p_copy.src.addr;
new_iph->sport() = RT_PORT;
new_iph->ttl() = 255;

// an error packet is a first class citizen, so we'll
// use handlePktWOSR to obtain a route if needed
handlePktWithoutSR(p_copy, false);
}

/*=====
==

```



```

    Callback for link layer transmission failures
    -----*/
struct filterfailedata {
    nsaddr_t dead_next_hop;
    int off_cmn_;
    CBRP_Agent *agent;
};

int
CBRP_FilterFailure(Packet *p, void *data)
{
    struct filterfailedata *ffd = (filterfailedata *) data;
    hdr_cmn *cmh = (hdr_cmn*)p->access(ffd->off_cmn_);
    int remove = cmh->next_hop() == ffd->dead_next_hop;
    if (remove) ffd->agent->undeliverablePkt(p,1);
    return remove;
}

void
CBRP_Agent::undeliverablePkt(Packet *pkt, int mine)
/* try our best to save the undeliverable packet using local
repair or cache */
{
    hdr_cbrp *cbrph = (hdr_cbrp*)pkt->access(off_cbrp_);
    hdr_ip *iph = (hdr_ip*)pkt->access(off_ip_);
    hdr_cmn *cmh = (hdr_cmn*)pkt->access(off_cmn_);

    CBRP_Packet p(pkt,cbrph);
    p.dest = ID(iph->dst(),::IP);
    p.src = ID(iph->src(),::IP);
    p.pkt = mine ? pkt : pkt->copy();

    cbrph = (hdr_cbrp*)p.pkt->access(off_cbrp_);
    iph = (hdr_ip*)p.pkt->access(off_ip_);
    cmh = (hdr_cmn*)p.pkt->access(off_cmn_);

    if ((cbrph->route_request()))
    {
        if (UnicastRREQ(p,iph->dst())) {
            trace("CBRP %.5f %d_ salvage-request %d->%d",
Scheduler::instance().clock(), myaddr_, iph->src(),cbrph-
>request_destination());
        }
        return;
    }
    else if (cbrph->route_reply())
    {
        // we have already dropped the gratuitous route reply
packet
        assert(!cbrph->route_repaired());
        //try to salvage the route reply
#ifdef CBRP_DEBUG

```



```

    printf("cbrp %d %.9f trying to salvage route reply %d-
>%di\n", net_id.getNSAddr_t(), Scheduler::instance().clock(), iph-
->src(), iph->dst());
#endif

    if ((ntable->my_status == CLUSTER_HEAD) && (cbrph-
>cur_addr() < (cbrph->num_addrs()-1))
        && (cbrph->addrs[cbrph->cur_addr()+1].addr ==
myaddr_)) {
        cbrph->cur_addr++;
        p.route.setIterator(cbrph->cur_addr());
    }
    if (cbrph->reply_addrs()[cbrph->route_reply_len()-
1].addr == net_id.getNSAddr_t()) {
        cbrph->route_reply_len()--;
    }

    p.dest.addr = iph->dst();
    p.dest.type = IP;
    if (handlerREP(p)) {
        trace("CBRP %.9f _%d_ salvage-reply %d-
>%d", Scheduler::instance().clock(), myaddr_, iph->src(), iph-
>dst());
    }
    return;
}
else if (cbrph->route_shortening())
{
    trace("CBRP %.5f _%d_ wrong-shorten %d...%d-
>%d...%d", Scheduler::instance().clock(), myaddr_, iph-
>src(), myaddr_, cbrph->addrs[cbrph->cur_addr()].addr, iph-
>dst());
    //My shortening of the path is wrong, don't do it
    cbrph->route_shortening() = 0;
    //reset the cur_addr_ pointer
    cbrph->cur_addr() = cbrph->cur_addr()-2;

    p.dest = ID(iph->dst(), ::IP);
    p.src = ID(iph->src(), ::IP);
    p.route.setIterator(cbrph->cur_addr());

    sendOutPacketWithRoute(p, false);

    //do not send any route error message
    return;
}

// it's a packet we're forwarding for someone, save it if we
can...
Path salvage_route;

//firstly reset the cur_addr_ pointer correctly
cbrph->cur_addr() -= 1;
p.route.setIterator(cbrph->cur_addr());
if (cbrph->cur_addr() >= (cbrph->num_addrs() - 1)) {

```



```

        trace("SDFU: route error beyond end of source route????");
        fprintf(stderr,"SDFU: route error beyond end of source
route????\n");
        Packet::free(p.pkt);
        return;
    }

    //salvage the route with local two hop topology information
    if (cbrp_local_repair && ((cbrph->route_repaired() <
MAX_LOCAL_REPAIR_TIMES ) && ((unsigned long)cbrph-
>addr[cbrph->cur_addr()].addr == net_id.addr)) )
    {
        if ((p.route.index()<p.route.length()-2) &&
            (cmh->next_hop() = ntable-
>GetQuickNextNode(p.route[p.route.index()+2].addr))
        {
            //try to reach the hop after next using another
intermediate node
            trace("CBRP %.5f _%d_ salvage-repair link %d ->
(%d) %d ", Scheduler::instance().clock(),myaddr_,myaddr_, cmh-
>next_hop(), cbrph->addr[cbrph->cur_addr()+2].addr);
            cbrph->addr[cbrph->cur_addr()+1].addr = cmh-
>next_hop();
            cbrph->addr[cbrph->cur_addr()+1].addr_type =
AF_INET;
            cbrph->cur_addr()++;
        }
        else if ((cmh->next_hop() = ntable-
>GetQuickNextNode((p.route[p.route.index()+1]).addr))
        {
            //try to reach the next hop using a new
intermediate node
            trace("CBRP %.5f _%d_ salvage-repair link %d ->
(%d) %d ", Scheduler::instance().clock(),myaddr_,myaddr_, cmh-
>next_hop(), cbrph->addr[cbrph->cur_addr()+1].addr);
            cbrph->cur_addr()++;
            for (int i = cbrph->num_addrs();i>cbrph-
>cur_addr();i--) {
                cbrph->addr[i] = cbrph->addr[i-1];
            }
            cbrph->num_addrs() += 1;
            cbrph->addr[cbrph->cur_addr()].addr = cmh-
>next_hop();
            cbrph->addr[cbrph->cur_addr()].addr_type =
AF_INET;
        }

        if (cmh->next_hop()) { //we've managed to save
using local repair, send it out

            if (cbrph->cur_addr() == 1) {
                assert(p.src == net_id);
                //amend the route directly in the cache, do not
need to mark repair flag
                Path route;

```



```

        //fill the path
        route.setLength(cbrph->num_addrs());
        for (int i=0;i<cbrph->num_addrs();i++) {
            route[i] = ID(cbrph->addrs[i]);
        }
        route.resetIterator();
        route_cache->addRoute(route,
Scheduler::instance().clock(), p.dest);
    }else {
        cbrph->route_repaired()++;
    }
    cmh->size() += cbrph->size();
    Scheduler::instance().schedule(11,p.pkt, 0);
    return;
}

}

    if (cbrp_salvage_with_cache && route_cache-
>findRoute(p.dest, salvage_route, 0))
    {
        p.route = salvage_route;
        p.route.setIterator(1);
        fillCBRPPath(p.route, cbrph);
        cmh->next_hop() = cbrph->addrs[p.route.index()].addr;
        trace("CBRP %.5f _%s_ salvage-cache %s -> %s --- %d
with %s",
                Scheduler::instance().clock(), net_id.dump(),
                p.src.dump(), p.dest.dump(), cmh->uid(),
p.route.dump());
        //prevent the route to be saved again in future.
        cbrph->route_repaired()=MAX_LOCAL_REPAIR_TIMES+1;
        cmh->size() += cbrph->size();
        Scheduler::instance().schedule(11,p.pkt, 0);
        return;
    }

    if (ID(iph->src(), ::IP) == net_id)
    {
        // it's our packet we couldn't send, we do another
RREQ
        trace("cbrp SAVVVE by re-handle it %d -> %d old src
route %s", iph->src(), iph->dst(), p
.route.dump());
        cbrph->init();
        handlePktWithoutSR(p, false);
        return;
    }

    //we failed in all previous attempts, no hope, say
byebye to the packet
    trace("Ssalv %.5f _%s_ dropping --- %d %s -> %s %s CBR
%d",
        Scheduler::instance().clock(),

```



```

        net_id.dump(), cmh->uid(), p.src.dump(),
p.dest.dump(), p.route.dump(), cbrph->valid());
        if (mine) drop(pkt, DROP_RTR_NO_ROUTE);

}

#ifdef USE_GOD_FEEDBACK
static int linkerr_is_wrong = 0;
#endif

void
CBRP_Agent::xmitFailed(Packet *pkt)
/* this function is called when the link layer fails to send
out a cbrp packet
we will firstly use this error information to update
neighbor table, adjacency table etc.
then we will run our local repair algorithm to save data
packets and RREP
as well as sending out route error message to the source if
it's data packet */
{
    hdr_cbrp *cbrph = (hdr_cbrp*)pkt->access(off_cbrp_);
    hdr_ip *iph = (hdr_ip*)pkt->access(off_ip_);
    hdr_cmn *cmh = (hdr_cmn*)pkt->access(off_cmn_);

    ID tell_id;
    ID from_id;
    ID to_id;

    assert(cmh->size() >= 0);

    if (verbose_)
        trace("SSendFailure %.9f %s_ --- %d - %s, (to %d) RREQ
%d(%d,%d), RREP %d",
            Scheduler::instance().clock(), net_id.dump(), cmh-
>uid(), cbrph->dump(), cmh->next_hop(), cbrph-
>request_destination(), iph->src(), cbrph->rtreq_seq(), cbrph-
>route_reply());

    //Jinyang -first update neighbor table to correct not-so-
fast timeout event leaveouts
    //again - we did not take into consideration of the uni-
directional links
    ntable_ent *ent;
    if ((ent = ntable->GetEntry(cmh->next_hop())) {
#ifdef CBRP_DEBUG
        trace("cbrp %d_ delete %d from neighbor table last update
%.9f", myaddr_, cmh->next_hop(), ent->last_update);
#endif
        ntable->DeleteEntry(cmh->next_hop());
    }
}

```



```

// if it's a gratuitous route reply about a repaired route,
just drop it silently
if (cbrph->route_reply() && (cbrph->route_repaired() || cbrph-
>route_shortened())) {
    Packet::free(pkt);
    return;
}

cmh->size() -= cbrph->size(); //recalculate the size

from_id = net_id;
to_id.addr = cmh->next_hop();
to_id.type = (ID_Type)AF_INET;

if (!cbrph->route_request() && !cbrph->route_reply() &&
!cbrph->route_repaired()) {
    //if this is a unrepaired data packet, we have to inform
the packet src about the error
    //the packet src may not be the packet originator
    tell_id.addr = cbrph->addrs[0].addr;
    tell_id.type = (ID_Type) AF_INET;
}

#ifdef USE_GOD_FEEDBACK
if (God::instance()->hops(from_id.getNSAddr_t(),
to_id.getNSAddr_t()) == 1)
{ /* god thinks this link is still valid */
    linkerr_is_wrong++;
    trace("SxmitFailed %.5f %s %d->%d god okays #d",
Scheduler::instance().clock(), net_id.dump(),
from_id.getNSAddr_t(), to_id.getNSAddr_t(),
linkerr_is_wrong);
    fprintf(stderr,
"xmitFailed on link %d->%d god okays - ignoring
& recycling #d\n"
,
from_id.getNSAddr_t(), to_id.getNSAddr_t(),
linkerr_is_wrong);
    /* put packet back on end of ifq for xmission */
    // make sure we aren't cycling packets
    assert(p.pkt->incoming == 0); // this is an outgoing
packet
    ll->recv(pkt, (Handler*) 0);
    return;
}
#endif

/* kill any routes we have using this link */
route_cache->noticeDeadLink(from_id, to_id,
Scheduler::instance().clock());

/* give ourselves a chance to save the packet, only nodes on
the original route save the pkt, local repair nodes do not */
undeliverablePkt(pkt->copy(), 1);
//if (Scheduler::instance().clock()>78.9) printf("2\n");

```



```

/* now kill all the other packets in the output queue that
would
use the same next hop. This is reasonable, since 802.11
has
already retried the xmission multiple times => a
persistent failure. */
Packet *r, *nr, *head = 0; // pkts to be recycled
while((r = ifq->filter(to_id.getNSAddr_t())) {
    r->next_ = head;
    head = r;
}

hdr_cbrp *cbrph_tmp;

for(r = head; r; r = nr) {
    nr = r->next_;
    cbrph_tmp = (hdr_cbrp*)r->access(off_cbrp_);
    cmh->size() -= cbrph_tmp->size();
    // if it's a gratuitous route reply about a repaired
route, just drop it silently
    if (cbrph_tmp->route_reply() && (cbrph_tmp->
>route_repaired() || cbrph_tmp->route_shortened())) {
        Packet::free(pkt);
        return;
    }
    undeliverablePkt(r, 1);
}

/* warp pkt into a route error message */
if (tell_id == net_id || tell_id == MAC_id )
{ // no need to send the route error if it's for us
    if (verbose_)
        trace("Sdebug %s_ not bothering to send route error
to myself",
            tell_id.dump());
    Packet::free(pkt);
    pkt = 0;
    return;
}

if (cbrph->route_request() || (cbrph->route_reply()) ||
cbrph->route_repaired() || cbrph->route_shortening()) {
    Packet::free(pkt);
    pkt = 0;
    return;
}

if (cbrph->num_route_errors() >= MAX_ROUTE_ERRORS)
{ // no more room in the error packet to nest an
additional error.
    // this pkt's been bouncing around so much, let's just
drop and let
    // the originator retry

```



```

        // Another possibility is to just strip off the outer
        error, and
        // launch a Route discovery for the inner error XXX -dam
6/5/98
        trace("SDFU %.5f %s_ dumping maximally nested error %s
%d -> %d",
            Scheduler::instance().clock(), net_id.dump(),
            tell_id.dump(),
            from_id.dump(),
            to_id.dump());
        Packet::free(pkt); // no drop needed
        pkt = 0;
        return;
    }

    cbrph->cur_addr()--;

    link_down_cbrp *deadlink = &(cbrph->down_links()[cbrph-
>num_route_errors()]);
    deadlink->addr_type = cbrph->addrs[cbrph-
>cur_addr()].addr_type;
    deadlink->from_addr = cbrph->addrs[cbrph->cur_addr()].addr;
    deadlink->to_addr = cbrph->addrs[cbrph->cur_addr()+1].addr;
    deadlink->tell_addr = cbrph->addrs[0].addr;
    cbrph->num_route_errors() += 1;

    if (verbose_)
        trace("Sdebug %.5f %s_ sending into dead-link (nest %d)
tell %d %d -> %d",
            Scheduler::instance().clock(), net_id.dump(),
            cbrph->num_route_errors(),
            deadlink->tell_addr,
            deadlink->from_addr,
            deadlink->to_addr);

    cbrph->valid() = 1;

    cbrph->route_error() = 1;
    cbrph->route_reply() = 0;
    cbrph->route_request() = 0;

    iph->dst() = deadlink->tell_addr;
    iph->dport() = RT_PORT;
    iph->src() = net_id.addr;
    iph->sport() = RT_PORT;
    iph->ttl() = 255;

    cmh->ptype() = PT_CBRP;
    cmh->num_forwards() = 0;
    cmh->size() = IP_HDR_LEN; //cut off data in the original
packet
    // assign this packet a new uid, since we're sending it
    cmh->uid() = uidcnt_++;
    cbrph->num_addrs() = cbrph->cur_addr()+1;

```



```

    CBRP_Packet p(pkt, cbrph);
    p.route.setLength(p.route.index()+1);
    p.route.reverseInPlace();
    p.route.resetIterator();

    fillCBRPPath(p.route, cbrph);

    p.dest = tell_id;
    p.src = net_id;

    /* send out the Route Error message */
    sendOutPacketWithRoute(p, true);
}

void
CBRP_XmitFailureCallback(Packet *pkt, void *data)
{
    CBRP_Agent *agent = (CBRP_Agent *)data; // cast of trust
    agent->xmitFailed(pkt);
}

/* fill the path information in CBRP header */
void CBRP_Agent::fillCBRPPath(Path &path, hdr_cbrp *& cbrph)
{
    for (int i=0; i<path.length(); i++) {
        path[i].fillSRAddr(cbrph->addrs[i]);
    }
    cbrph->num_addrs() = path.length();
    cbrph->cur_addr() = path.index();
}

```

Wireless scenarios

1.

```

=====
# Define options
#
=====
set val(chan) Channel/WirelessChannel ;# channel
type
set val(prop) Propagation/TwoRayGround ;# radio-
propagation model
set val(netif) Phy/WirelessPhy ;# network
interface type
set val(mac) Mac/802_11 ;# MAC type

```



```

set val(ifq)                Queue/DropTail/PriQueue   ;#
interface queue type
set val(ll)                 LL                       ;# link
layer type
set val(ant)                Antenna/OmniAntenna     ;# antenna
model
set val(ifqlen)             50                       ;# max
packet in ifq
set val(nn)                 2                       ;# number
of mobilenodes
set val(rp)                 DSDV                     ;# routing
protocol

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

#
# Initialize Global Variables
#
set ns_                      [new Simulator]
set tracefd                  [open simple.tr w]
set namtrace                  [open prowireless.nam w]
$ns_ namtrace-all-wireless $namtrace 500 500
$ns_ trace-all $tracefd

# set up topography object
set topo                      [new Topography]

$topo load_flatgrid 500 500

#
# Create God
#
create-god $val(nn)

#
# Create the specified number of mobilenodes [$val(nn)] and
"attach" them
# to the channel.
# Here two nodes are created : node(0) and node(1)

set chan [new $val(chan)]
# configure node

$ns_ node-config -adhocRouting $val(rp) \
                -llType $val(ll) \
                -macType $val(mac) \
                -ifqType $val(ifq) \
                -ifqLen $val(ifqlen) \

```



```

        -antType $val(ant) \
        -propType $val(prop) \
        -phyType $val(netif) \
        -channel $chan \
        -topoInstance $topo \
        -agentTrace ON \
        -routerTrace ON \
        -macTrace OFF \
        -movementTrace OFF

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

#
# Provide initial (X,Y, for now Z=0) co-ordinates for
mobilenodes
#
$node_(0) set X_ 5.0
$node_(0) set Y_ 2.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 390.0
$node_(1) set Y_ 385.0
$node_(1) set Z_ 0.0

#
# Now produce some simple node movements
# Node_(1) starts to move towards node_(0)
#
$ns_ at 50.0 "$node_(1) setdest 25.0 20.0 15.0"
$ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"

# Node_(1) then starts to move away from node_(0)
$ns_ at 100.0 "$node_(1) setdest 490.0 480.0 15.0"

# Setup traffic flow between nodes
# TCP connections between node_(0) and node_(1)

set tcp [new Agent/TCP]
$tcp set class_ 2
set sink [new Agent/TCPSink]
$ns_ attach-agent $node_(0) $tcp
$ns_ attach-agent $node_(1) $sink
$ns_ connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns_ at 10.0 "$ftp start"

#
# Tell nodes when the simulation ends
#
for {set i 0} {$i < $val(nn) } {incr i} {

```

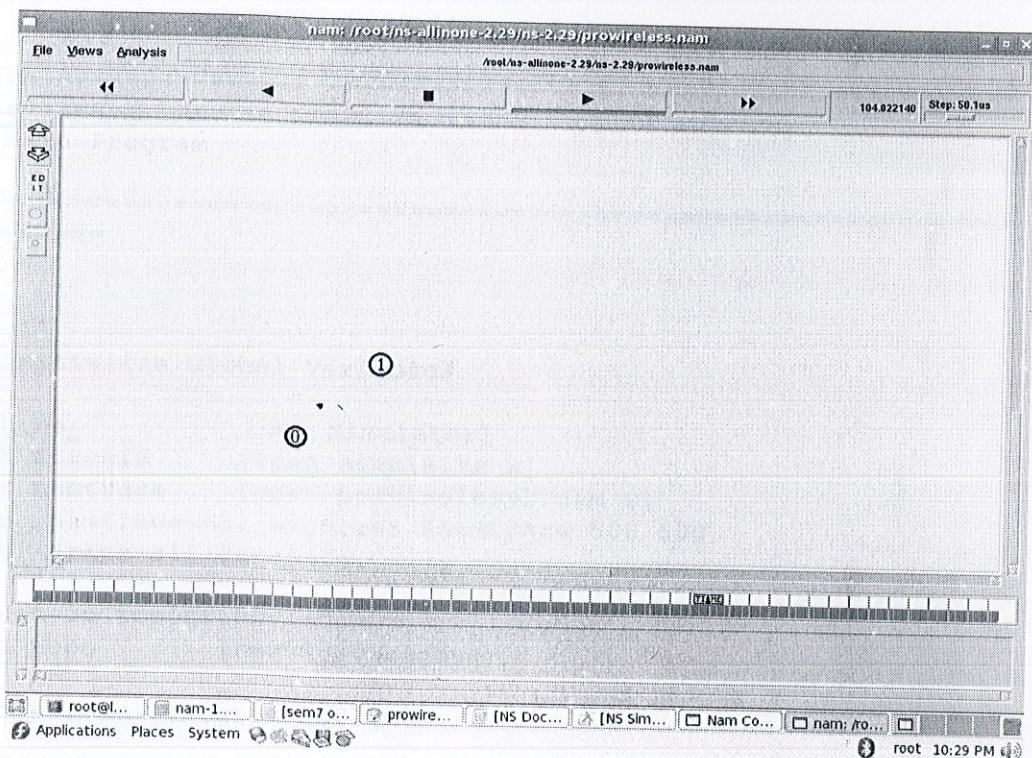


```

    $ns_ at 150.0 "$node_($i) reset";
}
$ns_ at 150.0001 "stop"
$ns_ at 150.0002 "puts \"NS EXITING...\" ; $ns_ halt"
proc stop {} {
    global ns_ tracefd namtrace
    $ns_ flush-trace
    close $tracefd
    close $namtrace
}

puts "Starting Simulation..."
$ns_ run

```



2.

```

#-----
=====
# Define options
#
#-----
=====
set val(chan)                Channel/WirelessChannel    ;# channel
type

```



```

set val(prop)          Propagation/TwoRayGround    ;# radio-
propagation model
set val(netif)        Phy/WirelessPhy           ;# network
interface type
set val(mac)          Mac/802_11                 ;# MAC type
set val(ifq)          Queue/DropTail/PriQueue    ;#
interface queue type
set val(ll)           LL                         ;# link
layer type
set val(ant)          Antenna/OmniAntenna       ;# antenna
model
set val(ifqlen)       50                         ;# max
packet in ifq
set val(nn)           5                          ;# number
of mobilenodes
set val(rp)           DSDV                       ;# routing
protocol

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

#
# Initialize Global Variables
#
set ns_                [new Simulator]
set tracefd            [open simple.tr w]
set namtrace           [open prowireless2.nam w]
$ns_ namtrace-all-wireless $namtrace 500 500
$ns_ trace-all $tracefd

# set up topography object
set topo               [new Topography]

$topo load_flatgrid 500 500

#
# Create God
#
create-god $val(nn)

#
# Create the specified number of mobilenodes [$val(nn)] and
"attach" them
# to the channel.
# Here two nodes are created : node(0) and node(1)

set chan [new $val(chan)]
# configure node

```



```

$ns_ node-config -adhocRouting $val(rp) \
    -llType $val(ll) \
    -macType $val(mac) \
    -ifqType $val(ifq) \
    -ifqLen $val(ifqlen) \
    -antType $val(ant) \
    -propType $val(prop) \
    -phyType $val(netif) \
    -channel $chan \
    -topoInstance $topo \
    -agentTrace ON \
    -routerTrace ON \
    -macTrace OFF \
    -movementTrace OFF

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

#
# Provide initial (X,Y, for now Z=0) co-ordinates for
mobilenodes
#
$node_(0) set X_ 5.0
$node_(0) set Y_ 2.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 390.0
$node_(1) set Y_ 385.0
$node_(1) set Z_ 0.0

$node_(2) set X_ 250.0
$node_(2) set Y_ 250.0
$node_(2) set Z_ 0.0

$node_(3) set X_ 150.0
$node_(3) set Y_ 145.0
$node_(3) set Z_ 0.0

$node_(4) set X_ 167.0
$node_(4) set Y_ 186.0
$node_(4) set Z_ 0.0

#
# Now produce some simple node movements
# Node_(1) starts to move towards node_(0)
#
$ns_ at 50.0 "$node_(1) setdest 25.0 20.0 15.0"
$ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"
$ns_ at 15.0 "$node_(2) setdest 167.0 186.0 2.0"
$ns_ at 25.0 "$node_(3) setdest 28.0 45.0 7.0"

```


REFERENCES

- [1] “Simplified clustering Scheme for Intrusion Detection in Mobile Ad-hoc Networks” – Kashan Samad, Ejaz Ahmed, Waqar Mahmood – NUST Institute of Information Technology(NIIT), Rawalpindi, Pakistan.
- [2] “Cluster-based Intrusion Detection (CBID) Architecture for Mobile Ad Hoc Networks” – Ejaz Ahmed, Kashan Samad, Waqar Mahmood – NUST Institute of Information Technology (NIIT), Rawalpindi, Pakistan.
- [3] NS Simulator for Beginners – Lecture notes, 2003-2004, University de Los Andes, Merida, Venezuela and ESSI, Sophia-Antipolis, France – Eitan Altman and Tania Jimenez, December 4, 2003.
- [4] The ns Manual, The VINT Project, A collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC, Edited by Kevin Fall and Kannan Varadhan, March 24, 2008.
- [5] “Sybil attack detection techniques in mobile ad-hoc networks” – Sarit Pal, Achyut Sharma, Amol Vasudeva, Sanatya Singh, Debojyoti Saha, Anand Mohan Sinha and S.K. Kak, March 2007.
- [6] Internet blogs – ns-users@isi.edu and sites – Wikipedia.org etc.


```

# Node_(1) then starts to move away from node_(0)
$ns_ at 100.0 "$node_(1) setdest 490.0 480.0 15.0"

# Setup traffic flow between nodes
# TCP connections between node_(0) and node_(1)

set tcp [new Agent/TCP]
$tcp set class_ 2
set sink [new Agent/TCPSink]
$ns_ attach-agent $node_(0) $tcp
$ns_ attach-agent $node_(1) $sink
$ns_ connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns_ at 10.0 "$ftp start"

#
# Tell nodes when the simulation ends
#
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at 150.0 "$node_($i) reset";
}
$ns_ at 150.0001 "stop"
$ns_ at 150.0002 "puts \"NS EXITING...\" ; $ns_ halt"
proc stop {} {
    global ns_ tracefd namtrace
    $ns_ flush-trace
    close $tracefd
    close $namtrace
}

puts "Starting Simulation..."
$ns_ run

```

