Comparative Performance Study of Standardized Ad-Hoc Routing Protocols and OSPF-MCDS

Palaniappan Annamalai

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

October, 2005
Blacksburg, Virginia

Keywords: Ad-hoc comparative study, ns-2 routing, ns2 implementation, OSPF-MCDS

Comparative Performance Study of Standardized Ad-Hoc Routing Protocols and OSPF-MCDS

Palaniappan Annamalai

Dr. Scott F. Midkiff, Chair
Electrical and Computer Engineering

ABSTRACT

The development of ubiquitous mobile computing devices has fueled the need for dynamic reconfigurable networks. Mobile ad-hoc network (MANET) routing protocols facilitate the creation of such networks, without centralized infrastructure. One of the challenges in the study of MANET routing protocols is the evaluation and design of an effective routing protocol that works at low data rates and responds to dynamic changes in network topology due to node mobility. Several routing protocols have been standardized by the Internet Engineering Task force (IETF) to address ad-hoc routing requirements. The performance of these protocols are investigated in detail in this thesis.

A relatively new approach to ad-hoc routing using the concept of a Minimal Connected Dominating Set (MCDS) has been developed at Virginia Tech. The OSPF-MCDS routing protocol is a modified version of the traditional Open Shortest Path First (OSPF) wired routing protocol which incorporates the MCDS framework. Enhancements to the protocol implementation to support multiple-interface routing are presented in this thesis. The protocol implementation was also ported to ns-2, a popular open source network simulator.

Several enhancements to the implementation and simulation model are discussed along with simulation specifics. New scenario visualization tools for mobility pattern generation and analysis are described. A generic framework and tutorial for developing new ad-hoc routing simulation models are also presented. The simulation model developed is used to compare the performance characteristics of OSPF-MCDS to three different standardized MANET routing protocols. Simulation results presented here show that no single protocol can achieve optimal performance for all mobility cases. Different observations from simulation experiments are summarized that support the likely candidate for different mobility scenarios.

Acknowledgements

I want to thank my advisors, Dr. Scott F. Midkiff and Dr. Jhang S. Park, for their unending patience and guidance during my graduate study and the course of this research. I am especially grateful to Dr. Midkiff for the opportunity he provided me to work with him, as well as his support, mentoring, invaluable insight and advice.

I wish to thank Dr. Thomas Hou for all his support and counsel during my graduate study and Dr. Shiwen Mao for his invaluable input in this research. I also want to express my sincere thanks to Tao Lin without whom this thesis topic and research would not have been possible.

I enjoyed working with members of the Laboratory for Advanced Networking and especially thank George C. Hadjichristofi, John D. Wells and Kaustubh Phanse for their enthusiasm and company. “Thank you” to all my friends for putting up with me when I needed a break from my research.

I am indebted to Karthik Channakeshava and Rashimi Kumar for all their feedback, support and encouragement; their input was really instrumental in shaping this thesis. The quality of this thesis has been vastly improved due to the editing efforts of Dr. Sara Thorne-Thomsen. Special thanks to my parents and brother for their unflagging support and encouragement.

I dedicate this thesis to my father Ganesan Annamalai, and my teacher, Mary George, without whom none of this would have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ever-increasing array of ubiquitous computing devices has fueled the need for a new class of routing techniques, because routing protocols for wired networks fall short of meeting expectations when put to use in wireless environments where there is a high degree of node mobility. A mobile ad-hoc network (MANET) consists of a group of mobile nodes forming a self-organized network, hence the name. It addresses the shortcomings in traditional wired network routing protocols, where a central router controls the different routes for traffic. In contrast, every host is forced to act as a router in a MANET.

MANET routing protocols are broadly classified as being either proactive or reactive. Proactive routing protocols use periodic messages to create and maintain routes, whereas reactive protocols create routes on demand. Some well-known proactive protocols are Optimized Link State Routing (OLSR) [1][2] and Topology Broadcast Based On Reverse-Path Forwarding (TBRPF) [3][4]. Some well-known reactive protocols are Dynamic Source Routing (DSR) [5][6], Temporally Ordered Routing Algorithm (TORA) [7][8] and Ad-Hoc On-Demand Distance Vector Routing (AODV) [9][10]. The Internet Engineering Task Force (IETF) [11] has standardized ADOV, OLSR, and TBRPF protocols, but the performance characteristics of each of these routing protocols under different mobility scenarios remains to be seen.

1.1 Problem Statement

A lot of competitive research is going on to find optimal solutions for MANET routing protocols. The challenges in this field are to design an effective routing protocol that responds to dynamic changes in node connectivity and works at low data rates. The primary concerns of ad-hoc routing protocols remain connectivity and reduced control overhead. Proactive routing protocols use different techniques to minimize their control overhead and achieve higher throughput. A new approach to ad-hoc routing using

Minimal Connected Dominating Set (MCDS) has been proposed by Tao Lin of Virginia Tech [12], a promising addition to the list of proactive protocols. This thesis, therefore, aims to provide an unbiased comparison of the existing standardized MANET routing protocols and the MCDS-based routing protocol to highlight the performance, strengths and weaknesses of each.

The new routing protocol, which is a modified version of the traditional Open Shortest Path First (OSPF) [13] wired routing protocol called OSPF-MCDS, incorporates the MCDS framework. In their specification draft [14], the designers of OSPF-MCDS have proposed the protocol specification and packet formats. The OSPF-MCDS routing agent has been implemented as a standalone Linux routing daemon. The present researcher has made improvements to the program and extended it to include a multiple interface version, thus making it possible to run the protocol on a multi-homed mobile node. With input from Lin, the original author of the algorithm, it has been possible to introduce a clustering concept, which closely resembles hierarchical routing, to implement multiple interface routing. So far, it appears that this is the first MANET routing implementation to offer multiple interface functionality. Since it is not possible to study the performance and feasibility of MANET routing protocols in an actual physical network with a large number of mobile nodes, the Linux implementation was ported to a popular open source network simulator ns-2 [15] to study scalability issues. A generic template based tutorial was created to facilitate the creation of new routing agents in ns2, which will help other researchers to easily port new routing protocols to ns-2. The three routing protocols OLSR, AODV and TBRBF are used as a basis of comparison for investigating the relative performance of the newly proposed OSPF-MCDS MANET routing protocol by means of simulation studies. Since one of the shortcomings of node mobility model generators is that they do not let researchers visualize the various scenarios, this study introduces some new tools to visualize node mobility patterns, because this can effectively help in studying mobility scenarios. In the course of this study a major bug in the network simulator was discovered, fixed and reported to the developers of these computer programs to benefit the research community.

The three standardized routing protocols, OSLR, AODV and TBRPF, were chosen for this study because they are available as IETF “Request for Comments” (RFCs). RFCs contain implementation specifics and aid in faster protocol evaluation. Furthermore, these three protocols represent a good sampling of the existing MANET routing protocol spectrum. The other advantage with standardized protocols is that their experimental evaluation results and simulation model are available.

1.2 Organization

The organization of this thesis is as follows: Chapter 2 provides a detailed introduction to routing protocols OLSR, AODV and TBRPF and the newly created OSPF-MCDS protocol. Chapter 3 discusses the implementation and simulation methodology used in the study. Chapter 4 is devoted to analysis of the different results obtained by various simulation runs. Chapter 5 includes the results, draws some conclusions about them and provides suggestions for further research on MANET.

Chapter 2

Background

Routing in infrastructure-based wireless and wired networks involves centralized routing, which computes optimal routes to a given destination using cost metrics like hop count or bandwidth. While this works well for wired networks, it does not scale well to dynamically changing environments such as a mobile ad-hoc network (MANET). In ad-hoc environments, mobile hosts, which want to communicate with one another, are required to form a self-organized network, which requires each node to perform multi-hop routing. The dynamic nature of ad-hoc networks, combined with topology changes caused by signal fading, interference loses, unidirectional links and link connectivity changes, needs more effective routing protocols to address the need for faster convergence and low control overhead in mobile scenarios.

2.1 Classification of Routing Protocols

Routing protocols are broadly classified as distance vector and link state routing. Distance vector routing is a decentralized routing algorithm. Each node that participates in routing exchanges its estimated least cost path to its directly connected neighbors to all other nodes in the network. Since no single node has a global view of the network in the distance vector, convergence is slow.

Link state routing is a global routing algorithm in which each node computes the shortest path to every other node in the network using global knowledge about the network. In link state routing protocols, each node reliably broadcasts the link state (cost) to its directly connected neighbors. This reliable flooding gives a global topology view to each node. Link state algorithms offer better reliability and solve count-to-infinity and looping issues associated with distance vector routing protocols. The widely-used Open Shortest Path First (OSPF) routing protocol is a link state protocol.

Topology-based wireless routing protocols are also broadly classified as proactive, reactive, and hybrid. Proactive routing protocols use periodic broadcasts to establish routes and maintain them; examples are Optimized Link State Routing (OLSR) [1][2] and Topology Broadcast Based On Reverse-Path Forwarding (TBRPF) [3][4]. Since they exchange topology information enabling each node to maintain an up-to-date view of the network, proactive protocols are also called table-driven protocols. The topology exchange can happen periodically (e.g. as in OLSR and TBRF) or on an event driven basis (e.g. as in DSDV and TORA). Proactive protocols can effectively route packets immediately to any other node in the network and do not suffer from a high starting latency. However, the periodic topology exchange results in a larger overhead especially when node mobility is high.

Reactive protocols create routes on demand by sending route request messages when a new route is needed. Reactive protocols trace the reply messages to construct optimum paths to the destination. Since route discovery is done only on an as-needed basis, the control overhead is smaller than it is in proactive protocols. However, these protocols suffer from high route determination latency. Some well-known reactive protocols are Dynamic Source Routing (DSR) [5] and Ad-Hoc On-Demand Distance Vector Routing (AODV) [9]. Protocols that combine both reactive and proactive measure are hybrid routing protocols. One example of a hybrid MANET routing protocol is Temporally Ordered Routing Algorithm (TORA) [7].

2.2 MANET Routing Protocol Expectations

MANET routing protocols are responsible for creating routes in a dynamically changing network with low bandwidth, low power and resource constrained computing nodes. Murthy and Manoj discuss in detail the design goals of MANET routing protocols [16]. These protocols are designed with the following primary expectations:

1. Provide stable loop free connectivity,
2. Have reduced control overhead,
3. Respond to dynamic changes in node mobility,
4. Have scalability and distributed routing,

5. Support QoS traffic prioritization, and
6. Provide secure routing.

Traditional link state protocols generate periodic broadcasts of link state messages. To create a controlled reliable flooding, wired network routers broadcast on all interfaces except the one on which they receive the broadcast. This eventually limits the broadcast scope. In a wireless network, such reliable flooding is not possible, because conventional wireless adapters broadcast in an omni-directional manner.

As outlined in the introduction, the four routing protocols, OLSR, AODV, TBRPF and OSPF-MCDS, under investigation are discussed in detail in the following sections. Each of these routing protocols uses different techniques to optimize the connectivity and reduce control overhead.

2.3 Optimized Link State Routing Protocol (OLSR)

The proactive OLSR [1] adapts a classical link state protocol for mobile ad hoc routing. As a proactive routing protocol, it uses periodic messages to update topology information at each node. In a classical link state protocol, the link state packet includes the entire neighbor list along with the associated link cost metric, thus generating large control packet overheard. Furthermore, these packets are broadcast to the entire network which does not scale well to the low bandwidth requirements of wireless ad-hoc networks. OLSR optimizes the classical link state protocol by reducing the control packet overhead and creating efficient flooding mechanisms.

OLSR tries to contain duplicate broadcasts and limit broadcast domain by using a Multi-Point Relay (MPR) set. The concept behind using MPR is to choose nodes in a network that will effectively cover the entire network. These nodes, called MPR nodes, are defined as one-hop neighbors with which there is bi-directional connectivity, and they, in turn, cover all the two-hop neighbors of a given node.

Each node maintains two sets of nodes, a MPRset and a MPRselectorset. The MPRset consists of the set of MPR nodes which the current node has selected, and the MPRselector consists of a set of nodes that have selected the current node as a MPR node. These MPR nodes act as the forwarding stations when they receive data from or destined to the nodes in its MPRselectorset. Selecting MPR nodes as the forwarding stations reduces the link state information because only the link state connectivity of the MPR node needs to be included in the link state control packets since a MPR node effectively represents its selector nodes. This reduces the size of the link state packets, thereby satisfying reducing overhead for OLSR.

Initially a node starts off with an empty MPRset and MPRselectorset. All one-hop neighbor nodes are considered as MPR nodes. This set decreases in size over time as the HELLO messages are received, because over a period of time the node will learn all its two-hop and MPR neighbors. OLSR uses three important elements: a neighbor sensing element, an efficient message flooding element, and a topology dissemination element. OLSR employs a simple neighbor sensing scheme to detect the neighbor link status, but does not rely on link level acknowledgements to detect link status. The link status can have three possible states: unidirectional, bidirectional and MPR. OLSR sends out a HELLO message periodically with a list of neighbors from which it has heard, along with the neighbor link status. A node receiving a HELLO message for the first time from a neighbor marks the link as unidirectional in the local neighbor table and includes the neighbor identifier (ID) in its next HELLO message. The neighbor receiving this HELLO message on finding its node ID determines the link is bidirectional. Then, for subsequent HELLO messages from this neighbor, the link is marked bidirectional.

Each node employs a distributed approximation algorithm to compute its MPRset and marks the corresponding node links as MPR in its local neighbor table. A neighbor marked MPR means the neighbor link is bidirectional and also the neighbor is a MPR for the current node. HELLO messages are only broadcasted to neighbors and are not relayed further. Each node learns all of its two hop neighbors through the periodic HELLO message. In addition the HELLO messages broadcast the transmitting nodes MPRset.

From the HELLO messages, the nodes know if they have been selected as a MPR. If they have, they place the corresponding node in its MPRselectorset.

To disseminate link state topology information in the network, each MPR node with a non-empty MPRselectorset periodically broadcasts a Topology Control (TC) message. The TC messages contain the MPR node ID and its MPRselectorset. Other MPR nodes receiving the broadcasts relay this information. Using the topology information obtained from TC messages, the nodes can compute the shortest path to every node in the network and form the routing table. An important point to note here is that the routes in OLSR always contain MPR nodes as the forwarding agents. Therefore, OLSR does not always construct a shortest path, but does guarantee a path to the destination.

According to the protocol specification [2], OLSR performs well in a highly dense network with sporadic node movement. This characteristic can be attributed to OLSR being a proactive protocol and having routes always available. Clausen, Hansen, Christensen and Behrmann have discussed a method of using random jitter delays to avoid packet collisions during transmission [17]. Avoiding packet collisions is a basic requirement of an efficient ad-hoc routing algorithm. The concept of using small random transmission delays is used by the authors of OSPF-MCDS as well.

2.3.1 MPR Selection Example

Figure 2.1 provides an example of a network where node 6 sends out its control packets. In a typical network flooding, the nodes broadcast the packet out to all their neighbors, which results in redundant broadcast traffic.

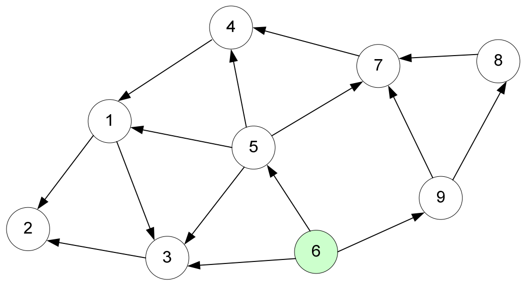


Figure 2.1: Traditional network flooding

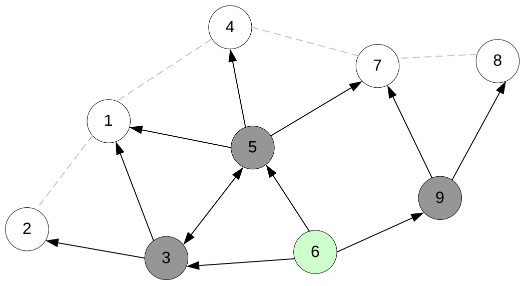


Figure 2.2: MPR selection
MPRset of node 6 = {3,5,9}
MPRSelectorset of node 3 = { 6, 2 }

In contrast, when OLSR is used as the routing protocol, as Figure 2.2 shows node 6 selects node 3, 5 and 9 as its MPR set. The arrows indicate the source of the transmission and the grayed-nodes are the MPR nodes of node 6. The MPR nodes forward only the control packets originating at node 6, thereby reducing the total number of transmissions needed to propagate the information to all the nodes.

2.3.2 Advantages and Disadvantages

The advantage of OLSR is that it reduces control information and efficiently minimizes broadcast traffic bandwidth usage. Although OLSR provides a path from source to destination, it is not necessarily the shortest path, because every route involves forwarding through a MPR node. A further disadvantage is that OLSR also has routing delays and bandwidth overhead at the MPR nodes as they act as localized forwarding routers.

2.4 Ad-Hoc On-Demand Distance Vector Routing (AODV)

AODV [9] uses a route discovery process to dynamically build new routes on an as need basis. AODV is a distributed algorithm using distance vector algorithms, such as the Bellman Ford algorithm. When a route to a destination is unknown, AODV creates a route request packet and broadcasts it to its neighbors. Route request messages contain the source ID, destination ID, source sequence numbers, destination sequence numbers, hop count and broadcast ID. The source sequence number and broadcast ID increment each time a new route request is generated. The destination sequence number is the source sequence number of the destination node as last recorded by the source node.

Each intermediate node receiving a route request caches the previous hop for the particular node originating the request; this helps to create a return path for the reply packets. AODV uses the destination sequence number to maintain freshness of routes. The destination node or any intermediate node can reply to a route request. If an intermediate node has previously learned the path to the destination node, it can reply with the next hop information only if it satisfies the following condition: the locally stored destination sequence number is higher or comparable to the destination sequence number in the route request packet. AODV relies heavily on the sequence numbers to avoid the count-to-infinity problem associated with distance vector protocols. The broadcast ID and source ID pair help in discarding any redundant requests that reach a node. The replying destination or intermediate node unicasts a route reply message to the specific source node that created the route request. Nodes receiving a route reply message store the source ID of the node forwarding the message as the next hop towards the destination in order to forward future traffic toward this destination.

The hop count in each message is incremented by one at each forwarding node, which helps track the distance to the source or destination node depending on the type of the message. A node generating a route request or route reply sets the hop count to zero, which is incremented at each intermediate forwarding node. This incrementing helps the intermediate node to determine the number of hops to reach the source or destination using the current path. The source node receiving a number of route replies from different paths uses the hop count in the route reply messages to choose the one with a lower hop count metric as the shortest route to the destination. Once a route is formed, AODV uses the current route until the route expires or any topology changes occur. Each node also maintains a “precursor list” [10] of nodes that help it identify the nodes it has to inform of a broken link. The “precursor list” is created from the route request packets and includes a list of nodes that are likely to use the current node as the next hop.

Each node monitors the status of each of its links, and when a link connectivity change occurs, the node creates a route error message and informs the members of the “precursor list” about the non-reachability of specific routes. AODV relies on medium access control (MAC) layer schemes or the use of beacon packets at periodic intervals to find the status of its directly connected neighbors. Topology changes or expiring timers associated with the route request, reply and beacon packets allow AODV to detect link failures.

AODV uses a progressive ring search technique to control the broadcast domain. Basically, it increases the time-to-live (TTL) value in each broadcast of the initial route request until it receives a route reply. AODV, however, only works on symmetric links although Nesargi and Prakash have proposed extensions for ADOV in environments with unidirectional links [18].

2.4.1 AODV Example

Figure 2.3 depicts a network where in node 1 desires to communicate to node 8. The AODV modules running on node 1 flood the network with route request (RREQ) messages. Each node receiving a RREQ message stores the previous hop and distance to source for the originating RREQ and forwards the RREQ to its neighbors.

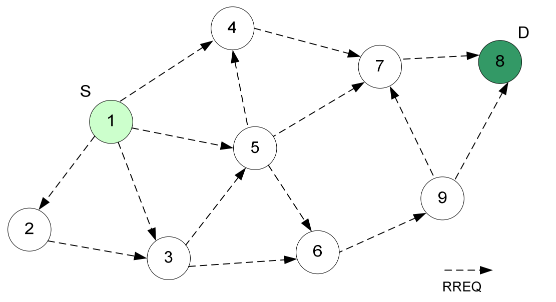


Figure 2.3: Route request (RREQ) flooding

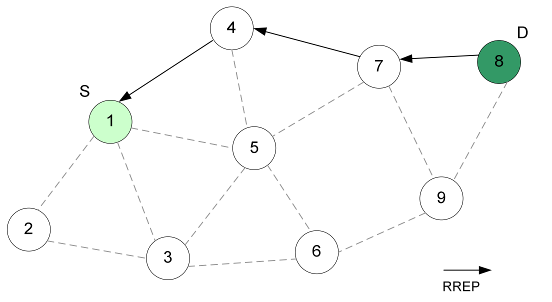


Figure 2.4: Route reply propagation

When the RREQ message reaches the designation node 8, the destination sends a unicast route reply (RREP) message back to the source using the previous hop on which it received the RREQ. Each node receiving the RREP message in turn forwards it to the next hop with the smallest distance to the source as shown in Figure 2.4. This process effectively builds the routing table at each node, and when any source destination pair establishes a route, the intermediate nodes learn the route as well.

2.4.2 AODV Advantages and Disadvantages

The advantage of AODV is that it creates routes only on demand, which greatly reduces the periodic control message overhead associated with proactive routing protocols. The disadvantage is that there is route setup latency when a new route is needed, because

ADOV queues data packets while discovering new routes and the queued packets are sent out only when new routes are found. This situation causes throughput loss in high mobility scenarios, because the packets get dropped quickly due to unstable route selection.

2.5 Topology Broadcast Based On Reverse-Path Forwarding (TBRPF)

TBRPF falls into the class of reactive protocols using efficient flooding techniques. Like OLSR, TBRPF has two goals, one to minimize overhead and the other to utilize efficient flooding of control packets.

In TBRPF each node maintains a spanning tree from all nodes to the source. The spanning trees are formed by the shortest path from all other nodes to the source. The messages broadcast by the source are forwarded only in the reverse path along the spanning tree previously generated. Dalal and Metclafe [19] and Bellur and Ogier [3] describe this process, which is called “Reverse Path Forwarding” (RPF). The parent nodes broadcast the control traffic originating at source V to all of its children. In other words, non-leaf nodes of the source tree rooted at each node forward the broadcast. In Figure 2.6, node 5 is the parent node and has children nodes 6 and 8 for a given source V (node 2 in this case).

TBRPF contains two modules, a neighbor discovery module and a routing module. The neighbor discovery module is unique to TBRPF in the sense that it employs differential HELLO messages, which report only changes in link status. TBRPF differs from other protocols, which broadcast the entire one-hop neighbor list in their HELLO messages. TBRPF periodically broadcasts HELLO messages, and these messages contain three categories of lists: neighbor request, neighbor reply and neighbor lost. Every HELLO message also contains an eight bit incremental sequence number HSEQ. The first time node A detects a new neighbor B, it creates a unidirectional link entry for node B in its neighbor table. The subsequent HELLO message from node A lists node B in the neighbor request category. Node B on receiving the HELLO message creates a similar table entry for node A and places node A in its neighbor reply list on the subsequent HELLO message. Thus node A and B determine they have bidirectional connectivity and update the local neighbor tables accordingly. The neighbor discovery messages are never forwarded; instead they are used for one-hop neighbor detection only.

TBRPF includes each status change of its neighbor in at least three consecutive HELLO messages to ensure the neighbors register the change in the link status. HELLO messages always contain a neighbor request list, even when there are no link changes to report, in order to let the neighbors sense the link status of the source node. If a node misses a number of HELLO messages (detected by missing sequence numbers or HSEQ) from a neighbor that exceeds a threshold, it marks the link as lost in its local neighbor-table and reports the change in the subsequent HELLO messages.

TBRPF has two modes of operation, full topology (FT) mode and partial topology (PT) mode. Bellur and Ogier describe the FT mode [3], which with Templin and Lewis they later modify to be the PT [4]. In PT each node receives only a PT graph that is just enough to construct the shortest path to all nodes. The PT mode is used in simulation experiments for this study.

Using a modified version of Dijkistra's algorithm, each node forms a shortest-path source tree to all reachable nodes. TBRPF reports only partial source trees in its link state updates instead of the entire neighbor link costs. The PT graph, which is called the "reportable subtree" (RT), is a sub-graph of the source tree at the node. Unlike other proactive routing protocols, TBRPF does not periodically broadcast FT updates. Instead it broadcasts a combination of periodic and differential updates of the RT. The changes to RT since the last full RT update are broadcast using differential updates; differential updates occur more periodically. TBRPF nodes also broadcast the full RT once every few seconds to allow newly detected nodes to construct the FT. The use of RT and differential updates minimizes control packet overhead.

The nodes forward the topology updates along the reverse path tree for any source. Reverse-path forwarding is achieved by using the information obtained by TBRPF operation. The TBRPF node forwards if the originating source node is a member of the Reportable Node (RN) set computed by TBRPF. Reportable Node set is a set which includes all neighbors *j* of a node *i* if node *i* determines that any of its neighbors might use node *i* as the next hop in the shortest path to node *j*. The algorithm to form a RT is given in TBRPF specifications. The RT at a node contains all the local links to its one-hop neighbors and the sub-trees of the source tree rooted at neighbors in the RN set [4].

2.5.1 TBRPF Example

The network topology shown in Figure 2.5 illustrates the TBRPF reportable tree mechanism. The shortest path tree rooted at node 2 is calculated and represents the source tree at node 2, as shown in Figure 2.6. Since all of the links in the source tree will be used to form shortest path trees to the neighbors of node 2, the reportable tree of node 2 is the same as its source tree, as shown in Figure 2.7.

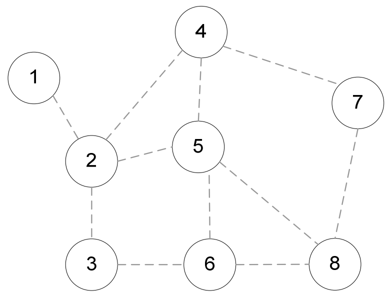


Figure 2.5: TBRPF Sample network topology

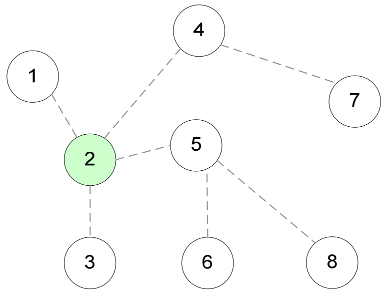


Figure 2.6: Source tree rooted at node 2

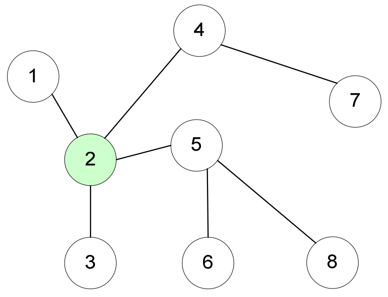


Figure 2.7: Reportable tree at node 2

When node 8 is considered, the source tree rooted at node 8 is shown in Figure 2.8. Here only the links of the source tree which may be used as part of the shortest path trees from node 8's neighbor is included in the reportable tree as shown in Figure 2.9.

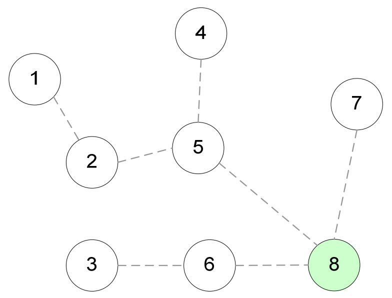


Figure 2.8: Source tree rooted at node 8

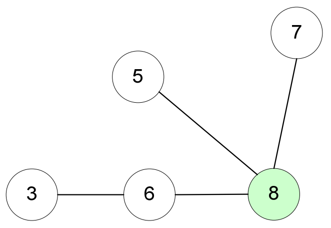


Figure 2.9: Reportable tree at node 8

2.5.2 Advantages and Disadvantages

TBRPF offers many advantages that OLSR and AODV do not. It produces less overhead when there is less node mobility, because the differential updates are related to the rate of the topology changes. TBRPF's reverse path forwarding creates less flooding traffic. The use of Reportable trees only when there is a change in the topology creates very low overhead in TBRPF. TBRPF allows using optional link metrics instead of just hop count to calculate the shortest paths based on the metric, thus making it useful in environments which need to route based on link quality rather than on minimum-hops [4]. TBRPF computes shortest path from every node to all its two hop neighbors; resulting in a computational overhead compared to pure flooding schemes.

2.6 Open Shortest Path First-Minimal Connected Dominating Set (OSPF-MCDS)

The comparatively new proactive protocol, OSPF-MCDS, is an adaptation of the traditional wired routing protocol OSPF. Blind broadcast of control messages as done in OSPF does not scale well in multi-hop low bandwidth MANETS. Protocols, like OLSR, which adapt classical link state protocols, try to minimize the blind-broadcast with various optimization techniques. OSPF-MCDS uses the concept of dominating sets to reduce the broadcast traffic. Let a set G be a set of wireless nodes V and E the edges connecting them, such that G = (V, E). In graph theory, a set S, such that $S \subseteq G$ and every vertex in G – S can connect to at least one vertex in S using an existing edge in G, is called a dominating set. When S is a connected graph, it is called a connected dominating set [20]. The premise here is that the dominating set covers all the nodes in the graph. There might be many such connected graphs covering all the nodes in G. These are the minimal connected dominating sets (MCDS). A good approximation algorithm produces a dominating set size almost equal to the average degree of the graph [20].

The problem of finding a connected dominating set is best solved by using approximation algorithms. Various approximation algorithms and their performance evaluation are presented in [12]. A localized hybrid version of Global Ripple CDS (GLRCDS) and an Updated Local Ripple CDS (ULRCDS) algorithm are used in the implementation OSPF-MCDS, which is documented in the protocol specification [14]. The authors provide the reasons for the use of this hybrid algorithm over the other CDS approximation algorithms [12]. OSPF-MCDS is made of three modules: neighbor discovery, CDS formation and link state topology synchronization. The OSPF-MCDS neighbor discovery mechanism derives from the neighbor discovery scheme in TBRPF, because it uses differential HELLO messages. Changes in link status are reported with Link State Description packets (LSD). When a new link is up between two neighbors, the neighbor with a higher node ID sends a LinkUP LSD message. Both the neighbors broadcast link down events as LinkDown LSD messages.

OSPF-MCDS protocol specifies a modified form of GLRCDS and ULRCDS algorithm for use in the CDS determination module. The algorithm works on a non-fully connected network, segregating the nodes into potential and primary MCDS nodes; after all the iterations the approximate MCDS set is formed. These nodes act as the forwarding agents of the broadcast link state topology updates. It is important to note here that unlike the MPR nodes in OLSR, the MCDS nodes do not act as forwarding routers for the data traffic.

OSPF-MCDS uses Link Database Description messages (LDD) to support periodic topology synchronization. Like OSPF, OSPF-MCDS uses Interface Prefix Description (IPD) messages to support declaring attached networks. LSD messages are broadcast only while using certain CDS approximation algorithms in order to help in the formation of the CDS set itself (p. 99, [21]). The MCDS nodes relay LDD packets, and non-MCDS nodes receive and process them, but do not relay them further.

Like OLSR, OSPF-MCDS uses jitter to avoid random collisions of data packets and broadcast packets. OSPF-MCDS also piggybacks on control messages to save on packet header bandwidth.

2.6.1 MCDS Selection Example

In the network topology shown in Figure 2.10, the set G = {1 – 8} and S is such that every vertex in G – S can connect to at least one vertex in S using an existing edge in G. If S is the set of gray nodes {2, 5, 8}, then G – S is the set {1, 3, 6, 4, 7}. As shown in Figure 2.11, every node in G – S can be connected to at least one vertex in S using an edge in G shown by the dark lines.

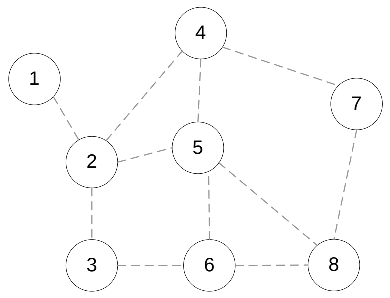


Figure 2.10: OSPF-MCDS sample network topology

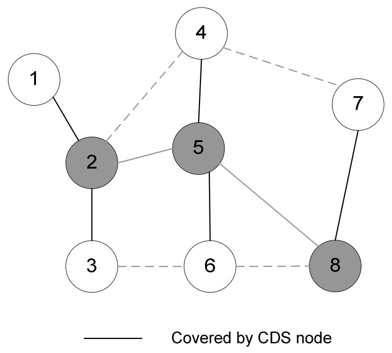


Figure 2.11: MCDS selection
G = { 1 – 8 }
dominating set, S = { 2 , 5, 8}

2.6.2 Advantages and Disadvantages

The advantage of OSPF-MCDS is that it forms shorter hop networks in most cases as the nodes receive a full topology, which is unlike other protocols which reduce the size of the topology broadcast creating sub-optimal routes. Furthermore, MCDS computation based on the approximation algorithm chosen could significantly add to computational complexity of the MANET routing process.

2.7 Related Work

Broch, Maltz, Johnson, Hu and Jetcheva have compared various early designs of MANET routing protocols like DSDV, TORA, DST and AODV [22]. A comparison of reactive protocols AODV and DSR is available in [23]. Clausen, Hausen, Christensen and Behrmann describe the performance of OLSR through emulation and simulation [17]. In [21], the author of OSPF-MCDS has conducted a similar study with a new two-state connectivity model using a smaller set of scenarios and an on-off traffic model. The work in this thesis is to use more widely used traditional mobility models and traffic sources to create observations based on more standardized methodology that can be used to compare OSPF-MCDS to other MANET protocols in similar simulations. The performance comparison portion of this study differs from the previous works by comparing three IETF ratified MANET routing standards and a promising new MANET routing protocol (OSPF-MCDS) by using large amounts of data sets comprising several replications, scenarios and radio-ranges using widely used mobility model and traffic sources. The comparison data consists of more than two-thousand simulation runs and almost 550 gigabytes of data. The simulation methodology is designed to easily replicate any future MANET routing protocol anyone wishes to include and compare against the current results. While porting OSPF-MCDS to ns-2, generic pseudo-code has been created to help MANET researchers create new ns-2 ad-hoc routing agents.

2.8 Summary

This chapter introduced the various classifications of MANET routing protocols and provided a discussion of the primary requirements of MANET routing protocols. It detailed the operation of the four MANET routing protocols, OLSR, AODV, TBRPF, and OSPF-MCDS, used in this investigation. It used examples to highlight how the protocols function and discussed the advantages and disadvantages of each of these protocols. It also introduced prior and related work in this area of study.

Chapter 3

Implementation Details and Simulation Methodology

This chapter discusses the implementation specifics related to the simulation model and the various components of the simulation environment. It also provides descriptions of the various simulation parameters and analysis used in this study.

3.1 Contributions to Implementation of OSPF-MCDS

The author of OSPF-MCDS created a preliminary version of OSPF-MCDS implementation to integrate in a MANET experimental test bed [24]. The contribution this study makes to the implementation is detailed below and is also acknowledged by the author of OSPF-MCDS in [21].

3.1.1 Subsystem Integration

Extensions were created to the implementation to communicate neighbor hop count table with other application layer software. This allowed integration with the policy-based quality of service (QoS) routing scheme [25], which required the MANET routing protocol to be able to communicate its neighbor hop count table. This requirement necessitated the creation of a client-server communication model to extract the needed information from every routing node. Any application layer software interested in the neighbor hop count table would be able to query the remote node's neighbor hop table by using client-server communication as follows.

```
Connect to remote <IP address of node> <port 17900>
Send command> getlinkstate
Receive> Remote Node's Neighbor Hop Table
```

3.1.2 Multiple Interface Routing

Most implementations of the MANET routing protocol are not designed to run on heterogeneous wireless environments on the same physical routing entity. For example, a heterogeneous wireless environment could consist of a custom radio and an IEEE 802.11 wireless local area network (WLAN) device operating on the same host. To overcome a similar situation, it was necessary to extend the routing protocol to accommodate multiple wireless segments and allow different physical layer devices to be connected to the same host.

The concept of local clustering is introduced at the multiple interface nodes. A cluster, which is defined as all interfaces in the same subnet and same media, is central to the implementation of this approach. If the node has directional antennas and if the line of sight is not overlapping, they are assumed to be different media. A multiple interface node is a cluster gateway acting as a relay agent between the nodes of the different clusters.

The network layout in Figure 3.1 illustrates this approach. Host X is running the multiple interface version of the OSPF-MCDS implementation. There is an instance (process) of OSPF-MCDS running on each interface of host X. The different processes are running a Transmission Control Protocol (TCP) communication channel on port 17900. Each process (per interface) queries all the peer interfaces for their topology table. On receiving this information each process adds every edge in the peer topology table to its local link state table along with the appropriate edge costs (hop count).

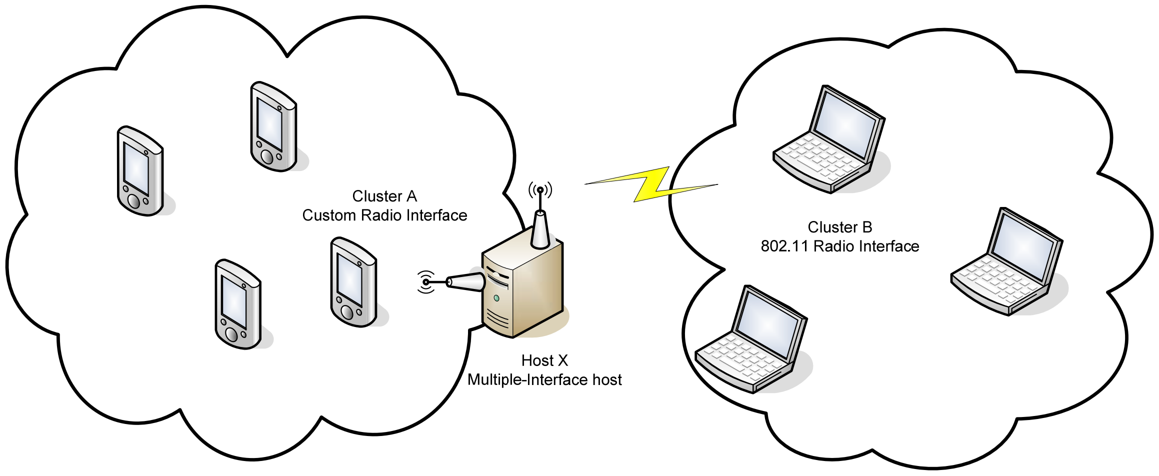


Figure 3.1: Multiple Interface Routing

The neighbors in each cluster receive topology information about nodes in other clusters through the multiple interface nodes. Therefore, the shortest path routes to nodes in the other cluster will have the multiple interface nodes as a hop. Since the multiple interface nodes do not relay LDD packets among clusters, the broadcast traffic size is reduced across different clusters. The multiple interface version of OSPF-MCDS was demonstrated in the experimental test bed as described in [24]. This version of OSPF-MCDS was run on a gateway node interconnecting wired and wireless networks with two physical interfaces. It should be noted that the wired nodes where also running a version of OSPF-MCDS and used a dynamic switch to emulate a MANET topology.

3.1.3 Modifications to OSPF-MCDS

The original implementation had stability issues due to the use of non-thread-safe functions and timers based on signals. Changing the timer implementation to one based on a C programming language and UNIX style “select” function with null file descriptors eliminated this problem. The timeout feature of the select function was used to create a thread-safe timer. However, using signal-based timers caused periodic “crashing” of the Linux protocol code. The use of the select function from the networking application program interface (API) solves this problem as select can be used in the re-entrant code. It is important to use re-entrant functions in multi-threaded programs. Experience with the protocol coding showed that the use of non-reentrant code, such as C style printf, inet_ntoa, inet_addr, gethostbyaddr and malloc functions, causes stability issues in multi-threaded applications.

Other modifications were also necessary to avoid use of non-thread-safe functions in the original code. A bug with the LDD duplicate messages had to be fixed. With the input of original authors of OSPFMCDS, several pieces of the protocol code were improved. However, all of these are not documented here, because they relate to more programming specific issues; the program source of OSPF-MCDS provides the details. During program development, it became necessary to inspect the protocol packets. Therefore, a packet logging feature to trace protocol messages while debugging was added.

3.2 Performance Metrics

Curson and Maclur define the performance metrics that can be used to quantitatively assess MANET routing protocols [26]. This comparative study uses the following performance metrics.

Packet Delivery Percentage: The ratio of total application data received at the destination and the application data sent from the source gives the packet delivery ratio. The percentage of this quantity is the packet delivery percentage. This metric defines the loss rate experienced by the application data and is related to the data throughput of the network.

End-to-End Delay: This is measured as the time delay between the application layer packet sent at the source node to the destination node. This metric describes the packet delivery time: the lower the end-to-end delay the better the application performance.

Hop Count: The number of hops a packet takes to reach from its source to a destination node is the hop count. This quantity helps to determine the path optimality of a given routing protocol over another.

Routing Overhead: The bandwidth consumed by all the control packets of the routing protocol is measured as control packet overhead. This quantity helps to determine the scalability of a given routing protocol. A lower control packet overhead with a higher throughput is a much desired optimization in MANETs.

These metrics are consistent with many similar studies found in the literature [22][26]. In the present simulation environment each of the above metrics is averaged over all of the nodes in the network.

3.3 Simulation Details

To obtain meaningful performance comparison results, a simulation study of a MANET routing protocol should be conducted using well tested mobility models, realistic radio ranges, traffic patterns, and physical layer interface effects such as collision, interface queues and parameters affecting radio propagation. This MANET routing study has three specially designed usage scenarios, as described below.

i) **Small Scenario:** A small node set emulating human movement speeds called the “Soldier Case.” It consists of a simulation area of 300×400 square units and 15 soldiers moving at seven miles per hour.
ii) **Medium Scenario:** A medium size set of nodes emulating small navy boats called the “Ship Case.” It consists of a simulation area of 1000×1200 square units, a grid ten times the previous scenario. There are 45 nodes representing ships traveling at an average of 20 knots (23 miles per hour).
iii) **Large Scenario:** A large node set emulating cars in a city block called the “Car Case.” It consists of a simulation area of 1400×1500 square units, with 80 nodes traveling at 45 miles per hour.

Each of these scenarios was simulated under varying radio ranges, and the performance metrics described in the previous section were measured. The scenarios are also subject to three different traffic loads to measure the effect of traffic load on these performance metrics. Table 1 provides a summary of these scenarios.

3.3.1 Mobility Model

Mobility models are sets of movement patterns generated to model the behavior of mobile nodes. The Random Waypoint mobility model use to generate the three scenarios is an entity mobility model [27], in which the node movements are independent of each other. In this model, a mobile node starts from its location in the simulation area and moves toward a random destination with a speed uniformly selected between Minspeed and Maxspeed. On reaching the destination, the mobile node pauses for a period of time and then moves again in a newly selected direction and speed. According to [27] and

[28], if the node movement speed is high and the pause time is large, then the scenario with the Random Waypoint model produces a more stable network than those with shorter pause times. Based on scenarios, it is estimated that a pause time of 20 seconds or higher will create a stable network. Camp, Boleng, and Davies [27] show that in the Random Waypoint mobility model the ratio of neighbor nodes to total number of mobile nodes in a network reaches a steady-state value after the initial 600 seconds of scenario generation.

3.3.2 Scenario Generation

The BonnMotion scenario generation and analysis tool [29] was used to create the mobile node usage scenarios discussed earlier. The implementation of Random Waypoint model in the BonnMotion scenario generator uses a random pause time up to the Maxpause time value defined in the scenario generation process. Eight independent node mobility scenarios were generated by using random seeds corresponding to different network topologies. Eliminating the first one-million seconds of the scenario generated ensures the mobile nodes have a steady-state ratio of neighbors. An analysis of the generated scenarios for the average node degree and number of partitions showed that as the radio range increases the node degree increases and the number of partitions decreases. In the soldier case, the average node degree increased progressively from 2 to 8 and the partitions decreased from 7 to 1.5. In the ship case, the average node degree increased progressively from 2 to 8 and the partitions decreased from 13 to 1.5. In the car case, the average node degree increased progressively from 3 to 10 and the partitions decreased from 15 to 1.5.

The scenario files generated from BonnMotion were converted to the ns-2 node movement format to be compatible with the ns-2 simulator requirements. To visualize the node movements and topology a tool[1] was created to extract node movements from any given node in the ns-2 scenario trace file. The initial random topology of nodes in each case is illustrated in Figure 3.2, Figure 3.3, and Figure 3.4.

[1] The listing for ns2 to plot tool is available in Appendix B.

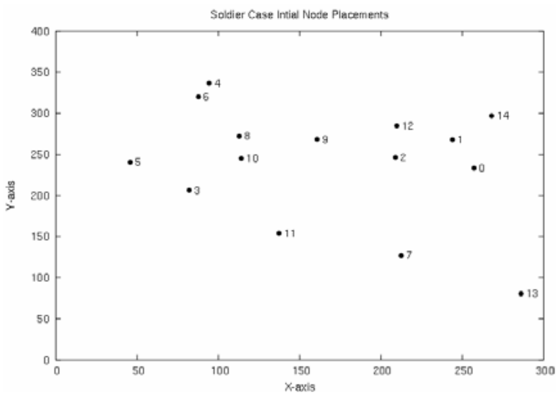


Figure 3.2: Soldier Case 15 nodes (seed index 10)

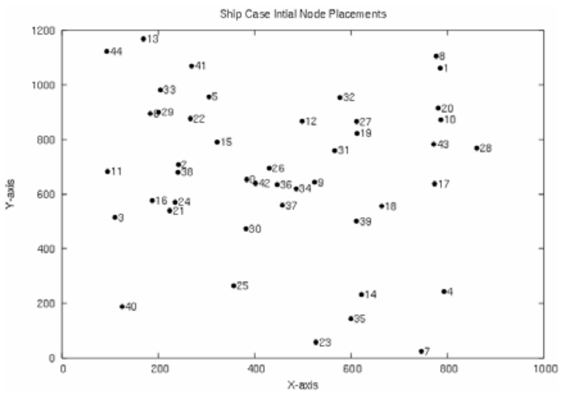


Figure 3.3: Ship Case 45 nodes (seed index 10)

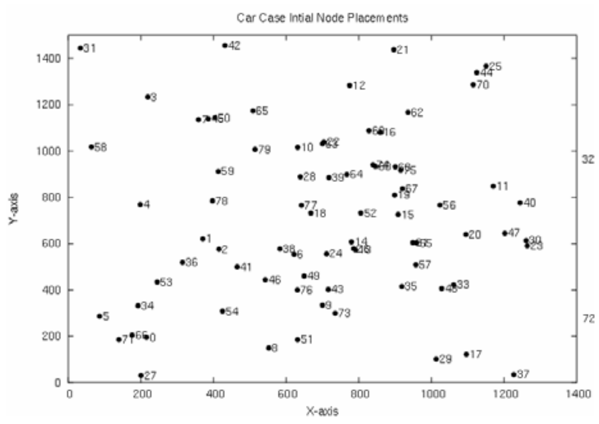


Figure 3.4: Car Case 80 nodes (seed index 10)

The graph in Figure 3.5 traces the path followed by node 0 in the Soldier Case (seed index 10) for a period of 5000 seconds.

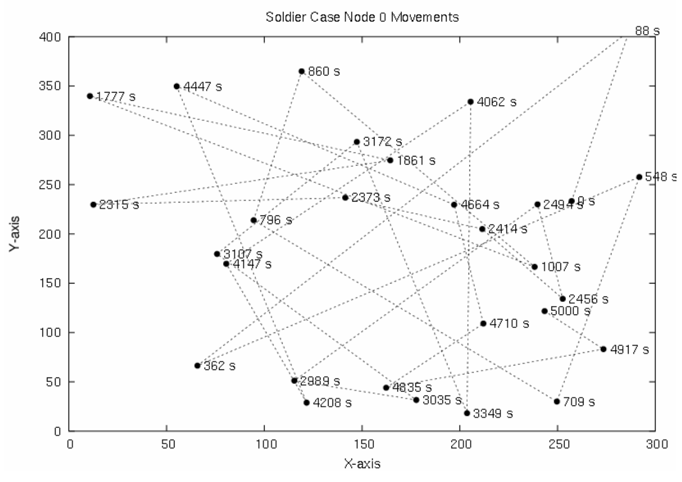


Figure 3.5*:* Soldier Case node 0 movements (seed index 10)

3.3.2.1 Node Movement Animation

Apart from the above static graph generation tools, a tool[2] was also created to generate movement animations of nodes. The tool, which takes an ns-2 scenario file and generates a network animation object capable of being viewed, using Network Animator (nam), has been submitted to BonnMotion for inclusion in its suite of scenario building tools. This animation tool will help other researchers study scenario files in greater detail by allowing them to visually inspect generated scenarios. Figure 3.6 shows an animation frame of the solider movements in this study. It is important that the reader remember this is not a post-simulation trace animation. The nam trace animation that ns-2 produces is voluminous and is disabled in large simulation runs. The tool creates a nam playable animation file directly from the ns-2 scenario file.

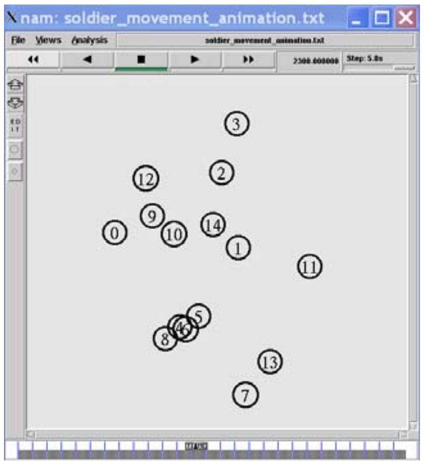

nam: soldier_movement_animation.txt
File Views Analysis
soldier_movement_animation.txt
Step: 5.0s
3
12
2
9
0
10
14
1
11
5
8
13
7

Figure 3.6: Soldier Case nodes movement animation (seed index 10)

3.3.2.2 ns-2 to nam Conversion Details

The details for converting from an ns-2 scenario file to a nam animation object are as follows. In the node movement shown in Figure 3.7, node 1 moves from position (X1, Y1) at time T_0 to (X2, Y2) in a two dimensional grid with a given speed S. The node's movement is represented in an ns-2 scenario file as

$ns_ at T_0 "$node_(1) setdest X2 Y2 S"

[2] The listing for ns2 trace to NAM tool is available in the appendix C.

The Network Animator accepts a different format as its input; it expects velocity in X and Y direction. Velocity is speed with direction, and speed is defined as distance divided by time. Let D be the distance traveled by node 1 from (X1, Y1) to (X2, Y2), then D is

$$D = \sqrt{(X2 - X1)^2 + (Y2 - Y1)^2} \quad (3.1)$$

Let T_t be the time taken to travel distance D, then

$$T_t = \frac{D}{S} \text{ (seconds)} \quad (3.2)$$

Let V_x and V_y be the velocity in X and Y direction. The sign of V_x and V_y indicates the direction of travel in the respective axis.

$$V_x = \frac{(X2 - X1)}{T_t} \text{ (units/second)}; \; V_y = \frac{(Y2 - Y1)}{T_t} \text{ (units/second)} \quad (3.3\text{-}3.4)$$

At the given velocity V_x and V_y the node will reach (X2, Y2) in time T_t from its current position. This information is the required format of nam as shown below:

n -t T_0 -s *node1* -x *X1* -y *Y1* -U V_x -V V_y -T T_t

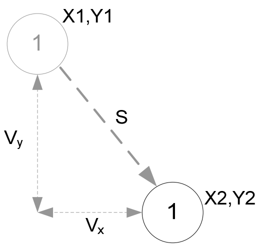


Figure 3.7: ns-2 node mobility

Use of the above technique makes it possible to convert an ns-2-compatible node movement scenario file to an animation object playable using nam. As a sophisticated animation player capable of producing frame animation and frame selection, nam will help researchers visualize the node positions at any given time. It is particularly useful for visualizing and finding neighbor nodes while debugging OSPF-MCDS simulation code.

Table 1 lists the complete scenario set, provides a summary of the scenario cases and the parameters discussed in this section.

Table 1: Summary of the Scenario Cases

Scenario	**Max. Speed** *(*Min. Speed of 0.1 m/s)	**Max. Pause Time** (seconds)	**Simulation Area** (Square Units)	**Scenario Replications**
Soldier Case	3m/s (7 mph)	15	300 * 400	Seed Index 10-17[3] (8 replications)
Ship Case	10 m/s (22.5 mph)	60	1000 * 1200	Seed Index 10-17 (8 replications)
Car Case	20 m/s (45 mph)	120	1400 * 1500	Seed Index 10-17 (8 replications)

3.3.3 Traffic Generation

One of the important uses of MANET is in deploying a dynamic rapid communication network setup in disaster relief and military operations. These operations usually involve video and voice communication applications, which are mainly constant data rate datagram applications. To emulate similar loads, constant bit rate (CBR) traffic was used as the application traffic model running over a User Datagram Protocol (UDP) transport connection. The CBR traffic generation scripts available in ns-2 were modified to generate random pairs of traffic. Although there is only a single traffic flow between any pair of nodes, a few nodes were incorporated to handle many traffic flows concurrently as either the source or the sink. In ns-2, the traffic source is the CBR traffic generating agent at the source node and the sink is a null agent at the destination node. The CBR sources generate one kilobyte of traffic every one second, and each CBR source is considered as one traffic flow.

To study of effects of several traffic flows a single flow was used in the whole network, and then the flows were increased based on each scenario. In the Soldier Case, the performance characteristics were studied with 1, 5, and 10 CBR flows. At 10 CBR flows with 15 nodes, every node is handling at least one source and one sink. For the Ship Case, the protocols under 1, 10, and 30 CBR flows were studied. Similarly for the Car

[3] Random seed index explained in the section on statistical considerations.

Case, the protocols under 1, 30, and 70 CBR flows were studied. It is important to note here that at 70 traffic flows the network is heavily loaded.

3.3.4 Statistical Considerations

Simulation engines rely on pseudorandom generators to create random node movements, traffic flows and so forth. A simulation is statistically sound only when a good pseudorandom generator is used. Pseudorandom generators rely on seed values, a quantity that helps setup the initial state of the pseudorandom generator. Jain describes an algorithm to generate independent random seeds for use in simulation studies [30].

Eight independent node mobility trace files were generated using eight independent seed values. The pseudorandom seeds used in the generation were calculated using the seedtool available in OmNet [31]. When a seed index *n* is specified the seedtool retrieves the n^{th} independent pseudorandom number using the algorithm described by Jain [30]. Seed indices 10 to 17 were used to generate the node movement scenario files.

As described in Section 3.3.2, to reach a steady state ratio of neighbors the node movements up to one million seconds were eliminated. Studies done in [21] indicate that warm-up times have to be considered to let routing protocols reach steady-state values. Eliminating the first 1000 seconds in the Soldier Case and Ship Case to account for the warm-up time resulted in 4000 seconds of simulation time for data analysis. Similarly, eliminating the first 500 seconds in the car case resulted in 1000 seconds of simulation time for data analysis.

While generating traffic flow patterns for the various replications and scenarios, random send index 10 was used, and the performance metrics were calculated using a 95% confidence interval.

3.3.5 Summary of Simulation Cases

To summarize these simulation experiments, three different usage cases and three different traffic flows for each case were used. These cases were studied under various radio ranges and the performance metrics described in Section 3.2 were measured as a function of the radio ranges. To obtain a higher confidence interval, these simulations were replicated eight times using independent and randomly generated scenario mobility trace files. The full simulation set is summarized in Table 2.

Table 2: Summary of Simulation Cases

Usage Scenario	**Topology - Node Mobility**	**Radio Ranges** (Meters)	**Traffic Flows** (Seed Index 10)	**Simulation Runs**
Soldier Case	8 replications *(Duration 5000 s)*	60,70,80,90,100,110,120 (7 ranges)	1, 5, 10 CBR Flows	8 replications *7 radio ranges *3 traffic flows = 168
Ship Case	8 replications *(Duration 5000 s)*	125,150,175,200, 225,250,275	1, 10, 30 CBR Flows	168
Car Case	8 replications *(Duration 1500 s)*	150,175,200,225, 250,275,300	1, 30, 70 CBR Flows	168
Number of simulations per protocol				504
Total number of simulation runs (4 protocols)				2016

3.3.6 Other Simulation Parameters

The default wireless channel model and IEEE 802.11 physical layer (PHY) and MAC were used to supply other simulation parameters. The interface queue is a droptail priority based queue, i.e. one that drops the last packet when the queue is full. The antenna model uses an omni-directional unity gain antenna. These parameters are summarized in Table 3.

Table 3: Other Simulation Parameters

Parameter	**Value**
Channel type	Wireless Channel
Radio-propagation model	Two Ray Ground
Network interface type	Wireless Physical Layer
MAC type	802.11
Interface queue type	DropTail/Priority Queue
Interface queue length	50 packets
Link layer type	Traditional Link Layer (LL)
Antenna model	Omni-directional (unity gain)
Receiving threshold	Two-Ray Rx Power(Radio Range)
Sensing threshold	Two-Ray Rx Power(Radio Range + 5 m)

3.3.7 Two-Ray Ground Reflection Radio Propagation Model

Radio propagation in direct line-of-sight (LOS) communication can be modeled using the Friis free space model [32]. The free space model computes received power at a distance *d*, when *d* is small using

$$P_r(d) = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2 d^2 L} \tag{3.5}$$

Here P_t is the power transmitted, G_t and G_r are the gain of transmitter and receiving antennas which is set to 1 in ns-2. L is the system loss, set to 1 in ns-2 [15]. λ is the radio signal wavelength.

When *d* is large, the propagation loss is more accurately modeled using the Two-Ray Ground Reflection model, which considers both the direct ray and the reflected ground ray. The Two-Ray Ground Reflection model computes received power at distance *d* by

$$P_r(d) = \frac{P_t G_t G_r h_t^2 h_r^2}{d^4 L} \tag{3.6}$$

Here G_t, G_r, L are same as for the free space model and h_t, h_r are the heights of the transmit and receive antenna, set to 1.5 in ns-2[4]. In the Two-Ray Ground Reflection model received power deteriorates faster with an increase in distance, but does not produce accurate results at a shorter distance.

The distance at which Two-Ray Ground Reflection model is accurate over the free space model is called cross-over distance d_c. The cross-over distance d_c is computed using

$$d_c = \frac{(4\pi h_t h_r)}{\lambda} \tag{3.7}$$

When the distance between nodes is less than d_c, the free space model is used and when distance between transmitting and receiving nodes is larger than d_c then the Two-Ray

[4] Available in ~ns2source/tcl/lib/ns-default.tcl

Ground Reflection model is used. This selection is done automatically by the simulator ns-2[5].

In the simulation for this study, the default wireless physical device available in ns-2, a Lucent WaveLAN direct-sequence spread-spectrum (DSSS) radio interface was used. The default for λ is 0.32822757, which corresponds to a frequency of 914 MHz and P_t is 0.28183815. The power received P_r computed from P_t and the propagation model described above should be greater than the receiving threshold (RXThreshold) for the packet to be received at another node. The value of RXThreshold for a given radio range can be calculated using the threshold utility[6].

3.4 Simulation Environment

Network Simulator version 2 (ns-2) is an object-oriented discrete event-driven network simulator. Computation delays do not affect the simulation parameters and metrics, which is a major benefit of discrete event simulators. The internal workings of ns-2 are documented in [15]. There are several other tutorials available for learning ns-2 basics at [33] and [34]. Before the discussion of the porting of OSPF-MCDS, a brief discussion of the simulator and its advantages follows to facilitate the understanding of the porting process.

3.4.1 Network Simulator 2 (ns-2) Overview

The simulator is written using a dual object-oriented design in C++ and OTcl. The C++ compiled components run the core simulation engine, event schedulers and agents. The OTcl based interpreter is used to setup the simulation configuration and controls of the C++ data path. The dual design benefits from the execution speed of the C++ compiled network objects and rapid reconfigure-ability of interpreted OTcl configuration objects. Most often in simulation studies the parameters change with every new simulation, but

[5] Available in ~ns2source/mobile/tworayground.cc
[6] Available in ~ns2source/indep-utils/propagation/threshold

the underlying protocols and data agents remain the same. Therefore, it is useful to have a rapidly reconfigurable simulator as the basis for using the dual interpreter/compiled class hierarchy. Since OTcl are interpreted changes in simulation parameters do not have to be recompiled, a researcher can run large sets of simulation with a one-time compilation of the C++ network objects. The control parameters and functions of the C++ compiled objects are exposed to the OTcl interpreter via OTcl linkage. For every OTcl object invoked in the interpreter hierarchy there is a mirrored object created in the C++ hierarchy.

In ns-2 the various network components are designed as class objects called agents. These include the routing agent, application agent, channel agents and so forth. Based on the simulation setup ns-2 links the various agents (called plumbing in ns-2) to create a complete network. Figure 3.8 shows a simple network setup of two nodes transferring data. The grey arrows are the normal links of the agents and the black arrows show the active linkage between the different agents. The source agent, in this case a CBR traffic source, sends a data packet through each network object down the network stack. At the routing object the data packet is tagged for its next hop. The Interface Queue (IFQ), link layer (LL) and MAC layers add delays to the packet based on queue lengths, collision and radio propagation time. The channel layer ns-2 is like a connecting media fabric; there is an channel classifier object at this layer that copies the packet from one node's MAC to the destination or to a multiple nodes if it is a broadcast.

The packet is sent to the respective designation node's MAC object, which once again passes it up the next linked object, the LL. It is important to observe here that in ns-2 a data packet on reaching the designation node is delivered directly to the sink agent (a null object in this case). Bypassing the routing layer this presents some interesting problems with Internet Protocol (IP) headers, as described in Section 3.4.3.

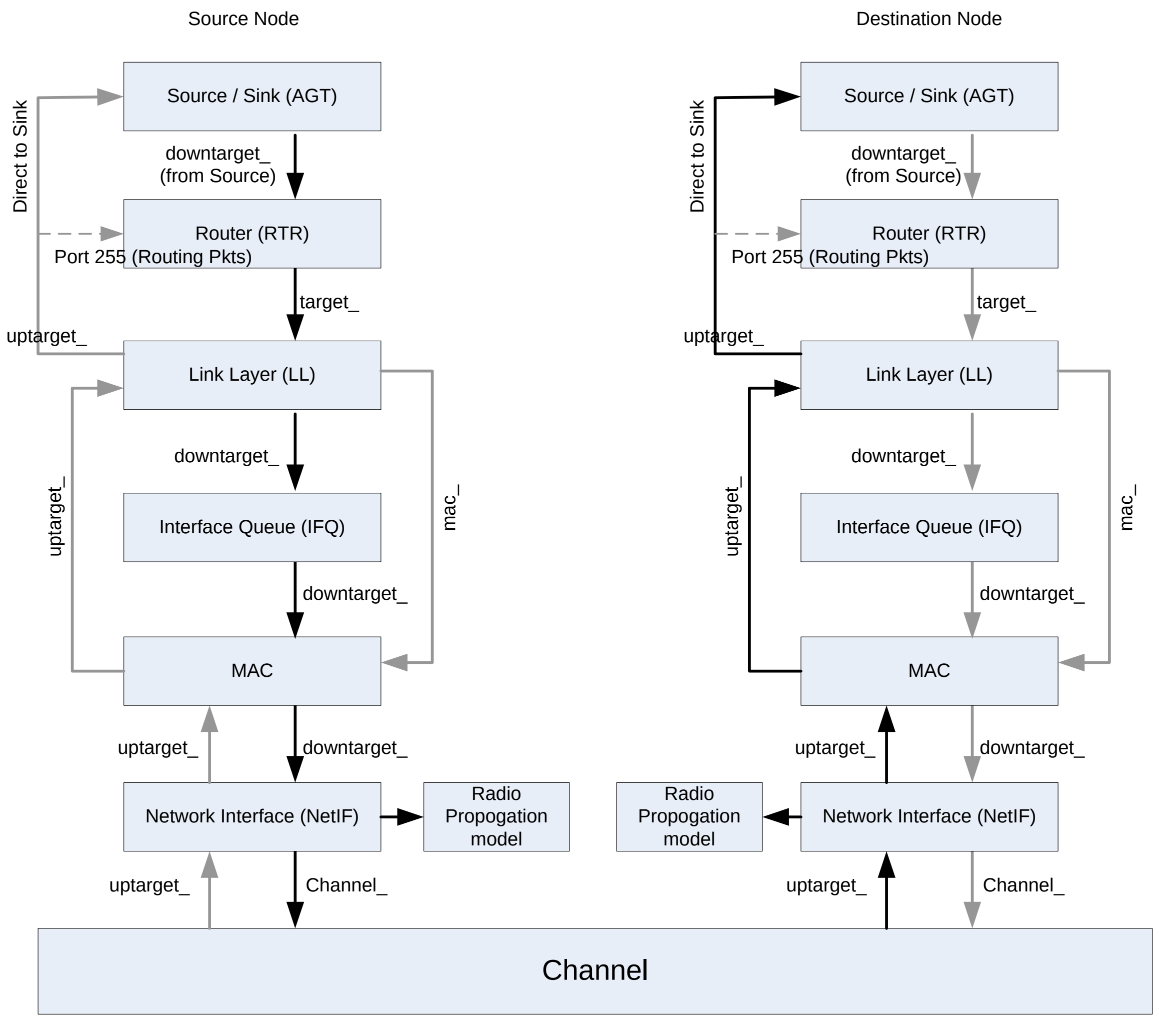


Figure 3.8: Internal schematic of an ns-2 mobile node [15]

The various agents have internal parameters that they expose to the OTcl layer to be used by the user to configure the agents. For example, to set the present CBR source to a packet interval of 1, 2, 3, and 4 seconds in each simulation run the user just needs to change the value of Y in the last line shown below.

```
set cbr_ [new Application/Traffic/CBR]
$cbr_ set packetSize_ 1000
$cbr_ set interval_ Y
```

3.4.2 Adapting OSPF-MCDS to ns-2

Since most of the simulations in this study were done in December 2003 and the version of ns-2 available at that time was version 2.26, the program code references presented here are relative to ns-2 version 2.26. However, these should be applicable to future versions of ns-2.

The labels on each link provided in Figure 3.8, show the appropriate object that has to be used in the scheduler to send the packet to the next network agent. For example, the routing layer (RTR) would send the packet to the link layer (LL) using the following function where *target_* is the next object reference:

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

One of the challenges facing simulation implementers in ns-2 is the lack of a generic framework for porting existing and new routing protocols to simulation. Therefore, providing such a generic template for implementing new routing protocols in ns-2 has been attempted by creating a dummy routing protocol called MyRouter, showing the essential glue framework required to port an existing operating system (OS) implementation. Since the discussion runs into greater depth about programming, Appendix A provides a tutorial explaining the implementation of MyRouter agent.

Since the available Linux implementation of OSPF-MCDS was written using C++ classes, it was relatively easy to port the routines to work under ns-2. A brief description of the interfaces created for ns-2 follows. Since ns-2 refers to network objects as agents the OSPF-MCDS routing protocol agent class is called *NS_OSPFMCDSAgent*.

The necessary OTcl linkage was created using *TclClass*, *TclObject* and *PacketHeaderClass* derived classes in the ns-2 OSPF-MCDS Agent. All agents in ns-2 should be derived from the base class “Agent.” The actual Linux routing protocol, referred to as *ospfmcds_agent* was instantiated in the ns-2 class to allow for the use of the existing implementation code inside ns-2.

```
class NS_OSPFMCDSAgent : public Agent {
 ….
          OSPFMCDS_Agent ospfmcds_agent;
…..
};
```

When the mobile node setup in ns-2 issues a *start* command, the parameters for the Linux implementation were set up and, in turn, a start command was issued to it. It is important to point out that the Linux agent was being masked to function inside the ns-2 simulation environment by proxying its *send*, *receive*, *start* and other functions from within ns-2.

```
void NS_OSPFMCDSAgent::Start() {
….
          ospfmcds_agent.command("id", ns-2_myaddr_); //set internal fields
          ospfmcds_agent.add_nic(ns2_myaddr_); // htonl is not required in ns-2
          ospfmcds_agent.command("start", 0);
….
}
```

When the NS_OSPFMCDS agent receives a packet from ns2 that is destined to the OSPFMCDS routing agent, it delivers this packet to the Linux implementation to process it. When the agent receives an OSPFMCDS routing protocol packet (*PT_OSPFMCDS*), it copies the packet's data to the *recv*() of *ospfmcds_agent*. The following code snippet illustrates the process:

```
void NS_OSPFMCDSAgent::recv(Packet *p, Handler *) {
{….
if (ch->ptype() == PT_OSPFMCDS) {
      ih->ttl_ -= 1; //Passed through router so ttl less 1
      //Access Packet Data
      u_int8_t *data = (u_int8_t*)p->accessdata();
      int packet_size = p->datalen();
      //Call the ospfmcds_agent (actual protocol implementation object) to process the
data
                ospfmcds_agent.recv(data,packet_size,ih->saddr());
….}
```

Just as the agent uses the Linux implementation within the ns-2 agent, it stores a reference to the ns-2 agent object in the actual *ospfmcds_agent* class.

```
class OSPFMCDS_Agent {
…
protected:
    NS_OSPFMCDSAgent *ns_ospfmcds_; //Holds the ns-2 object for the OSPFMCDS
Agent
};
```

Using the stored reference, the Linux implementation class can use *NS_OSPFMCDSAgent* packet handling functions such as *send* and *forward*.

```
void OSPFMCDS_Agent::send_packet(unsigned char * buf, int len){
  ns_ospfmcds_->send(buf,len); //Call the ns-2 object to send the packet
}

void OSPFMCDS_Agent::forward(Packet *p, IPAddr nexthop, double delay){
          ns_ospfmcds_->forward(p, nexthop, delay);
}
```

In the *NS_OSPFMCDSAgent* object, it takes the Linux *ospfmcds_agent* routing protocol packet and copies it as a payload of the ns-2 version packet *PT_OSPFMCDS*. The following code snippet shows the packet payload (*accessdata*) being filled with the protocol data received from *ospfcmds_agent*.

```
void NS_OSPFMCDSAgent::send(unsigned char *pkt_buf, int pkt_len) {
…..
memcpy(p->accessdata(), pkt_buf, pkt_len);
….
     Scheduler::instance().schedule(target_, p, OSPFMCDS_cfg.Jitter *
Random::uniform());
}
```

The ns_agent uses random jitters to avoid packet collisions. Observation shows reduced packet drops due to collisions when employing small uniform random packet jitters while transmitting packets.

The Linux implementation timers are inherited from the *TimerHandler* class in ns-2. The *TimerHandler* class allows for the creation of an expiring timer, reschedule the timer and check the timer pending status. The *expire* function gets called every time a schedule timer event expires. Based on the event type the event handler (*expire* function) sends out HELLO, Link State Database (LSD), Link Database Description (LLD) and Interface

Prefix Description (IPD) packets. The following snippet shows the class inheritance in *ospmcds_agent*.

```
class Generic_timer : public TimerHandler {
{ …
void expire(Event *e);
}
```

The x86 computer architecture uses the “little endian” byte ordering. When running on an x86 architecture processor, the ns-2 simulator also uses host byte ordering (little endian), making it unnecessary to use network byte order (big endian) conversion functions in ns-2.

3.4.3 ns-2 Porting Issues

In ns-2, the routing agent of the destination node does not receive the application packet. The simulator plumbing directly delivers the packet to the destination application agent, which poses a problem with the IP header. There is no entity stripping the additional IP header in the simulator, a bug is documented in [21]. To create a fair comparison, IP headers were not added to application packet as there are no entities to decrement the IP header overhead on the receiving end. Similarly ns-2’s Address Resolution Protocol (ARP) packet generation suffers from a timer expiry issue, as documented in [21].

3.5 Protocol Simulation Code

Of the four routing protocols compared in this study, only OSPF-MCDS has been ported to ns-2 as described above. The sources for the simulation code of OLSR, AODV and TBRP are documented here.

OLSR has a few implementations for ns-2 [35]. For this study the first OLSR patch for ns-2 version 2.1b7a was used. Of the many implementations of AODV available, this study used the one from Uppsala University, AODV-UU [36], which was found to be stable on ns-2 version 2.26. A TBRPF implementation is also available for ns-2 version 2.26 [37].

Some protocols including AODV try to use link layer neighbor detection mechanisms which give lower overhead. To ensure a fair comparison the link layer neighbor detection schemes in all the protocols were disabled. The IP header and ARP bug described above were also fixed in all of the routing protocols studied.

3.5.1 Protocol Specific Parameters

The HELLO interval for all the protocols was set at one second. The neighbor expiry was set at three seconds for OSPF-MCDS, OLSR and TBRPF. In AODV-UU after three HELLO messages were lost, the link was considered broken.

3.6 Simulation Trace File Size Limitation Bug

Some of the simulation, especially the 80-node dense Car Case scenario, created simulation trace files over two giga bytes (GB) in size. It became evident that the current versions of TCL [less than version 8.4.11] used by ns-2 contain a serious limitation in that they do not allow trace files to be larger than 2 GB. Linux kernels 2.4 and higher support the Large File System (LFS)[7], a scheme designed to enable up to two terabytes of storage per file on ext3/ext2 file systems. A relatively old but lesser known flag called O_LARGEFILE was introduced to facilitate the creation of large files using the "open" function available in C/C++. The sources of TCL were modified to support files larger than a 2-GB file. This bug has been reported to TCL[8] and ns-2 and the developer communities of these projects are working on ways to fix it.

Programmers using C/C++ "open" functions to do file handling can use the following flag additions to support files larger than 2 GB:

```
#define O_LARGEFILE     0100000
int fd = open("filename", O_LARGEFILE|O_WRONLY|O_CREAT, 0644);
```

[7] http://www.suse.de/~aj/linux_lfs.html
[8] http://sourceforge.net/tracker/index.php?func=detail&aid=1287638&group_id=10894&atid=110894

3.7 Simulation Data Analysis

Performance metrics are derived from simulation data analysis. ns-2 offers several ways to measure traffic metrics. One of them is to use special sink agents called LossMonitor, but because they have limited data reporting capability, they were not used in this study. Instead, the simulation trace file was post processed to obtain the metrics defined in Section 3.2.

Although several tutorials document the ns-2 trace file format [38] (p.112 in [34]), a brief introduction to the mobile trace file format of ns-2 is necessary to explain the data analysis methodology used in this study. The following lines are sample trace lines captured at different times during the simulation.

AODV-UU protocol HELLO packet sent by Routing agent

```
s 1.000284093 _2_ RTR  --- 2 AODVUU 40 [0 0 0 0] ------- [2:255 -1:255
1 0] [0x2 0 [2 0] 3000.000000] (HELLO)
```

AODV-UU protocol HELLO packet at sending MAC layer

```
s 1.000639093 _2_ MAC  --- 2 AODVUU 92 [0 ffffffff 2 800] -------
[2:255 -1:255 1 0] [0x2 0 [2 0] 3000.000000] (HELLO)
```

CBR traffic packet at receiving node

```
r 108.038780688 _29_ MAC  --- 4993 cbr 1000 [13a 1d 16 0] ------- [33:0
29:0 32 29] [58] 4 0
```

The complete trace format structure for various packets can be obtained from the ns-2 source. To help the reader understand the various headers recorded in the trace, Table 4 shows the tabulation of the above sample trace lines.

Table 4: Trace Format Explanation

Event Type	Time stamp	Src ID	Agent Name	Reason	Pkt ID	Pkt Type	Pkt Size	**[**Tx Duration	Dst MAC	Src MAC	MAC**]** Pkt Type
s(Tx)	1.x	_2_	RTR	---	2	AODVUU	40	0	0	0	0
s(Tx)	1.x	_2_	MAC	---	2	AODVUU	92	0	ffffffff	2	800
r(Rx)	108.x	29	MAC	---	4993	cbr	1000	13a	1d	16	0

------ Contd.	**[**src IP:port	Dst IP:port	TTL	Next Hop**]**	*CBR only* Seq No	*CBR only* Num Fwds	*CBR only* Optimal Fwds
	2:255	Bdcast:255	1	0	N/A	N/A	N/A
	2:255	Bdcast:255	1	0	N/A	N/A	N/A
	33:0	29:0	32	29	58	4	0

As discussed in Section 3.4, ns-2 creates agents for the various network objects, including the router, CBR source, physical interface and so forth. Each of these agents log data which contains at least the minimal information shown in Table 4 along with any agent specific information added to the trace. The e*vent id* shows the r-receiving, s-send, D-drop and f-forward status of the packets. The a*gent name* specifies which agent logs the packet and the *Tx Duration* is the estimated time for the actual transmission. The other headers are self explanatory. The important headers in this study are *Event id, Agent Name, Pkt Size, Time Stamp, CBR num fwds.*

The calculation of the packet delivery ratio uses the ratio of the total number of CBR packets received in the network to the total number of CBR packets sent during the simulation.

$$Pkt\_Delivery\% = \frac{\sum_{1}^{n} CBR_{recv}}{\sum_{1}^{n} CBR_{sent}} \times 100 \tag{3.8}$$

Once the time difference between every CBR packet sent and received was recorded, dividing the total time difference over the total number of CBR packets received gave the average end-to-end delay for the received packets.

$$Avg\_End-to-End\_Delay = \frac{\sum_{1}^{n} \left(CBR_{sentTime} - CBR_{recvTime} \right)}{\sum_{1}^{n} CBR_{recv}} \tag{3.9}$$

The path hop count taken by each CBR packet over the total number of received CBR packets gave the average hop count per application packet.

$$Avg\_Hop\_Count = \frac{\sum_{1}^{n} CBR_{numFwds}}{\sum_{1}^{n} CBR_{recv}} \tag{3.10}$$

The total capacity consumed by the routing protocol is the ratio of sum of the MAC layer packet size when the control packets are sent or forwarded at each node in the network to the duration of the simulation.

$$Routing\_Overhead = \frac{\sum_{1}^{n}\left[MAC(Control_{pkt})_{sentSize} + MAC(Control_{pkt})_{fwdSize}\right]}{\Delta T_{sim}} \qquad (3.11)$$

A custom program was used to analyze the trace files for the above metrics. This tool, which iterates through each line of the trace file, determines the appropriate agent and registers the metrics for each line. After analyzing all the trace lines it creates a summary report that was used to further automate and produce the performance graphs. The program listing is available in Appendix D.

3.8 Summary

Chapter 3 has outlined the contributions of this study to the implementation of OSPF-MCDS. The most important of these is the achievement of multiple interface routing and sub-system integration. It includes a discussion of the various performance metrics of interest for evaluation and an explanation of the Random Waypoint mobility model used in the simulation study. It has introduced new tools to visualize the node mobility patterns and positions. It has provided a discussion of the various simulation considerations, such as traffic generators, simulation parameters and so forth, as well as an explanation of the radio propagation model used. It has given a brief overview of the network simulator, ns-2, and identified a severe limitation in the ns-2 program as well as fixes for it. It has provided an analysis of the ns-2 trace formats and presented equations to obtain performance metrics. This chapter has also introduced a generic routing template and a trace analyzer for ns-2.

Chapter 4
Simulation Results

4.1 Comparison Study

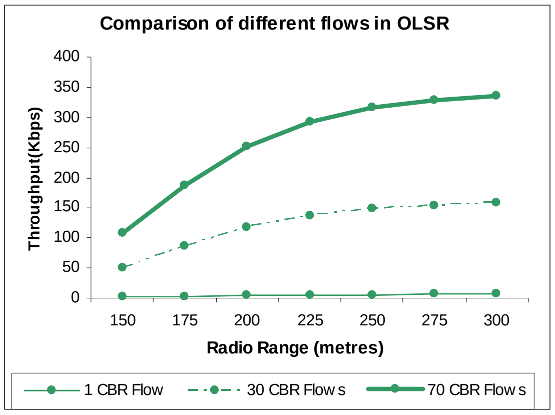
The goal of this study is to show the relative performance of each selected routing protocol with respect to varying scenarios and traffic loads. Pre-generated scenario files were used to subject each protocol to the same set of scenarios and traffic loads in an identical fashion to perform a fair comparison. The performance metrics considered were packet delivery percentage (throughput), average end-to-end delay, average hop count, and routing control overhead.

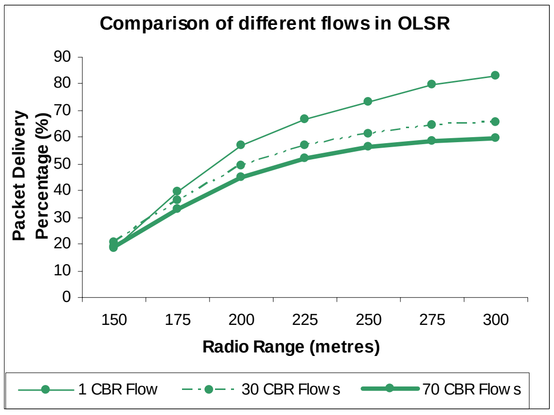
4.1.1 Throughput versus Packet Delivery Percentage

There are two representations of throughput; one is the amount of data transferred over the period of time expressed in kilobits per second (Kbps). The other is the packet delivery percentage obtained from a ratio of the number of data packets sent and the number of data packets received. Sometimes this is also referred to as the "goodput" of the system. Since different numbers of traffic flows were considered and compared to the different flows, the packet delivery percentage was used as a measure of throughput. If the Kbps measure was to be used, then when the traffic flows are increased, there will be an obvious increase in the throughput, but that does not necessarily indicate that the overall delivery percentage has improved.

Figure 4.1 shows the throughput and packet delivery percentage of OLSR for the large scenario case (car case) under different traffic flows. The throughput expressed in Kbps increases drastically as the number of traffic flows increases. While this is relative to the number of flows, it does not necessarily show an increase in performance. The packet delivery percentage actually drops as the number of flows increases, as is evident from

Figure 4.1(a). The packet delivery percentage is independent of the number of flows and, therefore, it is a good measure to show relative performance.





(a) Overall Throughput *(b) Packet Delivery Percentage*
Figure 4.1: Large Scenario – Car Case – OLSR Comparison with different traffic loads

4.1.2 Performance Constraints

Throughput alone does not indicate that a protocol A is better than a protocol B. How it achieves higher throughput when combined with scalability is a good measure of a better performance. Good performance indicators are:

- high throughput (packet delivery percentage);
- low average latency of delivered packets; and
- low control traffic overhead.

The performance reflected in the graphs is not representative of the computational complexity (order of computation) of the particular protocol. For example, OSPF-MCDS achieves a smaller hop count and a reduced broadcast domain at the cost of increased computation in selecting the MCDS set by using various approximation algorithms. The fact that OSPF-MCDS involves a slightly higher computation overhead is not accounted for in these graphs. From a practical stand point, MANET routing protocols are usually run on devices (viz. laptop computers and personal digital assistants), which force the network layer to provide reliable connectivity even at the cost of increased computational overhead.

The observations made in this study showed that OLSR and OSPF-MCDS almost take the same simulation time for a given scenario. In the simulation runs, OLSR and OSPF-MCDS took about seven times longer to simulate than TBRPF and AODV-UU. This does not necessarily mean they have a higher computation overhead than TBRPF or AODV-UU, because the longer time could also be due to un-optimized implementations, which were not analyzed in this study. However it is known that efficient flooding protocols such as OSPF-MCDS and OLSR, which use approximation algorithms to find the relay nodes, will have a slightly higher computing overhead than protocols which only use shortest-path algorithms.

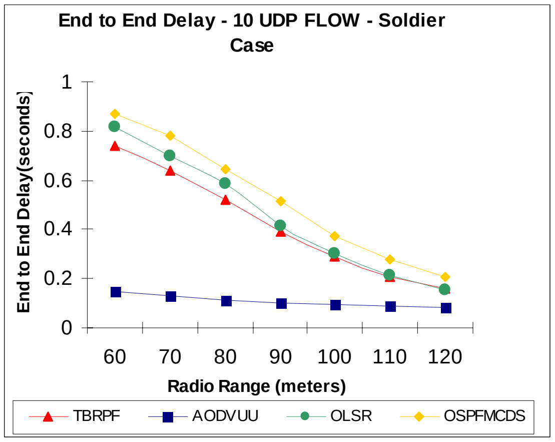
Each replication in this study's simulation experiment uses a different scenario, which was created using independent random seeds. Each of the eight independent node mobility scenarios corresponds to a different network topology. Since the performance of the routing protocols is sensitive to changes in network topology, consistent results are expected with more replications and larger traffic flows, as is evident with the large scenario cases discussed in Section 4.4. For the sake of clarity, only the average values in the graphs for the 95% confidence interval data are presented in Appendix F.

4.2 Small Scenario - Low Speed - Soldier Case

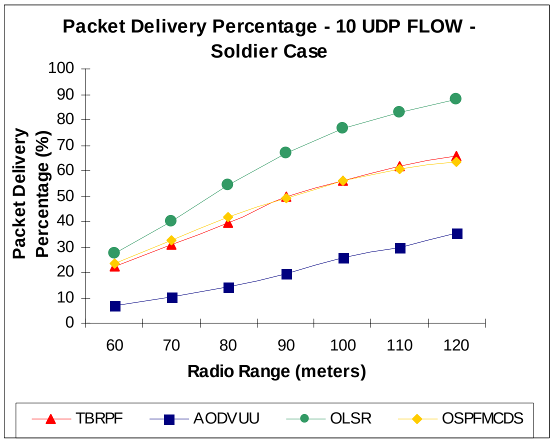
In the soldier movement scenario, which consists of a group of 15 soldiers (nodes) moving at a maximum speed of three m/s in a grid of 300×400 square units, the performance metrics were measured as a function of varying radio ranges. As the radio range increases, the node degree of the nodes increases and the number of partitions in the network decrease. At 120 meters radio range, the average number of network partitions decreases to less than two and achieves an almost fully connected network. The following analysis of the small scenario first focuses on the performance of each protocol with a fixed number of traffic flows and then compares the performance under different traffic flows.

4.2.1 Relative Performance

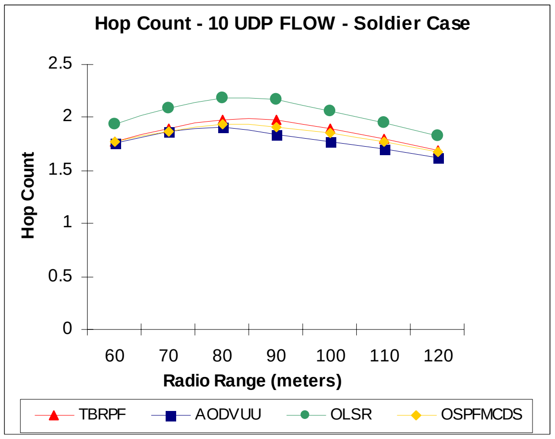
Figure 4.2 shows the relative protocol performance of the small scenario for TBRPF, AODV, OLSR and OSPF-MCDS as a function of radio ranges with 10 CBR flows in the network. The packet delivery percentage increases with increasing radio range as the network density increases and more routes become available to deliver the data packets.



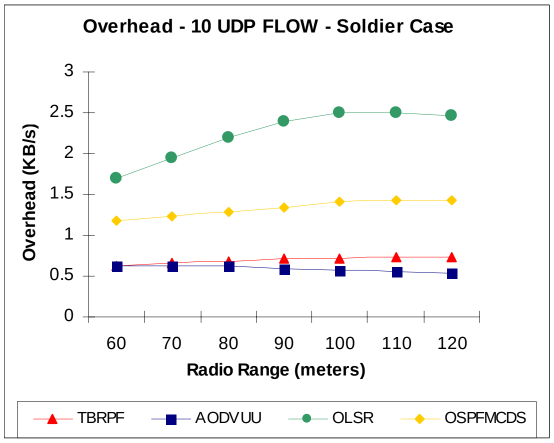
(a) *Average End-to-End Delay*



(b) *Packet Delivery Percentage*



(c) *Average Hop Count*



(d) *Overall Overhead*

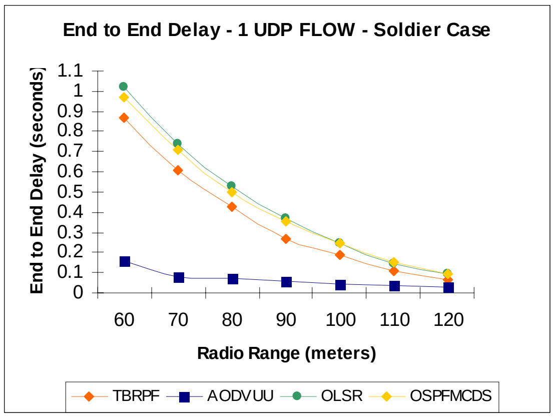
Figure 4.2: Relative Performance Comparison - Small Scenario – Soldier Case - 10 CBR Flows

Likewise, when the radio range increases, the network link density increases and more routes are available, thus reducing the end-to-end delay for TBRPF, OLSR and OSPF-MCDS, as shown in Figure 4.2(a). As the graph shows, the exception is AODV, which contradicts the theoretical observation that reactive protocols tend to have high setup latency. The simulation results reveal that AODV has the lowest end-to-end delay in most cases, but at the cost of lower throughput as shown in Figure 4.2(a) and 4.2(b). The hypothesis is AODV queues packets and sends them out as soon as a route is found. In a mobile scenario, if the path breaks before the packet reaches its destination, then the packet delivery ratio suffers, as is evident in Figure 4.2(b). However for those packets that do reach the destination, the end-to-end delay is minimal. AODV achieves low end-to-end delay at the cost of reduced packet delivery percentage.

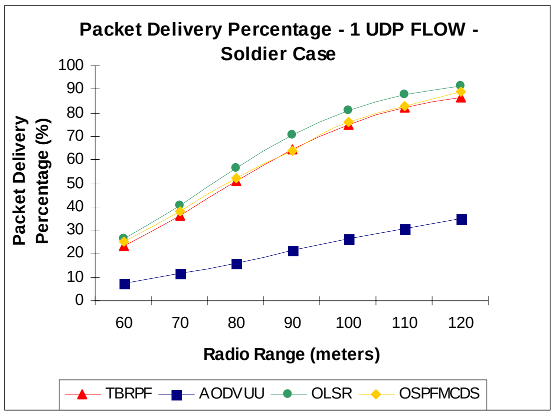
For a given scenario all protocols have an almost equal average hop count, as shown in Figure 4.2(c). OLSR has a slightly higher average hop count due to the use of MPR nodes. In OLSR MPR nodes are part of the route in OLSR, the routes might not always be the optimal shortest path. OSPF-MCDS forms shorter hop paths in most cases, as the nodes receive a full topology, unlike other protocols which reduce the size of the topology broadcast creating sub-optimal routes. OSPF-MCDS is an ideal candidate when the shortest path route is desired in a MANET network.

AODV has a low overhead proportional to the number of flows, because it is a reactive protocol. (A detailed comparison of overhead under different traffic flows follows in Section 4.4.2.1.) Among the proactive routing protocols, TBRPF has the lowest overhead, because it does differential updates and only includes partial source trees in its topology updates. OSPF-MCDS has an overhead twice that of TBRPF as it broadcasts the complete link state table and not reduced partial source trees as TBRPF does. However, OSPF-MCDS has half the overhead of OLSR, because, like TBRPF, OSPF-MCDS uses differential HELLO messages.

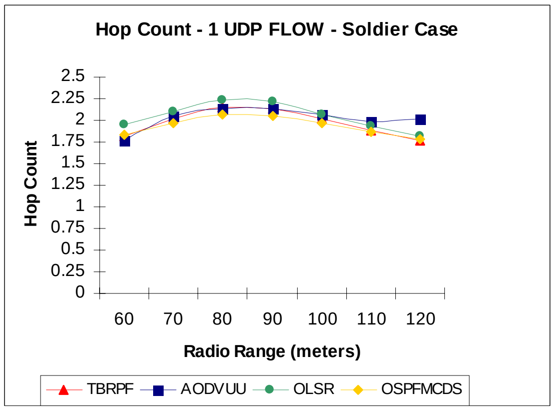
For the same soldier scenario case, when the traffic load is just one CBR flow, the traffic has negligible effect on the performance metrics. Figure 4.3(a) shows a slightly decreased end-to-end delay when the number of flows is small, which could be attributed to the reduced collision in the network. Likewise, the packet delivery percentage increases slightly for all the protocols, as shown in Figure 4.3(b).With only a single traffic flow in the network, the protocols behave ideally, as expected. In this case, because there is only one flow, the number of AODV route requests is low, thus giving it a low overhead, as shown in Figure 4.3(d).



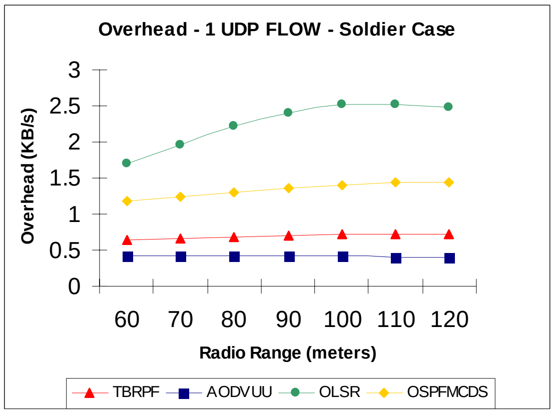
(a) Average End-to-End Delay



(b) Packet Delivery Percentage



(c) Average Hop Count



(d) Overall Overhead

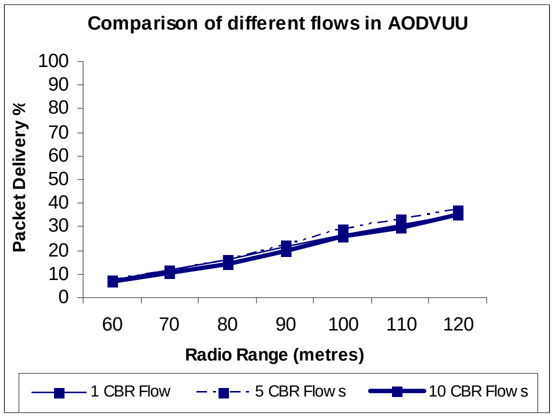
Figure 4.3: Relative Performance Comparison - Small Scenario – Soldier Case - 1 CBR Flow

4.2.2 Traffic Performance

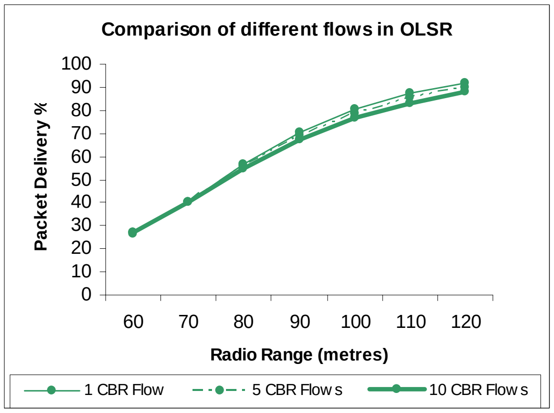
Since the above graphs have provided a means of observing and comparing only the relative performance of each protocol under a fixed traffic load, it is important to continue the comparison by observing the performance of each protocol under varying loads. In the soldier scenario case, an increase in traffic loads affects the packet delivery percentage and end-to-end delay. For brevity only the metrics that exhibit significant variations with different flows are presented here. Appendix E provides a complete set of soldier case results.

4.2.2.1 Packet Delivery Percentage

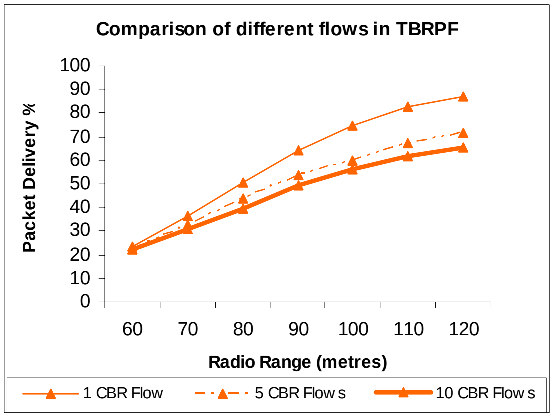
AODV and OLSR achieve a consistent packet delivery ratio irrespective of traffic load in the soldier case, as shown in Figure 4.4 (a) and 4.4 (b). In contrast, TBRPF and OSPF-MCDS show a decreased performance under increasing loads. TBRPF and OSPF-MCDS deliver only 65% and 63% of packets, respectively, at 120 m radio range, whereas OLSR delivers 88% of packets at the same radio range. This difference occurs because, as the traffic flow increases, TBRPF and OSPF-MCDS experience periodic collisions from the different forwarding nodes. In OLSR, the use of MPR nodes streamlines the data through a few nodes and reduces the spatial collision, because there are fewer contention sources. As the radio range increases, the node degree increases, thus creating a small set of MPR nodes, which, in turn, reduces spatial collision. This explains the linear increase in OLSR packet delivery compared to other protocols, with respect to increasing radio ranges. OLSR is an ideal candidate for MANET if a high throughput (packet delivery percentage) is required.



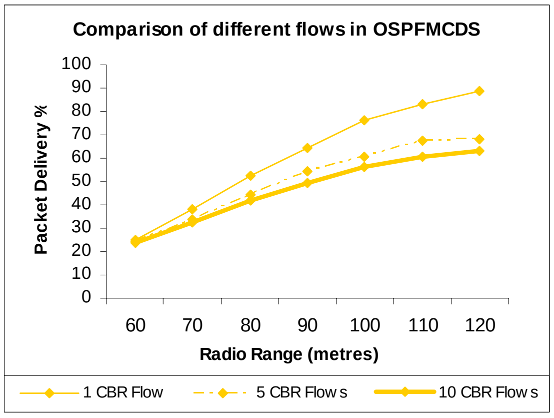
(a) *Packet Delivery Percentage in AODVUU*



(b) *Packet Delivery Percentage in OLSR*



(c) *Packet Delivery Percentage in TBRPF*



(d) *Packet Delivery Percentage in OSPFMCDS*

Figure 4.4: Soldier Case – Comparison of packet delivery percentage with different traffic flows

4.3 Medium Scenario - Medium Speed – Ship Case

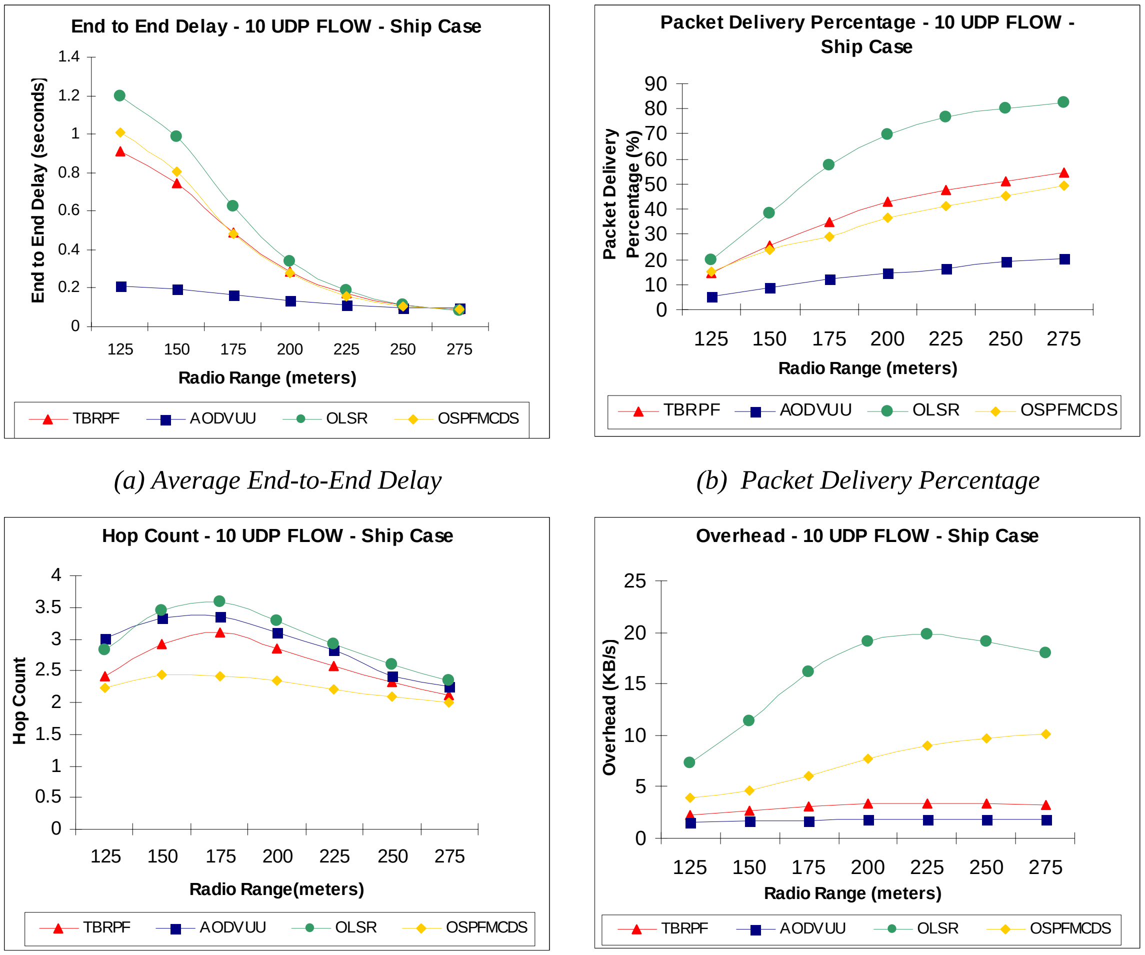
The ship movement scenario consists of a group of 45 navy ships (nodes) moving at a maximum speed of ten m/s with a maximum pause time of 60 seconds. The performance metrics in this scenario are measured as a function of varying radio ranges. As radio range increases the node degree of the nodes increases and the partitions in the network get reduced. At 275 meters radio range the network partition reduces to less than two and achieves an almost fully connected network link density. The following analysis of the medium scenario first focuses on the performance of each protocol with a fixed number of traffic flows and then compares the performance under different traffic flows.

4.3.1 Relative Performance

The relative performance is more complex in this scenario than in the soldier scenario because of the increase in the number of nodes and the speed at which they move. Therefore, the analysis is divided into comparisons of TBRPF, AODV, OLSR and OSPF-MCDS with the medium node set’s performance in a lightly and heavily loaded network.

4.3.1.1 Medium Node Set Lightly Loaded Network

In the ship movement scenario, Figure 4.5 shows the performance metrics under a fixed traffic load of 10 CBR flows. More specifically, Figure 4.5(a) shows that OLSR has the highest average end-to-end delay of the four protocols. This can be attributed to the fact that data packets have to take a longer path in OLSR. In OLSR the average hop count is larger than in the other protocols due to the MPR nodes. As shown in Figure 4.5(c), the hop count in OLSR increases steadily until it reaches a radio range of 200 meters, after which it starts to converge with that of the other protocols.



(a) Average End-to-End Delay *(b) Packet Delivery Percentage*

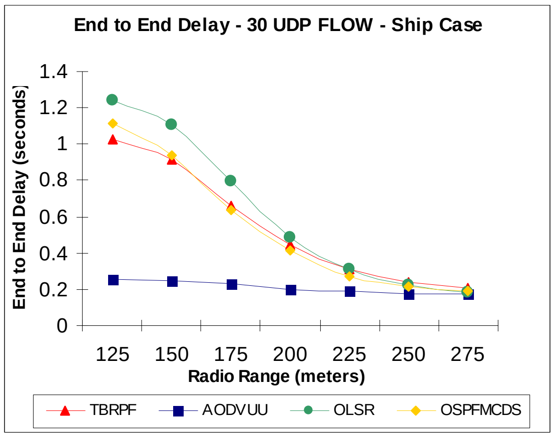
(c) Average Hop Count *(d) Overall Overhead*

Figure 4.5: Relative Performance Comparison - Medium Scenario – Ship Case - 10 CBR Flows

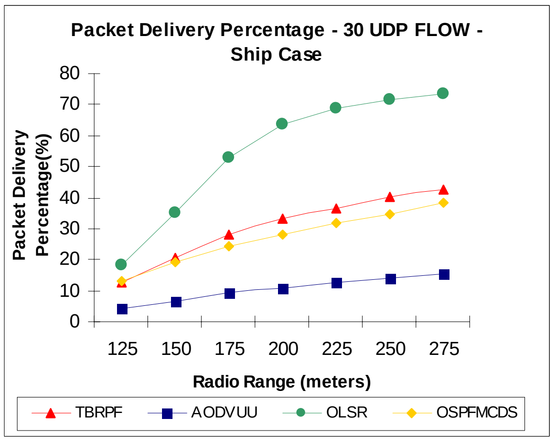
Figure 4.5(c) also shows that the hop count of OSPF-MCDS is the lowest at an average of 2.4 hops at 175 meters. In contrast, the other protocols have an average hop count of 3.6 for OLSR, 3.3 for AODV and 3.0 for TBRPF at 175 meters radio range.

4.3.1.2 Medium Node Set Heavily Loaded Network

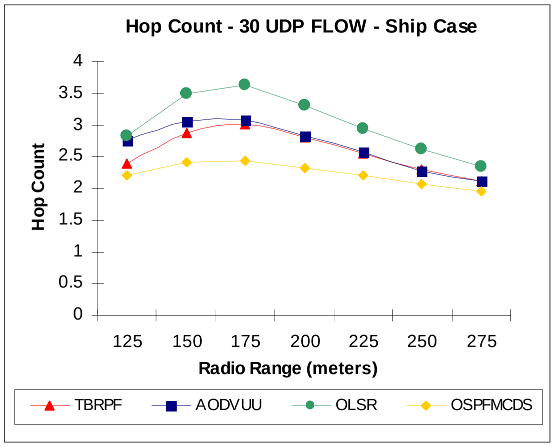
In the same ship scenario when the traffic load is increased to 30 CBR flows, all the protocols show decreased performance due to the increase in traffic load. With the exception of AODV, all the other protocol experience a 10% drop in packet delivery at 200 meters, mainly due to the increased collisions. The hop counts remain the same as those observed with 10 CBR flows. All three proactive protocols maintain a constant overhead, whereas AODV shows an increase in overhead from 1.8 KB/s at 10 CBR flows to 2.8 KB/s at 30 CBR flows (observed at 175 meters radio range).



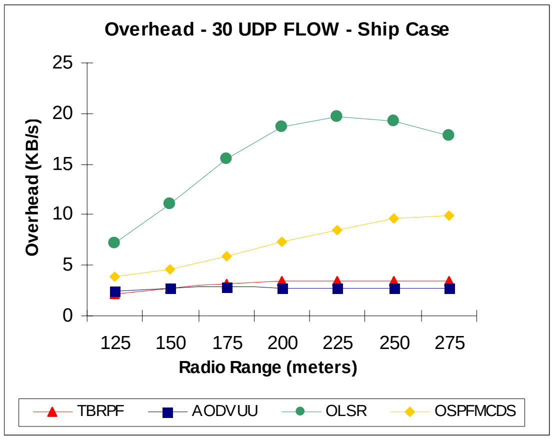
(a) Average End-to-End Delay



(b) Packet Delivery Percentage



(c) Average Hop Count



(d) Overall Overhead

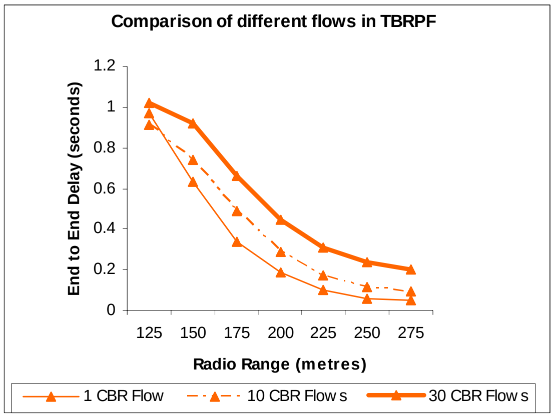
Figure 4.6: Relative Performance Comparison - Medium Scenario – Ship Case - 30 CBR Flows

4.3.2 Traffic Performance

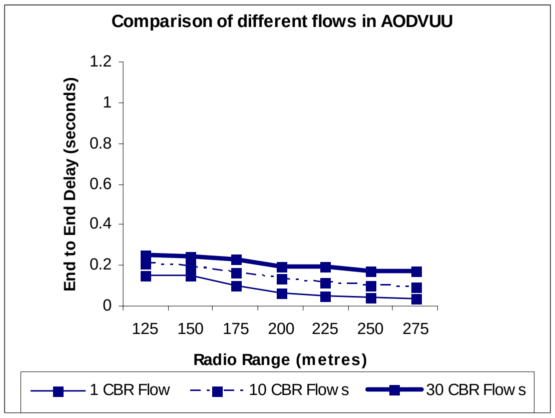
Observation of the same performance metrics under various traffic loads in the ship scenario case reveals that all metrics show variations, which are most pronounced in end-to-end delay. It is important to note that, according to the IEEE 802.11 standard [39], collision avoidance and backoff are not performed while transmitting broadcast packets. This is of significance to the present results, because all the MANET protocols send their control information as broadcast packets. The different broadcast packets themselves do not introduce delay, but when they collide with data packets they contribute to the increase in end-to-end delay and other subsequent performance degradations resulting from the collisions. For brevity only the metrics which exhibit significant variations with different flows are presented here. Appendix E provides the remaining set of ship case graphs.

4.3.2.1 End-to-End Delay

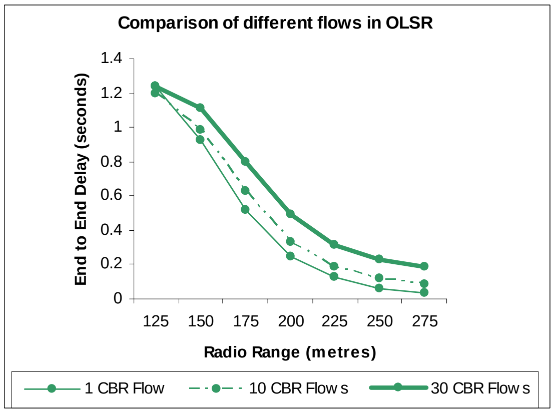
Compared to the soldier case, the ship case has a larger number of nodes and increased traffic loads. When the traffic loads increase, the number of contending sources also increases causing collisions and increased average end-to-end delay, as shown in Figure 4.7. In each protocol, the one CBR flow case produces the least end-to-end delay. It is important to note here that AODV's low end-to-end delay comes at the cost of a reduced packet delivery percentage (throughput) as shown in Figure 4.6(b).



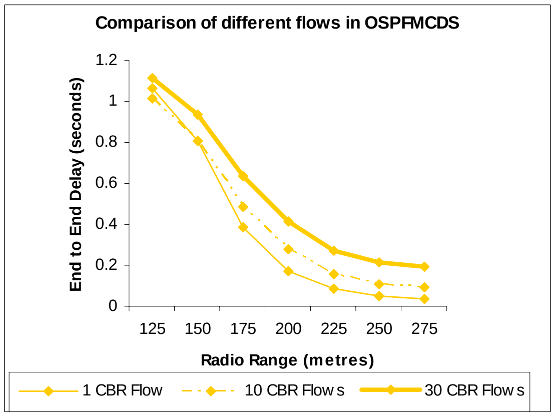
(a) End-to-End Delay in TBRPF



(b) End-to-End Delay in AODVUU



(c) End-to-End Delay in OLSR



(d) End-to-End Delay in OSPFMCDS

Figure 4.7: Ship Case – Comparison of end-to-end delay with different traffic flows

4.3.2.2 Average Hop Count

With a medium size set of nodes traveling at a medium speed in the ship scenario, the proactive protocols (OLSR, TBRPF and OSPF-MCDS) offer a consistent hop count even under increasing traffic loads. However, AODV, a reactive protocol, has a reduced hop count with increasing traffic loads, as Figure 4.8 shows, because in AODV the intermediate nodes use previous knowledge of a route to the destination. When the new flows create new route discovery processes, the intermediate nodes learn shorter routes. Eventually other source destination pairs use this knowledge, thereby reducing the average hop count as the flows increase.

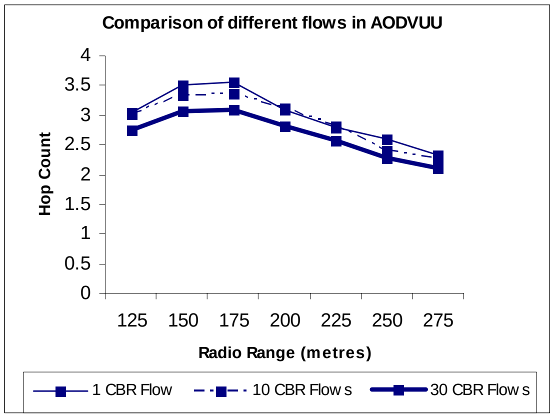


Figure 4.8: Ship Case – Comparison of AODVUU hop count with different traffic flows

4.4 Large Scenario - High Speed - Car Case

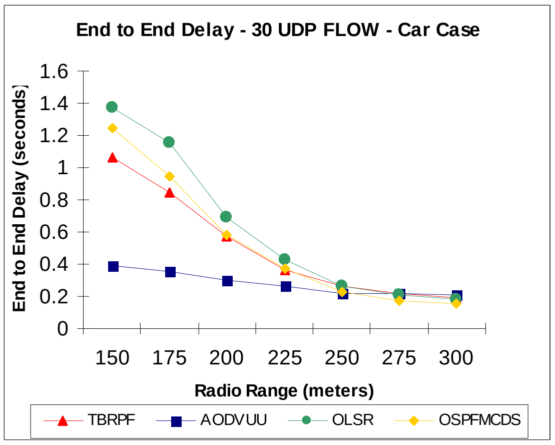
The car movement scenario consists of a group of 80 cars (nodes) moving at a maximum speed of 20 m/s with a maximum pause time of 60 seconds. The performance metrics are measured as a function of varying radio ranges. As the radio range increases, the node degree of the nodes increases and the partitions in the network decrease. At 300 meters radio range, the average number of network partitions decreases to less than 1.5 and an almost fully connected network is achieved with increased link density. First, the performance of each protocol with a fixed number of traffic flows is presented and, then, the performance under different traffic flows is compared.

4.4.1 Relative Performance

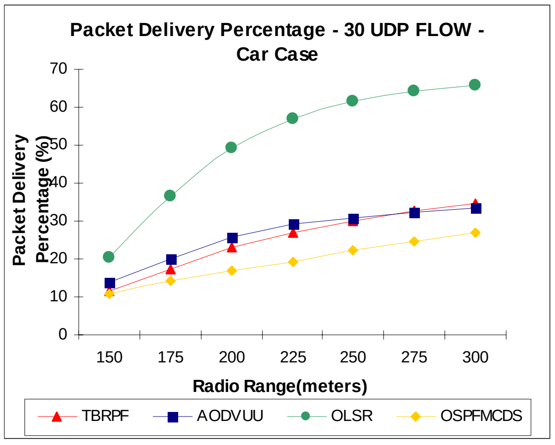
The relative performance of each protocol in this scenario illustrates the impact of large node sets at relatively higher speeds of node movement. The analysis below shows the comparison of large node set performance metrics under a medium and heavily loaded network.

4.4.1.1 Large Node Set Medium Loaded Network

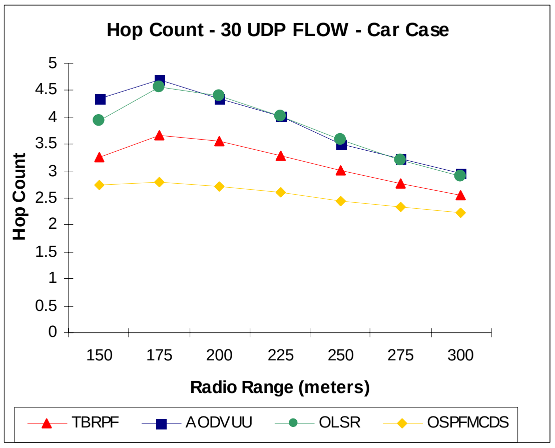
The car case with 30 CBR flows is well suited for observing the behavior of protocols under a large node set and medium traffic load, because it is more representative of a real-world scenario. Figure 4.9 shows the performance metrics for this case. As is evident from the graph in Figure 4.9(b), OSPF-MCDS exhibits scalability problems with large node sets. At 250 meters, the packet delivery is only 22%, the lowest of all the protocols studied. Also OSPF-MCDS has an increased overhead without any throughput (packet delivery percentage) improvement. OLSR shows the best packet delivery percentage, but at a cost of increased overhead, due to the non-optimized control packet data. AODV has the highest average path length in this case, because once AODV forms a route to the destination, it does not switch to a better route when one becomes available, unless the current path breaks. This situation leads to sub-optimal path selection. TBRPF shows modest packet delivery and low control overhead while at the same time delivering low end-to-end delay. TBRPF also has a 15 to 20% lower path length than AODV and OLSR. TBRPF is, therefore, the ideal candidate for low bandwidth, large scale MANET environments at a modest packet delivery performance. TBRPF would also be ideal for lower power applications as it has a low control packet overhead.



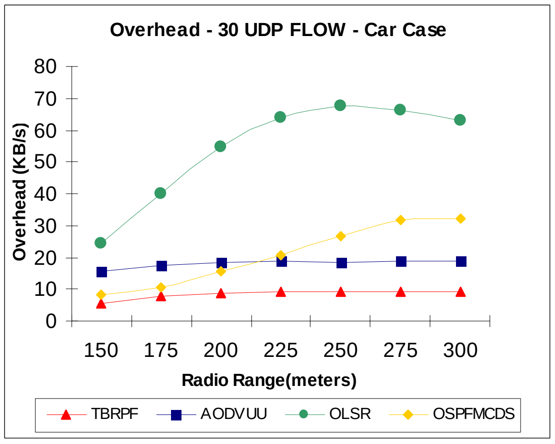
(a) Average End-to-End Delay



(b) Packet Delivery Percentage



(c) Average Hop Count

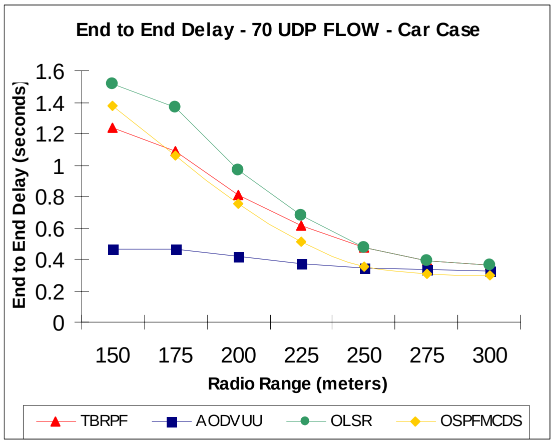


(d) Overall Overhead

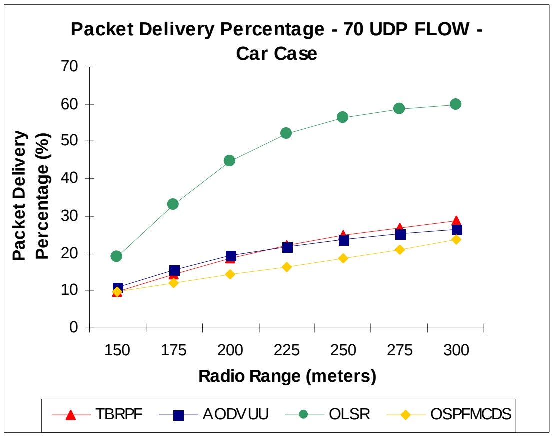
Figure 4.9: Relative Performance Comparison – Large Scenario – Car Case - 30 CBR Flows

4.4.1.2 Large Node Set Heavily Loaded Network

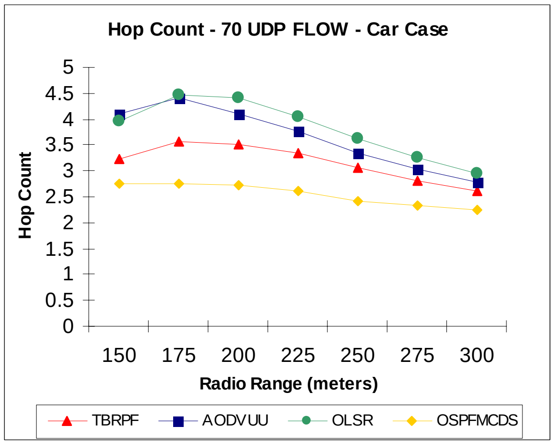
In this case, the protocols are subject to heavy traffic loads to simulate extreme conditions. There are 70 CBR flows in an 80 node scenario. At 70 CBR flows, every node in the network sources or sinks traffic for at least one pair of a CBR traffic flow. Figure 4.10 shows that the packet delivery performance for all of the protocols decreases with the increase in traffic load mainly because of packet drops resulting from increased collisions. Section 4.4.2.2 makes it clearer that the effects of increased traffic load only slightly reduce the performance metrics compared to the 30 CBR flows case. The overhead of TBRPF is almost constant, because it only reports differential topology changes. In contrast, although OSPF-MCDS uses differential updates for the neighbor detection HELLO messages, it includes the full neighbor link state descriptions, thus causing a gradual increase in OSPF-MCDS overhead, because when the link density increases, there are more neighbor links to report in the link state descriptions.



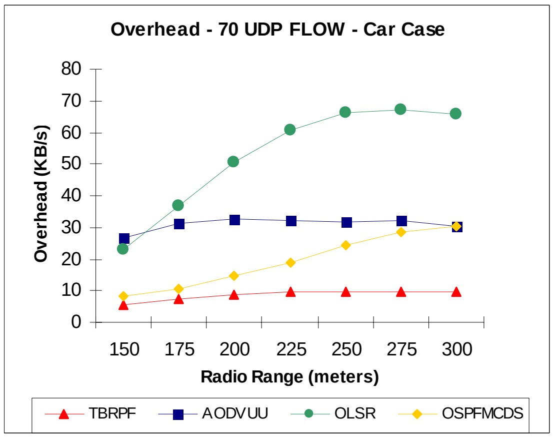
(a) *Average End-to-End Delay*



(b) Packet Delivery Percentage



(c) Average Hop Count



(d) Overall Overhead

Figure 4.10: Relative Performance Comparison - Large Scenario – Car Case - 70 CBR Flows

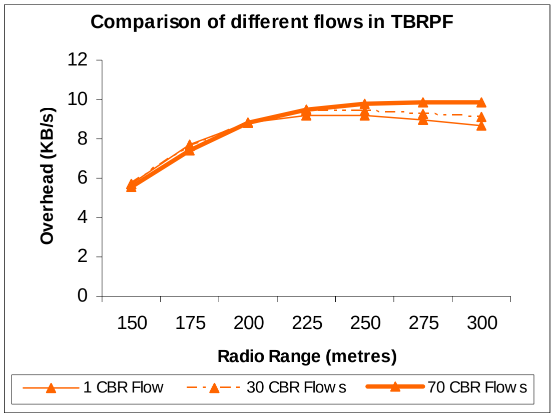
4.4.2 Traffic Performance

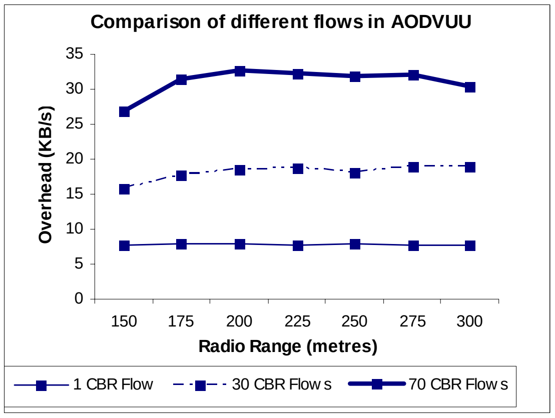
To illustrate the effects of increased traffic load we plot the performance metrics for different traffic loads were plotted as a function of radio range for each of the four protocols.

4.4.2.1 Routing Overhead

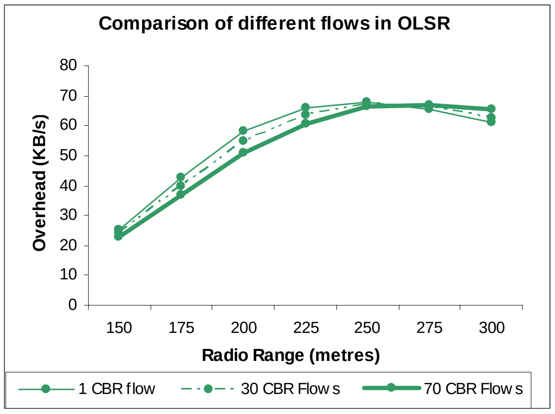
In the three proactive protocols, the routing overhead is independent of the traffic load, as shown in Figure 4.11(a), 4.11(c), and 4.11(d). However, with AODV, a reactive protocol, the routing overhead increases with the traffic load. Figure 4.11(b) shows AODV's routing overhead increasing by a factor of two for each increase in traffic flow because of the new route discovery process, which happens in AODV for every new traffic flow. This increase in overhead without any notable improvement in throughput (packet delivery percentage), as shown in Figure 4.10(d), demonstrates that AODV has a scalability problem with large networks and heavy traffic loads.

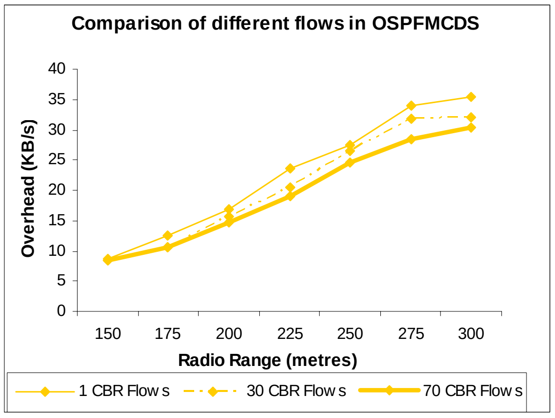
In comparison, TBRPF has the lowest overhead of all the protocols, as Figure 4.9(d) and Figure 4.11(a) show, because TBRPF uses reduced headers namely 8-bit sequence numbers and differential HELLO messages. Further, TBRPF broadcasts the changes only for a threshold number of times, typically three by default. TBRPF benefits from highly dense networks as the reportable partial tree becomes smaller in more dense networks.





(a) Routing Overhead in TBRPF *(b) Routing Overhead in AODVUU*



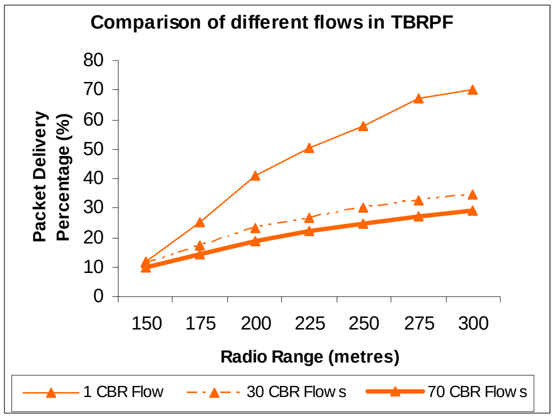


(c) Routing Overhead in OLSR *(d) Routing Overhead in OSPFMCDS*

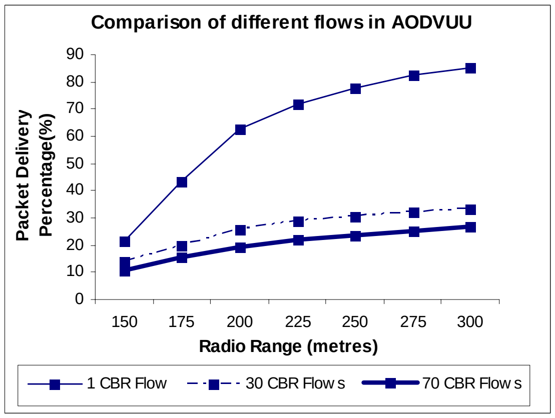
Figure 4.11: Car Case – Comparison of overhead with different traffic flows (Vertical Scale Adjusted for clarity)

4.4.2.2 Packet Delivery Percentage

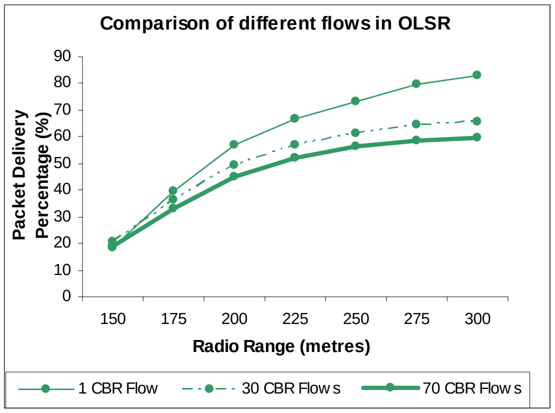
Figure 4.12(a) through 4.12 (d) show the packet delivery percentage as a function of radio ranges for all the protocols under investigation. It can be clearly seen that the performance significantly degrades from one CBR flow in the network to 30 CBR flows. However, the difference between 30 CBR flows and 70 CBR flows is marginal. Among all the protocols investigated, OLSR gives the best throughput (packet delivery percentage), followed by TBRPF, AODV and OSPF-MCDS. The effect of increased traffic load is minimal in OLSR compared to the other protocols. OLSR experiences a 12% reduction in performance from one CBR flow to 30 CBR flows at 250 meters radio range, whereas at the same radio range the other protocols suffer 20 to 40% reduction in performance. OLSR scales well at increased traffic loads.



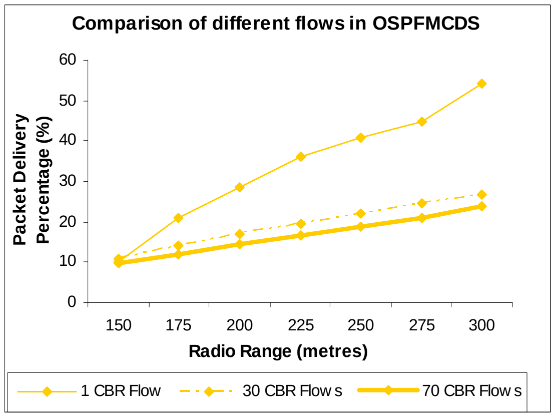
(a) Packet Delivery Percentage in TBRPF



(b) Packet Delivery Percentage in AODVUU



(a) Packet Delivery Percentage in OLSR



(b) Packet Delivery Percentage in OSPFMCDS

Figure 4.12: Car Case – Comparison of packet delivery percentage with different traffic flows (Vertical Scale Adjusted for clarity)

4.5 Summary

This chapter outlined performance constraints and metrics for the four protocols being investigated in this study. The three scenarios were then introduced and the relative performance of the protocols under each usage scenario was explained. This chapter provided a comparison of the performance of each protocol under different traffic loads. The results of the comparisons have highlighted the best candidates for certain MANET scenarios.

Chapter 5

Conclusions and Future Work

This study provided a detailed investigation of the operation and performance of three standardized MANET routing protocols and a relatively new protocol, OSPF-MCDS. Using simulation, the performance of these protocols was compared and recommendations made for the best candidates for different scenarios. The study also presented some improvements to OSPF-MCDS, as well as a simulation model of OSPF-MCDS and a generic routing template for the ns-2 simulator . The bugs discovered in TCL and ns-2 were discussed and the program code fixes for these bugs explained. The study has introduced some new tools to aid scenario visualizations. The simulation methodology presented has created a standard basis for the comparison of future ad-hoc routing protocols.

It is evident from this study that no single protocol is a panacea for all MANET routing needs. The dynamic nature of wireless networks requires certain approaches based on the expected mobility scenario. OLSR lives up to its protocol specifications because it performs well in a highly dense network even under varying load conditions. It gives a high throughput under most conditions, but at the cost of an increased overhead.

TBRPF is the ideal candidate for low bandwidth and low power applications, because it has a low overhead irrespective of the scenario size. Although it scales well to large node sets, it does degrade in throughput when traffic load increases. In contrast, AODV, the only reactive protocol assessed in this study, suffers from scalability problems. Because it is a reactive protocol, its overhead is directly proportional to the number of traffic flows. AODV performs well in static scenarios under low traffic loads, but with even small node movements it fails to maintain good throughput.

OSPF-MCDS offers the minimum shortest path over any of the protocols investigated in this study. It is an ideal candidate when shortest path is desired, for subsystem integration like QoS, load balancing and so forth. OSPF-MCDS performs well with sporadic traffic under any scenario. However, its performance degrades in large node set and heavy traffic loads.

5.1 Future Work

The results of the assessment of the four protocols indicate that there is scope for improvement in proactive protocols, particularly OLSR and OSPF-MCDS. The observations from simulation results have revealed that OLSR performs well in large dense networks, but requires increased overhead. OLSR’s overhead can be reduced by using differential schemes similar to the one used in TBPRF, especially for HELLO messages and link state updates. OSPF-MCDS’s overhead can be reduced further by limiting its topology information to differential changes in topology rather than the full topology in every update. Overhead reduction in OSPF-MCDS can also be achieved by using smaller and more efficient headers in the protocol update packets. Piggy backing topology updates with neighbor sensing packets offers one way to reduce the packet header overhead significantly. These observations can be integrated to form another new protocol, which combines the benefits of OLSR and TBRPF. The protocols discussed do not address security issues; it would be interesting to observe the effects of security additions to the performance of these protocols. The simulation study can be extended to any future MANET routing protocols to facilitate comparison of the new protocol to the existing ones investigated in this study.

Biblography

[1] T. H. Clausen, G. Hansen, L. Christensen, and G. Behrmann, "The Optimized Link State Routing Protocol, Evaluation Through Experiments and Simulation," *Proceedings of IEEE Symposium on Wireless Personal Mobile Communications*, September 2001, pp. 841-846.

[2] T. Clausen, P. Jacquet, A. Laouiti, P. Minet, P. Muhlethaler, A. Qayyum, and L. Viennot, "Optimized Link State Routing Protocol," Internet Engineering Task Force (IETF) draft,March, 2002. Available at http://www.ietf.org/internet-drafts/draft-ietfmanet-olsr-06.txt.

[3] B. Bellur and R.G. Ogier, "A Reliable, Efficient Topology Broadcast Protocol for Dynamic Networks," *Proceedings of INFOCOM*, 1999, pp. 178-186.

[4] R. G. Ogier, F. L. Templin, B. Bellur, and M. G. Lewis, "Topology Broadcast Based on Reverse-Path Forwarding (TBRPF)," Internet Engineering Task Force (IETF) draft, November 2002. Available at http://www.ietf.org/internet-drafts/draft-ietf-manet-tbrpf-06.txt.

[5] D. Johnson and D. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," *Mobile Computing*, Kluwer Academic, vol. 353, pp. 153-181, 1996.

[6] D. B. Johnson, D. A. Maltz, Y. C. Hu, and J. G. Jetcheva, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)," Internet Engineering Task Force (IETF) draft, Febuary 2002. Available at http://www.ietf.org/internet-drafts/draftietf-manet-dsr-07.txt.

[7] V. D. Park and M.S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," *Proceedings of INFOCOM*, 1997, pp. 1405-1413.

[8] V. Park and S. Corson, "Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification," Internet Engineering Task Force (IETF) draft, July 2001.

[9] C. E. Perkins and E. M. Royer, "Ad hoc On-demand Distance Vector Routing," *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 1999, pp 90–100.

[10] C. E. Perkins, E. M. Belding-Royer, and S. R. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," Internet Engineering Task Force (IETF) draft, November 2002. Available at http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-12.txt.

[11] MANET Charter, Internet Engineering Task Force (IETF). Available at http://www.ietf.org/html.charters/manet-charter.html.

[12] T. Lin, S. F. Midkiff, and J. S. Park, "Minimal Connected Dominating Set Algorithms and Application for a MANET Routing Protocol," *Proceedings of IEEE International Performance, Computing, and Communications Conference*, 2003, pp. 157-164.

[13] J. Moy, “OSPF Version 2,” Internet Engineering Task Force RFC 2328, April 1998. Available at http://www.ietf.org/rfc/rfc2328.txt.

[14] T. Lin, “draft-ospfmcds-00.txt,” IETF draft, *Mobile Ad-hoc Network Routing Protocols: Methodologies and Applications*, Sub-document of unpublished doctoral dissertation, Virginia Polytechnic Institute and State University, Blacksburg, pp 218-253, 2004.

[15] K. Fall and K. Varadhan. ns Notes and Documentation. Technical report, UC Berkeley, LBL, USC/ISI, and Xerox PARC, June 2003. Available at http://www.isi.edu/nsnam/ns/ns-documentation.html.

[16] C.S.R. Murthy and B.S. Manoj. *Ad Hoc Wireless Networks: Architectures and Protocols*. Pearson Education, pp. 207-208,304, 2004.

[17] T. H. Clausen, G. Hansen, L. Christensen, and G. Behrmann, “The Optimized Link State Routing Protocol, Evaluation through Experiments and Simulation,” *IEEE Symposium on Wireless Personal Mobile Communications*, 2001, pp. 841-846.

[18] S. Nesargi, and R. Prakash, “A tunneling approach to routing with unidirectional links in mobile ad-hoc networks,” *Proceedings of the IEEE International Conference on Computer Communications and Networks (ICCCN)*, 2000, pp. 522-527.

[19] Y. Dalal, and R. Metclafe, “Reverse Path Forwarding of Broadcast Packets,” *Communications of the ACM*, Vol. 21, No. 12, pp. 1040-1048, December 1978.

[20] D. B. West, *Introduction to Graph Theory*, 2nd edition, Prentice Hall, 2001.

[21] T. Lin, “Mobile Ad-hoc Network Routing Protocols: Methodologies and Applications,” unpublished doctoral dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, March 2004.

[22] J. Broch, D. Maltz, D. Johnson, Y-C. Hu, and J. Jetcheva, “A Performance Comparison of Multi-hop Wireless Ad hoc Network Routing Protocols,” *Proceedings of IEEE/ACM MOBICOM*, 1998, pp 85–97.

[23] S. Das, C. Perkins, and E. Royer, “Performance Comparison of Two On-demand Routing Protocols for Ad hoc Networks,” *Proceedings of IEEE INFOCOM*, 2000, pp. 3-12.

[24] L. DaSilva, S. Midkiff, J. Park, G. Hadjichristofi, K. Phanse, T. Lin, and N. Davis, “Network Mobility and Protocol Interoperability in Ad hoc Networks,” *IEEE Communications Magazine*, vol. 42, no. 11, pp. 88-96, November 2004.

[25] K. Phanse and L.A. DaSilva, “Protocol Support for Policy-Based Management of Mobile Ad Hoc Networks,” *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2004, pp 3-16.

[26] S. Corson, and J. Macker, “Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations,” Internet Engineering Task Force Network Working Group RFC 2501, January 1999.

[27] T. Camp, J. Boleng, and V. Davies, “A Survey of Mobility Models for Ad Hoc Network Research,” *Wireless Communications and Mobile Computing (WCMC): Special Issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, vol. 2, no. 5, pp. 483–502, 2002.

[28] B. Karp, *Geographic Routing for Wireless Networks*, unpublished doctoral dissertation, Cambridge, MA, Harvard University, 2000.

[29] C. de Waal, and M. Gerharz, “BonnMotion mobility scenario generator and analysis tool,” 2003. Available at http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/ BonnMotion/.

[30] R. Jain, *The Art of Computer Systems Performance Analysis*, JohnWiley & Sons, pp. 441-455, 1991.

[31] OmNet manual. Available at http://zeus.unex.es/~victor/ralteleco/software/descarga/ usman.pdf.

[32] H. T. Friis, “A note on a simple transmission formula,” *Proceedings of the Institute of Radio Engineers*, vol. 34, pp. 254-256, May 1946.

[33] J. Chung and M. Claypool, “NS by Example”. Available at http://nile.wpi.edu/NS/.

[34] E. Altman and T. Jimenez, ”NS Simulator for Beginners,” Lecture Notes, Autumn 2002, Univ. de Los Andes, Merida, Venezuela, July 2003.

[35] OLSR implementation and simulation code. Available at http://hipercom.inria.fr/olsr/.

[36] AODV Implementation from Uppsala University (AODV-UU) version 0.7.1. Available at http://www.docs.uu.se/docs/research/projects/scanet/aodv/aodvuu.shtml.

[37] TBRPF ns-2 source code available at http://www.erg.sri.com/projects/tbrpf/ sourcecode.html.

[38] NS-2 Trace Formats available at http://www.k-lug.org/%7Egriswold/NS2/ns2-trace-formats.html#wireless:old.

[39] IEEE Standards Board, “Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” ANSI/IEEE Std 802.11, *The Institute of Electrical and Electronics Engineers Inc.*, 1999 Edition.

Appendix A

Building a new MANET Ad-hoc Routing Agent in ns2

Palan Annamalai [palan AT vt d0t edu]
v1.0

There are many tutorials available to familiarize yourself with ns2 simulation setup and its internal workings [1][34]. I will try to briefly introduce ns2 and demonstrate how to create a new MANET Ad-hoc Routing protocol in ns2. If you're looking to convert an existing Linux/Windows routing code to ns2, this tutorial will provide you a generic template to port your existing routing code to ns2. If you're interested in just learning how to create a new agent in ns2 follow the code closely and it is well commented to show the key aspects of creating your new agent.

Network Simulator version 2(ns2) is an object oriented discrete event driven network simulator. The simulation parameters and metrics are not affected by the computation delays this is a major benefit of discrete event simulators. The internal workings of ns2 are documented in [3]. We will assume the reader is familiar with ns2 basics at least to the level "Extending NS" described in [1].

We will construct a new router agent in ns2 called MyRouter. This simple routing agent will send out dummy protocol packets of size 122 (seqno+IP header+100 byte payload) every 10 seconds. When the agent receives an application packet it will forward the packet to a fixed node (node2 in this case). Finally after creating the agent we will use a simple validation script to test our new MyRouter agent in action.

ns2 Interfaces:

Lets call our router protocol packets MyRTR (MyRouter). Our packet has to be defined before the last packet type (PT_NTYPE) in packet_t structure. Add the new packet type constant to ~ns2source/common/packet.h,

```
enum packet_t {
…..
    //Dummy Routing Protocol - Palan
    PT_MyRTR,
}
```

The enum structure will assign a number to PT_MyRTR, we need to associate the protocol name to the packet type number. In this case our protocol packet name is MyRouter, add it to ~ns2source/common/packet.h

```
p_info() {
        ……
  //Dummy Routing Protocol
```

```
        name_[PT_MyRTR] = "MyRouter"; //MyRouter is the packet type string
                ……
      }
```

Each time ns2 starts, our packet has to be initialized and added to the packet manager's list of packets. To create the packet when the simulator starts add your packet type string to the ~ns2source/tcl/lib/ns-packet.tcl

```
        foreach prot {
        …
        MyRouter
        }
```

We need to create the OTcl commands needed to setup our routing protocol for a mobile node, to achieve that add the following to ~ns2source/tcl/lib/ns-lib.tcl

```
        switch -exact $routingAgent_ {
                ……
             MyRouter {
                     set ragent [$self create-MyRouter-agent $node]
                 }
        ……}

        Simulator instproc create-MyRouter-agent { node } {
             #  Create MyRouter routing agent
             set ragent [new Agent/MyRouter [$node node-addr]]
             $self at 0.0 "$ragent start"     ;# start MyRouter Agent
             $node set ragent_ $ragent
             return $ragent
        }
```

When a user configures the mobile node to have a new routing agent using *$ns_ node-config –adhocRouting MyRouter* <…*other node parameters*>, the OTcl interpreter calls the create-MyRouter-agent function. This function instantiates the MyRouter agent and signals the agent to start functioning.

MyRouter Routing Agent:

Drop the program listings for myrouter_agent.cc and myrouter_agent.h in ~ns2source/ns-2.26/MyRouter. The source code for these files are well commented, however we will discuss some sections of the code in detail.

First we need to define the structure for our packet. The access and offset functions should be defined so that ns2 can manage the packet using its packet header manager object for more details on this refer to the chapter "Packet Headers and Formats" in ns2 doc[3].

```
struct hdr_MyRouter {
        u_int16_t seqno_;
        //ns2 specific packet header access functions
        static int offset_;
        inline static int& offset() { return offset_; }
        inline static hdr_MyRouter* access(const Packet* p) {
               return (hdr_MyRouter*) p->access(offset_);
        }
};
```

The packet header and agent classes have to create an OTcl linkage, for the user to create an instance of our routing agent. In our packet header class, the constructor informs the PacketHeader manager the size of the Packet and the variable (the variable expecting the position is offset_) at which to return the offset number for this particular packet in the entire ns2 simulation packet (to understand further refer to Bag of Bits in ns2 doc[3])

```
static class NS_MyRouter_PktHeaderClass : public PacketHeaderClass {
public:
NS_MyRouter_PktHeaderClass(): PacketHeaderClass("PacketHeader/MyRouter",sizeof(hdr_MyRouter)) {
                    bind_offset(&hdr_MyRouter::offset_);
            }
}
```

The TclClass derived constructor (NS_MyRouterClass in this case) creates the OTcl class (interpreted class) for MyRouter under Agent and inserts this object into the link list of TclClass objects. The name a user will invoke in OTcl space is Agent/MyRouter, this creates the actual C++ routing agent object of type TclObject. It would suffice to know that TclClass and TclObject are the primary classes that facilitate the OTcl linkage.

```
static class NS_MyRouterClass : public TclClass {
public:
            NS_MyRouterClass() : TclClass("Agent/MyRouter") {}
            TclObject *create(int argc, const char *const *argv) {
                        return (new NS_MyRouterAgent ((nsaddr_t)atoi(argv[4])));
                        //NS_MyRouterAgent is the actual class implementing the ns2 routing
            }
} NS_MyRouterClass_object; // This class object will get created as soon as ns2 starts
```

After the user creates a new instance of our routing agent, we need to provide the interface for the user to set the control parameters in our routing agent. This is done by using the command function. The code snippet below shows the routing agent receiving a command “start” which sets the routing agent to start its operation.

```
// --== Tcl Command Interface ==--
int NS_MyRouterAgent::command(int argc, const char *const *argv) {
……
            } else if (command_is("start", 0, argc, argv)) {
             // start simulation
            Start();   // This start function should activate your routing protocol fill in with appropriate code
            return TCL_OK;
            }
…}
```

Ns2 maintains a common header for each packet defined by struct hdr_cmn, this structure defines the direction of the packet, the packet type, its size etc. This header size is not included in the actual size of the protocol packet. The various headers in a received packet can be accessed using the HDR_IP() macro and alike as shown below.

```
void NS_MyRouterAgent::recv(Packet *p, Handler *) {
   /* Access common header */
   struct hdr_cmn *ch = HDR_CMN(p);
   /*Access ip header */
   struct hdr_ip *ih = HDR_IP(p);
   /* Access MyRouter header */
   struct hdr_MyRouter *myrtr = HDR_MyRTR(p);
if (ch->ptype() == PT_MyRTR) {
          ih->ttl_ -= 1; //Passed through router so ttl less 1
….}
```

The routing layer (RTR) in this case on receiving an application packet forwards it to a fixed node after setting the appropriate common header values. It would send the packet to the link layer (LL) by scheduling the packet to be received by the next linked target_ object.

```
          void NS_MyRouterAgent::forward(Packet *p, addr_t nexthop) {
          ….
             ch->prev_hop_ = myaddr_;
             ch->next_hop_ = nexthop;
             ch->addr_type() = NS_AF_INET;
             ch->direction() = hdr_cmn::DOWN;
          …
          Scheduler::instance().schedule(target_, p, 0);
          }
```

Timers in ns2 should be derived from TimerHandler class. The TimerHandler class allows us to create an expiring timer, reschedule the timer and check the timer pending status. In our MyRouter agent when the routing agent is started with the "start" command, the timer is set to call the expire function immediately.

```
void NS_MyRouterAgent::Start() {
          // TODO: Add your functions to initialize your Agent
          NS_MyRouter_Timer_object.sched(0); //Schedule the first timer event immediately at 0 seconds
}
```

The expire function calls the sendPkt function to send out the protocol packet and then reschedules the expiry in the next 10 seconds.

```
void NS_MyRouter_Timer::expire(Event *e) {
….
  NS_MyRouterAgentPtr_->sendPkt(mydata,100); /
  resched(10);     //Assuming 10 seconds once you want to send routing control packets
}
```

In the sendPkt function we create a new ns2 packet, fill the various protocol headers and attach our data payload to the newly created IP packet. There are two ways to access the data payload in a packet using acccessdata() and setdata(). In ns2 there are two primary packet palyload classess called AppData and PacketData. PacketData is derived from AppData with a fixed payload type called PACKET_DATA.

Method 1: Copy over our protocol packet buffer to the newly created ns2 packet as its data payload

```
        Packet *p = allocpkt(pkt_len);
        ….
        memcpy(p->accessdata(), pkt_buf, pkt_len);
```

Method 2: Create a new PacketData object and then use setdata()

```
        Packet *p = allocpkt(pkt_len);
        ……
        PacketData* pktdata = new PacketData(pkt_len);
        memcpy(pktdata->data(), pkt_buf, pkt_len);
        p->setdata(pktdata); // setdata takes a value of an AppData object but since PACKET_DATA is
        //the only Packet type we need - we can use a PacketData Object
```

Tracing and Logging the new protocol packets:

When tracing is enabled by the user the trace routines for wireless present in cmu-trace.cc need to be able to identify the packet of type PT_MyRTR to be able to log it in the trace file. This is done by the following code additions to ~ns2source/trace/cmu-trace.cc

```
        void CMUTrace::format(Packet* p, const char *why)
        {….
                case PT_MyRTR:
                     format_MyRouter(p,offset);
                     break;
        …}

        void CMUTrace::format_MyRouter(Packet *p, int offset)
        {
             u_int8_t *pktdata = (u_int8_t*)p->accessdata();
             //printf("--- %s ----\n", pktdata);
             if(strncmp((const char*)pktdata,"AAAAAAA",7) == 0) {
                  sprintf(pt_->buffer() + offset, " %s ", "MyRouter Dummy Packet");
             } else {
                  sprintf(pt_->buffer() + offset, " %s ", "Congrats! You nailed ns2");
             }
        }
```

We also need to declare our packet processing function format_MyRouter as a private member of class CMUTrace in ~ns2source/trace/cmu-trace.h

```
        void format_MyRouter(Packet *p, int offset);
```

To compile the new MyRouter agent add the object files to your ~ns2source/ns-2.26/makefile.in and compile using the linux make utility.

```
OBJ_CC = \
…
$(OBJ_MyRouter)

# Define all your .cc files as .o files here to be compiled into ns2
OBJ_MyRouter = MyRouter/myrouter_agent.o
```

Validating our new Agent

To validate the new created routing agent we will use a simple MANET simulation setup with four nodes. Remember we created a static forwarding in our agent (all traffic is forwarded to node 2). The nodes are setup such that traffic originating at node 0 will start flowing towards to node 2 when node 2 moves in the radio range of node 0. This can be visualized from the NAM trace result obtained as show in Figure A.1.

From your ns2 folder run:
bin/ns validate_MyRouter.tcl
bin/nam MyRouter.nam

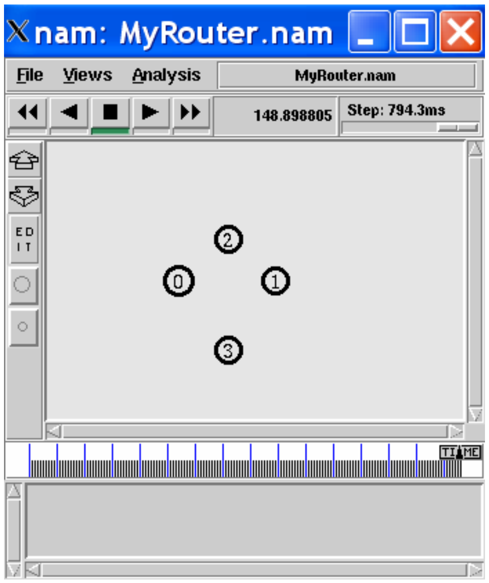


Figure A.1: Post Simulation Animation using NAM

The ns2 trace file format is documented in several tutorials [3][38]. The following lines are sample trace lines from *MyRouter.tr* from our simulation run.

CBR Traffic being dropped by the interface queue

```
D 66.000000000 _0_ IFQ  ARP 92 cbr 1000 [0 0 0 800] ------- [0:0 2:0 255 2] [64] 0 0
```

MyRouter protocol packet sent by the Routing agent

```
s 70.158726125 _1_ RTR  --- 98 MyRouter 122 [0 0 0 0] ------- [1:255 -1:255 255 0] SqNo. 36436
MyRouter Dummy Packet
```

CBR traffic packet received at the destination node

```
r 71.012227516 _2_ AGT  --- 102 cbr 1000 [13a 2 0 800] ------- [0:0 2:0 255 2] [70] 1 0
```

The complete trace format structure for various packets can be obtained from ns2 source. To understand the various headers recorded in the trace refer to Table A.1, it shows each filed of the trace line mapped to the corresponding headers.

Table A.1: Trace Format Explanation

Event Type	Time stamp	Src ID	Agent Name	Reason	Pkt ID	Pkt Type	Pkt Size	[Tx Duration	Dst MAC	Src MAC	MAC] Pkt Type
Drop	66.x	_0_	IFQ	ARP	92	cbr	1000	0	0	0	800
s(Tx)	70.x	_1_	RTR	---	98	MyRouter	122	0	0	0	0
r(Rx)	71.x	_2_	AGT	---	102	cbr	1000	13a	2	0	800

------ Contd.	[src IP:port	Dst IP:port	TTL	Next Hop]	*CBR/MyRTR only Seq No*	*CBR only* Num Fwds	*CBR only* Optimal Fwds
	0:0	2:0	255	2	64	0	0
	1:255	Bdcast:255	255	0	36436	N/A	N/A
	0:0	2:0	255	2	70	1	0

Each of the ns2 agents log data which contains at least the minimal information shown in Table A.1 along with any agent specific information added to the trace. *Event id* shows the r-receiving, s-send, D-drop and f-forward status of the packets. *Agent Name* specifies which agent logs the packet, the *Tx Duration* is the estimated time for the actual transmission. The other headers are self explanatory

Other Considerations:

It will be beneficial to know that ns2 simulator uses host byte order (little endian) the same as x86 computers hence you do not have to use network byte order (big endian) conversion functions in ns2 running on x86 computers.

Converting Linux Timer to ns2 Timer:

When using the ns2 resched() for timers, it expects a double type value as the input. Ns2 timers keep incrementing as a number from the start of the simulation. If your existing implementation uses linux timeval for its timer convert them to double value using the timeval2double and double2timeval routines listed below.

```
resched(timeval2double(linuxtimer_.value.it_value));

//Converts timeval structure to double
inline double timeval2double(struct timeval t_)
{
        double d;
        d=t_.tv_sec + (t_.tv_usec / 1e6);
        return d;
};

//Converts double to timeval structure
inline struct timeval double2timeval(double d)
{            struct timeval t_;
              t_.tv_sec=long(d);
              t_.tv_usec = long (fmod(d, 1)  * 1e6) ;
             return t_;
};
```

References:

[1] Jae Chung and Mark Claypool, "NS by Example," available at http://nile.wpi.edu/NS/.

[2] E. Altman, T. Jimenez, "NS Simulator for Beginners", Lecture Notes, Autumn 2002, Univ. de Los Andes, Merida, Venezuela, July 2003.

[3] Kevin Fall and Kannan Varadhan. The *ns* manual available at http://www.isi.edu/nsnam/ns/ns-documentation.html.

[4] NS-2 Trace Formats available at http://www.k-lug.org/%7Egriswold/NS2/ns2-trace-formats.html#wireless:old.

Code Listings:

Listing for MyRouter Agent Code:

```
/* File:  myrouter_agent.cc
Generic MANET Router ns2 Template Code
Author : Palan Annamalai ( palan AT vt d0t edu )

Virigina Tech Lab for Advanced Networking (VTLAN)

File: Routiness for NS_MyRouterAgent (myrouter_agent.cc)
Last Updated : August 5th, 2003

This file and the associated header file (myrouter_agent.[cc|h])
provide a glue framework for your existing linux based routing protocol
We ported our OSPFMCDS linux routing protocol to ns2 using this
glue framework. Hope it helps your efforts.

Salute to the ns2 Guruz
Knowledge is like free flowing river don't build a dam around it


Licence:
Copyright (c) 2003, Palan Annamalai
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions
are met:

    * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in
the documentation and/or other materials provided with the distribution.
    * Neither the name of the VTLAN nor the names of its contributors may be used to endorse or promote products derived from this
software without specific prior written permission.
```

```
-------------------

Parts of this program uses code from OLSR and ns2
Appropriate Licences Apply

OLSR ns2 implementation
Copyright (c) 2000, 2001 Lars Christensen
For full lincene refer OLSR-ns2 patch

*/

//Include - Header Files, Definitions
#include "myrouter_agent.h"
int hdr_MyRouter::offset_; //This definition is required bcoz we never instantiate hdr_MyRouter

// --== OTcl Linkage Interfaces ==--

// --== MyRouterAgent OTcl Linkage Class ==--
static class NS_MyRouterClass : public TclClass {
public:
        //The TclClass constuctor creates the OTcl class(interpreted class) for MyRouter under Agent
        // and inserts this object into the link list of Tcl Class objects
        NS_MyRouterClass() : TclClass("Agent/MyRouter") {} //The name a user will invoke in OTcl space  is Agent/MyRouter,
MyRouter is the protocol name
        // Create method is called when user invokes a new Agent/MyRouter
        TclObject *create(int argc, const char *const *argv) {
                return (new NS_MyRouterAgent ((nsaddr_t)atoi(argv[4])));
                //NS_MyRouterAgent is the actual class implementing the ns2 routing
        }
} NS_MyRouterClass_object; // This class object will get created as soon as ns2 starts


// --== MyRouter Header OTcl Linkage Class ==--
static class NS_MyRouter_PktHeaderClass : public PacketHeaderClass {
public:
    //Constructor informs the PacketHeader manager the size of the Packet and the variable (the variable expecting the position is
offset_ )
    // at which to return the offset number for this particular packet in the entire ns2 packet (refer to Bag of Bits in ns2 doc to
understand this)
        NS_MyRouter_PktHeaderClass() : PacketHeaderClass("PacketHeader/MyRouter",sizeof(hdr_MyRouter)) {
                bind_offset(&hdr_MyRouter::offset_);
        }
} NS_MyRouter_pkthdr_object;


// --== Utility functions ==--

double simulator_time() {
    return Scheduler::instance().clock();  // To get current Simulator Time
}

// --== MyRouter Implementation ==--

// Default Constructor
NS_MyRouterAgent::NS_MyRouterAgent(nsaddr_t id)
: Agent(PT_MyRTR), NS_MyRouter_Timer_object(this) //Constructor of NS_MyRotuerAgent has the packet of type PT_MyRTR
{
        myaddr_ = id; //Assign myaddr_ with the id that comes from the Tcl
        seq_num_ = int(Random::uniform(0,65535)); // If seq_num_ is 65535 then it wraps around
```

```
                //cout << "My node ID is : " << myaddr_ << endl ;
}


// --== Tcl Command Interface ==--
int NS_MyRouterAgent::command(int argc, const char *const *argv) {
                //cout << "Processing command " << argv[0] << argv[1] << endl ;
    if (command_is("id", 0, argc, argv)) {
        /* return the id of this node */
        Tcl &tcl = Tcl::instance();
        tcl.resultf("%d", myaddr_);
        return TCL_OK;

                } else if (command_is("start", 0, argc, argv)) {
                 // start simulation
                                Start();   // This start function should activate your routing protocol fill in with appropriate code
                                //cout << "Node issued start command" << endl ;
        return TCL_OK;

                } else if (command_is("index", 1, argc, argv)) {
        /* set the id of this router agent */
        myaddr_ = atoi(argv[2]);
        return TCL_OK;

                } else if (command_is("log-target", 1, argc, argv)
            || command_is("tracetarget", 1, argc, argv)) {
                //Checking to see if log-target is actually required
                //trace-target is required, default Agent::command will not handle this
       // set the trace object
       logtarget = (Trace*)TclObject::lookup(argv[2]);
       if (!logtarget)
           return TCL_ERROR;
       else
          return TCL_OK;
   }
   // let the superclass handle  the rest of the commands
   return Agent::command(argc, argv);
}

void NS_MyRouterAgent::Start() {
                // TODO: Add your functions to initialize your Agent
                NS_MyRouter_Timer_object.sched(0); //Schedule the first timer event immediately at 0 seconds
}


void NS_MyRouterAgent::recv(Packet *p, Handler *) {
   /* Access common header */
   struct hdr_cmn *ch = HDR_CMN(p);
   /*Access ip header */
   struct hdr_ip *ih = HDR_IP(p);
   /* Access MyRouter header */
   struct hdr_MyRouter *myrtr = HDR_MyRTR(p);

    //If I receive PT_MyRTR then it is from some other node
                if (ch->ptype() == PT_MyRTR) {
                ih->ttl_ -= 1; //Passed through router so ttl less 1
                u_int8_t *data = (u_int8_t*)p->accessdata(); //Accessdata gives hte pointer to the data in the packet
                int packet_size = p->datalen();
                                u_int16_t rx_seqno = myrtr->seqno_;
                                //TODO: Call your packet receive function here - to process the packet
                //MyRouter.recv(data,packet_size,ih->saddr(),rx_seqno);
                Packet::free(p); // Since no other agent needs to receive the packet free it
                return;
                }//End of if PT_MyRTR

                //If recevied packet is from myself and not yet forwarded - set TTL to MAX TTL
    if (ih->saddr() == myaddr_) {
       if (ch->num_forwards() == 0) {
           // ch->size() += IP_HDR_LEN; // --== Do not do this as destination node does not strip the additional bytes ==--
           ih->ttl_ = MAX_HOPS;
        } else { //already forwarded by someone could be loop - drop packet
            drop(p, DROP_RTR_ROUTE_LOOP);
                    return; // Return after drop
        }
    } else if (--ih->ttl_ == 0) {  // Source is not me and REDUCE TTL by 1
        drop(p, DROP_RTR_TTL); //TTL reached zero
        return;
    }

    //NOTE: TTL has been reduced by 1 if the packet was a multi-hop packet

    // Now Route the packet - based on local route table
    // TODO: Your routing class MyRouter should construct this table
    /*const routetable &rtable = MyRouter.getroutetable();
    routetable::const_iterator f_it = rtable.find(ih->daddr());
    if (f_it != rtable.end()) {
        forward(p, f_it->second.next_hop);
                */
                //TODO: For now route all packets to a fixed node
                u_int32_t next_hop=2;
                forward(p, next_hop);   //Multi-hop data packet received, forward it based on next hop from local routing table
    /*} else {
        drop(p, DROP_RTR_NO_ROUTE); //Route to destination not know
                return;
    }*/
}



void NS_MyRouterAgent::forward(Packet *p, addr_t nexthop) {
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);

    /* check ttl */
    if (ih->ttl_ == 0) {
        drop(p, DROP_RTR_TTL);
        return;
    }

    ch->prev_hop_ = myaddr_;
    ch->next_hop_ = nexthop;
    ch->addr_type() = NS_AF_INET; //NS_AF_INET is used for directed IP Packets
    ch->direction() = hdr_cmn::DOWN;

                // Schedule the packet for sending dont use Jitter in forward you dont want
                // to delay the packet any longer unless you want to show processing delay
    Scheduler::instance().schedule(target_, p, 0.0); //Schedule the packet for next agent to receive
    //Schedule is better way than target_->recv(p, (Handler*)0);
}

//Expire function gets called every time the timer schedule value expires
void NS_MyRouter_Timer::expire(Event *e) //Event e is dummy, to be used if your handler needs to handle multiple events
{
   unsigned char mydata[100];
   memset(mydata,'A',100); //For this dumy agent the MyRouter protocol data is just 100 A's
                //TODO: Instead of scheduling a sendpkt with this dummy data - fill your own packet data in the buffer
                //TODO: In your routing protocol to maintain various event timer - keep a timer function and set its subsequent expire
resched
   NS_MyRouterAgentPtr_->sendPkt(mydata,100); // For now dummy 100 bytes
   resched(10); //Assuming 10 seconds once you want to send routing control packets
   //cout << "Send Timer Expire " << endl ;
}

void NS_MyRouterAgent::sendPkt(unsigned char *pkt_buf, int pkt_len) {  //You can use the pkt_buf to set the data of the outgoing IP
packet
   Packet *p = allocpkt(pkt_len);

   // Fill common header
   struct hdr_cmn *ch = HDR_CMN(p);
   ch->ptype() = PT_MyRTR;
   ch->size() = IP_HDR_LEN + MyRTR_HDR_LEN + p->datalen();
   ch->iface() = -2;  //In ns2 -2 is unknow, -1 reserved for broadcast
   ch->error() = 0;
   ch->addr_type() = NS_AF_NONE; //NS_AF_NONE is used for broadcast packets and NS_AF_INET is used for directed IP Packets
   ch->prev_hop_ = myaddr_;


   // Fill IP header
   struct hdr_ip *ih = HDR_IP(p);
   ih->saddr() = myaddr_;
   ih->sport() = RT_PORT;
   ih->daddr() = IP_BROADCAST; //Since Routing Send for everyone
   ih->dport() = RT_PORT; //Note: This port is exclusive for Routing
   ih->ttl_ = MAX_HOPS;

   // Fill our Routing protocol MyRouter packet header
   struct hdr_MyRouter *myrtr = HDR_MyRTR(p);
   myrtr->seqno_ = seq_num_++;  // If you want to be sure on any platform it wraps then you can modulo this value by 65535 (16 bit
size)

   //Palan - pkt->accessdata is actually the old method try to use setdata; accessdata is provided only for backward comptability only
   // In this case - p->accessdata works better
   memcpy(p->accessdata(), pkt_buf, pkt_len); // Copy over our protocol packet buffer to the newly created ns2 packet as its data
payload
   /* The setdata method is as shown below
   PacketData* pktdata = new PacketData(pkt_len); // PacketData class is derived from AppData with Application type
PACKET_DATA
   memcpy(pktdata->data(), pkt_buf, pkt_len);
   p->setdata(pktdata); // setdata takes a value of an AppData object but since PACKET_DATA is the only Packet type we need - we
can use a PacketData Object
   */

    double Jitter_value=0.2;
    //Use a random jitter to avoid simultaneous packet send by many MyRouter Agents -- this reduces Collisions
    if (Jitter_value > 0) {
        Scheduler::instance().schedule(target_, p, Random::uniform(Jitter_value));
    } else {
                Scheduler::instance().schedule(target_, p, 0);
                //Schedule is better way than target_->recv(p, (Handler*)0);
    }
   // Note - you should not free the packet here if you do no other agent can receive it - the packet will be destroyed
}
```

Listing for MyRouter Agent Header File:

```
/* File:  myrouter_agent.h
Generic MANET Router ns2 Template Code
Author : Palan Annamalai ( palan AT vt d0t edu )

Virigina Tech Lab for Advanced Networking (VTLAN)

File: header for myrouter_agent (myrouter_agent.h)
Last Updated : August 4th, 2003

Salute to the ns2 Guruz
Knowledge is like free flowing river don't build a dam around it

For licence info use myrouter_agent.cc
*/

#ifndef NS_MyRouter_AGENT
#define NS_MyRouter_AGENT

#include <timer-handler.h>
#include <cmu-trace.h>
#include <packet.h>
#include <agent.h>
#include <tclcl.h>
#include <address.h>
#include <ip.h>
#include <string>
#include <iostream>
#include <random.h>
#define MAX_HOPS 255   // TTL Value set by your routing protocols
typedef u_int32_t addr_t;
#define HDR_MyRTR(p)   ((struct hdr_MyRouter*)hdr_MyRouter::access(p))   // We can use HDT_MyRouter(p) to access the fields
in our packet header eg. seqno
#define MyRTR_HDR_LEN  2    // Size of the fields in our MyRouter packet

// MyRouter Packet Header Structure Definition - MyRouter control packets are of this format
struct hdr_MyRouter {
                u_int16_t seqno_; // Lets use a seqno for our routing protocol - this is now a member of the ns2 MyRouter packet
                                // TODO: Put your own Packet header variables here - You can also leave this empty and just copy your entire
                                // Router agent packet buffer as data to the ns2 MyRouter packet
                // Ns2 Specific Packet header access functions - Should be present in all Packet Header Definitions
        static int offset_;  // Offset specifies the location of the packet in the Bag of Bits (BoB) refer ns2 doc for more info on this
        inline static int& offset() { return offset_; }
        inline static hdr_MyRouter* access(const Packet* p) {
                return (hdr_MyRouter*) p->access(offset_);
        }
};

class NS_MyRouterAgent;  // Have to define class name earlier since we are going to use in the Timer Class definitions

//Inheriting from TimerHandler is the better way to use Timer,
//TimerHandler is derived from Handler
//Advantage of using inheriting from TimerHandler is it supports checking Timer pending status
class NS_MyRouter_Timer : public TimerHandler {
 public:
                NS_MyRouter_Timer(NS_MyRouterAgent *a) : TimerHandler(), NS_MyRouterAgentPtr_(a)  //Default Constructor
intializes the NS_MyRouterAgent Pointer and Calls base class constructor TimerHandle()
                { /*cout << "Initialized Timer " << endl; */ }
                void expire(Event *e);   //Expire function is called every time the Timer expires
 protected:
                NS_MyRouterAgent *NS_MyRouterAgentPtr_;   // Ptr to the NS MyRouterAgent that called this timer - we can use this
ptr to access the Agents members
};

// NS_MyRouterAgent Implementation Class Definition
class NS_MyRouterAgent : public Agent
{
 public:
		// TODO: Instantiate your existing linux Routing Agent full class here ..... eg. MyRouter (note: NOT NS_)
		NS_MyRouterAgent(nsaddr_t id); //The default constructor

		void recv(Packet *p, Handler *);			// Ns2 Packet receive
		void forward(Packet *p, addr_t nexthop); // Ns2 Packet forward
		void sendPkt(unsigned char *, int); //This function should have the packet data buffer passed to it and the data len
		void Start();   // Function to implement starting of your Routing Agent
		int myaddr_;		// node id
		u_int16_t seq_num_;
		Trace *logtarget; // routing table logging
 protected:
		// TCL Interface
		int command(int, const char *const *);
		NS_MyRouter_Timer NS_MyRouter_Timer_object; //friend class is not needed

		/* check if the arg[cv] matches a specific command and a given number
		of additional arguments. returns 1 if it is <command> with
		<nargs>. */
		int command_is(const char *command, int nargs, int argc, const char *const *argv) {
				return (argc==2+nargs) && (strcmp(command, argv[1]) == 0);
}
};


#endif  //End of Define NS_MyRouter_AGENT
```

Listing for Validation Code:

```
# File:  validate_MyRouter.tcl
# Copyright (c) 1997 Regents of the University of California.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
# 1. Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
# 2. Redistributions in binary form must reproduce the above copyright
#    notice, this list of conditions and the following disclaimer in the
#    documentation and/or other materials provided with the distribution.
# 3. All advertising materials mentioning features or use of this software
#    must display the following acknowledgement:
#      This product includes software developed by the Computer Systems
#      Engineering Group at Lawrence Berkeley Laboratory.
# 4. Neither the name of the University nor of the Laboratory may be used
#    to endorse or promote products derived from this software without
#    specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
# ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
# ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
# FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
# DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
# OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
# HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
# OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# Adapted from simple-wireless.tcl
# A simple example for wireless simulation for validating MyRouter Dummy Router Agent
# Palan, VTLAN < palan AT vt d0t edu >

# ======================================================================
# 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)             4                          ;# number of mobilenodes
set val(rp)             MyRouter                   ;# routing protocol

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

#use radio*(1+10%) to calculate it, 25 m for Recving and 28m for Censing
#used threshod -m Tworayground 25
Phy/WirelessPhy set CSThresh_ 2.45253e-07
Phy/WirelessPhy set RXThresh_ 3.07645e-07


#
# Initialize Global Variables
#
set ns_      [new Simulator]
set tracefd     [open MyRouter.tr w]
```

```
$ns_ trace-all $tracefd

set namtrace [open MyRouter.nam w]
$ns_ namtrace-all-wireless $namtrace 100 100

# set up topography object
set topo       [new Topography]

$topo load_flatgrid 100 100
#Grid is only top quadrant

#
# 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)

#New way of attaching channels - Palan
set chan_1_ [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_1_ \
                         -topoInstance $topo \
                         -agentTrace ON \
                         -routerTrace ON \
                         -macTrace OFF \
                         -movementTrace ON

                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_ 20.0
$node_(0) set Y_ 50.0
$node_(0) set Z_ 0

$node_(2) set X_ 39.0
$node_(2) set Y_ 90.0
$node_(2) set Z_ 0

$node_(3) set X_ 39.0
$node_(3) set Y_ 50.0
$node_(3) set Z_ 0

$node_(1) set X_ 58.0
$node_(1) set Y_ 50.0
$node_(1) set Z_ 0.0

#node_(3) moves away from the nodes at 50 seconds and node_(2) takes it places

$ns_ at 0.0 "$node_(0) setdest 20.0 50.0 1.0"
$ns_ at 0.0 "$node_(2) setdest 39.0 90.0 1.0"
$ns_ at 0.0 "$node_(3) setdest 39.0 50.0 1.0"
```

```
$ns_ at 0.0 "$node_(1) setdest 58.0 50.0 1.0"

###Very critical X has to be different from the previous one otherwise node wont move in nam

# Node_(1) -  node_(0) link  breaks and forms. When routing works flow CBR from 0 to 1
$ns_ at 50.0 "$node_(3) setdest 40.0 10.0 35.0"
$ns_ at 70.0 "$node_(2) setdest 40.0 50.0 30.0"


# Set CBR (constant bit-rate traffic) generator at node 0 to 2
set udp_(0) [new Agent/UDP]
$ns_ attach-agent $node_(0) $udp_(0)
set cbr_(0) [new Application/Traffic/CBR]
$cbr_(0) set packetSize_ 1000
$cbr_(0) set interval_ 1
$cbr_(0) attach-agent $udp_(0)

set null_(0) [new Agent/Null]
$ns_ attach-agent $node_(2) $null_(0)

$ns_ connect $udp_(0) $null_(0)
$ns_ at 1.0 "$cbr_(0) start"
$ns_ at 150.0 "$cbr_(0) stop"


# 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
# $ns_ at 1.0 "$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 150.0 "stop"
$ns_ at 150.01 "puts \"NS EXITING...\" ; $ns_ halt"
proc stop {} {
   global ns_ tracefd
   $ns_ flush-trace
   close $tracefd
}

puts "Starting Simulation..."
$ns_ run

##### End of validate_MyRouter.tcl  #####
```

Appendix B

ns2 Scenario Trace to GNU Plot Converter Tool

```
#!/usr/bin/perl -w

# ns2scn2plot.pl
# ns2 scenario file to time, X, Y plotter
# Author: Palan Annamalai (palan@vt.edu)
# Licence: Same licence as found in MyRouter agent ( see elsewhere in this document )
#
# Usage ns2tr2plot.pl <ns2scenariofile> <node>
# BonnMotion and similar tools let you create a ns2 scenario file
# It would be useful to see the node placements graphically
# the script will generate a csv file of all the co-ordinates
# it is trival then to use CUT TO EXTRACT THE COLUMNS for GNUPLOT eg:
# cat testTr2plot.txt.plotable.wri | cut -f 3,4,2  -d ',' > ns2nodemnt.txt
# gnuplot> splot "ns2nodemnt.txt" with linespoint
#
# Limitations: NO Z co-ordinates from ns2 but it is easy to just one more like $val[5] and shift the indexes
of val


my $version = "1.01, 8/29/2005";
my $X=9999999;
my $Y=9999999;


# Set these values based on ur need
$PRINTSPEED=0; # Make 0 if you do not want to print speed
$intial_node_pos_only=0; # Make 1 if you ONLY want to print the intial location of the nodes
$NODE=0;          # -1 for all nodes or any node number to print that nodes movements

open(INPUT, $ARGV[0]) || die "Couldn't open input ns2 trace file";
open(OUTPUT,"> $ARGV[0].plotable.wri") || die "Couldn't open output plot file";

print "Outfile in Format\nNode, Time, X, Y, Speed\n";
while (<INPUT>)
{
  $line=$_;
  $line =~ s/\r//; #If line was generated in windows
  chomp($line);

  if ( $line =~ m/^\$ns.*at.*setdest.*/i )
        {
                if(!$intial_node_pos_only) {   # Do not print this if only intial node position is requested
                        @val =  $line =~
m/^\$ns.*at\s+(\d+.\d+).*node.*\((\d+)\)\s+setdest\s+(\d+.\d+)\s+(\d+.\d+)\s+(\d+.\d+).*/i; # Note the () is
impo it maps the result in  LIST context to  $tm if you dont give () in the match it will map the scalar
context to @val
                        #$val[1] = node id, $val[0] = time, $val[2] = x, $val[3] = y, $val[4] = speed
                        if ( $val[1] == $NODE  || $NODE == -1 ) { # -1 for all nodes
                                print OUTPUT "$val[1], $val[0], $val[2], $val[3]";
```

```
                                        print OUTPUT ", $val[4]\n" if($PRINTSPEED);
                                        print OUTPUT "\n" if(!$PRINTSPEED);
                              }
                    }
          }
  elsif ( $line =~ m/^\$node.*\s+set\s+[X|Y].*/i )
          {
                    @val = $line =~ m/^\$node.*\((\d+)\)\s+set\s+([X|Y])_\s+(\d+.\d+)/i;
                    #$val[0] = node id, $val[1] = X|Y, $val[2] = number
                    if ( $val[0] == $NODE || $NODE == -1 ) {   # -1 for all nodes
                              $X = $val[2] if($val[1] =~ "X" );
                              $Y = $val[2] if($val[1] =~ "Y" );
                              if ($X != 9999999 && $Y != 9999999 )
                                        {print OUTPUT "$val[0], 0.0, $X, $Y\n";
                                         $X = 9999999; $Y = 9999999; # Now that we have printed it reset it
for the next node
                               }
                    }
           }
  else
  { next; }           #Skip if first line not a ns command setdest command  or set X|Y command
}

close(INPUT);
close(OUTPUT);

print "Done! Output is in : $ARGV[0].plotable.wri\n"
```

Appendix C

ns2 Scenario Trace to NAM Animation Converter Tool

```
#!/usr/bin/perl -w

# Program : ns2scn2nam.pl
# ns2 scenario file to nam animation file convertor
# Author: Palan Annamalai (palan@vt.edu)
# Licence: Same licence as found in MyRouter agent ( see elsewhere in this document )
# Option -t 10000 sets play back rate to 10s : default 5seconds
# Option -s 40 sets size to 40 : default 30
# Option -x 300 setx X-axis to 300
# Option -y 400 setx Y-axis to 400

# # ns2 sceanrio file
# #
# # Provide initial (X,Y, for now Z=0) co-ordinates for mobilenodes
# #
# #
# $node_(0) set X_ 20.0
# $node_(0) set Y_ 50.0
# $node_(0) set Z_ 0

# $node_(2) set X_ 39.0
# $node_(2) set Y_ 90.0
# $node_(2) set Z_ 0

# $node_(3) set X_ 39.0
# $node_(3) set Y_ 50.0
# $node_(3) set Z_ 0

# $node_(1) set X_ 58.0
# $node_(1) set Y_ 50.0
# $node_(1) set Z_ 0.0

# #node_(3) moves away from the node
# ###Very critical X has to be different from the previous one otherwise node wont move in nam

# # Node_(1) then starts to move away from node_(0)
# $ns_ at 50.0 "$node_(3) setdest 40.0 10.0 35.0"
# $ns_ at 70.0 "$node_(2) setdest 40.0 50.0 30.0"

# >>>> We NEED to convert the above to the below

# File: test_movement.nam
# V -t * -v 1.0a5 -a 0
# W -t * -x 500 -y 500
# n -t 0.000000 -s 0 -x 20.000000 -y 50.000000 -U 0.000000 -V 0.000000 -T 0.000000
# n -t 0.000000 -s 2 -x 39.000000 -y 90.000000 -U 0.000000 -V 0.000000 -T 0.000000
# n -t 0.000000 -s 3 -x 39.000000 -y 50.000000 -U 0.000000 -V 0.000000 -T 0.000000
# n -t 0.000000 -s 1 -x 58.000000 -y 50.000000 -U 0.000000 -V 0.000000 -T 0.000000
# n -t 50.000000 -s 3 -x 39.000000 -y 50.000000 -U 0.874727 -V -34.989068 -T 1.143214
# n -t 70.000000 -s 2 -x 39.000000 -y 90.000000 -U 0.749766 -V -29.990629 -T 1.333750
# T -t 71.000000
# ~


my $gridX=50;
my $gridY=50;
my $X=9999999;
my $Y=9999999;
my @nodes;
my $MaxGridX=0;
my $MaxGridY=0;
$node_size = 30;   # Change this to change the node size in the animation
$play_back_rate = 5000; # Rate in ms

use Getopt::Std;
my %Options;
getopts('x:y:s:t:', \%Options);

$play_back_rate = $Options{t} if ($Options{t});   # Option -t 10000 sets play back rate to 10s : default
5seconds
$node_size = $Options{s} if ($Options{s}); # Option -s 40 sets size to 40 : default 30
$gridX=$Options{x} if ($Options{x}); # Option -x 300 setx X-axis to 300
$gridY=$Options{y} if ($Options{y}); # Option -y 400 setx Y-axis to 400

# Set Endline based on OS the script is running on
if($^O =~ /MS/i ) { $endl="\n"; } else { $endl="\r"; }

open(INPUT, $ARGV[0]) || die "Couldn't open input ns2 Scenario file";

while (<INPUT>)
{
  $line=$_;
  $line =~ s/\r//; #If line was generated in windows
  chomp($line);

  if ( $line =~ m/^\$ns.*at.*setdest.*/i )
        {
                @val =  $line =~
m/^\$ns.*at\s+(\d+.\d+)\s+\"\$node.*\((\d+)\)\s+setdest\s+(\d+.\d+)\s+(\d+.\d+)\s+(\d+.\d+)\"/i; # Note the
() is impo it maps the result in  LIST context to  $tm if you dont give () in the match it will map the scalar
context to @val

                #$val[1] = node id, $val[0] = time, $val[2] = x, $val[3] = y, $val[4] = speed
                @tmp=[$val[0], $val[1], $val[2], $val[3], $val[4]];
                $MaxGridX=$val[2] if ($val[2] >= $MaxGridX); # Max Grid X Value
                $MaxGridY=$val[3] if ($val[3] >= $MaxGridY); # Max Grid Y Value
                push @nodes,@tmp;
        }
  elsif ( $line =~ m/^\$node.*\s+set\s+[X|Y].*/i )
        {
                @val = $line =~ m/^\$node.*\((\d+)\)\s+set\s+([X|Y])_\s+(\d+.\d+)/i;
                #$val[0] = node id, $val[1] = X|Y, $val[2] = co-ordinate
                $X = $val[2] if($val[1] =~ "X" );
                $Y = $val[2] if($val[1] =~ "Y" );
                        if ($X != 9999999 && $Y != 9999999 )
```

```
                                        {
                                        @tmp=[0.0, $val[0], $X, $Y, 0.0];   # Time 0.0 and Speed 0.0 since
this is initial Node placement
                                        push @nodes, @tmp;
                                        $MaxGridX=$X if ($X >= $MaxGridX); # Max Grid X Value
                                        $MaxGridY=$Y if ($Y >= $MaxGridY); # Max Grid Y Value
                                        $X = 9999999; $Y = 9999999; # Now that we have printed it reset it
for the next node
                                 }
          }
}

# Automatic detection of Grid size if greater than grid X,Y defined.
$gridX=int($MaxGridX/10) * 10 if ($MaxGridX >= $gridX);
$gridY=int($MaxGridY/10) * 10 if ($MaxGridY >= $gridY);

# Sort the nodes array by time
for $list_ref ( sort { $a->[0] <=> $b->[0] } @nodes ) { #Sort on time
          push @time_sorted_nodes, [@$list_ref]; # Array context is required here
}

# Automatic detection of animation playback rate - animation will plack bay at atleast 1/100the the rate of
the full time period
$ending_time=($time_sorted_nodes[$#time_sorted_nodes]->[0]); #Last value of Time in the sorted time
array

# Start printing the nam file
print "V -t * -v 1.0a5 -a 0$endl";
print "W -t * -x $gridX -y $gridY$endl";
print "v -t 0.0 -e set_rate_ext ".$play_back_rate."ms 1$endl";

# Compute Velocity for X,Y direction
for $i ( 0 .. $#time_sorted_nodes ) {
          @val=@{$time_sorted_nodes[$i]};
          $node=$val[1];$X=$val[2]; $Y=$val[3]; $speed=$val[4];
          $Xvelocity=0;     $Yvelocity=0;
          if( $speed != 0 )   # If speed in not 0 then calculate Velocity along X and Y directions
          {
                    $X2[$node]=$X;   #If this node was noticed in the scenario file before then X1, Y1
should be filled
                    $Y2[$node]=$Y;
                    $hypo_distance=sqrt(($X2[$node]-$X1[$node])**2+($Y2[$node]-$Y1[$node])**2); #
find hypotenuse of right angle triangle
                    $time_to_travel=$hypo_distance/$speed;
                    if( $time_to_travel != 0) {
                              $Xvelocity=($X2[$node]-$X1[$node])/$time_to_travel;    # Velocity is distance
/ time with direction indicated by + or -
                              $Yvelocity=($Y2[$node]-$Y1[$node])/$time_to_travel;
                    }
                    print  "n -t $val[0] -s $node -x $X1[$node] -y $Y1[$node] -U $Xvelocity -V $Yvelocity -
T $time_to_travel$endl"; # Node has to start from here X1,Y1 at velocity U,V by time_to_travel;
          }
          $X1[$node]=$X; # Current Position for node becomes its X1, Y1
          $Y1[$node]=$Y;
   if( $val[4] == 0 )    # If the speed is 0 then nothing to be done leave the co-orindates as such (inital case)
          {
                print  "n -t 0.0000 -s $node -x $X -y $Y -U 0.000000 -z $node_size -V 0.000000 -T
0.000000$endl";
        }
}

$ending_time=$val[0]+2;
print "T -t $ending_time$endl"; #If this is not given the last node wont move

# If you want to assign the modified array back to the sorted nodes array use the following eg.
#$val[4] = $time_to_travel;
#$time_sorted_nodes[$i]=[ @val ];
```

Appendix D

ns2 Wireless Trace Performance Analysis Tool

```
#!/usr/bin/perl

# analyze.pl
# Analyze an ns2 trace file
# John Wells
# Modified by Tao Lin (taolin@vt.edu)
# This file has checks for MAC layer
# This file does not check RTR layer
# So please change "MAC" to "RTR" in necessary places if you want to
# collect data at RTR layer instead of Routing Agent layer

# Modified by Palan

use Getopt::Long;

#
# User options
$dataType = "cbr";

################################################################################
# Start of code
################################################################################
sub help_info() {
    print "$0 <-n node> [OPTIONS..] file..\n";
    print "  -n, --node=NUMBER                Node to analyze (If you want application throughput use the node
ID of the reciving agent\n";
    print "  -i, --interval=NUMBER:NUMBER     Only consider this time interval\n";
    print "  -p, --print                      Print trace file lines\n";
    print "Note: overhead is taken to be everything but data traffic\n";
    exit;
    }

sub ProcessArgs(@args) {
  &GetOptions("n|node=i", \$node, "i|interval=s", \$intervalStr,
    "p|print" => \$print);

  if ("$node" eq "") {
    $node = ".*";
  }

  if ($intervalStr ne "") { @interval = split /:/, $intervalStr; }
  else                    { @interval = (0, 1000000); }
}

sub PrintStats {
  print "---------------------------------\n";
  if("$node" eq ".*"){
            print "Statistics across interval ($interval[0], $interval[1])\n";
  }
  else {
```

```
            print "Statistics for node $node across interval ($interval[0], $interval[1])\n";
  }
  print "Note: Only count MANET at MAC layer.\n";                         # Palan  - only checks for MANET
traffic not the DATA traffic (DATA is commented out)
  print "      Only count received(r) Data at AGT layer.\n";
  print "MANET (f or s): $manet[0] packets, $manet[1] bytes\n";

  print "Recv Data Pkts: $recvdata[0] , Bytes $recvdata[1]\n";
  $t=$udp[0] * 8/1000;
  print "Total sent Data: $t kB\n";
  $appthroughput = $recvdata[1]/($interval[1] - $interval[0]) * 8/1000;
  $capacity_MANET = $manet[1] /($interval[1] - $interval[0]) / 1000; # MAC overhead per packet
$manet[0] * 52
  print "Average Hop Count : $recv_pkt_hop_count\n";
  print "End-to-end delay of application : $delay second\n";
  print "Throughput of Node (application) : $appthroughput kbps (total size / interval time)\n";
  $percent_throughput = ($udp[1]/$udp[0]) * 100;
  print "Delievery Percentage at Node : $percent_throughput %\n";
  print "Capacity consumed by MANET RP (MAC layer) : $capacity_MANET KB/s\n";
  print "Last line of trace file : $lastline \n";
  # Throughput calculation for recvdata was added later on August 8/18
}


sub main {
  #@data = (0, 0);
  @udp = (0, 0);
  @manet = (0, 0);
  #@total = (0, 0);
  @recvdata=(0,0);
  @send_pkt_at_agt;
  @recv_pkt_at_agt;
  $recv_pkt_hop_count=0;
  $no_send_at_agt=0;
  $no_recv_at_agt=0;
  @dataTypes = ("cbr", "ftp");
  @manetTypes = ("DSR", "AODV", "DSDV", "TORA", "OSPFMCDS", "AODVUU", "message",
"OLSR");

  ProcessArgs();

  $delay=-1.0;

  while (<>) {
    @line = split /\ /;
    $lastline=$_; # Palan - Store the last line sometimes the simulation stops and trace file needs to be
removed for space purposes hence good to know last line of trace file for Verification

    if("$node" eq ".*"){

              #s 53.000000000 _33_ AGT  --- 2355 cbr 1000 [0 0 0 0] ------- [33:0 29:0 32 0] [3] 0 0
              #0 1                                   2             3           4 5           6             7         8   9 10 11 12
13     14  15  16 17 18 19 20
              #Palan - note two spaes between node and agent
              # Palan : From CMUTrace::format_rtp(Packet *p, int offset) - the 19th field is the number of
forwards for that packet filled from the CMN ns2 header
```

```
              #
   # Count all nodes
           if (($line[0] eq "r") &&
        ($interval[0] <= $line[1] && $line[1] <= $interval[1]) &&
        $line[3] eq "AGT"){

                     if(!$send_pkt_at_agt[$line[6]] eq ""){
                              $recvdata[0]++;                      # Number of Pakcets received at Agent
                              $recvdata[1]+=$line[8]; # Bytes received at agent
                              $udp[1] += $line[8]; #Size of received packets
                              $recv_pkt_at_agt[$line[6]]=$line[1]; # Storing the time of the receivied packet
to track the source the delay
                              $recv_pkt_hop_count = $recv_pkt_hop_count + $line[19];  #Global Hop Count
- this is correct bcoz CBR trace contains this info on the 19th field.
                              $delay=$delay+$recv_pkt_at_agt[$line[6]] - $send_pkt_at_agt[$line[6]];
           # Global Delay
                              $no_recv_at_agt++;   # Number of Packets received at agent
                              if($recv_pkt_at_agt[$line[6]] - $send_pkt_at_agt[$line[6]] < 0) {   #If delay is
less than 0 something is obviously wrong - Good check
                                       print;
                                       print "$line[0] | $line[1] | $line[2] | $line[3] | $line[4] | $line[5] |
$line[6] | error!\n";
                                       print "$recv_pkt_at_agt[$line[6]] - $send_pkt_at_agt[$line[6]]  ";
                                       $d = $recv_pkt_at_agt[$line[6]] - $send_pkt_at_agt[$line[6]];
                                       print "$d\n";
                                       exit;
                              }
                              if($recv_pkt_at_agt[$line[6]] - $send_pkt_at_agt[$line[6]] > 10) { # If delay is
more than 10 seconds mark the packet as seen in OLSR_car_seed_10_rnge_159_1UDPFlow result
                                       print;
                                       $d = $recv_pkt_at_agt[$line[6]] - $send_pkt_at_agt[$line[6]];
                                       print "Pkt $line[6]  Delay $d      Sent $send_pkt_at_agt[$line[6]]
Received $recv_pkt_at_agt[$line[6]]\n";
                              }


                     }
           }
           if (($line[0] eq "s") &&
        ($interval[0] <= $line[1] && $line[1] <= $interval[1]) &&
        $line[3] eq "AGT"){   #If the packet is from the SOURCE
                     $send_pkt_at_agt[$line[6]]=$line[1];  # Starting time at source for delay calculations
                     $no_send_at_agt++;                    # Number of Packets sent by agent this is easy to
deduce as we know the number of seconds the simulation is run (the interval) this is not reported or used
anywhere
                     $udp[0] += $line[8];
           }
           # Palan - In TBRPF trace there is a sndUPD packet - but from tbrpf_route.c this is not a real
packet it is just a DEBUG information - so need to count it in MANET
           if (($line[0] eq "f" || $line[0] eq "s") &&
             ($interval[0] <= $line[1] && $line[1] <= $interval[1]) &&
             $line[3] eq "MAC"){
                     if  (grep(/^$line[7]$/, @dataTypes)){
                              #$data[0]++; $data[1] += $line[8];
                     }
                     elsif (grep(/^$line[7]$/, @manetTypes)){    # If packet is forwarded or sent at the MAC
layer and is of type MANET record it
```

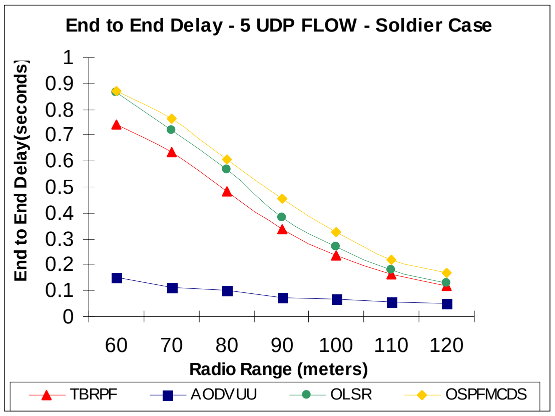
```
                              $manet[0]++; $manet[1] += $line[8];                    #manet[1] is the size of
the MANET packet and manet[0] is the number of MANET packets
                        }
                        #$total[0]++;
                        #$total[1] += $line[8];
                        if ($print) {
                              print;
                        }
                  }
            }
            else{

        # Count one node
                  if (($line[0] eq "r") &&
                    ($interval[0] <= $line[1] && $line[1] <= $interval[1]) &&
                    "_".$node."_" eq $line[2] && $line[3] eq "AGT"){
                        $recvdata[0]++;
                        $recvdata[1]+=$line[8];
                        $recv_pkt_at_agt[$line[5]]=$line[1];
                        $delay=$delay+$recv_pkt_at_agt[$line[5]] - $send_pkt_at_agt[$line[5]];
                        $recv_pkt_hop_count = $recv_pkt_hop_count + $line[19];
                  }

                  if (($line[0] eq "s") &&
                    ($interval[0] <= $line[1] && $line[1] <= $interval[1]) &&
                    "_".$node."_" eq $line[2] && $line[3] eq "AGT"){
                        $send_pkt_at_agt[$line[5]]=$line[1];
                  }

                  if (($line[0] eq "f" || $line[0] eq "s") &&
                    ($interval[0] <= $line[1] && $line[1] <= $interval[1]) &&
                    "_".$node."_" eq $line[2] && $line[3] eq "MAC"){
                        if  (grep(/^$line[7]$/, @dataTypes)){
                              #$data[0]++; $data[1] += $line[8];
                        }
                        elsif (grep(/^$line[7]$/, @manetTypes)){
                              $manet[0]++; $manet[1] += $line[8];
                        }
                        #$total[0]++;
                        #$total[1] += $line[8];
                        if ($print) {
                              print;
                        }
                  }
            }#end of if...else
        }#end of while


        if( $no_recv_at_agt != 0){
                  $delay=$delay / $no_recv_at_agt;
                  $recv_pkt_hop_count = $recv_pkt_hop_count /  $no_recv_at_agt;
        }
        else{
                  print "Overall delay is $delay second, there is no recv event at AGT\n";
                  #exit;
        }
```

```
  #print "$delay\n";
  #print "$no_recv_at_agt $no_send_at_agt\n";

  # If interval not given, then make it up
  if ($intervalStr eq "") {
    $interval[1] = $line[1];
  }

  PrintStats;
}


Main

# End of Code #
```

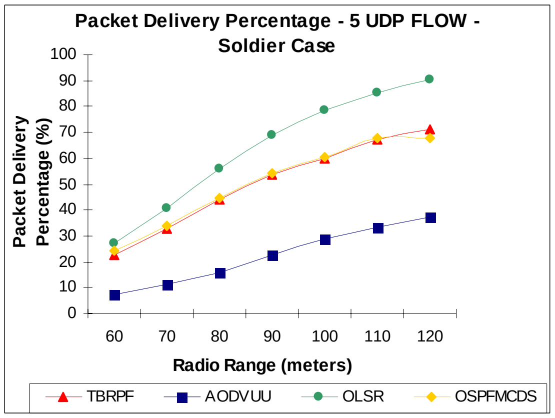
Appendix E

Additional Graphs

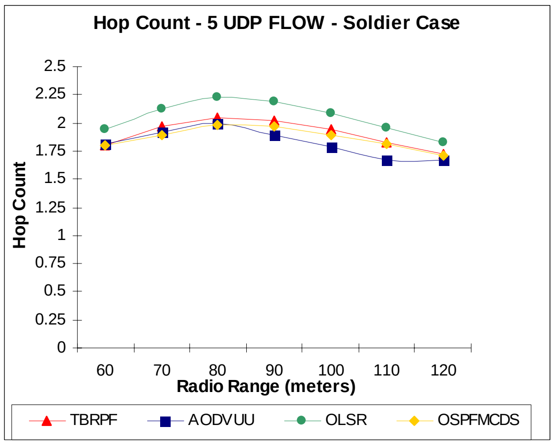
Small Scenario – Low Speed – Soldier Case



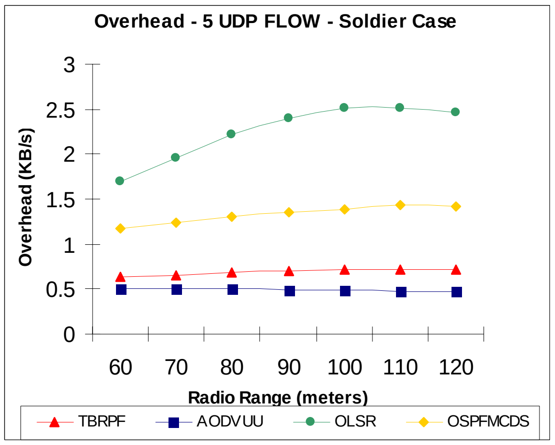
(a) Average End-to-End Delay



(b) Packet Delivery Percentage



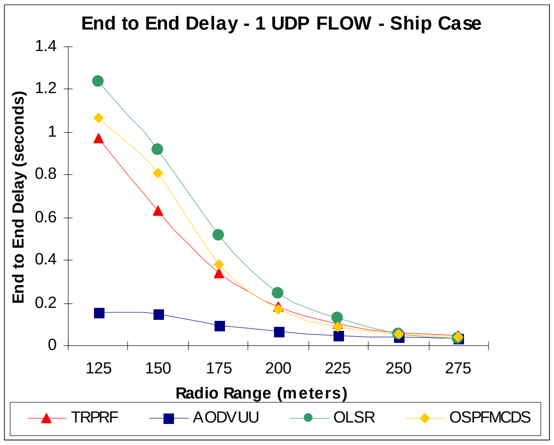
(c) Average Hop Count



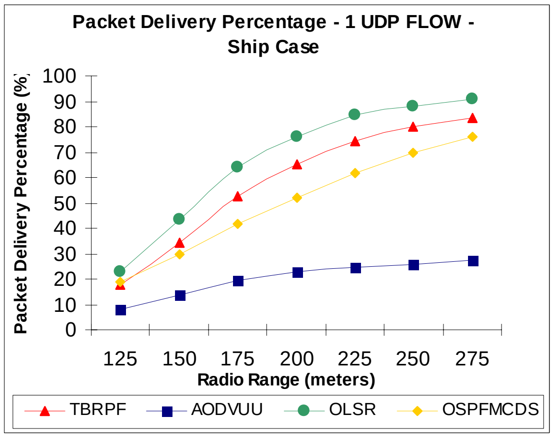
(d) Overall Overhead

Relative Performance Comparison - Small Scenario – Soldier Case - 5 CBR Flows

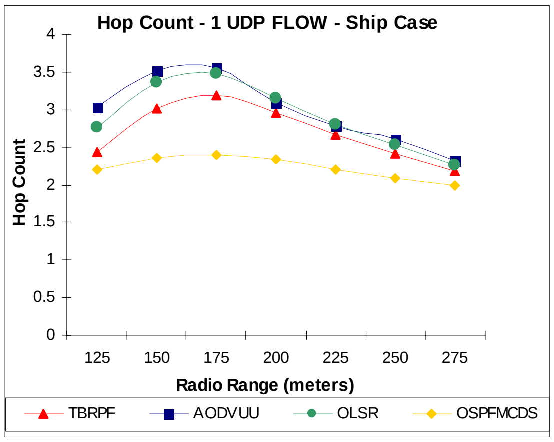
Medium Scenario – Medium Speed – Ship Case



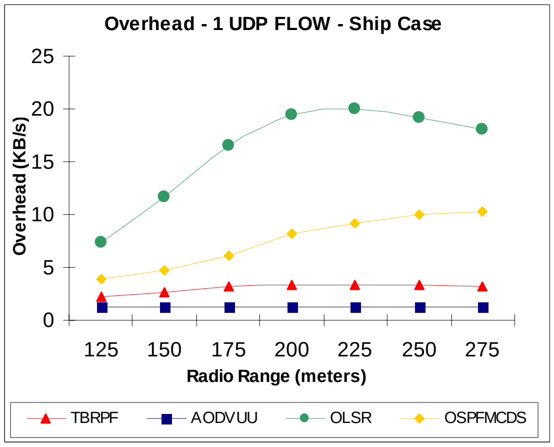
(a) Average End-to-End Delay



(b) Packet Delivery Percentage



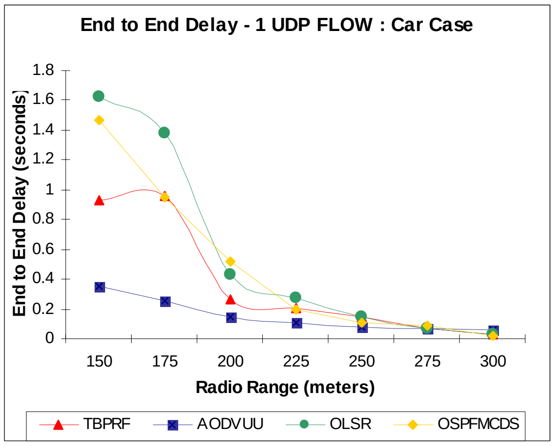
(c) Average Hop Count



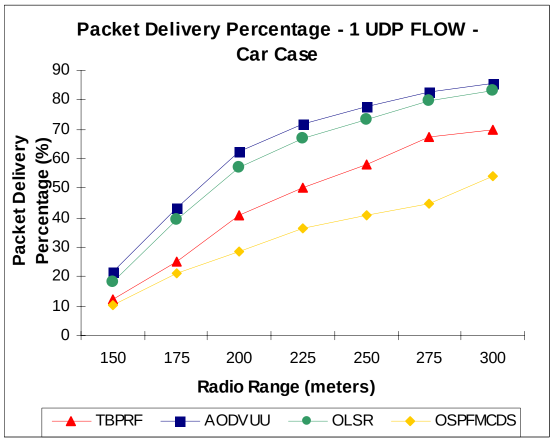
(d) Overall Overhead

Relative Performance Comparison – Medium Scenario – Ship Case - 1 CBR Flow

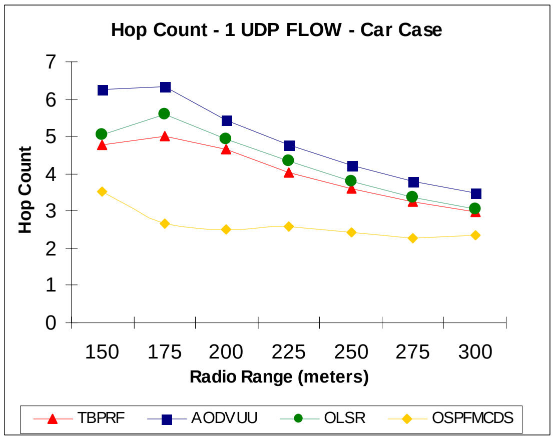
Large Scenario – Higher Speed – Car Case



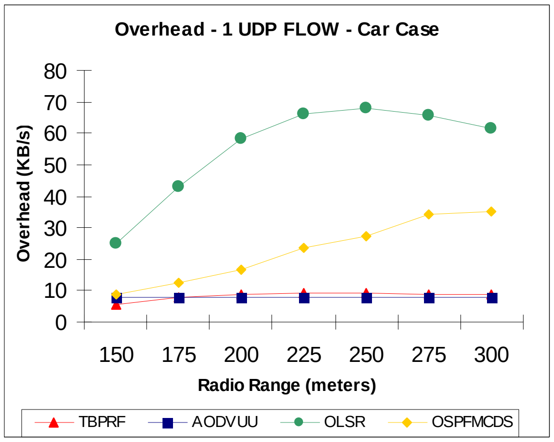
(a) Average End-to-End Delay



(b) Packet Delivery Percentage



(c) Average Hop Count



(d) Overall Overhead

Relative Performance Comparison - Large Scenario – Car Case – 1 CBR Flow

Appendix F

Tables of Confidence Intervals

Soldier Case

1 UDP Flow	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit
TBRPF												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.8690807	1.1186098	0.6195516	23.305403	28.974763	17.636042	1.8249558	2.0316297	1.6182819	0.6327393	0.6375	0.6279785
70	0.6092186	0.7648559	0.4535812	36.246248	42.220397	30.272099	2.0093579	2.2695097	1.7492061	0.6588226	0.6650404	0.6526048
80	0.424439	0.4770261	0.371852	50.862931	56.729496	44.996367	2.147061	2.3754949	1.9186271	0.6847623	0.6905127	0.6790118
90	0.2701541	0.3255807	0.2147275	64.182091	69.591264	58.772918	2.1341247	2.3488627	1.9193867	0.7043126	0.7085813	0.7000439
100	0.1854598	0.219508	0.1514117	74.934342	79.313804	70.55488	2.010433	2.2242317	1.7966342	0.7167144	0.7210641	0.7123647
110	0.1087983	0.1235904	0.0940062	82.444347	85.44292	79.445775	1.8836617	2.0688246	1.6984988	0.722279	0.7258944	0.7186636
120	0.0686665	0.0902091	0.0471239	86.777764	89.153259	84.402269	1.7692567	1.9488712	1.5896422	0.7226513	0.7258609	0.7194416
AODVUU												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.1601253	0.2211534	0.0990972	7.4568534	10.708836	4.2048704	1.7640964	2.041678	1.4865148	0.4110905	0.4154614	0.4067196
70	0.0826037	0.1176724	0.0475351	11.596423	14.266345	8.9265013	2.0439876	2.4674337	1.6205415	0.4121829	0.4170261	0.4073396
80	0.0728058	0.096142	0.0494696	16.201851	18.339499	14.064203	2.1310037	2.589529	1.6724784	0.4126188	0.4171773	0.4080602
90	0.0591969	0.0742073	0.0441864	21.77026	25.38351	18.157011	2.1337047	2.4884776	1.7789318	0.411769	0.4159943	0.4075437
100	0.0431384	0.0535432	0.0327336	26.575788	31.339887	21.811688	2.071963	2.3781185	1.7658075	0.4104565	0.4157435	0.4051695
110	0.0328251	0.0432752	0.0223751	30.687219	36.051341	25.323097	1.9856079	2.2520125	1.7192033	0.4085781	0.4137383	0.4034179
120	0.0298716	0.0415487	0.0181944	35.130065	40.924562	29.335568	2.0137856	2.2806543	1.746917	0.4060016	0.4100802	0.401923
OLSR												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	1.0219284	1.2655293	0.7783276	26.6008	32.457246	20.744355	1.9580908	2.18326	1.7329215	1.7016699	1.7472871	1.6560526
70	0.7377793	0.9100805	0.5654782	40.532766	46.728742	34.336791	2.0997158	2.3694671	1.8299646	1.9631449	2.0179224	1.9083674

80	0.5281825	0.5994451	0.45692	56.478239	62.404421	50.552057	2.2331175	2.4614725	2.0047626	2.2194159	2.2641333	2.1746985
90	0.3659667	0.4285255	0.303408	70.463357	75.591251	65.335462	2.2210601	2.433263	2.0088572	2.4039196	2.4355698	2.3722694
100	0.2430061	0.2811182	0.204894	80.777889	85.04096	76.514817	2.0738623	2.2825917	1.8651329	2.5152173	2.5519853	2.4784492
110	0.147211	0.1719863	0.1224358	87.556278	90.506261	84.606295	1.9346153	2.1205918	1.7486389	2.5236556	2.5608985	2.4864127
120	0.0940554	0.117668	0.0704427	91.548899	93.825963	89.271836	1.8163642	1.9965731	1.6361553	2.4708974	2.5105818	2.431213

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.9710923	1.2446189	0.6975657	25.125063	30.909812	19.340313	1.8372251	2.0340191	1.640431	1.1831704	1.1923683	1.1739725
70	0.7109609	0.8850281	0.5368937	38.128439	43.855313	32.401565	1.9663582	2.2018465	1.73087	1.2305616	1.2440733	1.2170499
80	0.4965372	0.586268	0.4068064	52.279265	58.70581	45.85272	2.0744085	2.2719341	1.8768828	1.2934676	1.3070815	1.2798538
90	0.3548715	0.4294686	0.2802744	64.072661	70.071926	58.073397	2.0419224	2.2455985	1.8382462	1.3600345	1.3773479	1.3427211
100	0.2463767	0.2939089	0.1988446	76.016133	81.200179	70.832087	1.9665102	2.1631819	1.7698384	1.4007271	1.4113964	1.3900578
110	0.1540398	0.1820766	0.1260031	83.047774	87.398511	78.697037	1.8596825	2.0357722	1.6835927	1.4356552	1.4550577	1.4162526
120	0.093406	0.1145474	0.0722645	88.9007	91.608716	86.192685	1.7844178	1.9551838	1.6136518	1.440676	1.4578776	1.4234744

5 UDP FLOW

TBRPF

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.742062	0.8307329	0.6533911	22.52939	26.659942	18.398837	1.7996501	1.8877817	1.7115185	0.6325838	0.6372176	0.6279499
70	0.633147	0.721869	0.544425	32.883942	37.608324	28.15956	1.96345	2.0612618	1.8656383	0.6590881	0.6652233	0.652953
80	0.4825645	0.5625192	0.4026098	44.037019	48.462181	39.611856	2.0430696	2.1373173	1.9488218	0.6849619	0.6907564	0.6791674
90	0.3369936	0.3931259	0.2808612	53.744997	57.670993	49.819002	2.0265531	2.1277239	1.9253822	0.7045763	0.708797	0.7003555
100	0.2360927	0.2704297	0.2017556	59.86931	62.090903	57.647717	1.9384041	2.0329054	1.8439029	0.7112034	0.7220085	0.7003982
110	0.16188	0.1852664	0.1384936	67.0504	69.756597	64.344204	1.8272207	1.9244201	1.7300214	0.7234943	0.7270901	0.7198984
120	0.1165942	0.1376436	0.0955449	71.311906	73.419934	69.203878	1.7165998	1.8155832	1.6176164	0.7236609	0.7266195	0.7207022

AODVUU

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.1494349	0.1924758	0.1063939	7.3736868	9.8593007	4.888073	1.8133602	2.0370412	1.5896792	0.5010628	0.5138618	0.4882637
70	0.1121581	0.1401466	0.0841696	11.223112	13.57014	8.8760827	1.9211913	2.1079268	1.7344558	0.5050516	0.5195833	0.4905199
80	0.0986377	0.1210813	0.0761941	15.961106	18.188601	13.73361	1.9924259	2.1911477	1.793704	0.5004624	0.513268	0.4876568

90	0.0750406	0.0904377	0.0596436	22.463107	26.71995	18.206263	1.8976463	2.1157125	1.6795801	0.485859	0.494363	0.477355
100	0.0648668	0.0720883	0.0576452	28.791896	32.801762	24.78203	1.7894597	1.8890354	1.689884	0.4814559	0.4899479	0.4729639
110	0.0571332	0.0639099	0.0503565	33.387319	38.322089	28.452549	1.6668103	1.7734161	1.5602046	0.4759475	0.4874097	0.4644853
120	0.0532583	0.0563803	0.0501363	37.259255	42.397038	32.121472	1.6677947	1.7610645	1.5745249	0.4646923	0.4735468	0.4558377
				37.259255	42.397038	32.121472						

OLSR

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.8637247	0.9895515	0.7378978	27.003502	31.804581	22.202423	1.9418091	2.0508667	1.8327514	1.6987134	1.74316	1.6542668
70	0.7172216	0.816478	0.6179651	40.442721	45.835657	35.049786	2.1292934	2.2385985	2.0199883	1.9594116	2.0139909	1.9048324
80	0.5702193	0.6493554	0.4910833	55.892946	60.903252	50.882641	2.225641	2.3280977	2.1231843	2.2112919	2.2564345	2.1661492
90	0.3812303	0.4370231	0.3254376	68.980115	73.380032	64.580198	2.1918216	2.3046595	2.0789836	2.3968623	2.4262462	2.3674783
100	0.2697942	0.3105786	0.2290097	78.667459	82.310445	75.024472	2.0831961	2.1984546	1.9679377	2.5053746	2.5436111	2.4671382
110	0.1808075	0.2102028	0.1514122	85.560905	88.23031	82.891501	1.9612461	2.0650877	1.8574046	2.5169236	2.5538179	2.4800294
120	0.1282615	0.15286	0.103663	90.165708	92.047259	88.284157	1.8324066	1.9326472	1.732166	2.4631505	2.5017109	2.4245901

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.8726985	0.9873925	0.7580044	24.19022	28.470069	19.910371	1.8043028	1.8854974	1.7231083	1.1810733	1.1900486	1.172098
70	0.7625061	0.8592134	0.6657987	33.7994	38.329985	29.268814	1.8956261	1.9877769	1.8034753	1.2345907	1.2469177	1.2222637
80	0.6094456	0.7037682	0.5151229	44.521011	48.614331	40.42769	1.9758567	2.0666343	1.8850791	1.2966019	1.3129898	1.280214
90	0.4533984	0.5213404	0.3854564	54.255878	58.39112	50.120636	1.9638406	2.0596189	1.8680623	1.3459742	1.3578338	1.3341146
100	0.3242282	0.3766052	0.2718512	60.406453	63.262703	57.550203	1.8940657	1.9907443	1.7973872	1.3834447	1.4001132	1.3667762
110	0.2176768	0.2554328	0.1799208	67.528764	70.200062	64.857467	1.8164858	1.9178436	1.7151279	1.4270313	1.4420801	1.4119824
120	0.1695288	0.2055461	0.1335115	67.97086	72.22508	63.716641	1.7116573	1.8019888	1.6213259	1.4217992	1.461808	1.3817903

10 UDP FLOW

TBRPF

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.7406438	0.8085112	0.6727763	22.002876	24.598953	19.4068	1.7721323	1.8324653	1.7117994	0.6325213	0.6373385	0.627704
70	0.6391525	0.6820302	0.5962747	30.687844	33.508826	27.866862	1.8961154	1.9415918	1.850639	0.6530831	0.6629933	0.6431729
80	0.5214304	0.5802404	0.4626204	39.560718	42.345123	36.776313	1.976292	2.022462	1.930122	0.6801884	0.6900511	0.6703257
90	0.3918857	0.4346782	0.3490933	49.479427	51.890037	47.068817	1.9745197	2.0571628	1.8918767	0.7061713	0.7104724	0.7018701

100	0.2880354	0.3138949	0.262176	56.257816	58.171275	54.344358	1.8964559	1.9808264	1.8120854	0.719935	0.7246037	0.7152663
110	0.2095758	0.2330269	0.1861247	61.581728	63.305812	59.857645	1.8006649	1.8733982	1.7279317	0.726802	0.7306841	0.7229199
120	0.1575548	0.173117	0.1419926	65.574037	67.18778	63.960294	1.6898662	1.7516099	1.6281225	0.7267843	0.7304737	0.7230948

AODVUU

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.1479086	0.1823045	0.1135127	6.8465483	8.3235383	5.3695583	1.7515432	1.9170166	1.5860698	0.6231706	0.6392682	0.607073
70	0.128783	0.1497578	0.1078083	10.258879	12.036643	8.4811161	1.8654381	1.9888261	1.7420501	0.6334003	0.6445447	0.6222558
80	0.1150389	0.135208	0.0948699	14.468172	16.509381	12.426963	1.9079616	2.0665637	1.7493594	0.6260724	0.6438777	0.608267
90	0.1019836	0.1180863	0.085881	19.673899	22.625189	16.72261	1.8455288	1.9912123	1.6998452	0.5867536	0.6014314	0.5720758
100	0.0953621	0.1065696	0.0841546	25.746936	29.233533	22.260339	1.7701386	1.8549701	1.6853072	0.5723778	0.588958	0.5557975
110	0.0887337	0.0976321	0.0798352	29.655765	32.656079	26.655452	1.6989531	1.7861205	1.6117857	0.5476105	0.568604	0.526617
120	0.0848084	0.0921472	0.0774697	35.299837	38.648456	31.951219	1.6181425	1.731453	1.504832	0.5371631	0.5482559	0.5260703

OLSR

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.8191815	0.9079412	0.7304218	27.199225	30.460368	23.938081	1.9302949	1.9998814	1.8607084	1.697121	1.7424242	1.6518178
70	0.6999588	0.7484852	0.6514323	39.974987	43.909011	36.040964	2.089047	2.1283493	2.0497448	1.9530465	2.0059957	1.9000973
80	0.5876064	0.6508017	0.524411	54.405953	58.152883	50.659023	2.184734	2.2377794	2.1316885	2.2003755	2.2431192	2.1576318
90	0.4169703	0.4577303	0.3762103	67.079477	70.060209	64.098746	2.1669769	2.2590658	2.074888	2.3856329	2.415391	2.3558748
100	0.3004289	0.3263683	0.2744894	76.503564	78.866003	74.141125	2.0658679	2.1583242	1.9734115	2.4955259	2.5307141	2.4603377
110	0.2138903	0.2370098	0.1907708	83.115308	84.843392	81.387223	1.9497792	2.028107	1.8714514	2.5070818	2.5418581	2.4723054
120	0.1564648	0.1759598	0.1369697	87.855803	89.002542	86.709064	1.8215056	1.8922517	1.7507595	2.4605703	2.4970784	2.4240621

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
60	0.8717505	0.9530904	0.7904105	23.471736	26.006714	20.936758	1.7660545	1.8217316	1.7103774	1.1816491	1.191082	1.1722161
70	0.7785819	0.8267905	0.7303733	32.311468	35.363697	29.259239	1.8691575	1.9100057	1.8283093	1.23081	1.2416644	1.2199556
80	0.646636	0.7045619	0.5887101	41.792459	44.738877	38.84604	1.9367027	1.9792103	1.8941951	1.2928187	1.3055803	1.2800572
90	0.5129746	0.5700084	0.4559408	49.254315	52.472577	46.036052	1.9081382	1.962315	1.8539614	1.3332093	1.3624269	1.3039917
100	0.3753157	0.421145	0.3294864	56.098987	58.061276	54.136698	1.8521812	1.9192986	1.7850638	1.4096371	1.4283085	1.3909657
110	0.280105	0.3184003	0.2418097	60.846986	63.386774	58.307198	1.7750637	1.839512	1.7106153	1.4211232	1.455333	1.3869134
120	0.2055391	0.2376722	0.1734059	63.422024	65.934565	60.909482	1.6786108	1.7518111	1.6054104	1.4254031	1.4714293	1.379377

Ship Case

1 UDP FLOW	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit
TBRPF												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	0.9691535	1.1208428	0.8174642	17.94022	22.664042	13.216398	2.4319334	2.7697242	2.0941425	2.1927536	2.2172594	2.1682478
150	0.63507	0.7982229	0.4719171	34.485993	41.943372	27.028614	3.0218344	3.2473272	2.7963416	2.6829551	2.7193505	2.6465598
175	0.3408835	0.4554551	0.2263119	52.448099	60.629082	44.267116	3.1904126	3.6255635	2.7552618	3.1408016	3.1833743	3.098229
200	0.1857083	0.2660695	0.105347	65.116933	71.811375	58.422492	2.9526186	3.3845763	2.5206609	3.3643558	3.4066583	3.3220532
225	0.1013934	0.1586571	0.0441297	74.434092	80.199486	68.668698	2.6659925	3.0302709	2.301714	3.3947318	3.4286558	3.3608077
250	0.059078	0.0915846	0.0265714	80.046273	84.183676	75.90887	2.4186806	2.7498858	2.0874754	3.345076	3.3697059	3.3204461
275	0.0485644	0.0761845	0.0209443	83.388569	87.565744	79.211395	2.1830618	2.4461013	1.9200223	3.2606895	3.2851034	3.2362756
AODVUU												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	0.1542261	0.2197472	0.0887051	7.7882691	9.9030517	5.6734866	3.0423596	3.6040303	2.4806889	1.2220576	1.2263656	1.2177496
150	0.1510373	0.2047831	0.0972915	13.475488	16.308965	10.64201	3.5085674	4.1397259	2.8774088	1.237132	1.2472524	1.2270116
175	0.0983776	0.1162196	0.0805357	19.25025	22.338236	16.162264	3.5471973	4.3188297	2.7755648	1.2392101	1.2535379	1.2248823
200	0.0664226	0.0823794	0.0504659	22.748874	25.273658	20.22409	3.0833839	3.8062044	2.3605634	1.2361579	1.2517771	1.2205386
225	0.0499125	0.0695442	0.0302808	24.771761	26.575716	22.967806	2.7882378	3.4985968	2.0778787	1.2325055	1.2459527	1.2190583
250	0.0410022	0.056276	0.0257284	25.912956	27.24077	24.585143	2.6010371	3.3054644	1.8966098	1.2314199	1.2440523	1.2187875
275	0.032586	0.0415039	0.023668	27.213607	29.0604	25.366813	2.3195197	2.916559	1.7224804	1.2287903	1.2418144	1.2157661
OLSR												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	1.2368306	1.4144479	1.0592133	22.823912	28.752064	16.89576	2.7655002	3.0097961	2.5212043	7.2982628	7.4724918	7.1240337
150	0.9208593	1.1009542	0.7407643	43.143447	51.538135	34.748758	3.3665618	3.6641366	3.068987	11.608983	11.8877	11.330266
175	0.5167089	0.6695842	0.3638337	63.750625	72.23699	55.264261	3.4811993	3.9365628	3.0258359	16.50256	16.881233	16.123887
200	0.2421515	0.3458124	0.1384906	76.234992	82.514762	69.955223	3.1410675	3.586576	2.695559	19.446643	20.000225	18.893062
225	0.1272734	0.1982528	0.0562939	84.301526	89.125229	79.477823	2.7990341	3.1921096	2.4059585	19.980721	20.397599	19.563843
250	0.0555352	0.092071	0.0189993	88.159705	91.354874	84.964536	2.5262215	2.8564652	2.1959778	19.225613	19.557211	18.894015
275	0.0332946	0.0588741	0.0077151	90.764132	93.345643	88.182621	2.2648943	2.544098	1.9856906	18.044354	18.313686	17.775021

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	1.0641491	1.2049289	0.9233694	18.875063	23.790485	13.959641	2.2043865	2.4508994	1.9578737	3.884692	3.9201552	3.8492288
150	0.8055229	0.9993703	0.6116755	29.905578	37.358929	22.452227	2.3644603	2.6242773	2.1046434	4.6807978	4.7795051	4.5820906
175	0.3825736	0.5086965	0.2564507	41.470735	51.192467	31.749004	2.3967527	2.5860735	2.2074319	6.1001952	6.3162217	5.8841686
200	0.1726641	0.2598707	0.0854575	51.735243	61.690149	41.780337	2.3300085	2.5153444	2.1446726	8.1386065	8.7706858	7.5065272
225	0.0887687	0.1287226	0.0488149	61.727739	72.312637	51.142841	2.1933298	2.3971356	1.989524	9.1901974	9.4247734	8.9556215
250	0.0531639	0.0795492	0.0267787	69.503502	76.854509	62.152495	2.0900683	2.32576	1.8543766	9.9445763	10.337851	9.5513021
275	0.0382599	0.0617485	0.0147714	75.878564	82.821138	68.935991	1.9949731	2.2518154	1.7381309	10.304453	10.560989	10.047916

10 UDP FLOW

TBRPF

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	0.9130987	1.0040586	0.8221389	14.6245	16.637881	12.611119	2.4174411	2.568959	2.2659232	2.19252	2.2172048	2.1678352
150	0.743114	0.8011032	0.6851249	25.330478	27.351953	23.309002	2.9306517	3.0822965	2.7790069	2.6789958	2.7158695	2.642122
175	0.4898944	0.5362095	0.4435793	34.921836	36.989106	32.854566	3.0948211	3.2294558	2.9601865	3.1116464	3.1518072	3.0714855
200	0.2870255	0.3166818	0.2573691	43.247249	45.20757	41.286927	2.8550668	2.9992002	2.7109333	3.3718718	3.413077	3.3306665
225	0.173302	0.1918841	0.1547198	47.476863	49.407054	45.546673	2.5635043	2.6868873	2.4401212	3.4009259	3.4315122	3.3703396
250	0.1158977	0.1273063	0.1044891	50.905765	52.973088	48.838443	2.3315645	2.4421946	2.2209344	3.354553	3.3918398	3.3172662
275	0.0914447	0.0988015	0.0840879	54.429402	55.987701	52.871103	2.125861	2.2211248	2.0305972	3.2884166	3.3140584	3.2627748

AODVUU

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	0.2089245	0.2543845	0.1634645	5.0075038	6.3146474	3.7003601	3.0128757	3.3833508	2.6424006	1.557104	1.6141785	1.5000295
150	0.195029	0.2267994	0.1632587	8.5836668	10.359393	6.8079407	3.3444061	3.7271794	2.9616329	1.6889341	1.779009	1.5988592
175	0.1651542	0.1947026	0.1356059	12.12919	13.898963	10.359416	3.3632174	3.7881079	2.9383269	1.7454634	1.8111088	1.679818
200	0.13449	0.1531501	0.1158299	14.716421	16.610188	12.822653	3.1141319	3.4209483	2.8073155	1.7558699	1.823274	1.6884658
225	0.113678	0.1254552	0.1019007	16.073349	18.669937	13.476761	2.8162767	3.092338	2.5402153	1.793688	1.8429074	1.7444686
250	0.0984337	0.109699	0.0871684	19.002939	21.127962	16.877916	2.4171511	2.6294164	2.2048857	1.7714291	1.8299828	1.7128754
275	0.0955493	0.1034681	0.0876304	20.539332	22.496503	18.582161	2.2588363	2.4732801	2.0443924	1.8341999	1.880278	1.7881217

OLSR

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	1.1937681	1.3218494	1.0656868	19.891196	22.221051	17.56134	2.8374609	2.9949887	2.679933	7.2412459	7.4107667	7.071725
150	0.9831296	1.0499497	0.9163095	38.116558	40.690209	35.542908	3.4543763	3.6028447	3.3059079	11.42706	11.70662	11.147501
175	0.6247632	0.6881618	0.5613646	57.71417	60.108699	55.319641	3.5945158	3.7568014	3.4322302	16.164404	16.56165	15.767157
200	0.3351158	0.3793524	0.2908791	69.733617	71.704691	67.762543	3.2926855	3.4464467	3.1389243	19.148384	19.716471	18.580297
225	0.1904561	0.2184487	0.1624634	76.396011	78.034012	74.758009	2.9158335	3.0580003	2.7736668	19.825144	20.24667	19.403618
250	0.1147192	0.1305206	0.0989178	80.283892	81.419405	79.148379	2.6013511	2.7263383	2.4763639	19.143483	19.480569	18.806396
275	0.082844	0.0916761	0.0740119	82.619122	83.556686	81.681558	2.3485607	2.454511	2.2426105	17.992717	18.262391	17.723043

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	1.0115527	1.140907	0.8821984	15.251376	17.312719	13.190033	2.2233842	2.3606525	2.0861159	3.8892519	3.9264737	3.85203
150	0.8083075	0.9099703	0.7066448	23.746873	25.813124	21.680623	2.4386969	2.5589079	2.3184859	4.6684009	4.7740763	4.5627255
175	0.4853828	0.523538	0.4472277	29.187719	32.092513	26.282925	2.4028628	2.4971554	2.3085702	6.0316907	6.2233501	5.8400312
200	0.2774808	0.3041359	0.2508258	36.423524	39.061306	33.785743	2.342297	2.428896	2.255698	7.7134128	7.9925036	7.4343219
225	0.1596738	0.185371	0.1339766	41.255628	43.472149	39.039107	2.2116499	2.2958934	2.1274063	9.0003119	9.1994291	8.8011946
250	0.1080992	0.121344	0.0948545	45.130378	48.452592	41.808163	2.0889227	2.1622942	2.0155512	9.7535328	10.065598	9.4414674
275	0.0921029	0.1018849	0.0823208	49.361243	51.810954	46.911532	1.9960898	2.0675756	1.924604	10.092539	10.455224	9.7298546

30 UDP FLOW

TBRPF

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	1.0227275	1.0954583	0.9499967	12.414124	13.242563	11.585685	2.3904356	2.4639351	2.316936	2.187104	2.2110419	2.1631661
150	0.9177797	0.9560446	0.8795147	20.445952	21.38763	19.504274	2.8849815	2.9436231	2.82634	2.6717928	2.7079013	2.6356842
175	0.6583073	0.7004353	0.6161792	27.84486	28.728816	26.960904	3.0125918	3.0936092	2.9315745	3.1250415	3.185001	3.065082
200	0.4444536	0.4627903	0.4261169	33.371061	34.089155	32.652966	2.8043953	2.8789994	2.7297913	3.405825	3.4429535	3.3686965
225	0.3079014	0.3248047	0.290998	36.479073	37.475851	35.482295	2.5446144	2.6104562	2.4787726	3.4428634	3.4944471	3.3912797
250	0.2401302	0.2496423	0.2306181	40.009067	40.62884	39.389295	2.3055555	2.3549285	2.2561825	3.4928428	3.5337171	3.4519684
275	0.2040556	0.2125192	0.195592	42.469047	43.404691	41.533403	2.1109594	2.1563522	2.0655665	3.4533659	3.4986094	3.4081224

AODVUU

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	0.2540841	0.2843259	0.2238423	4.1062198	4.7190813	3.4933583	2.7582333	2.9410763	2.5753903	2.4563879	2.5498578	2.3629179
150	0.247379	0.2710938	0.2236642	6.6259171	7.4961462	5.7556881	3.0667581	3.2732202	2.8602961	2.7706644	2.8735784	2.6677503
175	0.2273259	0.2507737	0.2038781	9.2077289	10.109113	8.3063451	3.0829566	3.331175	2.8347382	2.8360186	2.9091831	2.7628542
200	0.1975297	0.2096169	0.1854425	10.826768	11.696496	9.9570391	2.8304362	3.041113	2.6197594	2.7869326	2.8833241	2.6905412
225	0.1935828	0.2053232	0.1818424	12.514486	13.459937	11.569036	2.5772143	2.7612788	2.3931498	2.6933615	2.7971381	2.5895849
250	0.1746826	0.1907799	0.1585854	13.991683	15.036966	12.946401	2.2823888	2.4665772	2.0982003	2.7460953	2.8230323	2.6691582
275	0.1731912	0.1822059	0.1641764	15.582687	17.033214	14.13216	2.1086614	2.2403917	1.9769311	2.7578009	2.816776	2.6988258

OLSR

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	1.2392867	1.3261535	1.15242	18.121978	19.339156	16.904799	2.8364795	2.9124965	2.7604625	7.1659639	7.3245753	7.0073524
150	1.1080609	1.1481316	1.0679902	35.306403	36.720517	33.892289	3.5001892	3.5694646	3.4309138	11.046366	11.301315	10.791416
175	0.7978669	0.8475682	0.7481656	53.077059	54.212768	51.941351	3.6217126	3.7036061	3.5398192	15.492259	15.869213	15.115305
200	0.4879804	0.525659	0.4503019	63.50915	64.313678	62.704623	3.3163283	3.4016312	3.2310255	18.658528	19.260606	18.05645
225	0.3133595	0.3332972	0.2934219	68.72603	69.40696	68.0451	2.9342384	3.0044197	2.8640571	19.717358	20.157625	19.27709
250	0.2256306	0.2365912	0.2146699	71.692281	72.364591	71.019972	2.6121603	2.6729311	2.5513896	19.313203	19.597879	19.028528
275	0.1839865	0.1912997	0.1766732	73.414772	74.179457	72.650087	2.3491641	2.3980667	2.3002615	17.767343	18.058057	17.476628

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Count			Overhead		
125	1.1167925	1.1968686	1.0367163	12.886547	13.78149	11.991605	2.2021862	2.2460522	2.1583203	3.8664602	3.9097363	3.8231841
150	0.9364345	0.9872429	0.8856261	19.187406	20.14262	18.232192	2.4194469	2.4708568	2.3680369	4.6223067	4.6835855	4.5610279
175	0.6358489	0.6765098	0.5951881	24.316116	25.231167	23.401066	2.4353008	2.503629	2.3669725	5.8839131	6.0194337	5.7483925
200	0.4114549	0.451845	0.3710647	27.914895	28.823288	27.006502	2.3220135	2.395434	2.2485931	7.2738624	7.5451531	7.0025716
225	0.2695614	0.289819	0.2493038	31.938052	33.090063	30.786042	2.2055749	2.2724667	2.1386831	8.5127823	8.805874	8.2196907
250	0.2145944	0.2251939	0.2039948	34.80584	35.917424	33.694257	2.0586324	2.0962226	2.0210421	9.5720264	10.061852	9.082201
275	0.1910196	0.1996906	0.1823486	38.36783	38.903613	37.832046	1.9515217	2.001049	1.9019943	9.9498041	10.217324	9.6822842

Car Case

1 UDP FLOW	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit	Average	CI Upper Limit	CI Lower Limit
TBRPF												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	0.926491	1.3571093	0.4958726	12.05	16.274166	7.8258343	4.7808015	7.5925162	1.9690868	5.7285655	6.0453505	5.4117805
175	0.9586286	1.2058416	0.7114157	25	30.029241	19.970759	5.018816	6.7313201	3.3063118	7.719138	7.9484791	7.4897969
200	0.2651243	0.3438518	0.1863967	40.9625	49.042105	32.882895	4.6604889	6.2955952	3.0253826	8.7876065	8.8964632	8.6787498
225	0.2097017	0.2877234	0.13168	50.3375	60.353605	40.321395	4.0358118	5.1625935	2.9090302	9.22066	9.321763	9.119557
250	0.146169	0.2085785	0.0837595	57.9875	67.424407	48.550593	3.5861192	4.5274844	2.644754	9.1939565	9.296897	9.091016
275	0.0674961	0.0980478	0.0369445	67.275	73.880746	60.669254	3.2324518	4.0316939	2.4332097	8.937312	9.0686443	8.8059797
300	0.0306906	0.0401258	0.0212554	69.9625	77.778778	62.146222	2.9615423	3.6562332	2.2668513	8.6371875	8.795516	8.478859
AODVUU												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	0.3538341	0.4650118	0.2426564	21.575	26.765701	16.384299	6.2745289	8.4741911	4.0748666	7.6627125	7.7257843	7.5996407
175	0.2562961	0.3310591	0.181533	43.2125	47.286956	39.138044	6.3311962	7.8765357	4.7858568	7.831465	7.9233469	7.7395831
200	0.1455812	0.1876502	0.1035122	62.45	67.927765	56.972235	5.4254902	6.7864548	4.0645255	7.8435205	7.9631467	7.7238943
225	0.1032672	0.1376751	0.0688593	71.6125	77.898089	65.326911	4.7652904	5.8921058	3.638475	7.8110345	7.9309985	7.6910705
250	0.0807267	0.099799	0.0616544	77.55	82.874469	72.225531	4.2367901	5.220937	3.2526433	7.815078	7.9285597	7.7015963
275	0.0656044	0.0854509	0.0457579	82.65	86.926618	78.373382	3.7829625	4.7245999	2.8413251	7.7734255	7.8915988	7.6552522
300	0.0602137	0.0761704	0.0442571	85.3875	89.499495	81.275505	3.4880831	4.2667387	2.7094275	7.789136	7.8968026	7.6814694
OLSR												
Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.6272203	1.9714726	1.2829681	18.3625	24.211504	12.513496	5.0430123	7.405932	2.6800926	25.018621	28.23992	21.797321
175	1.3750022	1.9132937	0.8367108	39.5125	45.1765	33.8485	5.5975857	7.1656623	4.0295092	42.895052	46.442999	39.347104
200	0.433342	0.5423477	0.3243363	57.05	64.545329	49.554671	4.9232976	6.3199973	3.526598	58.097923	60.889334	55.306512
225	0.2779833	0.3774912	0.1784754	66.9	74.14723	59.65277	4.3307282	5.4150673	3.2463891	66.128434	67.953246	64.303621
250	0.1449125	0.2243854	0.0654396	73.1375	79.6903	66.5847	3.7871074	4.6906646	2.8835502	68.119979	69.417326	66.822631
275	0.0710717	0.104061	0.0380825	79.5625	85.056138	74.068862	3.3785618	4.123541	2.6335826	65.537143	66.535493	64.538792

300	0.0330653	0.0498305	0.0163002	82.9125	88.137538	77.687462	3.0659019	3.7437627	2.3880412	61.326374	62.373149	60.279598

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.4675509	1.8933713	1.0417306	10.233333	15.738821	4.727846	3.5271039	4.5913266	2.4628812	8.7521918	9.3493272	8.1550565
175	0.9481909	1.2882772	0.6081047	20.95	27.608471	14.291529	2.6405855	3.3301011	1.9510698	12.646926	13.641408	11.652444
200	0.5220153	0.7944749	0.2495557	28.533333	34.29087	22.775797	2.5141991	3.108175	1.9202232	16.862391	20.151077	13.573704
225	0.1995331	0.341167	0.0578993	36.25	43.376935	29.123065	2.5744946	2.9261093	2.22288	23.540329	26.769177	20.31148
250	0.110998	0.2135722	0.0084238	40.916667	51.262178	30.571155	2.4209406	2.686717	2.1551642	27.386161	29.738147	25.034175
275	0.0842409	0.1684033	7.847E-05	44.766667	54.317914	35.215419	2.2804326	2.5006869	2.0601782	34.010175	36.226427	31.793924
300	0.0212648	0.0241532	0.0183763	54.233333	63.71434	44.752326	2.3334798	2.6370835	2.029876	35.339211	37.327584	33.350837

30 UDP FLOW

TBRPF

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.0632454	1.1795493	0.9469415	11.465	13.243731	9.6862687	3.2485176	3.5326221	2.9644131	5.676966	5.9802368	5.3736952
175	0.8470587	0.9287118	0.7654055	17.461667	19.50957	15.413763	3.6694527	3.8524532	3.4864521	7.642026	7.8810046	7.4030474
200	0.5714635	0.6246949	0.518232	22.982083	25.562696	20.401471	3.5631923	3.6904931	3.4358915	8.8093405	8.938381	8.6803
225	0.366398	0.4004514	0.3323446	26.831667	29.254373	24.40896	3.3015436	3.4395684	3.1635188	9.3813445	9.4706742	9.2920148
250	0.2626905	0.2902972	0.2350838	29.950833	32.321521	27.580146	3.0127517	3.1654042	2.8600993	9.406379	9.5351933	9.2775647
275	0.2159835	0.2360255	0.1959415	32.70125	35.022486	30.380014	2.7686197	2.90313	2.6341094	9.284801	9.3975971	9.1720049
300	0.1913651	0.2069669	0.1757634	34.626667	36.931603	32.32173	2.5428405	2.6721653	2.4135156	9.0908685	9.2096448	8.9720922

AODVUU

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	0.3886201	0.4329098	0.3443303	14.037083	15.823393	12.250774	4.3556888	4.5679887	4.1433889	15.825045	16.675234	14.974855
175	0.3560588	0.3910728	0.3210449	20.087083	21.564196	18.609971	4.6905991	5.0041405	4.3770577	17.674718	18.527179	16.822257
200	0.3011842	0.3328946	0.2694739	25.6375	27.824207	23.450793	4.3453131	4.6817858	4.0088405	18.474314	19.579547	17.369081
225	0.2610159	0.279132	0.2428998	29.114583	30.462494	27.766673	4.0122492	4.2233455	3.8011529	18.800966	19.499859	18.102072
250	0.2223167	0.2378527	0.2067806	30.717917	32.598844	28.836989	3.5006613	3.7023596	3.2989631	18.162104	18.839946	17.484262
275	0.2144441	0.2252525	0.2036356	32.29	34.106767	30.473233	3.2426889	3.4281941	3.0571837	18.910601	19.378741	18.442461
300	0.2049631	0.2157501	0.1941761	33.426667	35.212056	31.641277	2.9586502	3.1246948	2.7926056	19.043056	19.642737	18.443375

OLSR

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.3682908	1.5589881	1.1775935	20.463333	23.845758	17.080909	3.9414292	4.2043819	3.6784766	24.144883	27.045983	21.243783
175	1.1582938	1.3133299	1.0032577	36.408333	40.245362	32.571304	4.5781534	4.7773872	4.3789196	39.951613	43.251618	36.651608
200	0.6951317	0.7823031	0.6079604	49.130833	53.387941	44.873726	4.4117181	4.6069514	4.2164848	54.905775	57.925196	51.886353
225	0.4316931	0.5002605	0.3631258	57.09375	59.905097	54.282403	4.0236213	4.2515739	3.7956687	63.716333	65.957749	61.474917
250	0.2639401	0.2999953	0.2278849	61.399583	63.680454	59.118713	3.5802109	3.8063309	3.3540908	67.36376	69.110052	65.617468
275	0.2058017	0.2302639	0.1813396	64.35625	66.33887	62.37363	3.217499	3.413666	3.021332	66.248601	67.514886	64.982316
300	0.1821297	0.1979231	0.1663363	65.752917	67.425446	64.080387	2.9168323	3.0950416	2.738623	62.773896	64.010247	61.537545

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.248962	1.4581751	1.0397489	10.720833	12.448127	8.9935392	2.7476162	2.9682606	2.5269719	8.467975	8.8353808	8.1005692
175	0.9468208	1.1060351	0.7876066	14.103889	16.121099	12.086679	2.7858235	2.9641387	2.6075083	10.722022	12.20206	9.2419839
200	0.5793618	0.7356678	0.4230559	16.842222	19.288852	14.395592	2.7046971	2.8122803	2.5971139	15.604581	18.209496	12.999665
225	0.3751955	0.4587704	0.2916205	19.413889	21.73321	17.094568	2.5955568	2.7242046	2.4669091	20.579994	23.386667	17.773321
250	0.227241	0.2615292	0.1929529	22.122778	25.421216	18.82434	2.448513	2.5579111	2.339115	26.596083	29.635923	23.556242
275	0.1722767	0.1860513	0.1585021	24.57	27.496342	21.643658	2.3299098	2.3951474	2.2646722	31.693074	33.767885	29.618263
300	0.1505886	0.1602411	0.1409361	26.772222	30.198655	23.345789	2.2331878	2.2956048	2.1707709	32.037487	32.810565	31.264408

70 UDP FLOW

TBRPF

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.2416803	1.3279011	1.1554595	9.7653571	10.861163	8.6695509	3.2174447	3.4662293	2.9686602	5.590395	5.8974586	5.2833314
175	1.0909012	1.1702553	1.0115471	14.434643	15.654415	13.214871	3.5806761	3.7121532	3.4491989	7.427417	7.6562523	7.1985817
200	0.8138968	0.8817886	0.7460051	18.73	20.251324	17.208676	3.5247449	3.6803254	3.3691644	8.810639	8.9707083	8.6505697
225	0.6099553	0.6421114	0.5777992	22.128036	23.450437	20.805634	3.3305992	3.463959	3.1972393	9.514071	9.6722086	9.3559334
250	0.4708438	0.4976025	0.444085	24.934464	26.270011	23.598918	3.0523477	3.165166	2.9395294	9.806629	9.9441091	9.6691489
275	0.3932113	0.4173181	0.3691044	27.025179	28.200572	25.849785	2.8217688	2.9271489	2.7163887	9.827541	9.933943	9.721139
300	0.3651342	0.3918317	0.3384368	28.955536	30.25031	27.660762	2.6150082	2.7042693	2.5257472	9.826342	9.9468037	9.7058803

AODVUU

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	0.4679262	0.4906517	0.4452007	10.919821	11.998213	9.8414302	4.1028082	4.2995413	3.9060752	26.860182	28.338314	25.382049
175	0.4691768	0.4921931	0.4461606	15.626607	16.688201	14.565013	4.4213025	4.5996508	4.2429541	31.452934	32.910857	29.995011
200	0.4150584	0.4414285	0.3886883	19.377679	20.525772	18.229585	4.0893907	4.2963269	3.8824545	32.719068	33.654121	31.784015
225	0.3761943	0.3926493	0.3597393	21.794286	22.819505	20.769067	3.7505826	3.9212772	3.5798881	32.338273	32.797425	31.87912
250	0.3456007	0.3631348	0.3280666	23.838393	24.9425	22.734286	3.3427538	3.4814556	3.204052	31.936139	32.466773	31.405505
275	0.3309366	0.3446916	0.3171816	25.156429	26.439254	23.873603	3.0403676	3.1751499	2.9055854	32.050817	33.380128	30.721505
300	0.3265279	0.339854	0.3132018	26.58375	27.558772	25.608728	2.7883226	2.9049756	2.6716696	30.423299	31.289423	29.557175

OLSR

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.5145207	1.6212233	1.4078182	19.140357	21.92771	16.353004	3.9524041	4.2113966	3.6934115	22.976452	25.556669	20.396234
175	1.3688788	1.4745102	1.2632474	33.186429	36.080646	30.292211	4.4707209	4.571837	4.3696048	36.764965	39.743836	33.786093
200	0.9668845	1.0667237	0.8670453	44.820714	47.696272	41.945156	4.3985387	4.5462243	4.250853	50.689213	53.956402	47.422024
225	0.6769363	0.7382823	0.6155903	52.264464	54.180604	50.348324	4.0537477	4.2077775	3.8997179	60.490469	63.338596	57.642342
250	0.4699235	0.5085446	0.4313024	56.334286	57.646051	55.022521	3.627115	3.775259	3.478971	66.324537	68.891071	63.758003
275	0.3946355	0.4214328	0.3678383	58.674643	59.588524	57.760761	3.2630446	3.3914879	3.1346013	67.087473	69.171434	65.003512
300	0.3645067	0.3880137	0.3409997	59.698571	60.548777	58.848366	2.9613303	3.0753783	2.8472824	65.684333	67.250232	64.118434

OSPFMCDS

Range	End-to-End Delay			Packet Delivery Percentage(%)			Hop Cpunt			Overhead		
150	1.3744267	1.5066718	1.2421816	9.7778571	10.617844	8.9378704	2.7510925	2.9913908	2.5107942	8.4416608	8.8382969	8.0450246
175	1.0594766	1.224013	0.8949402	11.917143	13.360468	10.473818	2.7600236	2.9128118	2.6072353	10.539394	11.917949	9.1608389
200	0.7509428	0.9441709	0.5577147	14.362857	15.733671	12.992043	2.7182659	2.8877276	2.5488043	14.731104	17.107855	12.354352
225	0.5114896	0.5913843	0.4315949	16.458333	17.991488	14.925179	2.6049144	2.7250348	2.484794	18.945295	21.126762	16.763828
250	0.3556429	0.4163389	0.2949469	18.702143	20.195651	17.208635	2.4129596	2.5670811	2.2588381	24.476294	27.085131	21.867457
275	0.3111	0.3442677	0.2779323	20.908571	22.445889	19.371254	2.3277432	2.4399296	2.2155567	28.395312	30.493872	26.296751
300	0.3011593	0.3269299	0.2753887	23.852857	25.473082	22.232632	2.2434847	2.3319871	2.1549822	30.247651	31.371921	29.12338