Performance Analysis of AODV MANET under Black Hole Attack

ENSC 894: Communications Network

Final Project

Project Website

Johnpaul Kosisochukwu Nwagwu

Student's Number: 301447651

Email: jkn14@sfu.ca

Group Number: 3

Simon Fraser University

April 11, 2023

Contents

List of Figures

List of Tables

Some List of Acronyms

MANET Mobile Ad Hoc Network

AODV Ad Hoc On-Demand Distance Vector

OLSR Optimized Link State Routing

PANs Personal Area Networks

BANs Body Area Networks

FANETs Flying Ad hoc Networks

IOT Internet of Things

IMANETs Internet Mobile Ad hoc Networks

DSDV Destination Sequenced Distance Vector

GSR Global State Routing

WRP Wireless Routing Protocol

TORA Temporally Ordered Routing Protocol

DSR Dynamic Source Routing

ZRP Zone Routing Protocol

Abstract

A Mobile Ad Hoc Network (MANET) is a group of independent mobile nodes that spontaneously creates a radio network without using any preexisting infrastructure. This type of network is vulnerable to a variety of network layer assaults because of its changing topology and lack of centralized administration. One such network attack is the black hole attack. In this project, the performance of MANETs under the Ad Hoc on-Demand Distance Vector Protocol(AODV) is simulated and analyzed with and without a black hole attack and performance metrics such as throughput, and Average End to End delay is computed and compared. Also, we consider the density of the nodes, speed of the nodes, mobility of the nodes and the mobility model of the nodes in order to have a simulation which deals with more general and more real scenarios. The simulation is done using the Network Simulator; NS-3.33 simulator.

Acknowledgements

Huge Thanks to God Almighty for his unending love, my professor and teaching assistant for their various inputs and contributions

Chapter 1

Introduction

Mobile Ad Hoc Networks (MANETs) are wireless networks consisting of mobile nodes that are not connected together with the use of physical wires. Due to the nature of this network, it's topology is constantly changing as nodes in the network communicate with each other. They have so many applications such as in the military as Tactical Networks, Smart Cities, Education, Entertainment and Mobile Conferencing among others. An example of MANET is shown in Figure 1.1.

MANETs are wireless networks that lack centralised network node management and infrastructure support. The nodes' mobility and even the potential for dynamic joining and departing of the network create frequent changes in the topology of the network. Due to this decentralized nature of the network and dynamic topology, nodes in MANETs may be susceptible to various assaults, such as Blackhole attacks, wormhole attacks, Route Request (RREQ) Flooding Attacks and so on.[1]

In contrast to wired networks, each node in the mobile ad hoc network can serve as a router and forward packets to neighbouring nodes and as such, there is no way to distinguish between legitimate users and malicious nodes controlled by attackers in the network. In this project, we concentrate on Ad hoc on-Demand Distance Vector (AODV) protocol and the security of the network layer in order to protect the network layer's routing and forwarding

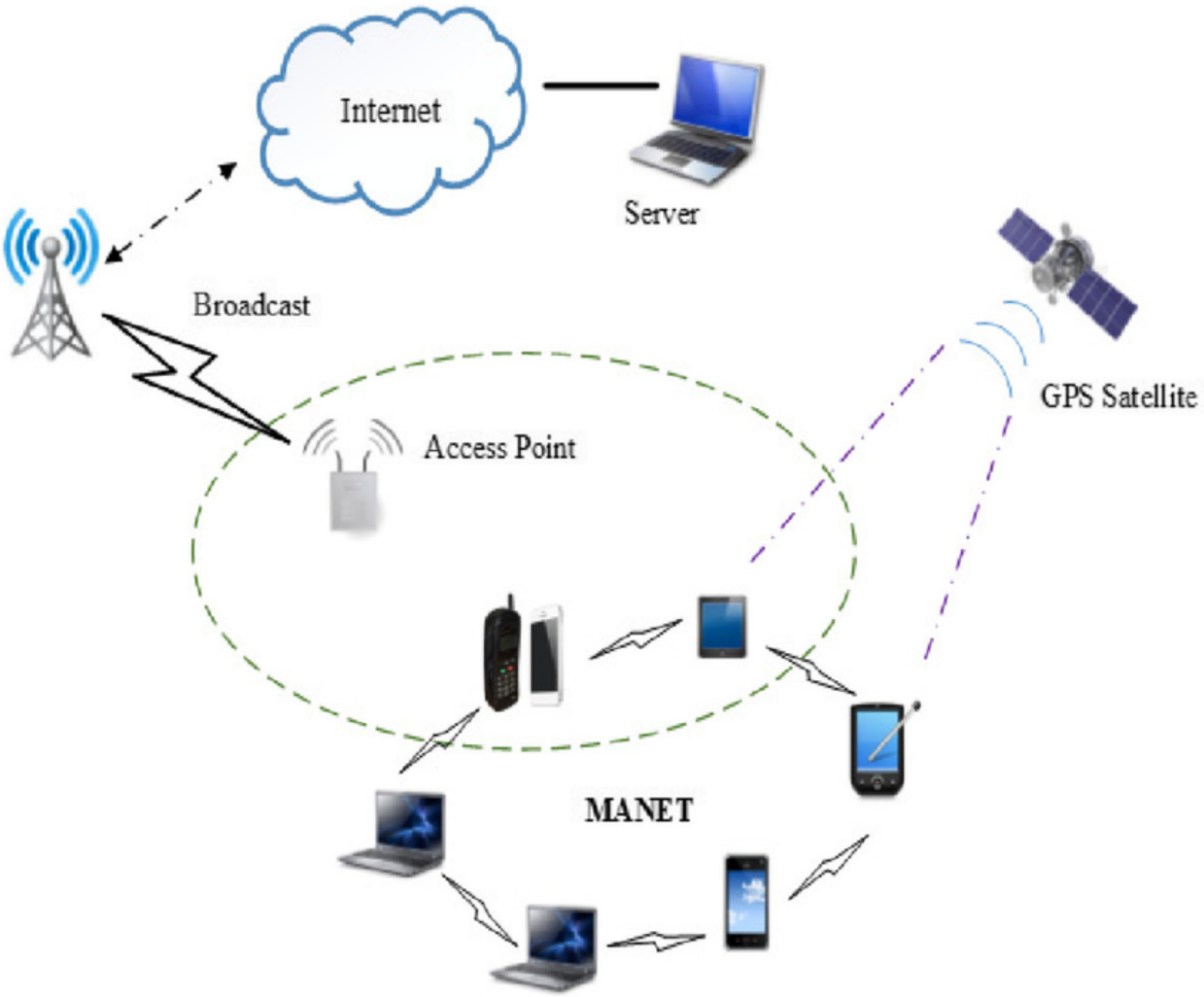


Figure 1.1: Mobile Ad Hoc Network [3]

function since it lacks a clear line of defence. [2]

The rest of the report is structured as follows. In Chapter 2 we discuss Related Work. Routing Protocols and Attacks on Mobile Ad hoc Networks(MANETs) are discussed in Chapters 3 and 4. The simulation scenario and setup are discussed in Chapter 5. We show simulation results in Chapter 6. Finally, we discuss the results and conclude and discuss future work in chapters 7, 8 and 9 respectively.

Chapter 2

Related Work

There is quite a good amount of work that has been done on MANETs and their vulnerabilities in various attack scenarios using various protocols such as AODV, OLSR and DSDV. Several Network layer Attacks have also been explored such as Worm Hole Attack, Flooding Attack, Gray Hole Attack and Black Hole Attack. In the following paragraphs, we discuss some related work.

Konagala Pavani and Damodaram Avula used NS-2.29 to simulate the performance of AODV routing protocol in mobile ad hoc networks under normal conditions and under a black hole attack by measuring metrics such as the packet delivery ratio, Throughput and Packet Loss for 20 nodes at 20 seconds, 40 seconds, 60 seconds and 80 seconds pause times. Their simulation showed that the throughput of the network is reduced when under black hole attack with increased packet loss and lower packet delivery ratio[1]

A study was done by Sakshi Jain and Ajay Khueta on detecting and overcoming black hole attacks in AODV mobile ad hoc networks by deploying a base node in the network. The base node simply helped to detect the malicious node and isolated it from the rest of the network. The simulation was carried out using NS-2.34 for 200 seconds and metrics chosen to monitor the performance of the proposed system were the throughput, packet delivery ratio and Average End to End Delay. The proposed network when under a black

hole attack showed an increase in packet delivery ratio with improved throughput but with higher delays when compared to the normal network under a black hole attack. [4]

Another study was done by Latha Tamilselvan and V. Sankaranarayanan on the prevention of black hole attacks in MANET. The simulation of the attack was done on the modified AODV protocol and compared to the basic AODV protocol using the global mobile simulator on 25 nodes for 5 minutes. The Routing overhead, Average End to End delay and Packet delivery ratio were calculated and compared. The results showed that the modified AODV protocol performed better than the basic AODV protocol but with slightly additional delay and overhead[5]

Chapter 3

MANET Routing Protocols

MANET routing protocols can be grouped into five categories; Proactive, Reactive, Hybrid, Hierarchical and Geographic position routing protocols[6]. The goal of the routing protocol in mobile ad hoc networks is to establish a cost-effective path between the packet's source and destination. They can be classified into five different categories as shown in Figure 3.1

The Proactive Routing protocol is also called a table-driven protocol. Each node has updated route information for all nodes on the network and each node's routing table is automatically updated if there is a change in the network topology. An example of this protocol is DSDV (Destination Sequence Distance Vector), Wireless Routing Protocol(WRP), and Optimized Link State Routing Protocol (OLSR).

The Reactive Routing protocol is also known as the On-demand protocol. This protocol does not maintain updated tables for all nodes in the network. It discovers routes for the transfer of packets only when there is a demand for them. If a route between the source and the destination is not already in the source node's route cache, the route discovery process is started. An example of this protocol is Ad hoc On-Demand Distance Vector (AODV), Temporally Ordered Routing Algorithm (TORA) and Dynamic Source Routing Protocol (DSR).

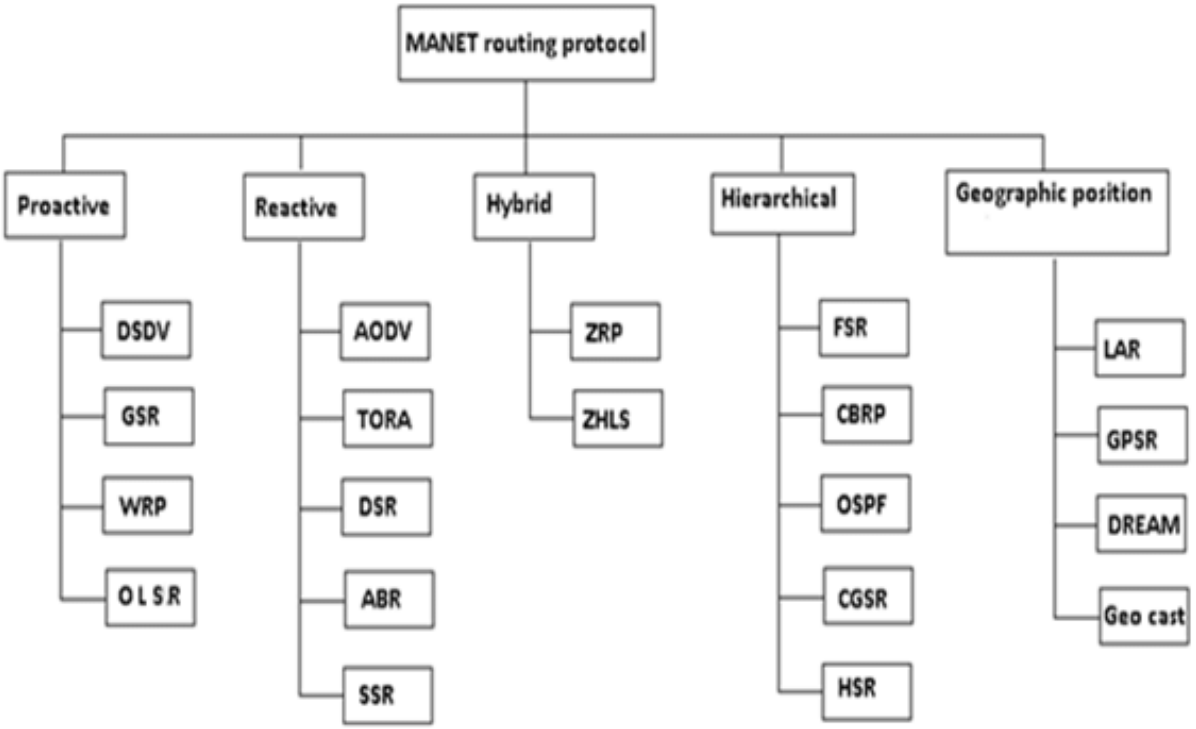


Figure 3.1: MANET Routing Protocols[7]

The Hybrid Routing protocol combines the functionalities of proactive and reactive routing protocol. This routing protocol was primarily created for larger, more complicated networks in order to benefit from both proactive and reactive routing protocols. An Example of such routing protocols include Zone Routing Protocol(ZRP)

Hierarchical Routing groups mobile nodes on multiple levels and clusters. It efficiently combines the qualities of both proactive and reactive protocols. The advantage of hierarchical routing is that the full route does not need to be recomputed in the event of a route failure. Examples of such routing protocols are Fisheye State Routing (FSR) and Cluster Gateway Switch Routing Protocol (CGSR)

Geographic Routing protocols employ topology information only a little and don't require updating routing tables, they are more organised and scalable for ad-hoc networks. Each node in this protocol is aware of its own geocoordinates and communicates this geoin-

formation by flooding the network. Examples include Location-Aided Routing (LAR), and Geocast Protocols.[6]

3.1 Ad Hoc on-Demand Distance Vector (AODV)

AODV is a reactive protocol and an improvement over the Destination-Sequenced Distance-Vector (DSDV) routing algorithm. It is designed for mobile ad hoc networks with many mobile nodes and can handle low, moderate, and relatively high mobility rates, as well as a variety of data traffic levels. AODV aims to reduce the number of broadcast messages forwarded throughout the network by discovering routes on-demand instead of keeping a complete update of route information, unlike the proactive protocols.

When a source node wants to send a data packet to a destination node, it checks its route table to see if it has a valid route to this destination. If a route exists, it forwards the packets to the next hop along the way to the destination. However, if there is no route in the table, the source node initiates a route discovery process by broadcasting a Route Request (RREQ) packet to its immediate neighbours. The RREQ packet contains information such as the IP address of the source node, the current sequence number, the IP address of the destination node, and the last known sequence number.[8]

Intermediate nodes can reply to the RREQ packet only if they have a destination sequence number that is greater than or equal to the number contained in the RREQ packet header. The Route Reply (RREP) packet is forwarded along the established reverse path, and the forward route entry is stored in the routing table. In case of link failure during packet transmission, a Route Error (RERR) packet is sent to notify the sending nodes in the network. However, if either the destination or the intermediate node moves away, route maintenance is performed. Route maintenance is performed by sending a link failure message to each of its neighbours to ensure the removal of that failed part of the route.[9][3]

Chapter 4

Attacks in MANETs

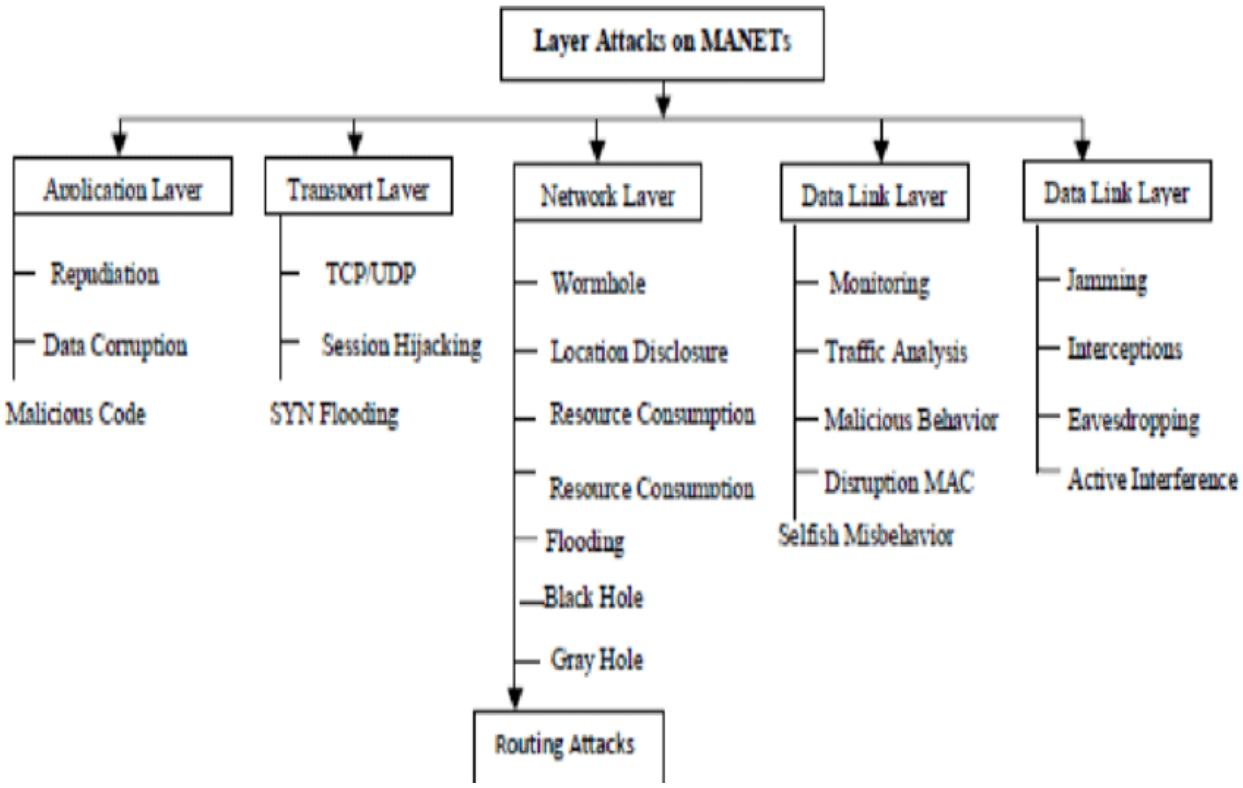


Figure 4.1: MANET Attacks [6]

There are different types of attacks in MANETs as shown in Figure 4.1. These security loopholes are exploited at various layers of the network such as the application layer, Transport layer, Network layer, and Data Link layer.

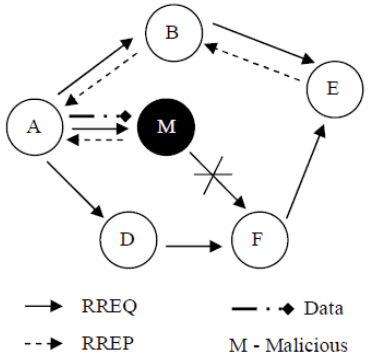


Figure 4.2: Black Hole Attack[11]

In the application layer, malicious code may be added to the application software which may result in data loss or corruption. In the Transport layer, attacks such as session hijacking and SYN flooding can occur. In the Data Link layer, MANETs can be exposed to attacks such as Traffic Analysis, Jamming, Eavesdropping and link frame monitoring.[2]

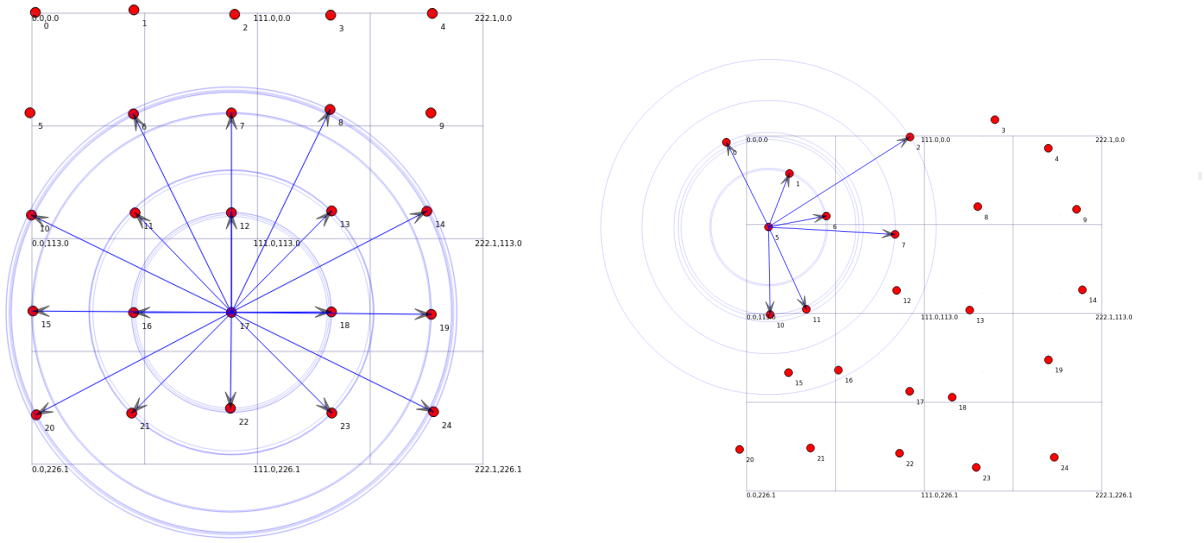
One of the most common layers commonly exploited layer of MANETs is the network layer. This layer is exposed to a variety of attacks such as Flooding attacks, Black Hole attacks, Worm Hole attacks, and Gray Hole attacks.

4.0.1 Black Hole Attack in AODV MANETs

A Black hole in a mobile ad hoc network is simply a malicious node that advertises itself as the cost-effective path to a node seeking a route to a destination node, takes this packet and drops it or sends it to a non-existent IP address as shown in Figure 4.2. This black hole drops packets repeatedly without informing the sending node that the packet did not reach its intended destination.[8] [10]

Chapter 5

Network Simulation and Attack Setup



(a) (b)

Figure 5.1: NetAnim view of the 25 node MANETS (a) static nodes (b) mobile nodes

The overall set-up consists of 25 nodes equally spaced and distanced from each other by 50m. During a black hole attack, one malicious node is placed at the centre depending on the position of the nodes. A simple Network Animation view of this network is shown in Figure 5.1. The concentric circles indicate the region within the reach of the node sending a packet. The remaining information can be found in Table 5.1 below.

Parameter	**Definition**
Routing Protocol	AODV
Simulation time	100 seconds
Attack time	50 to 80 seconds
Packet Size	64 Bits
Traffic Rate	2048 bps
Traffic Source	ConstantBitRate
Number of nodes	25 nodes, 49 nodes
Grid Spacing	50m
Grid Size	5 x 5, 7 x 7
Number of Sinks	1,3, and 5
Mobility Model	ConstantMobilityModel, RandomWalk2DMobilityModel
Propagation Model	Constant Speed
Node Speed	20 m/s and 50 m/s
Node Pause time	0 second
Attack Type	Black Hole Attack
Wifi Propagation Loss Model	FriisPropagationLossModel
Number of nodes	25,49
Wifi	AdhocWifiMac
Wifi Standard	802.11B
WifiDataMode	DsssRate11Mbps
WifiControlMode	DsssRate11Mbps
Protocol	User Datagram Protocol(UDP)
Performance Metrics	Average ThroughPut and Average End-to-End Delay

Table 5.1: Simulation Parameters

This simulation was done using NetAnim 3.33 and Network Animation module to view the transfer of packets from the generated XML file. The simulation code for these Experiments is contained in Appendix C with Appendix A and B serving as header and source files in aodv-routing-protocol.h and aodv-routing-protocol.cc. To run this experiment simply copy the helper and source files to ns3.33/src/aodv/model. The implementation of the Black Hole Attack is provided in [12]

The Attack starts at 50 seconds and ends at 80 seconds for each scenario. After each simulation for a source-sink pair, there is a 10 second time offset for the next simulation. Each scenario simulates the throughput and average End-to-End delay for one source-sink pair, three source-sink pairs and five source-sink pairs which are randomly assigned to the nodes in the network.

Chapter 6

Results

6.1 Network Operating Regularly: No Malicious Node

This involves 25 nodes spaced 50m apart equally. We simulate two scenarios: normal density (25 nodes) and high density(49 nodes) for 1,3, and 5 source-sink pairs. Each scenario runs for a total of 100 seconds.

Figures 6.1 through 6.2 show the results of a network with no malicious nodes and static nodes with constant position mobility models and 1,3, and 5 source-sink pairs. We present a series of time evolution graphs for the Throughput and Average End-to-End Delay.

Figures 6.3 through 6.6 show the results of a network with no malicious nodes and mobile nodes with Random Walk 2D mobility models and 1,3, and 5 source-sink pairs. We present a series of time evolution graphs for the Throughput and Average End-to-End Delay.

Static Nodes and Normal Density

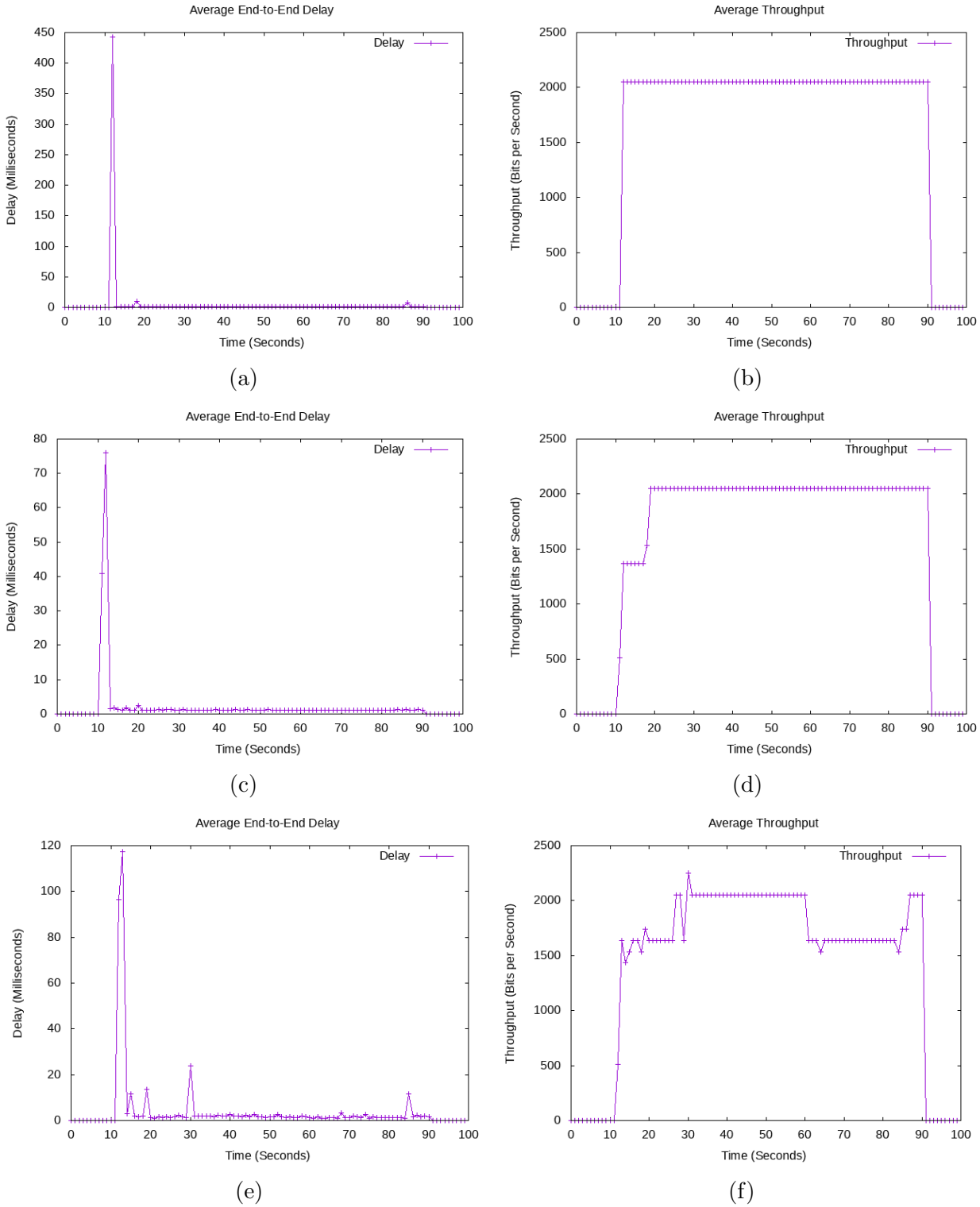


Figure 6.1: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with Static Nodes, Normal Density and Constant Mobility Model. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

Static Nodes and High Density

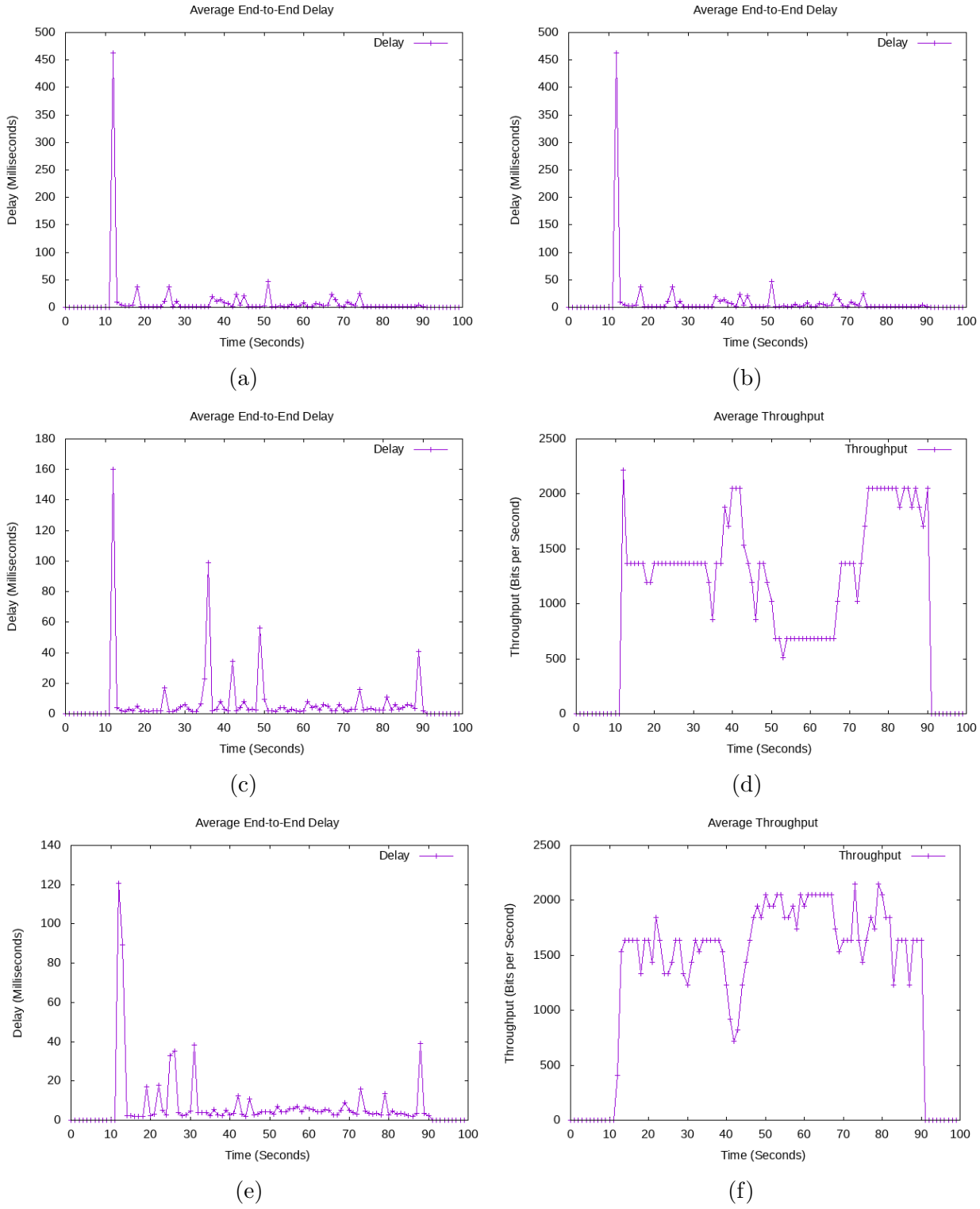


Figure 6.2: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with Static Nodes, High Density and Constant Mobility Model. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

Normal Speed and Normal Density

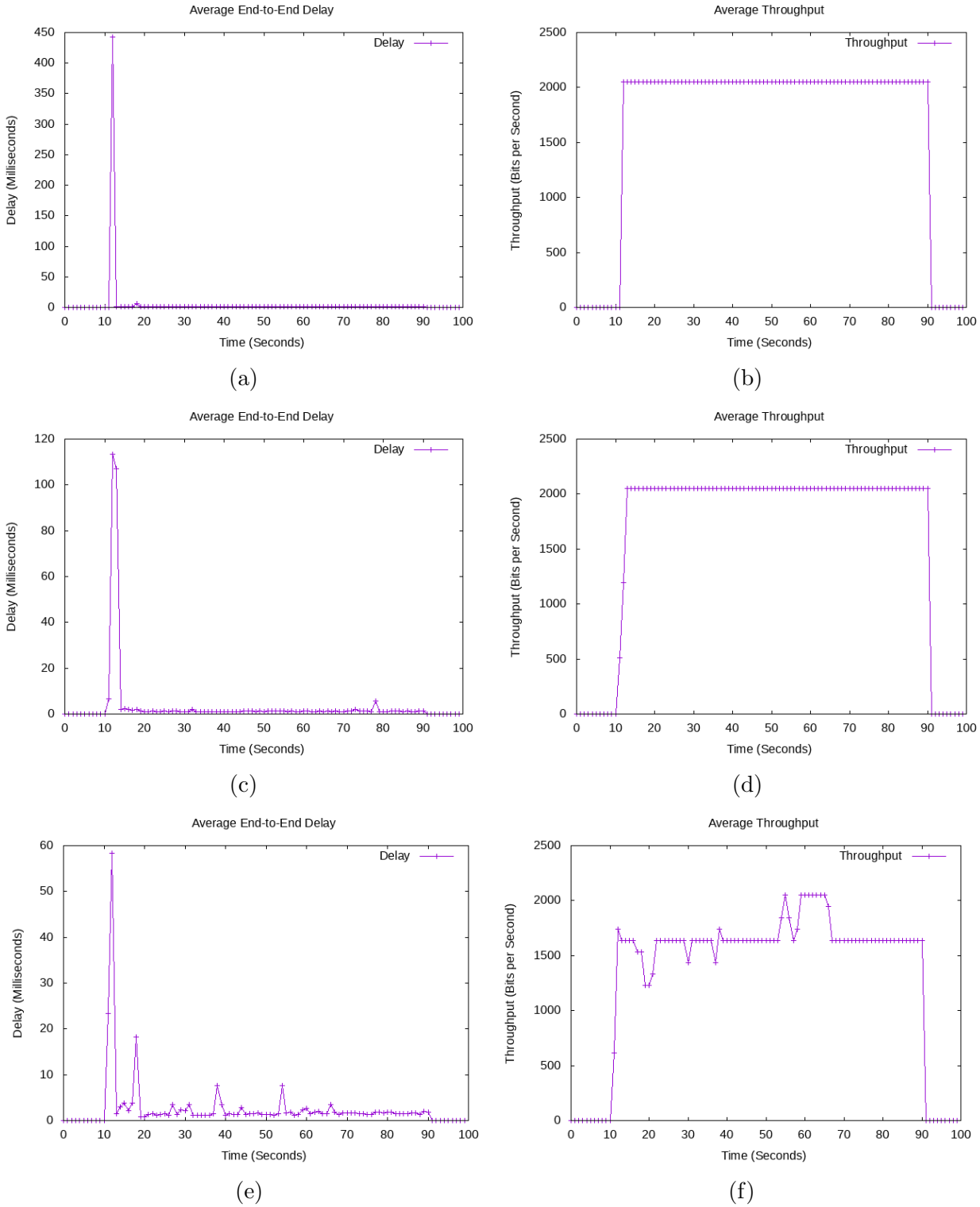


Figure 6.3: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with Mobile Nodes, Normal Speed, Normal Density and RandomWalk2D Mobility Model. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

Normal Speed and High Density

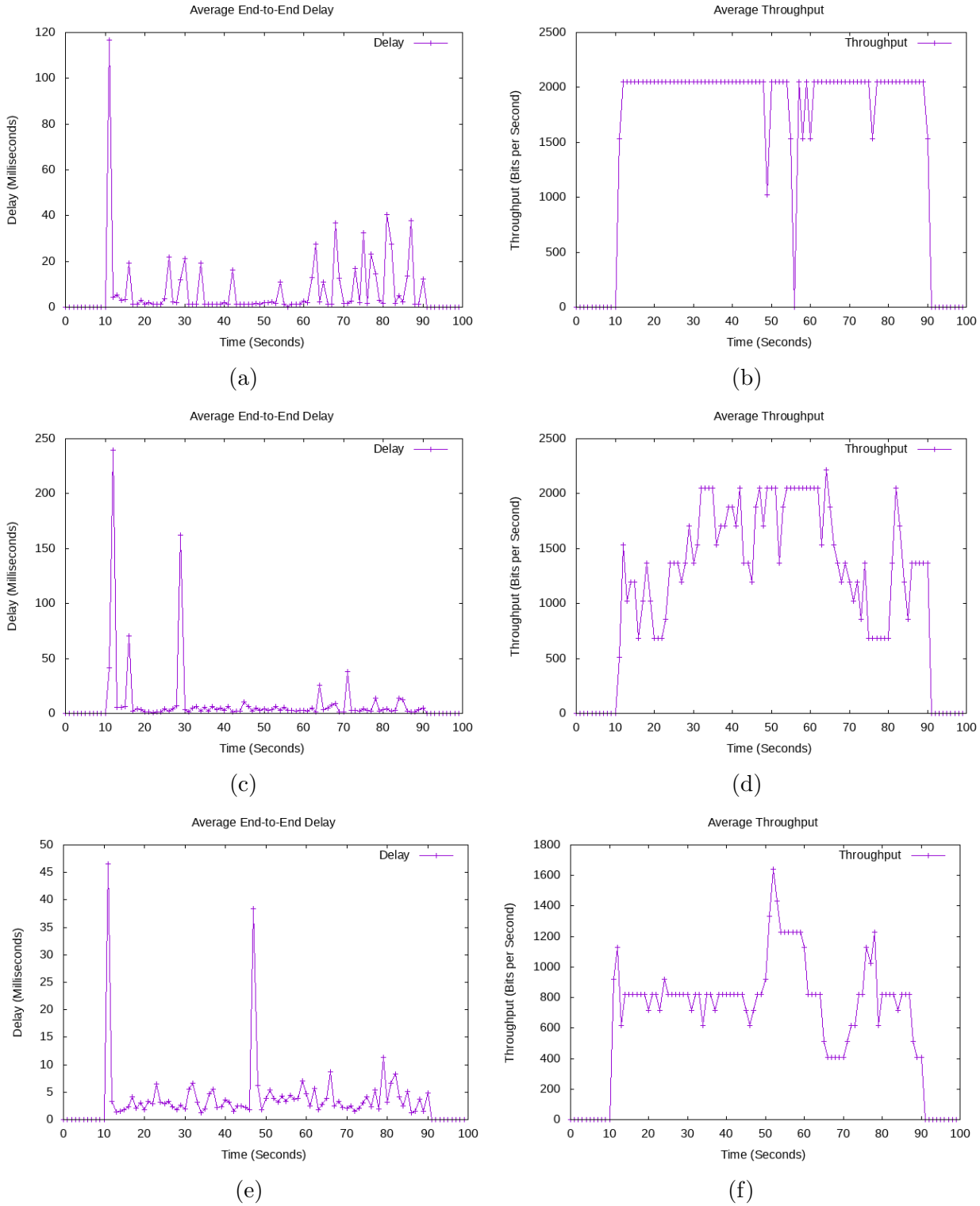


Figure 6.4: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with Mobile Nodes, Normal Speed, High Density and RandomWalk2D Mobility Model. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

High Speed and High Density

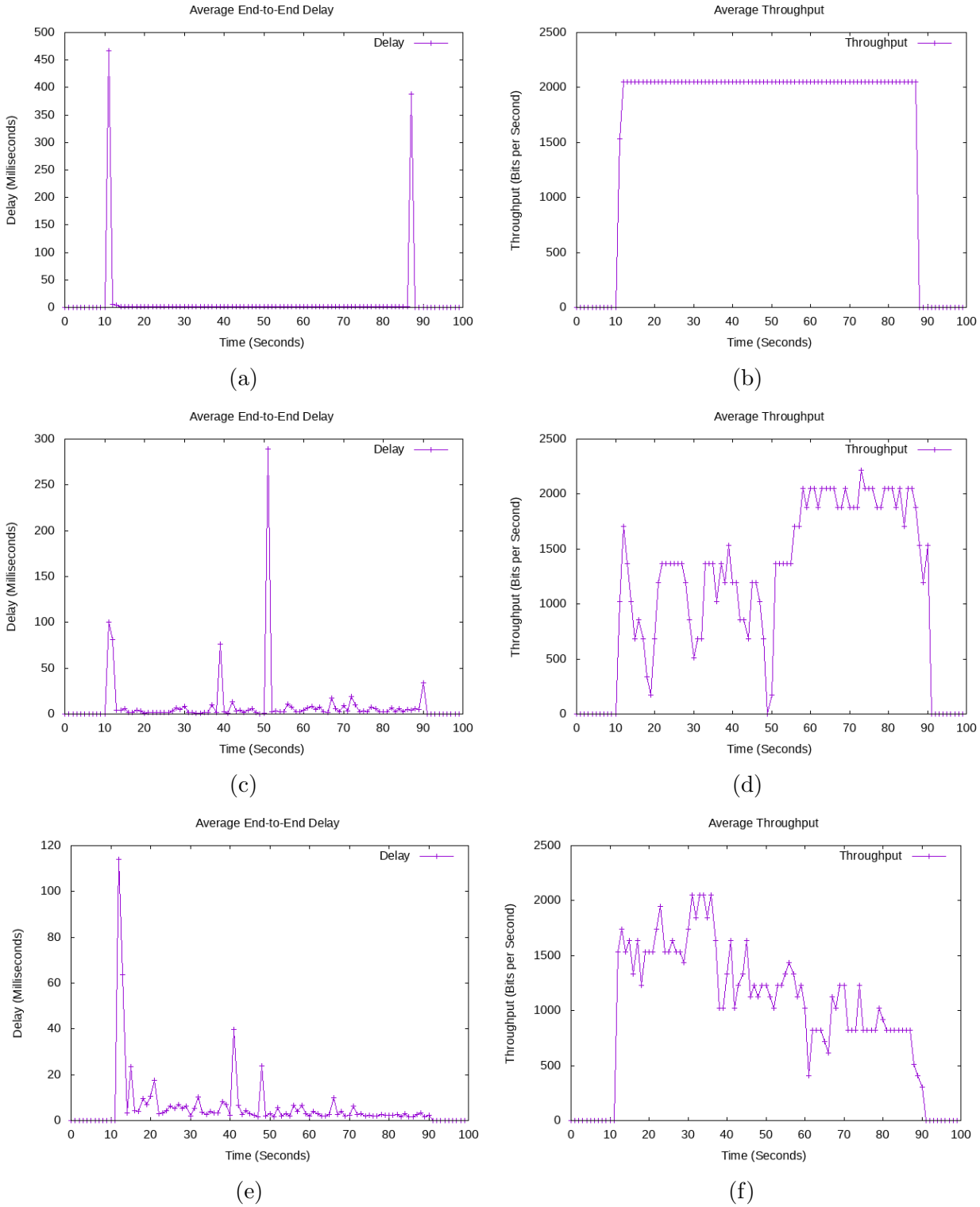


Figure 6.5: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with Mobile Nodes, High Speed Velocity, High Density and Random Walk 2D Mobility Model. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

High Speed and Normal Density

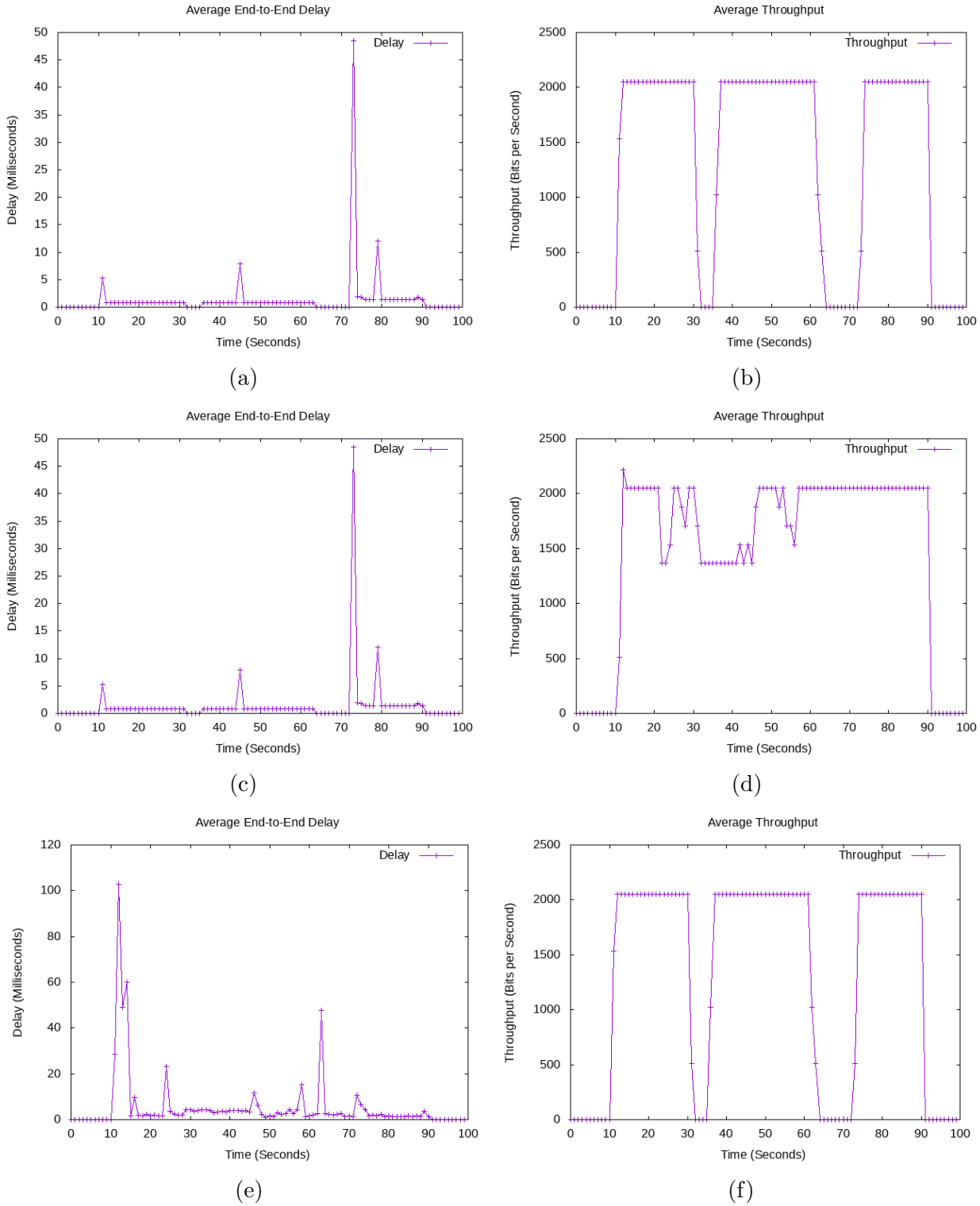


Figure 6.6: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with Mobile Nodes, High Speed, Normal Density and RandomWalk2D Mobility Model. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

6.2 Black Hole Attack: One Malicious node

Here we introduce a black hole node in our 5 x 5 node network evenly spaced 50m apart. To account for the varying effect of the black hole node due to its position in the network, we manually place the malicious node at the centre of the network depending on the initial position of the static or moving nodes when the simulation is run. This somewhat prevents us from running the same scenario multiple times to introduce some randomness and average out the effect of the black hole node on the Average Throughput and Average End-to-End Delay due to its position.

We simulate six different scenarios with each scenario running for one, three and five sending and receiving nodes. Then we plot the time evolution of the network Throughput and Delays. The attack starts at 50 seconds and ends at 80 seconds.

The six different scenarios include static nodes and Normal Density, Static Nodes and High Density, Normal Speed and Normal Density, Normal Speed and High Density, High Speed and Normal Density and High Speed and High Density. Please note that for the high speed, we use 50 m/s and for the high density we use 49 nodes instead of 25. Also, note that the as we increase the number of sending and receiving nodes for each scenario, the node size is constant.

Figures 6.7 through 6.8 show the results of a network with one malicious node and static nodes with a Constant Position mobility model and 1,3, and 5 source-sink pairs. We present a series of time evolution graphs for the Throughput and Average End-to-End Delay.

Figures 6.9 through 6.12 show the results of a network with one malicious node and mobile nodes with Random Walk 2D mobility models and 1,3, and 5 source-sink pairs. We present a series of time evolution graphs for the Throughput and Average End-to-End Delay.

Static Nodes and Normal Density

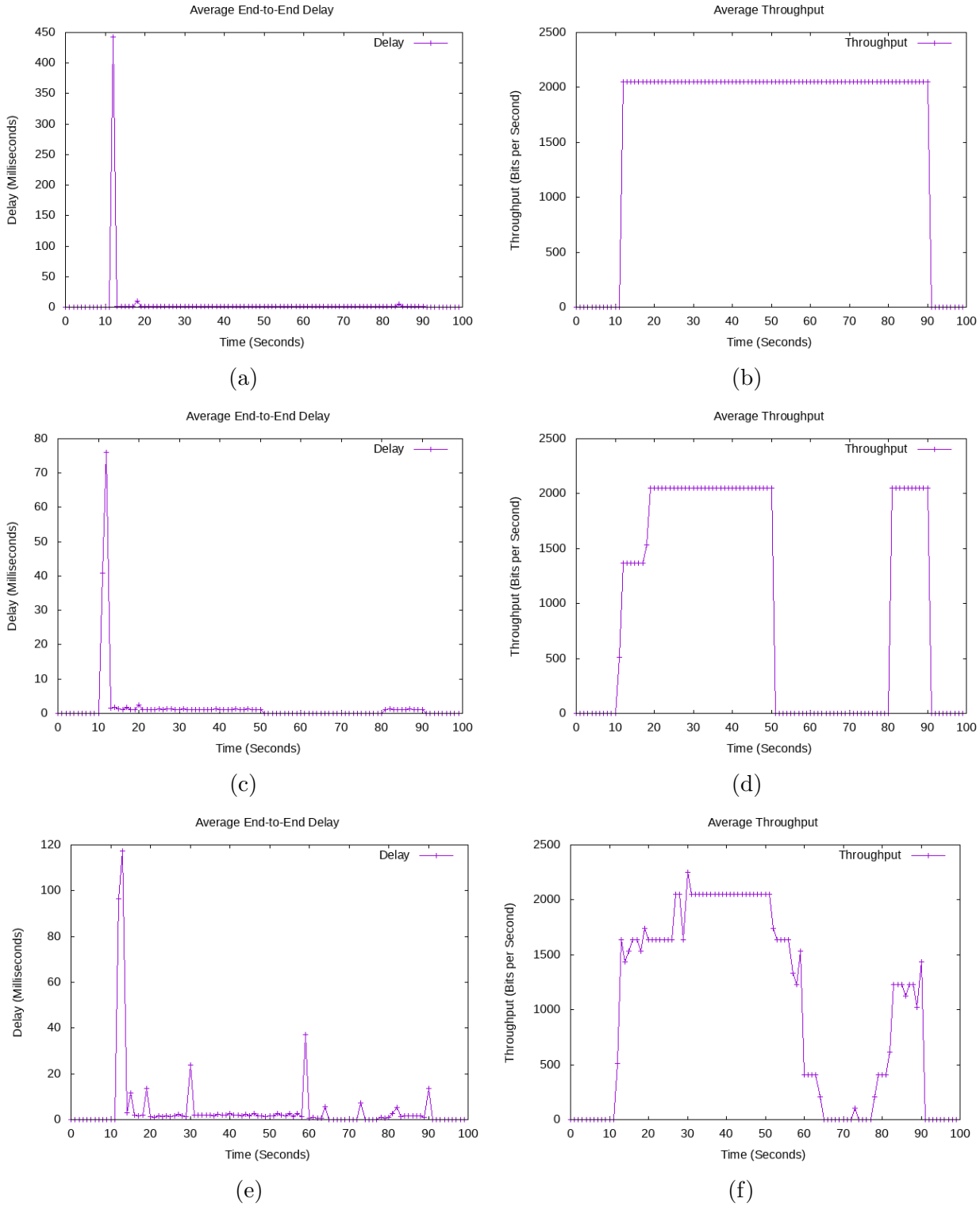


Figure 6.7: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with static Nodes, Normal Speed, Normal Density, Constant Mobility Model and one malicious node. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f).

Static Nodes and High Density

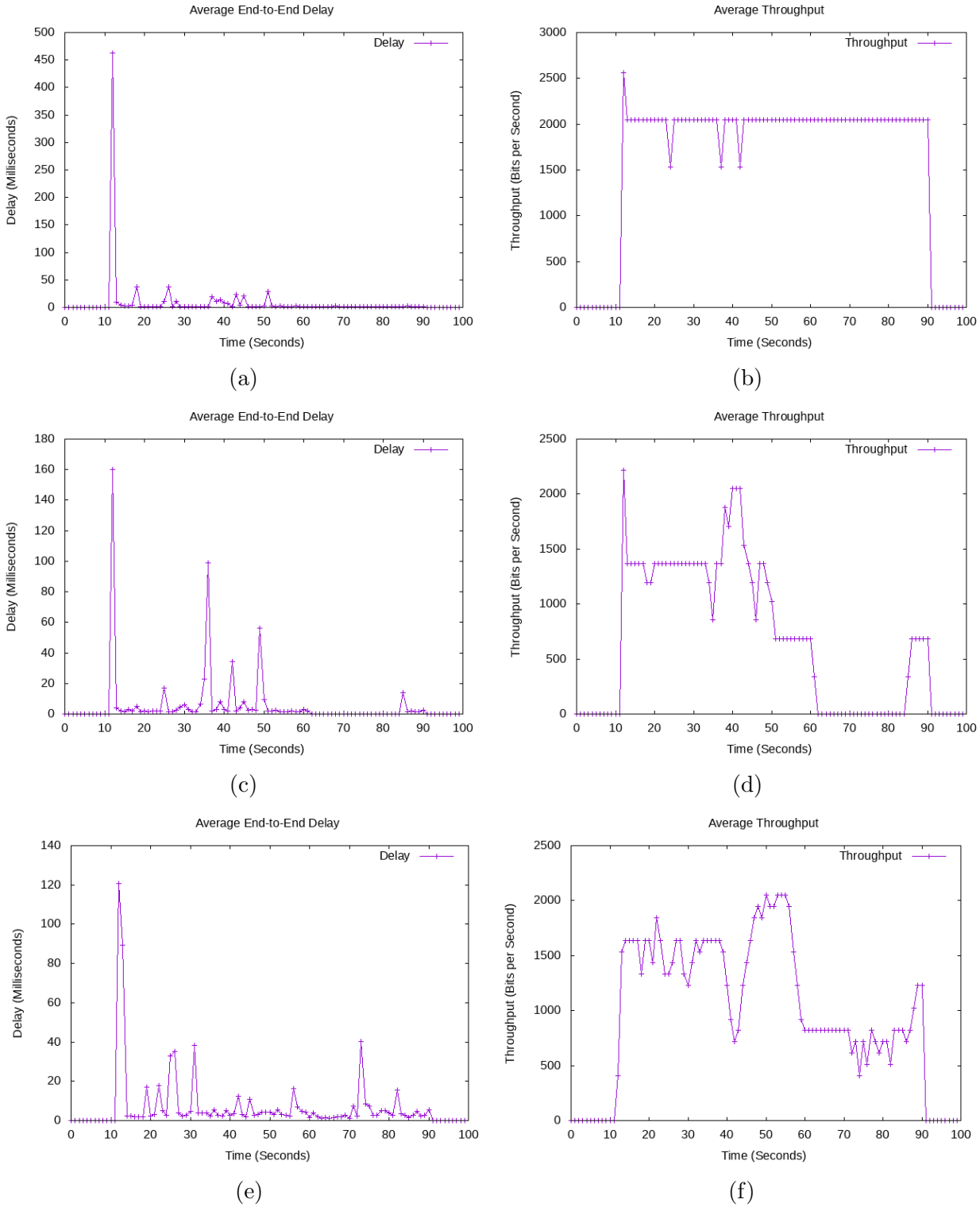


Figure 6.8: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with static Nodes, Normal Speed, High Density, Constant Mobility Model and one malicious node. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

Normal Speed and Normal Density

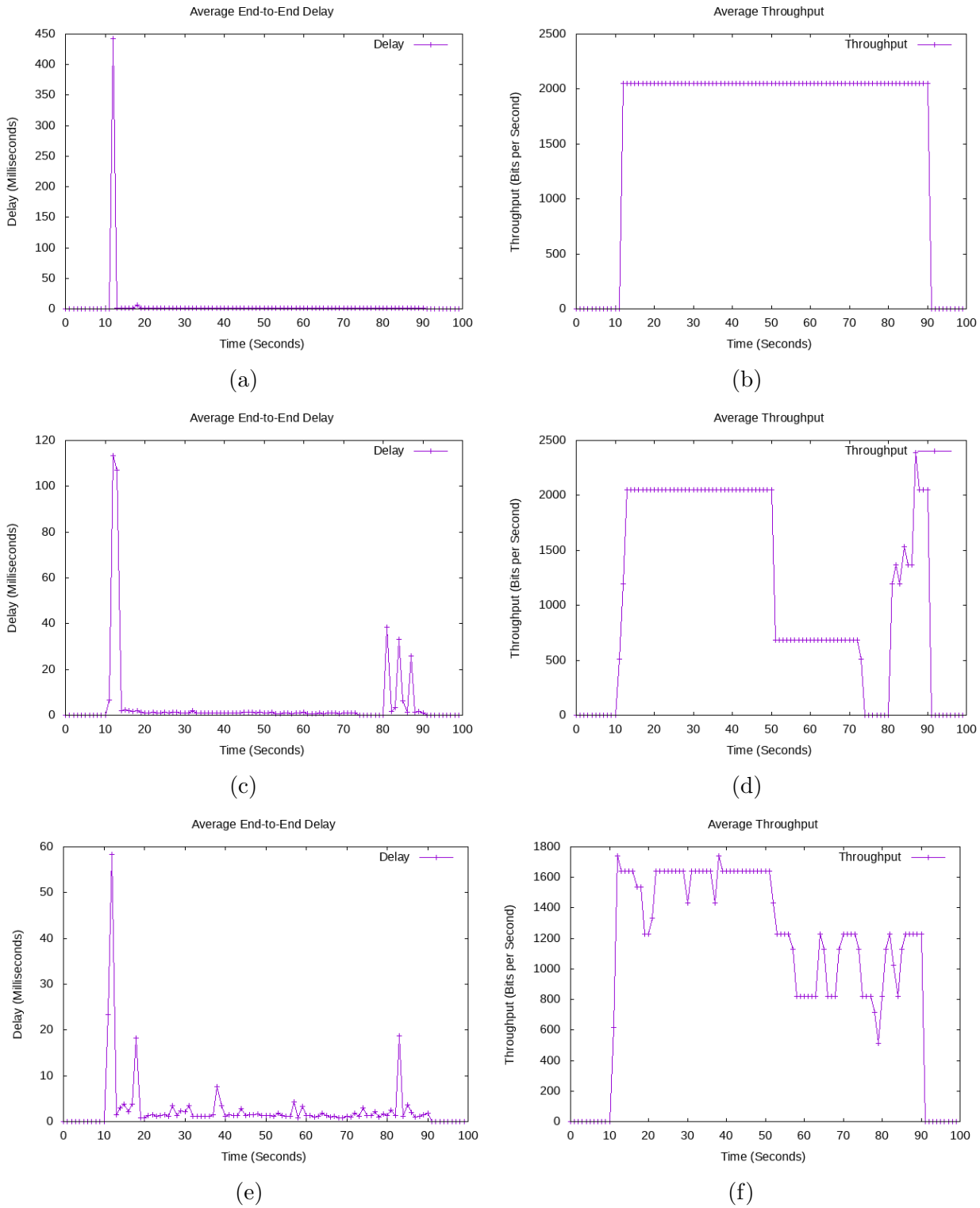


Figure 6.9: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with mobile Nodes, Normal Speed, Normal Density, RandomWalk 2D Mobility Model and one malicious node. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

Normal Speed and High Density

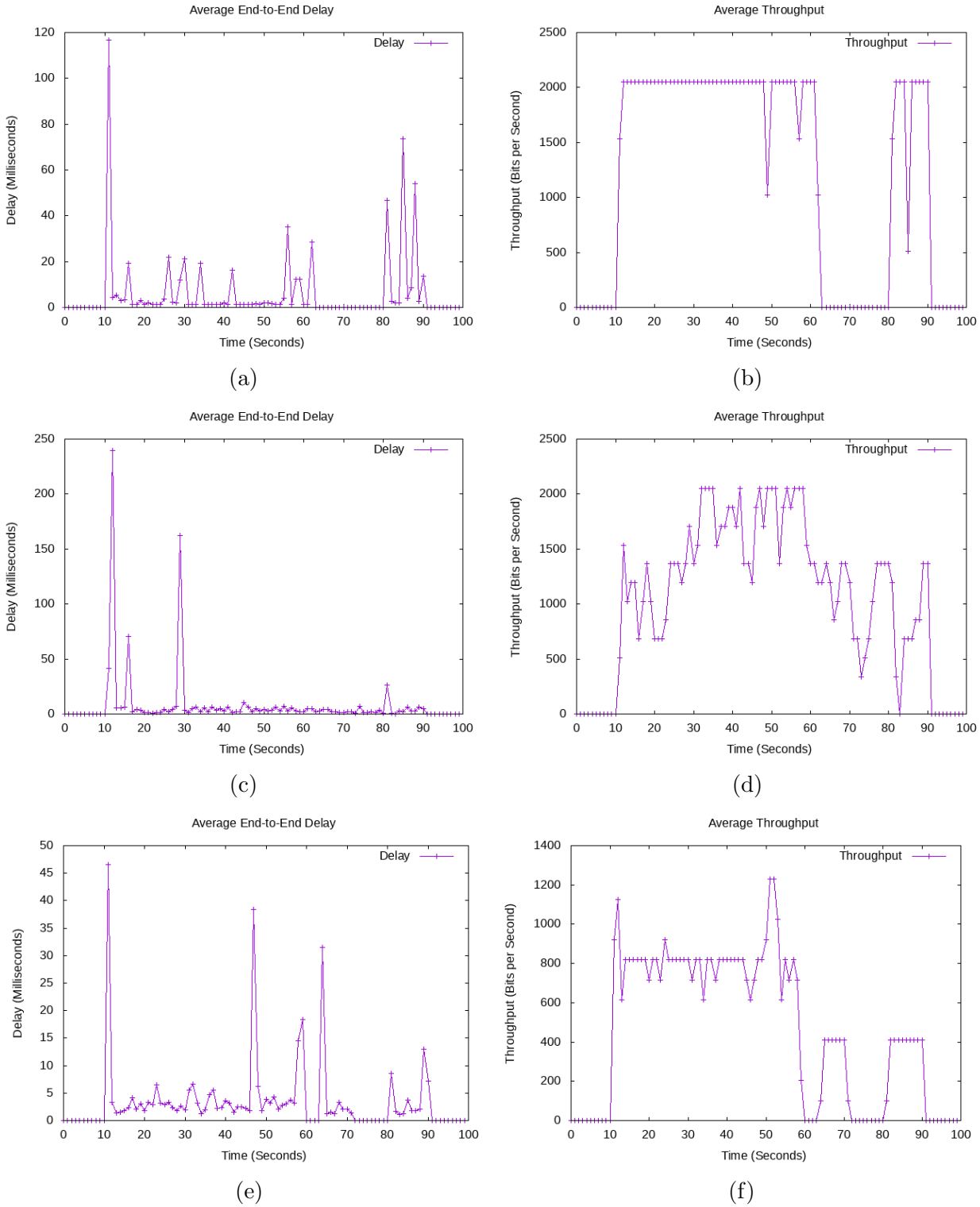


Figure 6.10: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with mobile Nodes, Normal Speed, High Density, RandomWalk 2D Mobility Model and one malicious node. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

High Speed and High Density

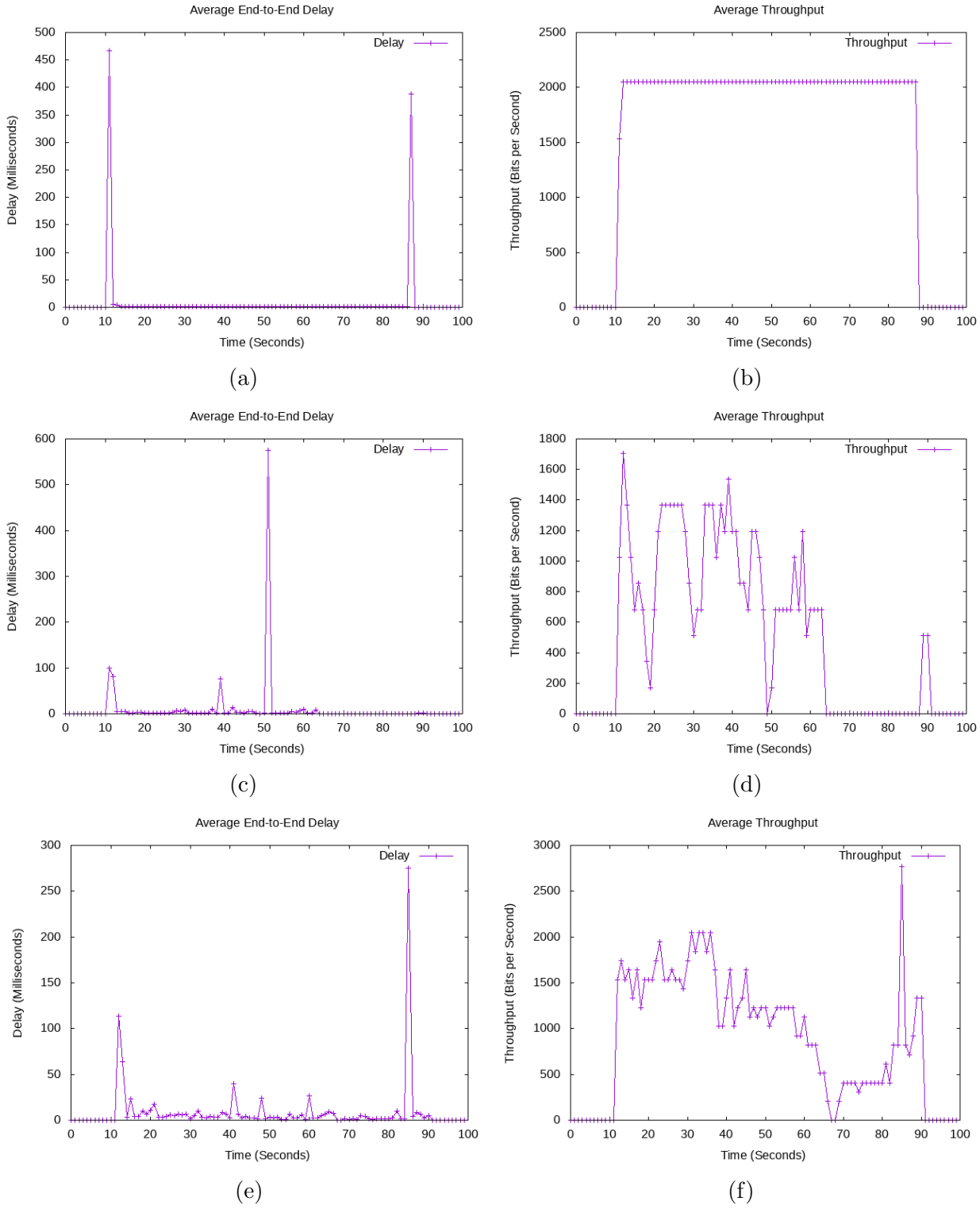


Figure 6.11: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with mobile Nodes, High Speed, High Density, RandomWalk 2D Mobility Model and one malicious node. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f).

High Speed and Normal Density

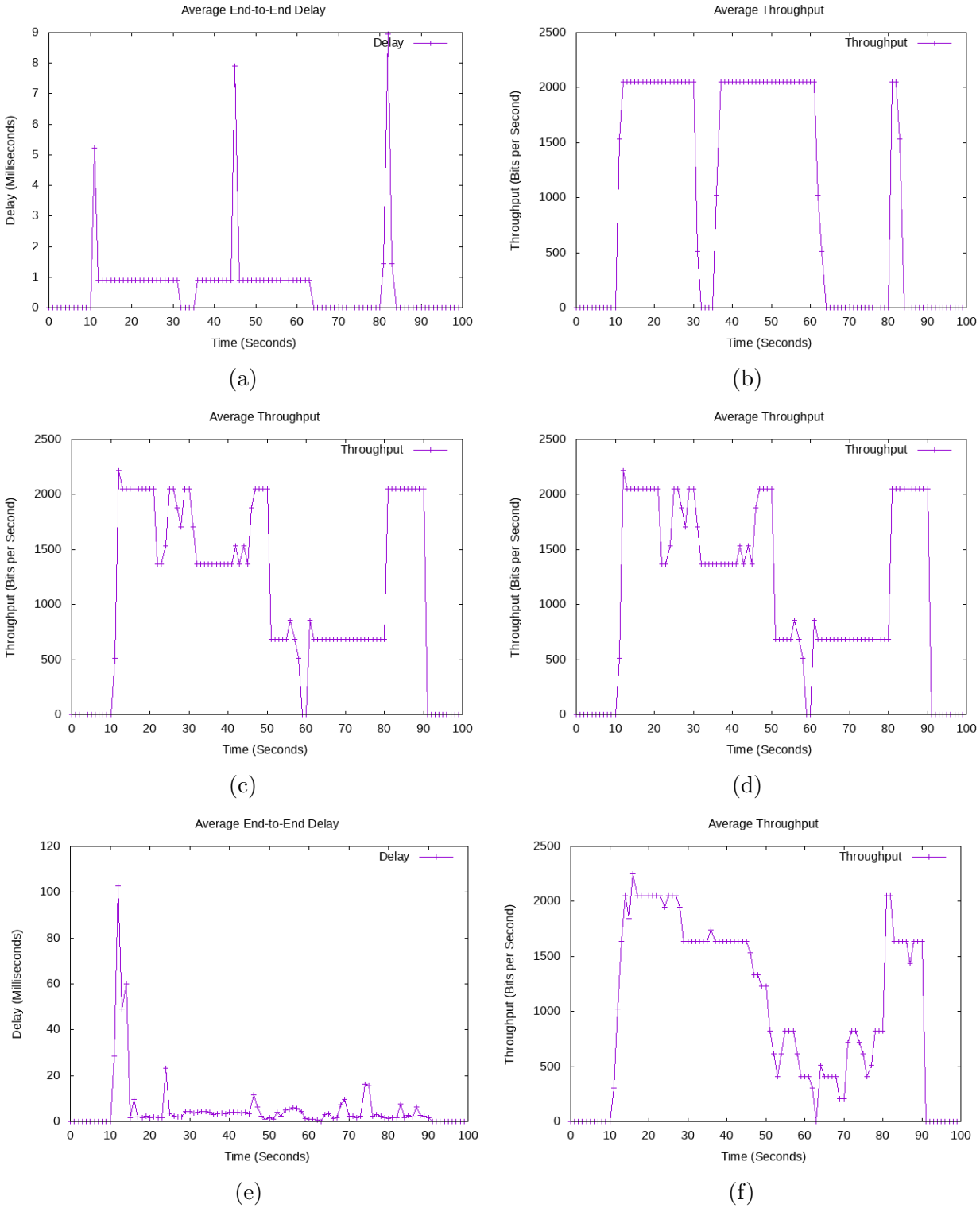


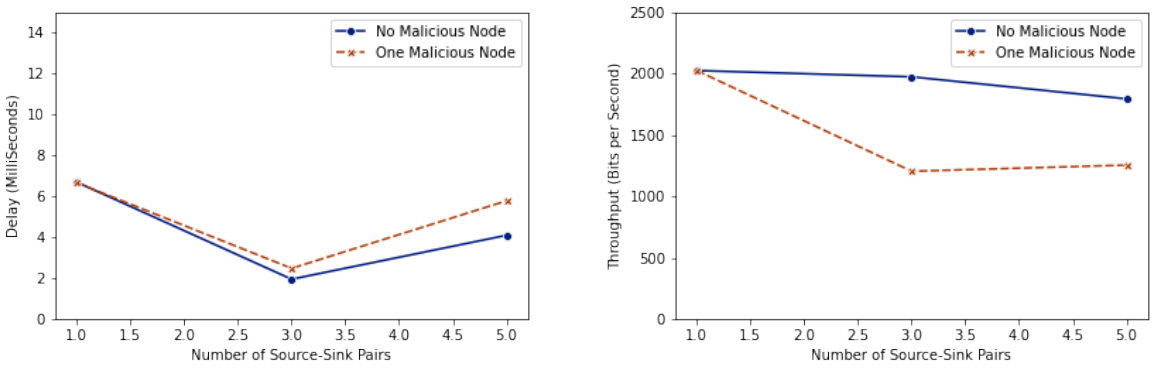
Figure 6.12: Time graph of Average End-to-End Delay and Average Throughput between Source-Sink Pairs with mobile Nodes, High Speed, Normal Density, RandomWalk 2D Mobility Model and one malicious node. One Source-Sink Pair (a) and (b), Three Source-Sink Pairs (c) and (d), Five Source-Sink Pairs (e) and (f)

Chapter 7

Result Summary and Discussion

7.0.1 Constant Position Mobility Model

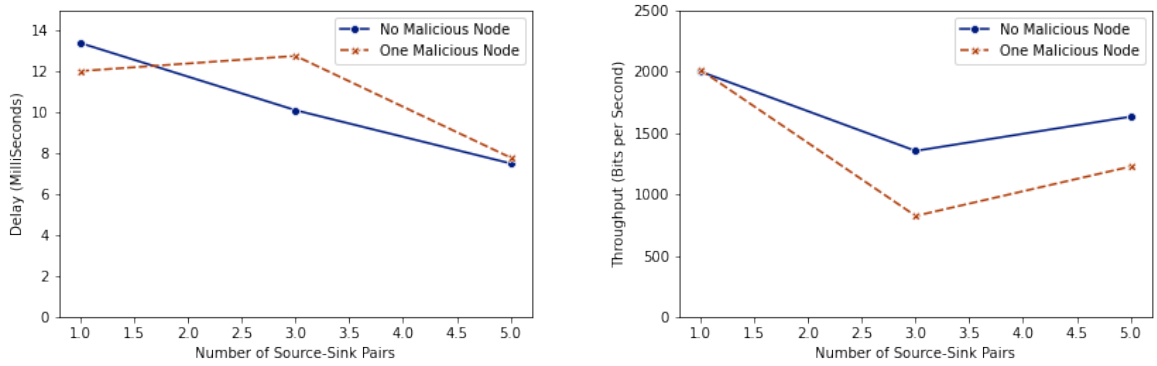
Constant Position Mobility Model: Normal Density



(a) (b)

Figure 7.1: Average End-to-End Delay (a) and Average Throughput (b) between One, Three and Five Source-Sink Pairs with Static Nodes, Normal Density and Constant Position Mobility Model

Constant Position Mobility Model: High Density

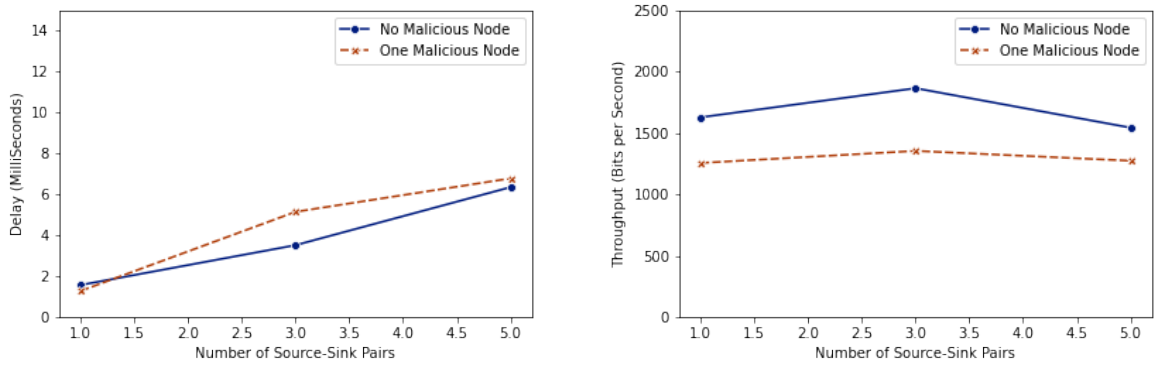


(a) (b)

Figure 7.2: Average End-to-End Delay (a) and Average Throughput (b) between One, Three and Five Source-Sink Pairs with Static Nodes, High Density and Constant Position Mobility Model

7.0.2 Random Walk 2D Mobility Model

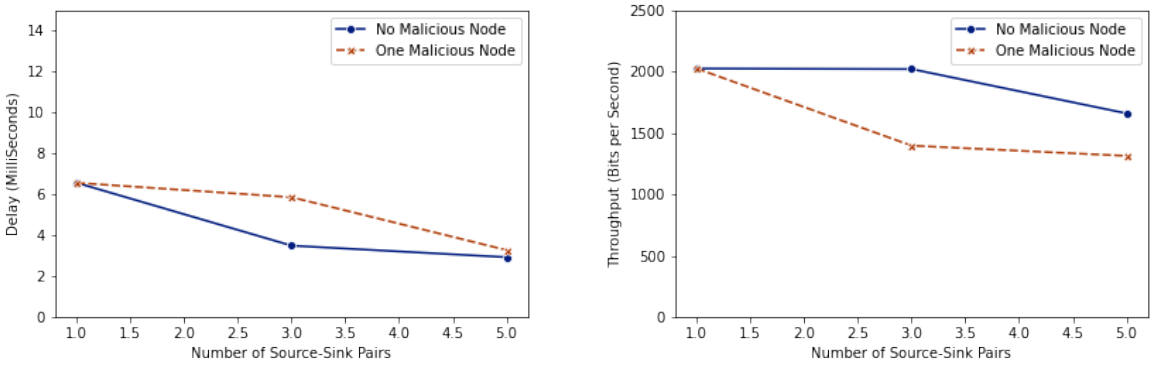
Random Walk 2D Mobility Model: Normal Speed



(a) (b)

Figure 7.3: Average End-to-End Delay (a) and Average Throughput (b) between One, Three and Five Source-Sink Pairs with Mobile Nodes, Normal Density, Normal Speed and Random Walk 2D Mobility Model

Random Walk 2D Mobility Model: High Speed



(a) (b)

Figure 7.4: Average End-to-End Delay (a) and Average Throughput (b) between One, Three and Five Source-Sink Pairs with Mobile Nodes, Normal Density, High Speed and Random Walk 2D Mobility Model

Result Discussion

From our simulation results, it is evident that a black hole reduces the Average Throughput and increases the Average End-to-End Delay of a mobile ad hoc network using ad hoc on-Demand Distance Vector protocol. This makes perfect sense as the sole objective of a black hole in a MANET is to drop packets. This leads to packet loss and hence, reduced throughput and increased delay since packets never reach their intended destination.

Also, the mobility of the network reduces the effect of a black hole attack in a network. This is probably due to the random movement of nodes and it is common knowledge that randomness generally increases efficiency. My thought is that when nodes are mobile, it is difficult for the malicious node to find itself in favourable positions to the sending node as frequently as it does when the nodes are static.

Another point worth noting is that the mobility model, speed and density of the network affect its performance of the network. In our case and based on our simulation and results, using a randomwaypoint2D mobility model and increasing the speed of the nodes seemed to reduce the effect of the black hole attack. In most general cases, this holds true except for cases of abnormal speed which could introduce some sort of reverse effect on the network.

Finally, it is worth noting that from our simulation and experimental setup, we maintained a constant density as we increased the number of sending and receiving nodes in the network.

Chapter 8

Conclusion

The inherent security challenges of mobile ad hoc networks are still an open problem and should be studied, and researched and new countermeasures should be proposed to mitigate attacks at all layers in MANETs, especially the network layer.

In this project, we analyzed a security threat faced at the network layer in mobile ad hoc networks: A black hole attack. We have implemented a Black hole attack using AODV protocol in Network Simulator -3.33. We have analyzed the performance of this network under various scenarios involving node mobility, node density, node speed and under a black hole attack. We showed how these attacks affect the network throughput and delay using the time graph for the Average Throughput and Average End-to-End Delay.

From our results and discussion in the preceding chapter, we can conclude that a black hole can reduce the average throughput in a network and increase the average end-to-end delay. Secondly, The mobility of nodes can reduce the effect of a black hole attack. Finally, The mobility model, density and speed of nodes can affect the average throughput and end-to-end delay in MANETs.

Chapter 9

Future Work

In this project, we simulated mobile nodes using the Random Walk2D mobility model. It will be interesting to simulate random direction 2D, and Random waypoint mobility models and see how these models compare to the Random Walk2D mobility model in terms of reaction to black hole node, average throughput and end-to-end delay. Also, besides average throughput and end-to-end delay as our performance metrics, more network performance metrics could be explored such as packet loss, receive rate and packet delivery ratio among others.

It would also be worthwhile to investigate the effects of certain parameters such as the data rate, packet size, bit rate and loss model to gain a more in-depth understanding of how all these different pieces connect together and relate as a whole. Who knows, one may discover something or more optimal parameters. Another important thing to do would be to run the simulation for a longer duration like say 200 seconds or up until a steady state is achieved. Then, see how it compares to our results. In the project, we ran each scenario for 100 seconds due to multiple scenarios being compared and evaluated. Finally, it will be good to simulate using a different protocol such as OLSR or DSDV and compare results. It is possible that one protocol may perform better than the other using a similar simulation scenario and under a black hole attack or any other network layer attack.

Appendix A

Modified AODV Header File

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2009 IITP RAS
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Based on
 *      NS-2 AODV model developed by the CMU/MONARCH group and optimized and
 *      tuned by Samir Das and Mahesh Marina, University of Cincinnati;
 *
 *      AODV-UU implementation by Erik Nordstr m of Uppsala University
 *      http://core.it.uu.se/core/index.php/AODV-UU
 *
 * Authors: Elena Buchatskaia <borovkovaes@iitp.ru>
 *          Pavel Boyko <boyko@iitp.ru>
```

```
 */
#ifndef AODVROUTINGPROTOCOL_H
#define AODVROUTINGPROTOCOL_H

#include "aodv-rtable.h"
#include "aodv-rqueue.h"
#include "aodv-packet.h"
#include "aodv-neighbor.h"
#include "aodv-dpd.h"
#include "ns3/node.h"
#include "ns3/random-variable-stream.h"
#include "ns3/output-stream-wrapper.h"
#include "ns3/ipv4-routing-protocol.h"
#include "ns3/ipv4-interface.h"
#include "ns3/ipv4-l3-protocol.h"
#include <map>

namespace ns3 {

class WifiMacQueueItem;
enum WifiMacDropReason : uint8_t;  // opaque enum declaration

namespace aodv {
/**
 * \ingroup aodv
 *
 * \brief AODV routing protocol
 */
class RoutingProtocol : public Ipv4RoutingProtocol
{
public:
  /**
   * \brief Get the type ID.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);
  static const uint32_t AODV_PORT;

  /// constructor
  RoutingProtocol ();
  virtual ~RoutingProtocol ();
```

```
virtual void DoDispose ();

// Inherited from Ipv4RoutingProtocol
Ptr<Ipv4Route> RouteOutput (Ptr<Packet> p, const Ipv4Header &header, Ptr<NetDevice> oif,
    Socket::SocketErrno &sockerr);
bool RouteInput (Ptr<const Packet> p, const Ipv4Header &header, Ptr<const NetDevice> idev
    ,
                 UnicastForwardCallback ucb, MulticastForwardCallback mcb,
                 LocalDeliverCallback lcb, ErrorCallback ecb);
virtual void NotifyInterfaceUp (uint32_t interface);
virtual void NotifyInterfaceDown (uint32_t interface);
virtual void NotifyAddAddress (uint32_t interface, Ipv4InterfaceAddress address);
virtual void NotifyRemoveAddress (uint32_t interface, Ipv4InterfaceAddress address);
virtual void SetIpv4 (Ptr<Ipv4> ipv4);
virtual void PrintRoutingTable (Ptr<OutputStreamWrapper> stream, Time::Unit unit = Time::
    S) const;

// Handle protocol parameters
/**
 * Get maximum queue time
 * \returns the maximum queue time
 */
Time GetMaxQueueTime () const
{
  return m_maxQueueTime;
}
/**
 * Set the maximum queue time
 * \param t the maximum queue time
 */
void SetMaxQueueTime (Time t);
/**
 * Get the maximum queue length
 * \returns the maximum queue length
 */
uint32_t GetMaxQueueLen () const
{
  return m_maxQueueLen;
}
/**
 * Set the maximum queue length
```

```
   * \param len the maximum queue length
   */
  void SetMaxQueueLen (uint32_t len);
  /**
   * Get destination only flag
   * \returns the destination only flag
   */
  bool GetDestinationOnlyFlag () const
  {
    return m_destinationOnly;
  }
  /**
   * Set destination only flag
   * \param f the destination only flag
   */
  void SetDestinationOnlyFlag (bool f)
  {
    m_destinationOnly = f;
  }
  /**
   * Get gratuitous reply flag
   * \returns the gratuitous reply flag
   */
  bool GetGratuitousReplyFlag () const
  {
    return m_gratuitousReply;
  }
  /**
   * Set gratuitous reply flag
   * \param f the gratuitous reply flag
   */
  void SetGratuitousReplyFlag (bool f)
  {
    m_gratuitousReply = f;
  }
  /**
   * Set hello enable
   * \param f the hello enable flag
   */
  void SetHelloEnable (bool f)
  {
```

```
  m_enableHello = f;
}
/**
 * Get hello enable flag
 * \returns the enable hello flag
 */
bool GetHelloEnable () const
{
  return m_enableHello;
}
/**
 * Set broadcast enable flag
 * \param f enable broadcast flag
 */
void SetBroadcastEnable (bool f)
{
  m_enableBroadcast = f;
}
/**
 * Get broadcast enable flag
 * \returns the broadcast enable flag
 */
bool GetBroadcastEnable () const
{
  return m_enableBroadcast;
}

void SetMaliciousEnable (bool f) { IsMalicious = f; }                          // Method
    declared for Blackhole Attack Simulation - Shalini Satre
bool GetMaliciousEnable () const { return IsMalicious; }                       // Method
    declared for Blackhole Attack Simulation - Shalini Satre

/**
 * Assign a fixed random variable stream number to the random variables
 * used by this model.  Return the number of streams (possibly zero) that
 * have been assigned.
 *
 * \param stream first stream index to use
 * \return the number of stream indices assigned by this model
 */
int64_t AssignStreams (int64_t stream);
```

```
protected:
  virtual void DoInitialize (void);
private:
  /**
   * Notify that an MPDU was dropped.
   *
   * \param reason the reason why the MPDU was dropped
   * \param mpdu the dropped MPDU
   */
  void NotifyTxError (WifiMacDropReason reason, Ptr<const WifiMacQueueItem> mpdu);

  // Protocol parameters.
  uint32_t m_rreqRetries;             ///< Maximum number of retransmissions of RREQ with
      TTL = NetDiameter to discover a route
  uint16_t m_ttlStart;                ///< Initial TTL value for RREQ.
  uint16_t m_ttlIncrement;            ///< TTL increment for each attempt using the
      expanding ring search for RREQ dissemination.
  uint16_t m_ttlThreshold;            ///< Maximum TTL value for expanding ring search, TTL
       = NetDiameter is used beyond this value.
  uint16_t m_timeoutBuffer;           ///< Provide a buffer for the timeout.
  uint16_t m_rreqRateLimit;           ///< Maximum number of RREQ per second.
  uint16_t m_rerrRateLimit;           ///< Maximum number of REER per second.
  Time m_activeRouteTimeout;          ///< Period of time during which the route is
      considered to be valid.
  uint32_t m_netDiameter;             ///< Net diameter measures the maximum possible
      number of hops between two nodes in the network
  /**
   *  NodeTraversalTime is a conservative estimate of the average one hop traversal time
       for packets
   *  and should include queuing delays, interrupt processing times and transfer times.
   */
  Time m_nodeTraversalTime;
  Time m_netTraversalTime;             ///< Estimate of the average net traversal time.
  Time m_pathDiscoveryTime;            ///< Estimate of maximum time needed to find route
      in network.
  Time m_myRouteTimeout;               ///< Value of lifetime field in RREP generating by
      this node.
  /**
   * Every HelloInterval the node checks whether it has sent a broadcast  within the last
       HelloInterval.
```

```
 * If it has not, it MAY broadcast a  Hello message
 */
Time m_helloInterval;
uint32_t m_allowedHelloLoss;         ///< Number of hello messages which may be loss for
    valid link
/**
 * DeletePeriod is intended to provide an upper bound on the time for which an upstream
     node A
 * can have a neighbor B as an active next hop for destination D, while B has invalidated
      the route to D.
 */
Time m_deletePeriod;
Time m_nextHopWait;                  ///< Period of our waiting for the neighbour's
    RREP_ACK
Time m_blackListTimeout;             ///< Time for which the node is put into the
    blacklist
uint32_t m_maxQueueLen;              ///< The maximum number of packets that we allow a
    routing protocol to buffer.
Time m_maxQueueTime;                 ///< The maximum period of time that a routing
    protocol is allowed to buffer a packet for.
bool m_destinationOnly;              ///< Indicates only the destination may respond to
    this RREQ.
bool m_gratuitousReply;              ///< Indicates whether a gratuitous RREP should be
    unicast to the node originated route discovery.
bool m_enableHello;                  ///< Indicates whether a hello messages enable
bool m_enableBroadcast;              ///< Indicates whether a a broadcast data packets
    forwarding enable
//\}

/// IP protocol
Ptr<Ipv4> m_ipv4;
/// Raw unicast socket per each IP interface, map socket -> iface address (IP + mask)
std::map< Ptr<Socket>, Ipv4InterfaceAddress > m_socketAddresses;
/// Raw subnet directed broadcast socket per each IP interface, map socket -> iface
    address (IP + mask)
std::map< Ptr<Socket>, Ipv4InterfaceAddress > m_socketSubnetBroadcastAddresses;
/// Loopback device used to defer RREQ until packet will be fully formed
Ptr<NetDevice> m_lo;

/// Routing table
RoutingTable m_routingTable;
```

```
  /// A "drop-front" queue used by the routing layer to buffer packets to which it does not
       have a route.
  RequestQueue m_queue;
  /// Broadcast ID
  uint32_t m_requestId;
  /// Request sequence number
  uint32_t m_seqNo;
  /// Handle duplicated RREQ
  IdCache m_rreqIdCache;
  /// Handle duplicated broadcast/multicast packets
  DuplicatePacketDetection m_dpd;
  /// Handle neighbors
  Neighbors m_nb;
  /// Number of RREQs used for RREQ rate control
  uint16_t m_rreqCount;
  /// Number of RERRs used for RERR rate control
  uint16_t m_rerrCount;
  /// Set node as malicious. Dropping every packet received.
  bool IsMalicious;                                          // Variable declared for
      Blackhole Attack Simulation - Shalini Satre


private:
  /// Start protocol operation
  void Start ();
  /**
   * Queue packet and send route request
   *
   * \param p the packet to route
   * \param header the IP header
   * \param ucb the UnicastForwardCallback function
   * \param ecb the ErrorCallback function
   */
  void DeferredRouteOutput (Ptr<const Packet> p, const Ipv4Header & header,
      UnicastForwardCallback ucb, ErrorCallback ecb);
  /**
   * If route exists and is valid, forward packet.
   *
   * \param p the packet to route
   * \param header the IP header
   * \param ucb the UnicastForwardCallback function
```

```
   * \param ecb the ErrorCallback function
   * \returns true if forwarded
   */
  bool Forwarding (Ptr<const Packet> p, const Ipv4Header & header, UnicastForwardCallback
      ucb, ErrorCallback ecb);
  /**
   * Repeated attempts by a source node at route discovery for a single destination
   * use the expanding ring search technique.
   * \param dst the destination IP address
   */
  void ScheduleRreqRetry (Ipv4Address dst);
  /**
   * Set lifetime field in routing table entry to the maximum of existing lifetime and lt,
       if the entry exists
   * \param addr - destination address
   * \param lt - proposed time for lifetime field in routing table entry for destination
       with address addr.
   * \return true if route to destination address addr exist
   */
  bool UpdateRouteLifeTime (Ipv4Address addr, Time lt);
  /**
   * Update neighbor record.
   * \param receiver is supposed to be my interface
   * \param sender is supposed to be IP address of my neighbor.
   */
  void UpdateRouteToNeighbor (Ipv4Address sender, Ipv4Address receiver);
  /**
   * Test whether the provided address is assigned to an interface on this node
   * \param src the source IP address
   * \returns true if the IP address is the node's IP address
   */
  bool IsMyOwnAddress (Ipv4Address src);
  /**
   * Find unicast socket with local interface address iface
   *
   * \param iface the interface
   * \returns the socket associated with the interface
   */
  Ptr<Socket> FindSocketWithInterfaceAddress (Ipv4InterfaceAddress iface) const;
  /**
   * Find subnet directed broadcast socket with local interface address iface
```

```
 *
 * \param iface the interface
 * \returns the socket associated with the interface
 */
Ptr<Socket> FindSubnetBroadcastSocketWithInterfaceAddress (Ipv4InterfaceAddress iface)
    const;
/**
 * Process hello message
 *
 * \param rrepHeader RREP message header
 * \param receiverIfaceAddr receiver interface IP address
 */
void ProcessHello (RrepHeader const & rrepHeader, Ipv4Address receiverIfaceAddr);
/**
 * Create loopback route for given header
 *
 * \param header the IP header
 * \param oif the output interface net device
 * \returns the route
 */
Ptr<Ipv4Route> LoopbackRoute (const Ipv4Header & header, Ptr<NetDevice> oif) const;

///\name Receive control packets
//\{
/**
 * Receive and process control packet
 * \param socket input socket
 */
void RecvAodv (Ptr<Socket> socket);
/**
 * Receive RREQ
 * \param p packet
 * \param receiver receiver address
 * \param src sender address
 */
void RecvRequest (Ptr<Packet> p, Ipv4Address receiver, Ipv4Address src);
/**
 * Receive RREP
 * \param p packet
 * \param my destination address
 * \param src sender address
```

```
 */
void RecvReply (Ptr<Packet> p, Ipv4Address my, Ipv4Address src);
/**
 * Receive RREP_ACK
 * \param neighbor neighbor address
 */
void RecvReplyAck (Ipv4Address neighbor);
/**
 * Receive RERR
 * \param p packet
 * \param src sender address
 */
/// Receive  from node with address src
void RecvError (Ptr<Packet> p, Ipv4Address src);
//\}

///\name Send
//\{
/** Forward packet from route request queue
 * \param dst destination address
 * \param route route to use
 */
void SendPacketFromQueue (Ipv4Address dst, Ptr<Ipv4Route> route);
/// Send hello
void SendHello ();
/** Send RREQ
 * \param dst destination address
 */
void SendRequest (Ipv4Address dst);
/** Send RREP
 * \param rreqHeader route request header
 * \param toOrigin routing table entry to originator
 */
void SendReply (RreqHeader const & rreqHeader, RoutingTableEntry const & toOrigin);
/** Send RREP by intermediate node
 * \param toDst routing table entry to destination
 * \param toOrigin routing table entry to originator
 * \param gratRep indicates whether a gratuitous RREP should be unicast to destination
 */
void SendReplyByIntermediateNode (RoutingTableEntry & toDst, RoutingTableEntry & toOrigin
    , bool gratRep);
```

```
/** Send RREP_ACK
 * \param neighbor neighbor address
 */
void SendReplyAck (Ipv4Address neighbor);
/** Initiate RERR
 * \param nextHop next hop address
 */
void SendRerrWhenBreaksLinkToNextHop (Ipv4Address nextHop);
/** Forward RERR
 * \param packet packet
 * \param precursors list of addresses of the visited nodes
 */
void SendRerrMessage (Ptr<Packet> packet,  std::vector<Ipv4Address> precursors);
/**
 * Send RERR message when no route to forward input packet. Unicast if there is reverse
     route to originating node, broadcast otherwise.
 * \param dst - destination node IP address
 * \param dstSeqNo - destination node sequence number
 * \param origin - originating node IP address
 */
void SendRerrWhenNoRouteToForward (Ipv4Address dst, uint32_t dstSeqNo, Ipv4Address origin
    );
/// @}

/**
 * Send packet to destination scoket
 * \param socket - destination node socket
 * \param packet - packet to send
 * \param destination - destination node IP address
 */
void SendTo (Ptr<Socket> socket, Ptr<Packet> packet, Ipv4Address destination);

/// Hello timer
Timer m_htimer;
/// Schedule next send of hello message
void HelloTimerExpire ();
/// RREQ rate limit timer
Timer m_rreqRateLimitTimer;
/// Reset RREQ count and schedule RREQ rate limit timer with delay 1 sec.
void RreqRateLimitTimerExpire ();
/// RERR rate limit timer
```

```
  Timer m_rerrRateLimitTimer;
  /// Reset RERR count and schedule RERR rate limit timer with delay 1 sec.
  void RerrRateLimitTimerExpire ();
  /// Map IP address + RREQ timer.
  std::map<Ipv4Address, Timer> m_addressReqTimer;
  /**
   * Handle route discovery process
   * \param dst the destination IP address
   */
  void RouteRequestTimerExpire (Ipv4Address dst);
  /**
   * Mark link to neighbor node as unidirectional for blacklistTimeout
   *
   * \param neighbor the IP address of the neightbor node
   * \param blacklistTimeout the black list timeout time
   */
  void AckTimerExpire (Ipv4Address neighbor, Time blacklistTimeout);

  /// Provides uniform random variables.
  Ptr<UniformRandomVariable> m_uniformRandomVariable;
  /// Keep track of the last bcast time
  Time m_lastBcastTime;
};

} //namespace aodv
} //namespace ns3

#endif /* AODVROUTINGPROTOCOL_H */
```

Appendix B

Modified AODV Source File

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2009 IITP RAS
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Based on
 *      NS-2 AODV model developed by the CMU/MONARCH group and optimized and
 *      tuned by Samir Das and Mahesh Marina, University of Cincinnati;
 *
 *      AODV-UU implementation by Erik Nordstr m of Uppsala University
 *      http://core.it.uu.se/core/index.php/AODV-UU
 *
 * Authors: Elena Buchatskaia <borovkovaes@iitp.ru>
 *          Pavel Boyko <boyko@iitp.ru>
 */
```

```
#define NS_LOG_APPEND_CONTEXT                                           \
  if (m_ipv4) { std::clog << "[node " << m_ipv4->GetObject<Node> ()->GetId () << "] "; }

#include "aodv-routing-protocol.h"
#include "ns3/log.h"
#include "ns3/boolean.h"
#include "ns3/random-variable-stream.h"
#include "ns3/inet-socket-address.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/udp-socket-factory.h"
#include "ns3/udp-l4-protocol.h"
#include "ns3/udp-header.h"
#include "ns3/wifi-net-device.h"
#include "ns3/adhoc-wifi-mac.h"
#include "ns3/wifi-mac-queue-item.h"
#include "ns3/string.h"
#include "ns3/pointer.h"
#include <algorithm>
#include <limits>

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("AodvRoutingProtocol");

namespace aodv {
NS_OBJECT_ENSURE_REGISTERED (RoutingProtocol);

/// UDP Port for AODV control traffic
const uint32_t RoutingProtocol::AODV_PORT = 654;

/**
* \ingroup aodv
* \brief Tag used by AODV implementation
*/
class DeferredRouteOutputTag : public Tag
{

public:
  /**
   * \brief Constructor
   * \param o the output interface
```

```
 */
DeferredRouteOutputTag (int32_t o = -1) : Tag (),
                                          m_oif (o)
{
}

/**
 * \brief Get the type ID.
 * \return the object TypeId
 */
static TypeId GetTypeId ()
{
  static TypeId tid = TypeId ("ns3::aodv::DeferredRouteOutputTag")
    .SetParent<Tag> ()
    .SetGroupName ("Aodv")
    .AddConstructor<DeferredRouteOutputTag> ()
  ;
  return tid;
}

TypeId  GetInstanceTypeId () const
{
  return GetTypeId ();
}

/**
 * \brief Get the output interface
 * \return the output interface
 */
int32_t GetInterface () const
{
  return m_oif;
}

/**
 * \brief Set the output interface
 * \param oif the output interface
 */
void SetInterface (int32_t oif)
{
  m_oif = oif;
```

```
  }

  uint32_t GetSerializedSize () const
  {
    return sizeof(int32_t);
  }

  void  Serialize (TagBuffer i) const
  {
    i.WriteU32 (m_oif);
  }

  void  Deserialize (TagBuffer i)
  {
    m_oif = i.ReadU32 ();
  }

  void  Print (std::ostream &os) const
  {
    os << "DeferredRouteOutputTag: output interface = " << m_oif;
  }

private:
  /// Positive if output device is fixed in RouteOutput
  int32_t m_oif;
};

NS_OBJECT_ENSURE_REGISTERED (DeferredRouteOutputTag);


//-----------------------------------------------------------------------------
RoutingProtocol::RoutingProtocol ()
  : m_rreqRetries (2),
    m_ttlStart (1),
    m_ttlIncrement (2),
    m_ttlThreshold (7),
    m_timeoutBuffer (2),
    m_rreqRateLimit (10),
    m_rerrRateLimit (10),
    m_activeRouteTimeout (Seconds (3)),
    m_netDiameter (35),
```

```
    m_nodeTraversalTime (MilliSeconds (40)),
    m_netTraversalTime (Time ((2 * m_netDiameter) * m_nodeTraversalTime)),
    m_pathDiscoveryTime ( Time (2 * m_netTraversalTime)),
    m_myRouteTimeout (Time (2 * std::max (m_pathDiscoveryTime, m_activeRouteTimeout))),
    m_helloInterval (Seconds (1)),
    m_allowedHelloLoss (2),
    m_deletePeriod (Time (5 * std::max (m_activeRouteTimeout, m_helloInterval))),
    m_nextHopWait (m_nodeTraversalTime + MilliSeconds (10)),
    m_blackListTimeout (Time (m_rreqRetries * m_netTraversalTime)),
    m_maxQueueLen (64),
    m_maxQueueTime (Seconds (30)),
    m_destinationOnly (false),
    m_gratuitousReply (true),
    m_enableHello (false),
    m_routingTable (m_deletePeriod),
    m_queue (m_maxQueueLen, m_maxQueueTime),
    m_requestId (0),
    m_seqNo (0),
    m_rreqIdCache (m_pathDiscoveryTime),
    m_dpd (m_pathDiscoveryTime),
    m_nb (m_helloInterval),
    m_rreqCount (0),
    m_rerrCount (0),
    m_htimer (Timer::CANCEL_ON_DESTROY),
    m_rreqRateLimitTimer (Timer::CANCEL_ON_DESTROY),
    m_rerrRateLimitTimer (Timer::CANCEL_ON_DESTROY),
    m_lastBcastTime (Seconds (0))
{
  m_nb.SetCallback (MakeCallback (&RoutingProtocol::SendRerrWhenBreaksLinkToNextHop, this))
      ;
}

TypeId
RoutingProtocol::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::aodv::RoutingProtocol")
    .SetParent<Ipv4RoutingProtocol> ()
    .SetGroupName ("Aodv")
    .AddConstructor<RoutingProtocol> ()
    .AddAttribute ("HelloInterval", "HELLO messages emission interval.",
                   TimeValue (Seconds (1)),
```

```
                MakeTimeAccessor (&RoutingProtocol::m_helloInterval),
                MakeTimeChecker ())
  .AddAttribute ("TtlStart", "Initial TTL value for RREQ.",
                UintegerValue (1),
                MakeUintegerAccessor (&RoutingProtocol::m_ttlStart),
                MakeUintegerChecker<uint16_t> ())
  .AddAttribute ("TtlIncrement", "TTL increment for each attempt using the expanding ring
      search for RREQ dissemination.",
                UintegerValue (2),
                MakeUintegerAccessor (&RoutingProtocol::m_ttlIncrement),
                MakeUintegerChecker<uint16_t> ())
  .AddAttribute ("TtlThreshold", "Maximum TTL value for expanding ring search, TTL =
      NetDiameter is used beyond this value.",
                UintegerValue (7),
                MakeUintegerAccessor (&RoutingProtocol::m_ttlThreshold),
                MakeUintegerChecker<uint16_t> ())
  .AddAttribute ("TimeoutBuffer", "Provide a buffer for the timeout.",
                UintegerValue (2),
                MakeUintegerAccessor (&RoutingProtocol::m_timeoutBuffer),
                MakeUintegerChecker<uint16_t> ())
  .AddAttribute ("RreqRetries", "Maximum number of retransmissions of RREQ to discover a
      route",
                UintegerValue (2),
                MakeUintegerAccessor (&RoutingProtocol::m_rreqRetries),
                MakeUintegerChecker<uint32_t> ())
  .AddAttribute ("RreqRateLimit", "Maximum number of RREQ per second.",
                UintegerValue (10),
                MakeUintegerAccessor (&RoutingProtocol::m_rreqRateLimit),
                MakeUintegerChecker<uint32_t> ())
  .AddAttribute ("RerrRateLimit", "Maximum number of RERR per second.",
                UintegerValue (10),
                MakeUintegerAccessor (&RoutingProtocol::m_rerrRateLimit),
                MakeUintegerChecker<uint32_t> ())
  .AddAttribute ("NodeTraversalTime", "Conservative estimate of the average one hop
      traversal time for packets and should include "
                "queuing delays, interrupt processing times and transfer times.",
                TimeValue (MilliSeconds (40)),
                MakeTimeAccessor (&RoutingProtocol::m_nodeTraversalTime),
                MakeTimeChecker ())
  .AddAttribute ("NextHopWait", "Period of our waiting for the neighbour's RREP_ACK = 10
      ms + NodeTraversalTime",
```

```
                TimeValue (MilliSeconds (50)),
                MakeTimeAccessor (&RoutingProtocol::m_nextHopWait),
                MakeTimeChecker ())
.AddAttribute ("ActiveRouteTimeout", "Period of time during which the route is
    considered to be valid",
                TimeValue (Seconds (3)),
                MakeTimeAccessor (&RoutingProtocol::m_activeRouteTimeout),
                MakeTimeChecker ())
.AddAttribute ("MyRouteTimeout", "Value of lifetime field in RREP generating by this
    node = 2 * max(ActiveRouteTimeout , PathDiscoveryTime)",
                TimeValue (Seconds (11.2)),
                MakeTimeAccessor (&RoutingProtocol::m_myRouteTimeout),
                MakeTimeChecker ())
.AddAttribute ("BlackListTimeout", "Time for which the node is put into the blacklist =
    RreqRetries * NetTraversalTime",
                TimeValue (Seconds (5.6)),
                MakeTimeAccessor (&RoutingProtocol::m_blackListTimeout),
                MakeTimeChecker ())
.AddAttribute ("DeletePeriod", "DeletePeriod is intended to provide an upper bound on
    the time for which an upstream node A "
                "can have a neighbor B as an active next hop for destination D, while B
                    has invalidated the route to D."
                " = 5 * max (HelloInterval , ActiveRouteTimeout)",
                TimeValue (Seconds (15)),
                MakeTimeAccessor (&RoutingProtocol::m_deletePeriod),
                MakeTimeChecker ())
.AddAttribute ("NetDiameter", "Net diameter measures the maximum possible number of
    hops between two nodes in the network",
                UintegerValue (35),
                MakeUintegerAccessor (&RoutingProtocol::m_netDiameter),
                MakeUintegerChecker<uint32_t> ())
.AddAttribute ("NetTraversalTime", "Estimate of the average net traversal time = 2 *
    NodeTraversalTime * NetDiameter",
                TimeValue (Seconds (2.8)),
                MakeTimeAccessor (&RoutingProtocol::m_netTraversalTime),
                MakeTimeChecker ())
.AddAttribute ("PathDiscoveryTime", "Estimate of maximum time needed to find route in
    network = 2 * NetTraversalTime",
                TimeValue (Seconds (5.6)),
                MakeTimeAccessor (&RoutingProtocol::m_pathDiscoveryTime),
                MakeTimeChecker ())
```

```
.AddAttribute ("MaxQueueLen", "Maximum number of packets that we allow a routing
    protocol to buffer.",
                UintegerValue (64),
                MakeUintegerAccessor (&RoutingProtocol::SetMaxQueueLen,
                                      &RoutingProtocol::GetMaxQueueLen),
                MakeUintegerChecker<uint32_t> ())
.AddAttribute ("MaxQueueTime", "Maximum time packets can be queued (in seconds)",
                TimeValue (Seconds (30)),
                MakeTimeAccessor (&RoutingProtocol::SetMaxQueueTime,
                                  &RoutingProtocol::GetMaxQueueTime),
                MakeTimeChecker ())
.AddAttribute ("AllowedHelloLoss", "Number of hello messages which may be loss for
    valid link.",
                UintegerValue (2),
                MakeUintegerAccessor (&RoutingProtocol::m_allowedHelloLoss),
                MakeUintegerChecker<uint16_t> ())
.AddAttribute ("GratuitousReply", "Indicates whether a gratuitous RREP should be
    unicast to the node originated route discovery.",
                BooleanValue (true),
                MakeBooleanAccessor (&RoutingProtocol::SetGratuitousReplyFlag,
                                     &RoutingProtocol::GetGratuitousReplyFlag),
                MakeBooleanChecker ())
.AddAttribute ("DestinationOnly", "Indicates only the destination may respond to this
    RREQ.",
                BooleanValue (false),
                MakeBooleanAccessor (&RoutingProtocol::SetDestinationOnlyFlag,
                                     &RoutingProtocol::GetDestinationOnlyFlag),
                MakeBooleanChecker ())
.AddAttribute ("EnableHello", "Indicates whether a hello messages enable.",
                BooleanValue (true),
                MakeBooleanAccessor (&RoutingProtocol::SetHelloEnable,
                                     &RoutingProtocol::GetHelloEnable),
                MakeBooleanChecker ())
.AddAttribute ("EnableBroadcast", "Indicates whether a broadcast data packets
    forwarding enable.",
                BooleanValue (true),
                MakeBooleanAccessor (&RoutingProtocol::SetBroadcastEnable,
                                     &RoutingProtocol::GetBroadcastEnable),
                MakeBooleanChecker ())
.AddAttribute ("UniformRv",
                "Access to the underlying UniformRandomVariable",
```

```
                    StringValue ("ns3::UniformRandomVariable"),
                    MakePointerAccessor (&RoutingProtocol::m_uniformRandomVariable),
                    MakePointerChecker<UniformRandomVariable> ())
    .AddAttribute ("IsMalicious", "Is the node malicious",
                    BooleanValue (false),
                    MakeBooleanAccessor (&RoutingProtocol::SetMaliciousEnable,
                                         &RoutingProtocol::GetMaliciousEnable),
                    MakeBooleanChecker ())

  ;
  return tid;
}

void
RoutingProtocol::SetMaxQueueLen (uint32_t len)
{
  m_maxQueueLen = len;
  m_queue.SetMaxQueueLen (len);
}
void
RoutingProtocol::SetMaxQueueTime (Time t)
{
  m_maxQueueTime = t;
  m_queue.SetQueueTimeout (t);
}

RoutingProtocol::~RoutingProtocol ()
{
}

void
RoutingProtocol::DoDispose ()
{
  m_ipv4 = 0;
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::iterator iter =
         m_socketAddresses.begin (); iter != m_socketAddresses.end (); iter++)
    {
      iter->first->Close ();
    }
  m_socketAddresses.clear ();
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::iterator iter =
```

```
        m_socketSubnetBroadcastAddresses.begin (); iter !=
            m_socketSubnetBroadcastAddresses.end (); iter++)
    {
      iter->first->Close ();
    }
  m_socketSubnetBroadcastAddresses.clear ();
  Ipv4RoutingProtocol::DoDispose ();
}

void
RoutingProtocol::PrintRoutingTable (Ptr<OutputStreamWrapper> stream, Time::Unit unit) const
{
  *stream->GetStream () << "Node: " << m_ipv4->GetObject<Node> ()->GetId ()
                        << "; Time: " << Now ().As (unit)
                        << ", Local time: " << m_ipv4->GetObject<Node> ()->GetLocalTime ().
                            As (unit)
                        << ", AODV Routing table" << std::endl;

  m_routingTable.Print (stream, unit);
  *stream->GetStream () << std::endl;
}

int64_t
RoutingProtocol::AssignStreams (int64_t stream)
{
  NS_LOG_FUNCTION (this << stream);
  m_uniformRandomVariable->SetStream (stream);
  return 1;
}

void
RoutingProtocol::Start ()
{
  NS_LOG_FUNCTION (this);
  if (m_enableHello)
    {
      m_nb.ScheduleTimer ();
    }
  m_rreqRateLimitTimer.SetFunction (&RoutingProtocol::RreqRateLimitTimerExpire,
                                    this);
  m_rreqRateLimitTimer.Schedule (Seconds (1));
```

```
  m_rerrRateLimitTimer.SetFunction (&RoutingProtocol::RerrRateLimitTimerExpire,
                                    this);
  m_rerrRateLimitTimer.Schedule (Seconds (1));

}

Ptr<Ipv4Route>
RoutingProtocol::RouteOutput (Ptr<Packet> p, const Ipv4Header &header,
                              Ptr<NetDevice> oif, Socket::SocketErrno &sockerr)
{
  NS_LOG_FUNCTION (this << header << (oif ? oif->GetIfIndex () : 0));
  if (!p)
    {
      NS_LOG_DEBUG ("Packet is == 0");
      return LoopbackRoute (header, oif); // later
    }
  if (m_socketAddresses.empty ())
    {
      sockerr = Socket::ERROR_NOROUTETOHOST;
      NS_LOG_LOGIC ("No aodv interfaces");
      Ptr<Ipv4Route> route;
      return route;
    }
  sockerr = Socket::ERROR_NOTERROR;
  Ptr<Ipv4Route> route;
  Ipv4Address dst = header.GetDestination ();
  RoutingTableEntry rt;
  if (m_routingTable.LookupValidRoute (dst, rt))
    {
      route = rt.GetRoute ();
      NS_ASSERT (route != 0);
      NS_LOG_DEBUG ("Exist route to " << route->GetDestination () << " from interface " <<
          route->GetSource ());
      if (oif != 0 && route->GetOutputDevice () != oif)
        {
          NS_LOG_DEBUG ("Output device doesn't match. Dropped.");
          sockerr = Socket::ERROR_NOROUTETOHOST;
          return Ptr<Ipv4Route> ();
        }
      UpdateRouteLifeTime (dst, m_activeRouteTimeout);
```

```
      UpdateRouteLifeTime (route->GetGateway (), m_activeRouteTimeout);
      return route;
    }

  // Valid route not found, in this case we return loopback.
  // Actual route request will be deferred until packet will be fully formed,
  // routed to loopback, received from loopback and passed to RouteInput (see below)
  uint32_t iif = (oif ? m_ipv4->GetInterfaceForDevice (oif) : -1);
  DeferredRouteOutputTag tag (iif);
  NS_LOG_DEBUG ("Valid Route not found");
  if (!p->PeekPacketTag (tag))
    {
      p->AddPacketTag (tag);
    }
  return LoopbackRoute (header, oif);
}

void
RoutingProtocol::DeferredRouteOutput (Ptr<const Packet> p, const Ipv4Header & header,
                                      UnicastForwardCallback ucb, ErrorCallback ecb)
{
  NS_LOG_FUNCTION (this << p << header);
  NS_ASSERT (p != 0 && p != Ptr<Packet> ());

  QueueEntry newEntry (p, header, ucb, ecb);
  bool result = m_queue.Enqueue (newEntry);
  if (result)
    {
      NS_LOG_LOGIC ("Add packet " << p->GetUid () << " to queue. Protocol " << (uint16_t)
          header.GetProtocol ());
      RoutingTableEntry rt;
      bool result = m_routingTable.LookupRoute (header.GetDestination (), rt);
      if (!result || ((rt.GetFlag () != IN_SEARCH) && result))
        {
          NS_LOG_LOGIC ("Send new RREQ for outbound packet to " << header.GetDestination ()
              );
          SendRequest (header.GetDestination ());
        }
    }
}
```

```
bool
RoutingProtocol::RouteInput (Ptr<const Packet> p, const Ipv4Header &header,
                             Ptr<const NetDevice> idev, UnicastForwardCallback ucb,
                             MulticastForwardCallback mcb, LocalDeliverCallback lcb,
                                 ErrorCallback ecb)
{
  NS_LOG_FUNCTION (this << p->GetUid () << header.GetDestination () << idev->GetAddress ())
      ;
  if (m_socketAddresses.empty ())
    {
      NS_LOG_LOGIC ("No␣aodv␣interfaces");
      return false;
    }
  NS_ASSERT (m_ipv4 != 0);
  NS_ASSERT (p != 0);
  // Check if input device supports IP
  NS_ASSERT (m_ipv4->GetInterfaceForDevice (idev) >= 0);
  int32_t iif = m_ipv4->GetInterfaceForDevice (idev);

  Ipv4Address dst = header.GetDestination ();
  Ipv4Address origin = header.GetSource ();

  // Deferred route request
  if (idev == m_lo)
    {
      DeferredRouteOutputTag tag;
      if (p->PeekPacketTag (tag))
        {
          DeferredRouteOutput (p, header, ucb, ecb);
          return true;
        }
    }

  // Duplicate of own packet
  if (IsMyOwnAddress (origin))
    {
      return true;
    }

  // AODV is not a multicast routing protocol
  if (dst.IsMulticast ())
```

```
  {
    return false;
  }

// Broadcast local delivery/forwarding
for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
       m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
  {
    Ipv4InterfaceAddress iface = j->second;
    if (m_ipv4->GetInterfaceForAddress (iface.GetLocal ()) == iif)
      {
        if (dst == iface.GetBroadcast () || dst.IsBroadcast ())
          {
            if (m_dpd.IsDuplicate (p, header))
              {
                NS_LOG_DEBUG ("Duplicated packet " << p->GetUid () << " from " << origin
                    << ". Drop.");
                return true;
              }
            UpdateRouteLifeTime (origin, m_activeRouteTimeout);
            Ptr<Packet> packet = p->Copy ();
            if (lcb.IsNull () == false)
              {
                NS_LOG_LOGIC ("Broadcast local delivery to " << iface.GetLocal ());
                lcb (p, header, iif);
                // Fall through to additional processing
              }
            else
              {
                NS_LOG_ERROR ("Unable to deliver packet locally due to null callback " <<
                     p->GetUid () << " from " << origin);
                ecb (p, header, Socket::ERROR_NOROUTETOHOST);
              }
            if (!m_enableBroadcast)
              {
                return true;
              }
            if (header.GetProtocol () == UdpL4Protocol::PROT_NUMBER)
              {
                UdpHeader udpHeader;
                p->PeekHeader (udpHeader);
```

```
                if (udpHeader.GetDestinationPort () == AODV_PORT)
                  {
                    // AODV packets sent in broadcast are already managed
                    return true;
                  }
              }
            if (header.GetTtl () > 1)
              {
                NS_LOG_LOGIC ("Forward broadcast. TTL " << (uint16_t) header.GetTtl ());
                RoutingTableEntry toBroadcast;
                if (m_routingTable.LookupRoute (dst, toBroadcast))
                  {
                    Ptr<Ipv4Route> route = toBroadcast.GetRoute ();
                    ucb (route, packet, header);
                  }
                else
                  {
                    NS_LOG_DEBUG ("No route to forward broadcast. Drop packet " << p->
                        GetUid ());
                  }
              }
            else
              {
                NS_LOG_DEBUG ("TTL exceeded. Drop packet " << p->GetUid ());
              }
            return true;
          }
      }
  }

// Unicast local delivery
if (m_ipv4->IsDestinationAddress (dst, iif))
  {
    UpdateRouteLifeTime (origin, m_activeRouteTimeout);
    RoutingTableEntry toOrigin;
    if (m_routingTable.LookupValidRoute (origin, toOrigin))
      {
        UpdateRouteLifeTime (toOrigin.GetNextHop (), m_activeRouteTimeout);
        m_nb.Update (toOrigin.GetNextHop (), m_activeRouteTimeout);
      }
    if (lcb.IsNull () == false)
```

```
        {
          NS_LOG_LOGIC ("Unicast local delivery to " << dst);
          lcb (p, header, iif);
        }
      else
        {
          NS_LOG_ERROR ("Unable to deliver packet locally due to null callback " << p->
              GetUid () << " from " << origin);
          ecb (p, header, Socket::ERROR_NOROUTETOHOST);
        }
      return true;
    }

  // Check if input device supports IP forwarding
  if (m_ipv4->IsForwarding (iif) == false)
    {
      NS_LOG_LOGIC ("Forwarding disabled for this interface");
      ecb (p, header, Socket::ERROR_NOROUTETOHOST);
      return true;
    }

  // Forwarding
  return Forwarding (p, header, ucb, ecb);
}

bool
RoutingProtocol::Forwarding (Ptr<const Packet> p, const Ipv4Header & header,
                             UnicastForwardCallback ucb, ErrorCallback ecb)
{
  NS_LOG_FUNCTION (this);
  Ipv4Address dst = header.GetDestination ();
  Ipv4Address origin = header.GetSource ();
  m_routingTable.Purge ();
  RoutingTableEntry toDst;
  /* Code added by Shalini Satre, Wireless Information Networking Group (WiNG), NITK
      Surathkal for simulating Blackhole Attack */
  /* Check if the node is suppose to behave maliciously */
  if(IsMalicious) {
    //When malicious node receives packet it drops the packet.
    //std :: cout <<"Launching Blackhole Attack! Packet dropped . . . \n";
        NS_LOG_DEBUG("Launching Blackhole Attack! Packet dropped...");
```

```
  return false;
}
/* Code for Blackhole attack simulation ends here */
if (m_routingTable.LookupRoute (dst, toDst))
  {
    if (toDst.GetFlag () == VALID)
      {
        Ptr<Ipv4Route> route = toDst.GetRoute ();
        NS_LOG_LOGIC (route->GetSource () << " forwarding to " << dst << " from " <<
            origin << " packet " << p->GetUid ());

        /*
         *  Each time a route is used to forward a data packet, its Active Route
         *  Lifetime field of the source, destination and the next hop on the
         *  path to the destination is updated to be no less than the current
         *  time plus ActiveRouteTimeout.
         */
        UpdateRouteLifeTime (origin, m_activeRouteTimeout);
        UpdateRouteLifeTime (dst, m_activeRouteTimeout);
        UpdateRouteLifeTime (route->GetGateway (), m_activeRouteTimeout);
        /*
         *  Since the route between each originator and destination pair is expected to
             be symmetric, the
         *  Active Route Lifetime for the previous hop, along the reverse path back to
             the IP source, is also updated
         *  to be no less than the current time plus ActiveRouteTimeout
         */
        RoutingTableEntry toOrigin;
        m_routingTable.LookupRoute (origin, toOrigin);
        UpdateRouteLifeTime (toOrigin.GetNextHop (), m_activeRouteTimeout);

        m_nb.Update (route->GetGateway (), m_activeRouteTimeout);
        m_nb.Update (toOrigin.GetNextHop (), m_activeRouteTimeout);

        ucb (route, p, header);
        return true;
      }
    else
      {
        if (toDst.GetValidSeqNo ())
          {
```

```
                SendRerrWhenNoRouteToForward (dst, toDst.GetSeqNo (), origin);
                NS_LOG_DEBUG ("Drop␣packet␣" << p->GetUid () << "␣because␣no␣route␣to␣forward
                    ␣it.");
                return false;
              }
          }
      }
    NS_LOG_LOGIC ("route␣not␣found␣to␣" << dst << ".␣Send␣RERR␣message.");
    NS_LOG_DEBUG ("Drop␣packet␣" << p->GetUid () << "␣because␣no␣route␣to␣forward␣it.");
    SendRerrWhenNoRouteToForward (dst, 0, origin);
    return false;
}

void
RoutingProtocol::SetIpv4 (Ptr<Ipv4> ipv4)
{
    NS_ASSERT (ipv4 != 0);
    NS_ASSERT (m_ipv4 == 0);

    m_ipv4 = ipv4;

    // Create lo route. It is asserted that the only one interface up for now is loopback
    NS_ASSERT (m_ipv4->GetNInterfaces () == 1 && m_ipv4->GetAddress (0, 0).GetLocal () ==
        Ipv4Address ("127.0.0.1"));
    m_lo = m_ipv4->GetNetDevice (0);
    NS_ASSERT (m_lo != 0);
    // Remember lo route
    RoutingTableEntry rt (/*device=*/ m_lo, /*dst=*/ Ipv4Address::GetLoopback (), /*know
        seqno=*/ true, /*seqno=*/ 0,
                                  /*iface=*/ Ipv4InterfaceAddress (Ipv4Address::
                                      GetLoopback (), Ipv4Mask ("255.0.0.0")),
                                  /*hops=*/ 1, /*next hop=*/ Ipv4Address::GetLoopback (),
                                  /*lifetime=*/ Simulator::GetMaximumSimulationTime ());
    m_routingTable.AddRoute (rt);

    Simulator::ScheduleNow (&RoutingProtocol::Start, this);
}

void
RoutingProtocol::NotifyInterfaceUp (uint32_t i)
{
```

```
NS_LOG_FUNCTION (this << m_ipv4->GetAddress (i, 0).GetLocal ());
Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
if (l3->GetNAddresses (i) > 1)
  {
    NS_LOG_WARN ("AODV does not work with more then one address per each interface.");
  }
Ipv4InterfaceAddress iface = l3->GetAddress (i, 0);
if (iface.GetLocal () == Ipv4Address ("127.0.0.1"))
  {
    return;
  }

// Create a socket to listen only on this interface
Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),
                                           UdpSocketFactory::GetTypeId ());
NS_ASSERT (socket != 0);
socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
socket->BindToNetDevice (l3->GetNetDevice (i));
socket->Bind (InetSocketAddress (iface.GetLocal (), AODV_PORT));
socket->SetAllowBroadcast (true);
socket->SetIpRecvTtl (true);
m_socketAddresses.insert (std::make_pair (socket, iface));

// create also a subnet broadcast socket
socket = Socket::CreateSocket (GetObject<Node> (),
                               UdpSocketFactory::GetTypeId ());
NS_ASSERT (socket != 0);
socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
socket->BindToNetDevice (l3->GetNetDevice (i));
socket->Bind (InetSocketAddress (iface.GetBroadcast (), AODV_PORT));
socket->SetAllowBroadcast (true);
socket->SetIpRecvTtl (true);
m_socketSubnetBroadcastAddresses.insert (std::make_pair (socket, iface));

// Add local broadcast record to the routing table
Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (iface.GetLocal
    ()));
RoutingTableEntry rt (/*device=*/ dev, /*dst=*/ iface.GetBroadcast (), /*know seqno=*/
   true, /*seqno=*/ 0, /*iface=*/ iface,
                                 /*hops=*/ 1, /*next hop=*/ iface.GetBroadcast (), /*
                                     lifetime=*/ Simulator::GetMaximumSimulationTime ())
```

```
                                        ;
  m_routingTable.AddRoute (rt);

  if (l3->GetInterface (i)->GetArpCache ())
    {
      m_nb.AddArpCache (l3->GetInterface (i)->GetArpCache ());
    }

  // Allow neighbor manager use this interface for layer 2 feedback if possible
  Ptr<WifiNetDevice> wifi = dev->GetObject<WifiNetDevice> ();
  if (wifi == 0)
    {
      return;
    }
  Ptr<WifiMac> mac = wifi->GetMac ();
  if (mac == 0)
    {
      return;
    }

  mac->TraceConnectWithoutContext ("DroppedMpdu", MakeCallback (&RoutingProtocol::
      NotifyTxError, this));
}

void
RoutingProtocol::NotifyTxError (WifiMacDropReason reason, Ptr<const WifiMacQueueItem> mpdu)
{
  m_nb.GetTxErrorCallback ()(mpdu->GetHeader ());
}

void
RoutingProtocol::NotifyInterfaceDown (uint32_t i)
{
  NS_LOG_FUNCTION (this << m_ipv4->GetAddress (i, 0).GetLocal ());

  // Disable layer 2 link state monitoring (if possible)
  Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
  Ptr<NetDevice> dev = l3->GetNetDevice (i);
  Ptr<WifiNetDevice> wifi = dev->GetObject<WifiNetDevice> ();
  if (wifi != 0)
    {
```

```
      Ptr<WifiMac> mac = wifi->GetMac ()->GetObject<AdhocWifiMac> ();
      if (mac != 0)
        {
          mac->TraceDisconnectWithoutContext ("DroppedMpdu",
                                              MakeCallback (&RoutingProtocol::NotifyTxError
                                                  , this));
          m_nb.DelArpCache (l3->GetInterface (i)->GetArpCache ());
        }
    }

  // Close socket
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (m_ipv4->GetAddress (i, 0));
  NS_ASSERT (socket);
  socket->Close ();
  m_socketAddresses.erase (socket);

  // Close socket
  socket = FindSubnetBroadcastSocketWithInterfaceAddress (m_ipv4->GetAddress (i, 0));
  NS_ASSERT (socket);
  socket->Close ();
  m_socketSubnetBroadcastAddresses.erase (socket);

  if (m_socketAddresses.empty ())
    {
      NS_LOG_LOGIC ("No aodv interfaces");
      m_htimer.Cancel ();
      m_nb.Clear ();
      m_routingTable.Clear ();
      return;
    }
  m_routingTable.DeleteAllRoutesFromInterface (m_ipv4->GetAddress (i, 0));
}

void
RoutingProtocol::NotifyAddAddress (uint32_t i, Ipv4InterfaceAddress address)
{
  NS_LOG_FUNCTION (this << " interface " << i << " address " << address);
  Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
  if (!l3->IsUp (i))
    {
      return;
```

```
  }
if (l3->GetNAddresses (i) == 1)
  {
    Ipv4InterfaceAddress iface = l3->GetAddress (i, 0);
    Ptr<Socket> socket = FindSocketWithInterfaceAddress (iface);
    if (!socket)
      {
        if (iface.GetLocal () == Ipv4Address ("127.0.0.1"))
          {
            return;
          }
        // Create a socket to listen only on this interface
        Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),
                                                   UdpSocketFactory::GetTypeId ());
        NS_ASSERT (socket != 0);
        socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv,this));
        socket->BindToNetDevice (l3->GetNetDevice (i));
        socket->Bind (InetSocketAddress (iface.GetLocal (), AODV_PORT));
        socket->SetAllowBroadcast (true);
        m_socketAddresses.insert (std::make_pair (socket, iface));

        // create also a subnet directed broadcast socket
        socket = Socket::CreateSocket (GetObject<Node> (),
                                       UdpSocketFactory::GetTypeId ());
        NS_ASSERT (socket != 0);
        socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
        socket->BindToNetDevice (l3->GetNetDevice (i));
        socket->Bind (InetSocketAddress (iface.GetBroadcast (), AODV_PORT));
        socket->SetAllowBroadcast (true);
        socket->SetIpRecvTtl (true);
        m_socketSubnetBroadcastAddresses.insert (std::make_pair (socket, iface));

        // Add local broadcast record to the routing table
        Ptr<NetDevice> dev = m_ipv4->GetNetDevice (
            m_ipv4->GetInterfaceForAddress (iface.GetLocal ()));
        RoutingTableEntry rt (/*device=*/ dev, /*dst=*/ iface.GetBroadcast (), /*know
            seqno=*/ true,
                                        /*seqno=*/ 0, /*iface=*/ iface, /*hops=*/ 1,
                                        /*next hop=*/ iface.GetBroadcast (), /*lifetime
                                            =*/ Simulator::GetMaximumSimulationTime ())
                                            ;
```

```
            m_routingTable.AddRoute (rt);
          }
      }
    else
      {
        NS_LOG_LOGIC ("AODV does not work with more then one address per each interface.
            Ignore added address");
      }
}

void
RoutingProtocol::NotifyRemoveAddress (uint32_t i, Ipv4InterfaceAddress address)
{
  NS_LOG_FUNCTION (this);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (address);
  if (socket)
    {
      m_routingTable.DeleteAllRoutesFromInterface (address);
      socket->Close ();
      m_socketAddresses.erase (socket);

      Ptr<Socket> unicastSocket = FindSubnetBroadcastSocketWithInterfaceAddress (address);
      if (unicastSocket)
        {
          unicastSocket->Close ();
          m_socketAddresses.erase (unicastSocket);
        }

      Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
      if (l3->GetNAddresses (i))
        {
          Ipv4InterfaceAddress iface = l3->GetAddress (i, 0);
          // Create a socket to listen only on this interface
          Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),
                                                     UdpSocketFactory::GetTypeId ());
          NS_ASSERT (socket != 0);
          socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
          // Bind to any IP address so that broadcasts can be received
          socket->BindToNetDevice (l3->GetNetDevice (i));
          socket->Bind (InetSocketAddress (iface.GetLocal (), AODV_PORT));
          socket->SetAllowBroadcast (true);
```

```
            socket->SetIpRecvTtl (true);
            m_socketAddresses.insert (std::make_pair (socket, iface));

            // create also a unicast socket
            socket = Socket::CreateSocket (GetObject<Node> (),
                                           UdpSocketFactory::GetTypeId ());
            NS_ASSERT (socket != 0);
            socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
            socket->BindToNetDevice (l3->GetNetDevice (i));
            socket->Bind (InetSocketAddress (iface.GetBroadcast (), AODV_PORT));
            socket->SetAllowBroadcast (true);
            socket->SetIpRecvTtl (true);
            m_socketSubnetBroadcastAddresses.insert (std::make_pair (socket, iface));

            // Add local broadcast record to the routing table
            Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (iface.
                GetLocal ()));
            RoutingTableEntry rt (/*device=*/ dev, /*dst=*/ iface.GetBroadcast (), /*know
                seqno=*/ true, /*seqno=*/ 0, /*iface=*/ iface,
                                  /*hops=*/ 1, /*next hop=*/ iface.GetBroadcast
                                      (), /*lifetime=*/ Simulator::
                                      GetMaximumSimulationTime ());
            m_routingTable.AddRoute (rt);
          }
        if (m_socketAddresses.empty ())
          {
            NS_LOG_LOGIC ("No aodv interfaces");
            m_htimer.Cancel ();
            m_nb.Clear ();
            m_routingTable.Clear ();
            return;
          }
      }
    else
      {
        NS_LOG_LOGIC ("Remove address not participating in AODV operation");
      }
}

bool
RoutingProtocol::IsMyOwnAddress (Ipv4Address src)
```

```
{
  NS_LOG_FUNCTION (this << src);
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
    {
      Ipv4InterfaceAddress iface = j->second;
      if (src == iface.GetLocal ())
        {
          return true;
        }
    }
  return false;
}

Ptr<Ipv4Route>
RoutingProtocol::LoopbackRoute (const Ipv4Header & hdr, Ptr<NetDevice> oif) const
{
  NS_LOG_FUNCTION (this << hdr);
  NS_ASSERT (m_lo != 0);
  Ptr<Ipv4Route> rt = Create<Ipv4Route> ();
  rt->SetDestination (hdr.GetDestination ());
  //
  // Source address selection here is tricky.  The loopback route is
  // returned when AODV does not have a route; this causes the packet
  // to be looped back and handled (cached) in RouteInput() method
  // while a route is found. However, connection-oriented protocols
  // like TCP need to create an endpoint four-tuple (src, src port,
  // dst, dst port) and create a pseudo-header for checksumming.  So,
  // AODV needs to guess correctly what the eventual source address
  // will be.
  //
  // For single interface, single address nodes, this is not a problem.
  // When there are possibly multiple outgoing interfaces, the policy
  // implemented here is to pick the first available AODV interface.
  // If RouteOutput() caller specified an outgoing interface, that
  // further constrains the selection of source address
  //
  std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses.begin
      ();
  if (oif)
    {
```

```
      // Iterate to find an address on the oif device
      for (j = m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
        {
          Ipv4Address addr = j->second.GetLocal ();
          int32_t interface = m_ipv4->GetInterfaceForAddress (addr);
          if (oif == m_ipv4->GetNetDevice (static_cast<uint32_t> (interface)))
            {
              rt->SetSource (addr);
              break;
            }
        }
    }
  else
    {
      rt->SetSource (j->second.GetLocal ());
    }
  NS_ASSERT_MSG (rt->GetSource () != Ipv4Address (), "Valid AODV source address not found")
      ;
  rt->SetGateway (Ipv4Address ("127.0.0.1"));
  rt->SetOutputDevice (m_lo);
  return rt;
}

void
RoutingProtocol::SendRequest (Ipv4Address dst)
{
  NS_LOG_FUNCTION ( this << dst);
  // A node SHOULD NOT originate more than RREQ_RATELIMIT RREQ messages per second.
  if (m_rreqCount == m_rreqRateLimit)
    {
      Simulator::Schedule (m_rreqRateLimitTimer.GetDelayLeft () + MicroSeconds (100),
                           &RoutingProtocol::SendRequest, this, dst);
      return;
    }
  else
    {
      m_rreqCount++;
    }
  // Create RREQ header
  RreqHeader rreqHeader;
  rreqHeader.SetDst (dst);
```

```
RoutingTableEntry rt;
// Using the Hop field in Routing Table to manage the expanding ring search
uint16_t ttl = m_ttlStart;
if (m_routingTable.LookupRoute (dst, rt))
  {
    // NS_LOG_UNCOND("bingo " << dst);
    if (rt.GetFlag () != IN_SEARCH)
      {
        ttl = std::min<uint16_t> (rt.GetHop () + m_ttlIncrement, m_netDiameter);
      }
    else
      {
        ttl = rt.GetHop () + m_ttlIncrement;
        if (ttl > m_ttlThreshold)
          {
            ttl = m_netDiameter;
          }
      }
    if (ttl == m_netDiameter)
      {
        rt.IncrementRreqCnt ();
      }
    if (rt.GetValidSeqNo ())
      {
        rreqHeader.SetDstSeqno (rt.GetSeqNo ());
      }
    else
      {
        rreqHeader.SetUnknownSeqno (true);
      }
    rt.SetHop (ttl);
    rt.SetFlag (IN_SEARCH);
    rt.SetLifeTime (m_pathDiscoveryTime);
    m_routingTable.Update (rt);
  }
else
  {
    rreqHeader.SetUnknownSeqno (true);
    Ptr<NetDevice> dev = 0;
    RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ dst, /*validSeqNo=*/ false, /*
```

```
        seqno=*/ 0,
                                                /*iface=*/ Ipv4InterfaceAddress (),/*hop=*/
                                                    ttl,
                                                /*nextHop=*/ Ipv4Address (), /*lifeTime=*/
                                                    m_pathDiscoveryTime);
    // Check if TtlStart == NetDiameter
    if (ttl == m_netDiameter)
      {
        newEntry.IncrementRreqCnt ();
      }
    newEntry.SetFlag (IN_SEARCH);
    m_routingTable.AddRoute (newEntry);
  }

if (m_gratuitousReply)
  {
    rreqHeader.SetGratuitousRrep (true);
  }
if (m_destinationOnly)
  {
    rreqHeader.SetDestinationOnly (true);
  }

m_seqNo++;
rreqHeader.SetOriginSeqno (m_seqNo);
m_requestId++;
rreqHeader.SetId (m_requestId);

// Send RREQ as subnet directed broadcast from each interface used by aodv
for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
       m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
  {
    Ptr<Socket> socket = j->first;
    Ipv4InterfaceAddress iface = j->second;

    rreqHeader.SetOrigin (iface.GetLocal ());
    m_rreqIdCache.IsDuplicate (iface.GetLocal (), m_requestId);

    Ptr<Packet> packet = Create<Packet> ();
    SocketIpTtlTag tag;
    tag.SetTtl (ttl);
```

```
        packet->AddPacketTag (tag);
        packet->AddHeader (rreqHeader);
        TypeHeader tHeader (AODVTYPE_RREQ);
        packet->AddHeader (tHeader);
        // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
        Ipv4Address destination;
        if (iface.GetMask () == Ipv4Mask::GetOnes ())
          {
            destination = Ipv4Address ("255.255.255.255");
          }
        else
          {
            destination = iface.GetBroadcast ();
          }
        NS_LOG_DEBUG ("Send RREQ with id " << rreqHeader.GetId () << " to socket");
        m_lastBcastTime = Simulator::Now ();
        Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable->GetInteger (0, 10))
            ), &RoutingProtocol::SendTo, this, socket, packet, destination);
      }
    ScheduleRreqRetry (dst);
}

void
RoutingProtocol::SendTo (Ptr<Socket> socket, Ptr<Packet> packet, Ipv4Address destination)
{
  socket->SendTo (packet, 0, InetSocketAddress (destination, AODV_PORT));
}

void
RoutingProtocol::ScheduleRreqRetry (Ipv4Address dst)
{
  NS_LOG_FUNCTION (this << dst);
  if (m_addressReqTimer.find (dst) == m_addressReqTimer.end ())
    {
      Timer timer (Timer::CANCEL_ON_DESTROY);
      m_addressReqTimer[dst] = timer;
    }
  m_addressReqTimer[dst].SetFunction (&RoutingProtocol::RouteRequestTimerExpire, this);
  m_addressReqTimer[dst].Cancel ();
  m_addressReqTimer[dst].SetArguments (dst);
  RoutingTableEntry rt;
```

```
  m_routingTable.LookupRoute (dst, rt);
  Time retry;
  if (rt.GetHop () < m_netDiameter)
    {
      retry = 2 * m_nodeTraversalTime * (rt.GetHop () + m_timeoutBuffer);
    }
  else
    {
      // NS_LOG_UNCOND("REQ COUNT " << rt.GetRreqCnt ());
      // NS_ABORT_MSG_UNLESS (rt.GetRreqCnt () > 0, "Unexpected value for GetRreqCount ()")
          ;
      uint16_t backoffFactor = rt.GetRreqCnt () - 1;
      NS_LOG_LOGIC ("Applying binary exponential backoff factor " << backoffFactor);
      retry = m_netTraversalTime * (1 << backoffFactor);
      if (retry.IsNegative()) {
        retry = m_netTraversalTime * (1 << (backoffFactor - 1));
      }
    }
    // NS_LOG_UNCOND("retry " << retry);
  m_addressReqTimer[dst].Schedule (retry);
  NS_LOG_LOGIC ("Scheduled RREQ retry in " << retry.As (Time::S));
}

void
RoutingProtocol::RecvAodv (Ptr<Socket> socket)
{
  NS_LOG_FUNCTION (this << socket);
  Address sourceAddress;
  Ptr<Packet> packet = socket->RecvFrom (sourceAddress);
  InetSocketAddress inetSourceAddr = InetSocketAddress::ConvertFrom (sourceAddress);
  Ipv4Address sender = inetSourceAddr.GetIpv4 ();
  Ipv4Address receiver;

  if (m_socketAddresses.find (socket) != m_socketAddresses.end ())
    {
      receiver = m_socketAddresses[socket].GetLocal ();
    }
  else if (m_socketSubnetBroadcastAddresses.find (socket) !=
      m_socketSubnetBroadcastAddresses.end ())
    {
      receiver = m_socketSubnetBroadcastAddresses[socket].GetLocal ();
```

```
    }
  else
    {
      NS_ASSERT_MSG (false, "Received a packet from an unknown socket");
    }
  NS_LOG_DEBUG ("AODV node " << this << " received a AODV packet from " << sender << " to "
       << receiver);

  UpdateRouteToNeighbor (sender, receiver);
  TypeHeader tHeader (AODVTYPE_RREQ);
  packet->RemoveHeader (tHeader);
  if (!tHeader.IsValid ())
    {
      NS_LOG_DEBUG ("AODV message " << packet->GetUid () << " with unknown type received: "
           << tHeader.Get () << ". Drop");
      return; // drop
    }
  switch (tHeader.Get ())
    {
    case AODVTYPE_RREQ:
      {
        RecvRequest (packet, receiver, sender);
        break;
      }
    case AODVTYPE_RREP:
      {
        RecvReply (packet, receiver, sender);
        break;
      }
    case AODVTYPE_RERR:
      {
        RecvError (packet, sender);
        break;
      }
    case AODVTYPE_RREP_ACK:
      {
        RecvReplyAck (sender);
        break;
      }
    }
}
```

```
bool
RoutingProtocol::UpdateRouteLifeTime (Ipv4Address addr, Time lifetime)
{
  NS_LOG_FUNCTION (this << addr << lifetime);
  RoutingTableEntry rt;
  if (m_routingTable.LookupRoute (addr, rt))
    {
      if (rt.GetFlag () == VALID)
        {
          NS_LOG_DEBUG ("Updating VALID route");
          rt.SetRreqCnt (0);
          rt.SetLifeTime (std::max (lifetime, rt.GetLifeTime ()));
          m_routingTable.Update (rt);
          return true;
        }
    }
  return false;
}

void
RoutingProtocol::UpdateRouteToNeighbor (Ipv4Address sender, Ipv4Address receiver)
{
  NS_LOG_FUNCTION (this << "sender " << sender << " receiver " << receiver);
  RoutingTableEntry toNeighbor;
  if (!m_routingTable.LookupRoute (sender, toNeighbor))
    {
      Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver))
          ;
      RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ sender, /*know seqno=*/ false,
          /*seqno=*/ 0,
                                                /*iface=*/ m_ipv4->GetAddress (m_ipv4->
                                                    GetInterfaceForAddress (receiver), 0),
                                                /*hops=*/ 1, /*next hop=*/ sender, /*lifetime
                                                    =*/ m_activeRouteTimeout);
      m_routingTable.AddRoute (newEntry);
    }
  else
    {
      Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver))
          ;
```

```
      if (toNeighbor.GetValidSeqNo () && (toNeighbor.GetHop () == 1) && (toNeighbor.
          GetOutputDevice () == dev))
        {
          toNeighbor.SetLifeTime (std::max (m_activeRouteTimeout, toNeighbor.GetLifeTime ()
              ));
        }
      else
        {
          RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ sender, /*know seqno=*/
              false, /*seqno=*/ 0,
                                      /*iface=*/ m_ipv4->GetAddress (m_ipv4->
                                          GetInterfaceForAddress (receiver), 0)
                                          ,
                                      /*hops=*/ 1, /*next hop=*/ sender, /*
                                          lifetime=*/ std::max (
                                          m_activeRouteTimeout, toNeighbor.
                                          GetLifeTime ()));
          m_routingTable.Update (newEntry);
        }
    }

}

void
RoutingProtocol::RecvRequest (Ptr<Packet> p, Ipv4Address receiver, Ipv4Address src)
{
  NS_LOG_FUNCTION (this);
  RreqHeader rreqHeader;
  p->RemoveHeader (rreqHeader);

  // A node ignores all RREQs received from any node in its blacklist
  RoutingTableEntry toPrev;
  if (m_routingTable.LookupRoute (src, toPrev))
    {
      if (toPrev.IsUnidirectional ())
        {
          NS_LOG_DEBUG ("Ignoring␣RREQ␣from␣node␣in␣blacklist");
          return;
        }
    }
```

```
uint32_t id = rreqHeader.GetId ();
Ipv4Address origin = rreqHeader.GetOrigin ();

/*
 *  Node checks to determine whether it has received a RREQ with the same Originator IP
     Address and RREQ ID.
 *  If such a RREQ has been received, the node silently discards the newly received RREQ.
 */
if (m_rreqIdCache.IsDuplicate (origin, id))
  {
    NS_LOG_DEBUG ("Ignoring RREQ due to duplicate");
    return;
  }

// Increment RREQ hop count
uint8_t hop = rreqHeader.GetHopCount () + 1;
rreqHeader.SetHopCount (hop);

/*
 *  When the reverse route is created or updated, the following actions on the route are
     also carried out:
 *  1. the Originator Sequence Number from the RREQ is compared to the corresponding
     destination sequence number
 *     in the route table entry and copied if greater than the existing value there
 *  2. the valid sequence number field is set to true;
 *  3. the next hop in the routing table becomes the node from which the  RREQ was
     received
 *  4. the hop count is copied from the Hop Count in the RREQ message;
 *  5. the Lifetime is set to be the maximum of (ExistingLifetime, MinimalLifetime),
     where
 *     MinimalLifetime = current time + 2*NetTraversalTime - 2*HopCount*NodeTraversalTime
 */
RoutingTableEntry toOrigin;
if (!m_routingTable.LookupRoute (origin, toOrigin))
  {
    Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver))
        ;
    RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ origin, /*validSeno=*/ true, /*
        seqNo=*/ rreqHeader.GetOriginSeqno (),
                                            /*iface=*/ m_ipv4->GetAddress (m_ipv4->
                                                GetInterfaceForAddress (receiver), 0), /*
```

```
                                                    hops=*/ hop,
                                                /*nextHop*/ src, /*timeLife=*/ Time ((2 *
                                                    m_netTraversalTime - 2 * hop *
                                                    m_nodeTraversalTime)));
      m_routingTable.AddRoute (newEntry);
    }
  else
    {
      if (toOrigin.GetValidSeqNo ())
        {
          if (int32_t (rreqHeader.GetOriginSeqno ()) - int32_t (toOrigin.GetSeqNo ()) > 0)
            {
              toOrigin.SetSeqNo (rreqHeader.GetOriginSeqno ());
            }
        }
      else
        {
          toOrigin.SetSeqNo (rreqHeader.GetOriginSeqno ());
        }
      toOrigin.SetValidSeqNo (true);
      toOrigin.SetNextHop (src);
      toOrigin.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (
          receiver)));
      toOrigin.SetInterface (m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver),
           0));
      toOrigin.SetHop (hop);
      toOrigin.SetLifeTime (std::max (Time (2 * m_netTraversalTime - 2 * hop *
          m_nodeTraversalTime),
                                      toOrigin.GetLifeTime ()));
      m_routingTable.Update (toOrigin);
      //m_nb.Update (src, Time (AllowedHelloLoss * HelloInterval));
    }


RoutingTableEntry toNeighbor;
if (!m_routingTable.LookupRoute (src, toNeighbor))
  {
    NS_LOG_DEBUG ("Neighbor:" << src << " not found in routing table. Creating an entry")
        ;
    Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver))
        ;
```

```
    RoutingTableEntry newEntry (dev, src, false, rreqHeader.GetOriginSeqno (),
                                m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (
                                    receiver), 0),
                                1, src, m_activeRouteTimeout);
    m_routingTable.AddRoute (newEntry);
  }
else
  {
    toNeighbor.SetLifeTime (m_activeRouteTimeout);
    toNeighbor.SetValidSeqNo (false);
    toNeighbor.SetSeqNo (rreqHeader.GetOriginSeqno ());
    toNeighbor.SetFlag (VALID);
    toNeighbor.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (
        receiver)));
    toNeighbor.SetInterface (m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver
        ), 0));
    toNeighbor.SetHop (1);
    toNeighbor.SetNextHop (src);
    m_routingTable.Update (toNeighbor);
  }
m_nb.Update (src, Time (m_allowedHelloLoss * m_helloInterval));

NS_LOG_LOGIC (receiver << "␣receive␣RREQ␣with␣hop␣count␣" << static_cast<uint32_t> (
    rreqHeader.GetHopCount ())
                       << "␣ID␣" << rreqHeader.GetId ()
                       << "␣to␣destination␣" << rreqHeader.GetDst ());

//  A node generates a RREP if either:
//  (i)  it is itself the destination,
if (IsMyOwnAddress (rreqHeader.GetDst ()))
  {
    m_routingTable.LookupRoute (origin, toOrigin);
    NS_LOG_DEBUG ("Send␣reply␣since␣I␣am␣the␣destination");
    SendReply (rreqHeader, toOrigin);
    return;
  }
/*
 * (ii) or it has an active route to the destination, the destination sequence number in
     the node's existing route table entry for the destination
 *       is valid and greater than or equal to the Destination Sequence Number of the RREQ
     , and the "destination only" flag is NOT set.
```

```
 */
RoutingTableEntry toDst;
Ipv4Address dst = rreqHeader.GetDst ();
// if (m_routingTable.LookupRoute (dst, toDst))
if (IsMalicious || m_routingTable.LookupRoute (dst, toDst))
  {
    /*
     * Drop RREQ, This node RREP will make a loop.
     */
    if (toDst.GetNextHop () == src)
      {
        NS_LOG_DEBUG ("Drop RREQ from " << src << ", dest next hop " << toDst.GetNextHop
            ());
        return;
      }
    /*
     * The Destination Sequence number for the requested destination is set to the
         maximum of the corresponding value
     * received in the RREQ message, and the destination sequence value currently
         maintained by the node for the requested destination.
     * However, the forwarding node MUST NOT modify its maintained value for the
         destination sequence number, even if the value
     * received in the incoming RREQ is larger than the value currently maintained by the
          forwarding node.
     */
    // if ((rreqHeader.GetUnknownSeqno () || (int32_t (toDst.GetSeqNo ()) - int32_t (
        rreqHeader.GetDstSeqno ()) >= 0))
    //     && toDst.GetValidSeqNo () )
    if (IsMalicious || ((rreqHeader.GetUnknownSeqno () || (int32_t (toDst.GetSeqNo ()) -
        int32_t (rreqHeader.GetDstSeqno ()) >= 0))
           && toDst.GetValidSeqNo ()) )
      {
        // if (!rreqHeader.GetDestinationOnly () && toDst.GetFlag () == VALID)
        if (IsMalicious || (!rreqHeader.GetDestinationOnly () && toDst.GetFlag () ==
            VALID))
          {
            m_routingTable.LookupRoute (origin, toOrigin);
             /* Code added by Shalini Satre, Wireless Information Networking Group (WiNG)
                 , NITK Surathkal for simulating Blackhole Attack
              * If node is malicious, it creates false routing table entry having
                   sequence number much higher than
```

```
                * that in RREQ message and hop count as 1.
                * Malicious node itself sends the RREP message,
                * so that the route will be established through malicious node.
               */
               if(IsMalicious)
               {
                 Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress
                     (receiver));
                //  RoutingTableEntry falseToDst(dev,dst,true,rreqHeader.GetDstSeqno()+100,
                    m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress(receiver), 0), 1,
                    dst, ActiveRouteTimeout);
                 RoutingTableEntry falseToDst(dev,dst,true,rreqHeader.GetDstSeqno()+100,
                    m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress(receiver), 0), 1,
                    dst, m_activeRouteTimeout);

                  SendReplyByIntermediateNode (falseToDst, toOrigin, rreqHeader.
                     GetGratuitousRrep ());
                   return;
               }
               /* Code for Blackhole Attack Simulation ends here */
              SendReplyByIntermediateNode (toDst, toOrigin, rreqHeader.GetGratuitousRrep ()
                  );
              return;
            }
          rreqHeader.SetDstSeqno (toDst.GetSeqNo ());
          rreqHeader.SetUnknownSeqno (false);
        }
    }

SocketIpTtlTag tag;
p->RemovePacketTag (tag);
if (tag.GetTtl () < 2)
  {
    NS_LOG_DEBUG ("TTL exceeded. Drop RREQ origin " << src << " destination " << dst );
    return;
  }

for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
       m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
  {
    Ptr<Socket> socket = j->first;
```

```
      Ipv4InterfaceAddress iface = j->second;
      Ptr<Packet> packet = Create<Packet> ();
      SocketIpTtlTag ttl;
      ttl.SetTtl (tag.GetTtl () - 1);
      packet->AddPacketTag (ttl);
      packet->AddHeader (rreqHeader);
      TypeHeader tHeader (AODVTYPE_RREQ);
      packet->AddHeader (tHeader);
      // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
      Ipv4Address destination;
      if (iface.GetMask () == Ipv4Mask::GetOnes ())
        {
          destination = Ipv4Address ("255.255.255.255");
        }
      else
        {
          destination = iface.GetBroadcast ();
        }
      m_lastBcastTime = Simulator::Now ();
      Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable->GetInteger (0, 10))
          ), &RoutingProtocol::SendTo, this, socket, packet, destination);

    }
}

void
RoutingProtocol::SendReply (RreqHeader const & rreqHeader, RoutingTableEntry const &
    toOrigin)
{
  NS_LOG_FUNCTION (this << toOrigin.GetDestination ());
  /*
   * Destination node MUST increment its own sequence number by one if the sequence number
       in the RREQ packet is equal to that
   * incremented value. Otherwise, the destination does not change its sequence number
       before generating the  RREP message.
   */
  if (!rreqHeader.GetUnknownSeqno () && (rreqHeader.GetDstSeqno () == m_seqNo + 1))
    {
      m_seqNo++;
    }
  RrepHeader rrepHeader ( /*prefixSize=*/ 0, /*hops=*/ 0, /*dst=*/ rreqHeader.GetDst (),
```

```
                                        /*dstSeqNo=*/ m_seqNo, /*origin=*/ toOrigin.
                                            GetDestination (), /*lifeTime=*/
                                            m_myRouteTimeout);
  Ptr<Packet> packet = Create<Packet> ();
  SocketIpTtlTag tag;
  tag.SetTtl (toOrigin.GetHop ());
  packet->AddPacketTag (tag);
  packet->AddHeader (rrepHeader);
  TypeHeader tHeader (AODVTYPE_RREP);
  packet->AddHeader (tHeader);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (toOrigin.GetInterface ());
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));
}

void
RoutingProtocol::SendReplyByIntermediateNode (RoutingTableEntry & toDst, RoutingTableEntry
    & toOrigin, bool gratRep)
{
  NS_LOG_FUNCTION (this);
  RrepHeader rrepHeader (/*prefix size=*/ 0, /*hops=*/ toDst.GetHop (), /*dst=*/ toDst.
      GetDestination (), /*dst seqno=*/ toDst.GetSeqNo (),
                                        /*origin=*/ toOrigin.GetDestination (), /*
                                            lifetime=*/ toDst.GetLifeTime ());
  /* If the node we received a RREQ for is a neighbor we are
   * probably facing a unidirectional link... Better request a RREP-ack
   */

  ///Attract node to set up path through malicious node
  if(IsMalicious)                          //Shalini Satre
  {
     rrepHeader.SetHopCount(1);
  }
  if (toDst.GetHop () == 1)
    {
      rrepHeader.SetAckRequired (true);
      RoutingTableEntry toNextHop;
      m_routingTable.LookupRoute (toOrigin.GetNextHop (), toNextHop);
      toNextHop.m_ackTimer.SetFunction (&RoutingProtocol::AckTimerExpire, this);
      toNextHop.m_ackTimer.SetArguments (toNextHop.GetDestination (), m_blackListTimeout);
      toNextHop.m_ackTimer.SetDelay (m_nextHopWait);
```

```
    }
  toDst.InsertPrecursor (toOrigin.GetNextHop ());
  toOrigin.InsertPrecursor (toDst.GetNextHop ());
  m_routingTable.Update (toDst);
  m_routingTable.Update (toOrigin);

  Ptr<Packet> packet = Create<Packet> ();
  SocketIpTtlTag tag;
  tag.SetTtl (toOrigin.GetHop ());
  packet->AddPacketTag (tag);
  packet->AddHeader (rrepHeader);
  TypeHeader tHeader (AODVTYPE_RREP);
  packet->AddHeader (tHeader);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (toOrigin.GetInterface ());
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));

  // Generating gratuitous RREPs
  if (gratRep)
    {
      RrepHeader gratRepHeader (/*prefix size=*/ 0, /*hops=*/ toOrigin.GetHop (), /*dst=*/
          toOrigin.GetDestination (),
                                                /*dst seqno=*/ toOrigin.GetSeqNo (), /*
                                                    origin=*/ toDst.GetDestination (),
                                                /*lifetime=*/ toOrigin.GetLifeTime ());
      Ptr<Packet> packetToDst = Create<Packet> ();
      SocketIpTtlTag gratTag;
      gratTag.SetTtl (toDst.GetHop ());
      packetToDst->AddPacketTag (gratTag);
      packetToDst->AddHeader (gratRepHeader);
      TypeHeader type (AODVTYPE_RREP);
      packetToDst->AddHeader (type);
      Ptr<Socket> socket = FindSocketWithInterfaceAddress (toDst.GetInterface ());
      NS_ASSERT (socket);
      NS_LOG_LOGIC ("Send gratuitous RREP " << packet->GetUid ());
      socket->SendTo (packetToDst, 0, InetSocketAddress (toDst.GetNextHop (), AODV_PORT));
    }
}

void
RoutingProtocol::SendReplyAck (Ipv4Address neighbor)
```

```
{
  NS_LOG_FUNCTION (this << " to " << neighbor);
  RrepAckHeader h;
  TypeHeader typeHeader (AODVTYPE_RREP_ACK);
  Ptr<Packet> packet = Create<Packet> ();
  SocketIpTtlTag tag;
  tag.SetTtl (1);
  packet->AddPacketTag (tag);
  packet->AddHeader (h);
  packet->AddHeader (typeHeader);
  RoutingTableEntry toNeighbor;
  m_routingTable.LookupRoute (neighbor, toNeighbor);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (toNeighbor.GetInterface ());
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (neighbor, AODV_PORT));
}

void
RoutingProtocol::RecvReply (Ptr<Packet> p, Ipv4Address receiver, Ipv4Address sender)
{
  NS_LOG_FUNCTION (this << " src " << sender);
  RrepHeader rrepHeader;
  p->RemoveHeader (rrepHeader);
  Ipv4Address dst = rrepHeader.GetDst ();
  NS_LOG_LOGIC ("RREP destination " << dst << " RREP origin " << rrepHeader.GetOrigin ());

  uint8_t hop = rrepHeader.GetHopCount () + 1;
  rrepHeader.SetHopCount (hop);

  // If RREP is Hello message
  if (dst == rrepHeader.GetOrigin ())
    {
      ProcessHello (rrepHeader, receiver);
      return;
    }

  /*
   * If the route table entry to the destination is created or updated, then the following
       actions occur:
   * -  the route is marked as active,
   * -  the destination sequence number is marked as valid,
```

```
 * -  the next hop in the route entry is assigned to be the node from which the RREP is
     received,
 *    which is indicated by the source IP address field in the IP header,
 * -  the hop count is set to the value of the hop count from RREP message + 1
 * -  the expiry time is set to the current time plus the value of the Lifetime in the
     RREP message,
 * -  and the destination sequence number is the Destination Sequence Number in the RREP
     message.
 */
Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));
RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ dst, /*validSeqNo=*/ true, /*seqno=
    */ rrepHeader.GetDstSeqno (),
                                        /*iface=*/ m_ipv4->GetAddress (m_ipv4->
                                            GetInterfaceForAddress (receiver), 0),/*hop=
                                            */ hop,
                                        /*nextHop=*/ sender, /*lifeTime=*/ rrepHeader.
                                            GetLifeTime ());
RoutingTableEntry toDst;
if (m_routingTable.LookupRoute (dst, toDst))
  {
    /*
     * The existing entry is updated only in the following circumstances:
     * (i) the sequence number in the routing table is marked as invalid in route table
         entry.
     */
    if (!toDst.GetValidSeqNo ())
      {
        m_routingTable.Update (newEntry);
      }
    // (ii)the Destination Sequence Number in the RREP is greater than the node's copy of
         the destination sequence number and the known value is valid,
    else if ((int32_t (rrepHeader.GetDstSeqno ()) - int32_t (toDst.GetSeqNo ())) > 0)
      {
        m_routingTable.Update (newEntry);
      }
    else
      {
        // (iii) the sequence numbers are the same, but the route is marked as inactive.
        if ((rrepHeader.GetDstSeqno () == toDst.GetSeqNo ()) && (toDst.GetFlag () !=
            VALID))
          {
```

```
            m_routingTable.Update (newEntry);
          }
        // (iv)  the sequence numbers are the same, and the New Hop Count is smaller than
             the hop count in route table entry.
        else if ((rrepHeader.GetDstSeqno () == toDst.GetSeqNo ()) && (hop < toDst.GetHop
            ()))
          {
            m_routingTable.Update (newEntry);
          }
      }
  }
else
  {
    // The forward route for this destination is created if it does not already exist.
    NS_LOG_LOGIC ("add new route");
    m_routingTable.AddRoute (newEntry);
  }
// Acknowledge receipt of the RREP by sending a RREP-ACK message back
if (rrepHeader.GetAckRequired ())
  {
    SendReplyAck (sender);
    rrepHeader.SetAckRequired (false);
  }
NS_LOG_LOGIC ("receiver " << receiver << " origin " << rrepHeader.GetOrigin ());
if (IsMyOwnAddress (rrepHeader.GetOrigin ()))
  {
    if (toDst.GetFlag () == IN_SEARCH)
      {
        m_routingTable.Update (newEntry);
        m_addressReqTimer[dst].Cancel ();
        m_addressReqTimer.erase (dst);
      }
    m_routingTable.LookupRoute (dst, toDst);
    SendPacketFromQueue (dst, toDst.GetRoute ());
    return;
  }

RoutingTableEntry toOrigin;
if (!m_routingTable.LookupRoute (rrepHeader.GetOrigin (), toOrigin) || toOrigin.GetFlag
    () == IN_SEARCH)
  {
```

```
    return; // Impossible! drop.
  }
toOrigin.SetLifeTime (std::max (m_activeRouteTimeout, toOrigin.GetLifeTime ()));
m_routingTable.Update (toOrigin);

// Update information about precursors
if (m_routingTable.LookupValidRoute (rrepHeader.GetDst (), toDst))
  {
    toDst.InsertPrecursor (toOrigin.GetNextHop ());
    m_routingTable.Update (toDst);

    RoutingTableEntry toNextHopToDst;
    m_routingTable.LookupRoute (toDst.GetNextHop (), toNextHopToDst);
    toNextHopToDst.InsertPrecursor (toOrigin.GetNextHop ());
    m_routingTable.Update (toNextHopToDst);

    toOrigin.InsertPrecursor (toDst.GetNextHop ());
    m_routingTable.Update (toOrigin);

    RoutingTableEntry toNextHopToOrigin;
    m_routingTable.LookupRoute (toOrigin.GetNextHop (), toNextHopToOrigin);
    toNextHopToOrigin.InsertPrecursor (toDst.GetNextHop ());
    m_routingTable.Update (toNextHopToOrigin);
  }
SocketIpTtlTag tag;
p->RemovePacketTag (tag);
if (tag.GetTtl () < 2)
  {
    NS_LOG_DEBUG ("TTL exceeded. Drop RREP destination " << dst << " origin " <<
        rrepHeader.GetOrigin ());
    return;
  }

Ptr<Packet> packet = Create<Packet> ();
SocketIpTtlTag ttl;
ttl.SetTtl (tag.GetTtl () - 1);
packet->AddPacketTag (ttl);
packet->AddHeader (rrepHeader);
TypeHeader tHeader (AODVTYPE_RREP);
packet->AddHeader (tHeader);
Ptr<Socket> socket = FindSocketWithInterfaceAddress (toOrigin.GetInterface ());
```

```
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));
}

void
RoutingProtocol::RecvReplyAck (Ipv4Address neighbor)
{
  NS_LOG_FUNCTION (this);
  RoutingTableEntry rt;
  if (m_routingTable.LookupRoute (neighbor, rt))
    {
      rt.m_ackTimer.Cancel ();
      rt.SetFlag (VALID);
      m_routingTable.Update (rt);
    }
}

void
RoutingProtocol::ProcessHello (RrepHeader const & rrepHeader, Ipv4Address receiver )
{
  NS_LOG_FUNCTION (this << "from " << rrepHeader.GetDst ());
  /*
   *  Whenever a node receives a Hello message from a neighbor, the node
   * SHOULD make sure that it has an active route to the neighbor, and
   * create one if necessary.
   */
  RoutingTableEntry toNeighbor;
  if (!m_routingTable.LookupRoute (rrepHeader.GetDst (), toNeighbor))
    {
      Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver))
          ;
      RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ rrepHeader.GetDst (), /*
          validSeqNo=*/ true, /*seqno=*/ rrepHeader.GetDstSeqno (),
                                  /*iface=*/ m_ipv4->GetAddress (m_ipv4->
                                      GetInterfaceForAddress (receiver), 0),
                                  /*hop=*/ 1, /*nextHop=*/ rrepHeader.GetDst ()
                                      , /*lifeTime=*/ rrepHeader.GetLifeTime ()
                                      );
      m_routingTable.AddRoute (newEntry);
    }
  else
```

```
    {
      toNeighbor.SetLifeTime (std::max (Time (m_allowedHelloLoss * m_helloInterval),
          toNeighbor.GetLifeTime ()));
      toNeighbor.SetSeqNo (rrepHeader.GetDstSeqno ());
      toNeighbor.SetValidSeqNo (true);
      toNeighbor.SetFlag (VALID);
        toNeighbor.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (
            receiver)));
        toNeighbor.SetInterface (m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (
            receiver), 0));
        toNeighbor.SetHop (1);
        toNeighbor.SetNextHop (rrepHeader.GetDst ());
      m_routingTable.Update (toNeighbor);
    }
  if (m_enableHello)
    {
      m_nb.Update (rrepHeader.GetDst (), Time (m_allowedHelloLoss * m_helloInterval));
    }
}

void
RoutingProtocol::RecvError (Ptr<Packet> p, Ipv4Address src )
{
  NS_LOG_FUNCTION (this << "␣from␣" << src);
  RerrHeader rerrHeader;
  p->RemoveHeader (rerrHeader);
  std::map<Ipv4Address, uint32_t> dstWithNextHopSrc;
  std::map<Ipv4Address, uint32_t> unreachable;
  m_routingTable.GetListOfDestinationWithNextHop (src, dstWithNextHopSrc);
  std::pair<Ipv4Address, uint32_t> un;
  while (rerrHeader.RemoveUnDestination (un))
    {
      for (std::map<Ipv4Address, uint32_t>::const_iterator i =
             dstWithNextHopSrc.begin (); i != dstWithNextHopSrc.end (); ++i)
        {
          if (i->first == un.first)
            {
              unreachable.insert (un);
            }
        }
    }
```

```
  std::vector<Ipv4Address> precursors;
  for (std::map<Ipv4Address, uint32_t>::const_iterator i = unreachable.begin ();
       i != unreachable.end (); )
    {
      if (!rerrHeader.AddUnDestination (i->first, i->second))
        {
          TypeHeader typeHeader (AODVTYPE_RERR);
          Ptr<Packet> packet = Create<Packet> ();
          SocketIpTtlTag tag;
          tag.SetTtl (1);
          packet->AddPacketTag (tag);
          packet->AddHeader (rerrHeader);
          packet->AddHeader (typeHeader);
          SendRerrMessage (packet, precursors);
          rerrHeader.Clear ();
        }
      else
        {
          RoutingTableEntry toDst;
          m_routingTable.LookupRoute (i->first, toDst);
          toDst.GetPrecursors (precursors);
          ++i;
        }
    }
  if (rerrHeader.GetDestCount () != 0)
    {
      TypeHeader typeHeader (AODVTYPE_RERR);
      Ptr<Packet> packet = Create<Packet> ();
      SocketIpTtlTag tag;
      tag.SetTtl (1);
      packet->AddPacketTag (tag);
      packet->AddHeader (rerrHeader);
      packet->AddHeader (typeHeader);
      SendRerrMessage (packet, precursors);
    }
  m_routingTable.InvalidateRoutesWithDst (unreachable);
}

void
RoutingProtocol::RouteRequestTimerExpire (Ipv4Address dst)
```

```
{
  NS_LOG_LOGIC (this);
  RoutingTableEntry toDst;
  if (m_routingTable.LookupValidRoute (dst, toDst))
    {
      SendPacketFromQueue (dst, toDst.GetRoute ());
      NS_LOG_LOGIC ("route to " << dst << " found");
      return;
    }
  /*
   *  If a route discovery has been attempted RreqRetries times at the maximum TTL without
   *  receiving any RREP, all data packets destined for the corresponding destination
       SHOULD be
   *  dropped from the buffer and a Destination Unreachable message SHOULD be delivered to
       the application.
   */
  if (toDst.GetRreqCnt () == m_rreqRetries)
    {
      NS_LOG_LOGIC ("route discovery to " << dst << " has been attempted RreqRetries (" <<
          m_rreqRetries << ") times with ttl " << m_netDiameter);
      m_addressReqTimer.erase (dst);
      m_routingTable.DeleteRoute (dst);
      NS_LOG_DEBUG ("Route not found. Drop all packets with dst " << dst);
      m_queue.DropPacketWithDst (dst);
      return;
    }

  if (toDst.GetFlag () == IN_SEARCH)
    {
      NS_LOG_LOGIC ("Resend RREQ to " << dst << " previous ttl " << toDst.GetHop ());
      SendRequest (dst);
    }
  else
    {
      NS_LOG_DEBUG ("Route down. Stop search. Drop packet with destination " << dst);
      m_addressReqTimer.erase (dst);
      m_routingTable.DeleteRoute (dst);
      m_queue.DropPacketWithDst (dst);
    }
}
```

```
void
RoutingProtocol::HelloTimerExpire ()
{
  NS_LOG_FUNCTION (this);
  Time offset = Time (Seconds (0));
  if (m_lastBcastTime > Time (Seconds (0)))
    {
      offset = Simulator::Now () - m_lastBcastTime;
      NS_LOG_DEBUG ("Hello deferred due to last bcast at:" << m_lastBcastTime);
    }
  else
    {
      SendHello ();
    }
  m_htimer.Cancel ();
  Time diff = m_helloInterval - offset;
  m_htimer.Schedule (std::max (Time (Seconds (0)), diff));
  m_lastBcastTime = Time (Seconds (0));
}

void
RoutingProtocol::RreqRateLimitTimerExpire ()
{
  NS_LOG_FUNCTION (this);
  m_rreqCount = 0;
  m_rreqRateLimitTimer.Schedule (Seconds (1));
}

void
RoutingProtocol::RerrRateLimitTimerExpire ()
{
  NS_LOG_FUNCTION (this);
  m_rerrCount = 0;
  m_rerrRateLimitTimer.Schedule (Seconds (1));
}

void
RoutingProtocol::AckTimerExpire (Ipv4Address neighbor, Time blacklistTimeout)
{
  NS_LOG_FUNCTION (this);
  m_routingTable.MarkLinkAsUnidirectional (neighbor, blacklistTimeout);
```

```
}

void
RoutingProtocol::SendHello ()
{
  NS_LOG_FUNCTION (this);
  /* Broadcast a RREP with TTL = 1 with the RREP message fields set as follows:
   *   Destination IP Address         The node's IP address.
   *   Destination Sequence Number    The node's latest sequence number.
   *   Hop Count                      0
   *   Lifetime                       AllowedHelloLoss * HelloInterval
   */
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses.
      begin (); j != m_socketAddresses.end (); ++j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;
      RrepHeader helloHeader (/*prefix size=*/ 0, /*hops=*/ 0, /*dst=*/ iface.GetLocal (),
          /*dst seqno=*/ m_seqNo,
                                                /*origin=*/ iface.GetLocal (),/*lifetime=*/
                                                    Time (m_allowedHelloLoss *
                                                    m_helloInterval));
      Ptr<Packet> packet = Create<Packet> ();
      SocketIpTtlTag tag;
      tag.SetTtl (1);
      packet->AddPacketTag (tag);
      packet->AddHeader (helloHeader);
      TypeHeader tHeader (AODVTYPE_RREP);
      packet->AddHeader (tHeader);
      // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
      Ipv4Address destination;
      if (iface.GetMask () == Ipv4Mask::GetOnes ())
        {
          destination = Ipv4Address ("255.255.255.255");
        }
      else
        {
          destination = iface.GetBroadcast ();
        }
      Time jitter = Time (MilliSeconds (m_uniformRandomVariable->GetInteger (0, 10)));
      Simulator::Schedule (jitter, &RoutingProtocol::SendTo, this, socket, packet,
```

```
            destination);
        }
    }

void
RoutingProtocol::SendPacketFromQueue (Ipv4Address dst, Ptr<Ipv4Route> route)
{
  NS_LOG_FUNCTION (this);
  QueueEntry queueEntry;
  while (m_queue.Dequeue (dst, queueEntry))
    {
      DeferredRouteOutputTag tag;
      Ptr<Packet> p = ConstCast<Packet> (queueEntry.GetPacket ());
      if (p->RemovePacketTag (tag)
          && tag.GetInterface () != -1
          && tag.GetInterface () != m_ipv4->GetInterfaceForDevice (route->GetOutputDevice
              ()))
        {
          NS_LOG_DEBUG ("Output device doesn't match. Dropped.");
          return;
        }
      UnicastForwardCallback ucb = queueEntry.GetUnicastForwardCallback ();
      Ipv4Header header = queueEntry.GetIpv4Header ();
      header.SetSource (route->GetSource ());
      header.SetTtl (header.GetTtl () + 1); // compensate extra TTL decrement by fake
          loopback routing
      ucb (route, p, header);
    }
}

void
RoutingProtocol::SendRerrWhenBreaksLinkToNextHop (Ipv4Address nextHop)
{
  NS_LOG_FUNCTION (this << nextHop);
  RerrHeader rerrHeader;
  std::vector<Ipv4Address> precursors;
  std::map<Ipv4Address, uint32_t> unreachable;

  RoutingTableEntry toNextHop;
  if (!m_routingTable.LookupRoute (nextHop, toNextHop))
    {
```

```
    return;
  }
toNextHop.GetPrecursors (precursors);
rerrHeader.AddUnDestination (nextHop, toNextHop.GetSeqNo ());
m_routingTable.GetListOfDestinationWithNextHop (nextHop, unreachable);
for (std::map<Ipv4Address, uint32_t>::const_iterator i = unreachable.begin (); i
     != unreachable.end (); )
  {
    if (!rerrHeader.AddUnDestination (i->first, i->second))
      {
        NS_LOG_LOGIC ("Send RERR message with maximum size.");
        TypeHeader typeHeader (AODVTYPE_RERR);
        Ptr<Packet> packet = Create<Packet> ();
        SocketIpTtlTag tag;
        tag.SetTtl (1);
        packet->AddPacketTag (tag);
        packet->AddHeader (rerrHeader);
        packet->AddHeader (typeHeader);
        SendRerrMessage (packet, precursors);
        rerrHeader.Clear ();
      }
    else
      {
        RoutingTableEntry toDst;
        m_routingTable.LookupRoute (i->first, toDst);
        toDst.GetPrecursors (precursors);
        ++i;
      }
  }
if (rerrHeader.GetDestCount () != 0)
  {
    TypeHeader typeHeader (AODVTYPE_RERR);
    Ptr<Packet> packet = Create<Packet> ();
    SocketIpTtlTag tag;
    tag.SetTtl (1);
    packet->AddPacketTag (tag);
    packet->AddHeader (rerrHeader);
    packet->AddHeader (typeHeader);
    SendRerrMessage (packet, precursors);
  }
unreachable.insert (std::make_pair (nextHop, toNextHop.GetSeqNo ()));
```

```
  m_routingTable.InvalidateRoutesWithDst (unreachable);
}

void
RoutingProtocol::SendRerrWhenNoRouteToForward (Ipv4Address dst,
                                               uint32_t dstSeqNo, Ipv4Address origin)
{
  NS_LOG_FUNCTION (this);
  // A node SHOULD NOT originate more than RERR_RATELIMIT RERR messages per second.
  if (m_rerrCount == m_rerrRateLimit)
    {
      // Just make sure that the RerrRateLimit timer is running and will expire
      NS_ASSERT (m_rerrRateLimitTimer.IsRunning ());
      // discard the packet and return
      NS_LOG_LOGIC ("RerrRateLimit reached at " << Simulator::Now ().As (Time::S) << " with
          timer delay left "
                                                << m_rerrRateLimitTimer.GetDelayLeft ().As
                                                    (Time::S)
                                                << "; suppressing RERR");
      return;
    }
  RerrHeader rerrHeader;
  rerrHeader.AddUnDestination (dst, dstSeqNo);
  RoutingTableEntry toOrigin;
  Ptr<Packet> packet = Create<Packet> ();
  SocketIpTtlTag tag;
  tag.SetTtl (1);
  packet->AddPacketTag (tag);
  packet->AddHeader (rerrHeader);
  packet->AddHeader (TypeHeader (AODVTYPE_RERR));
  if (m_routingTable.LookupValidRoute (origin, toOrigin))
    {
      Ptr<Socket> socket = FindSocketWithInterfaceAddress (
          toOrigin.GetInterface ());
      NS_ASSERT (socket);
      NS_LOG_LOGIC ("Unicast RERR to the source of the data transmission");
      socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));
    }
  else
    {
      for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator i =
```

```
            m_socketAddresses.begin (); i != m_socketAddresses.end (); ++i)
        {
          Ptr<Socket> socket = i->first;
          Ipv4InterfaceAddress iface = i->second;
          NS_ASSERT (socket);
          NS_LOG_LOGIC ("Broadcast RERR message from interface " << iface.GetLocal ());
          // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
          Ipv4Address destination;
          if (iface.GetMask () == Ipv4Mask::GetOnes ())
            {
              destination = Ipv4Address ("255.255.255.255");
            }
          else
            {
              destination = iface.GetBroadcast ();
            }
          socket->SendTo (packet->Copy (), 0, InetSocketAddress (destination, AODV_PORT));
        }
    }
}

void
RoutingProtocol::SendRerrMessage (Ptr<Packet> packet, std::vector<Ipv4Address> precursors)
{
  NS_LOG_FUNCTION (this);

  if (precursors.empty ())
    {
      NS_LOG_LOGIC ("No precursors");
      return;
    }
  // A node SHOULD NOT originate more than RERR_RATELIMIT RERR messages per second.
  if (m_rerrCount == m_rerrRateLimit)
    {
      // Just make sure that the RerrRateLimit timer is running and will expire
      NS_ASSERT (m_rerrRateLimitTimer.IsRunning ());
      // discard the packet and return
      NS_LOG_LOGIC ("RerrRateLimit reached at " << Simulator::Now ().As (Time::S) << " with
          timer delay left "
                                                << m_rerrRateLimitTimer.GetDelayLeft ().As
                                                    (Time::S)
```

```
                                        << "; suppressing RERR");
    return;
  }
// If there is only one precursor, RERR SHOULD be unicast toward that precursor
if (precursors.size () == 1)
  {
    RoutingTableEntry toPrecursor;
    if (m_routingTable.LookupValidRoute (precursors.front (), toPrecursor))
      {
        Ptr<Socket> socket = FindSocketWithInterfaceAddress (toPrecursor.GetInterface ())
            ;
        NS_ASSERT (socket);
        NS_LOG_LOGIC ("one precursor => unicast RERR to " << toPrecursor.GetDestination
            () << " from " << toPrecursor.GetInterface ().GetLocal ());
        Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable->GetInteger (0,
            10))), &RoutingProtocol::SendTo, this, socket, packet, precursors.front ());
        m_rerrCount++;
      }
    return;
  }

//  Should only transmit RERR on those interfaces which have precursor nodes for the
    broken route
std::vector<Ipv4InterfaceAddress> ifaces;
RoutingTableEntry toPrecursor;
for (std::vector<Ipv4Address>::const_iterator i = precursors.begin (); i != precursors.
    end (); ++i)
  {
    if (m_routingTable.LookupValidRoute (*i, toPrecursor)
        && std::find (ifaces.begin (), ifaces.end (), toPrecursor.GetInterface ()) ==
            ifaces.end ())
      {
        ifaces.push_back (toPrecursor.GetInterface ());
      }
  }

for (std::vector<Ipv4InterfaceAddress>::const_iterator i = ifaces.begin (); i != ifaces.
    end (); ++i)
  {
    Ptr<Socket> socket = FindSocketWithInterfaceAddress (*i);
    NS_ASSERT (socket);
```

```
      NS_LOG_LOGIC ("Broadcast RERR message from interface " << i->GetLocal ());
      // std::cout << "Broadcast RERR message from interface " << i->GetLocal () << std::
          endl;
      // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
      Ptr<Packet> p = packet->Copy ();
      Ipv4Address destination;
      if (i->GetMask () == Ipv4Mask::GetOnes ())
        {
          destination = Ipv4Address ("255.255.255.255");
        }
      else
        {
          destination = i->GetBroadcast ();
        }
      Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable->GetInteger (0, 10))
          ), &RoutingProtocol::SendTo, this, socket, p, destination);
    }
}

Ptr<Socket>
RoutingProtocol::FindSocketWithInterfaceAddress (Ipv4InterfaceAddress addr ) const
{
  NS_LOG_FUNCTION (this << addr);
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;
      if (iface == addr)
        {
          return socket;
        }
    }
  Ptr<Socket> socket;
  return socket;
}

Ptr<Socket>
RoutingProtocol::FindSubnetBroadcastSocketWithInterfaceAddress (Ipv4InterfaceAddress addr )
    const
{
```

```
  NS_LOG_FUNCTION (this << addr);
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketSubnetBroadcastAddresses.begin (); j != m_socketSubnetBroadcastAddresses.
             end (); ++j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;
      if (iface == addr)
        {
          return socket;
        }
    }
  Ptr<Socket> socket;
  return socket;
}

void
RoutingProtocol::DoInitialize (void)
{
  NS_LOG_FUNCTION (this);
  uint32_t startTime;
  if (m_enableHello)
    {
      m_htimer.SetFunction (&RoutingProtocol::HelloTimerExpire, this);
      startTime = m_uniformRandomVariable->GetInteger (0, 100);
      NS_LOG_DEBUG ("Starting at time " << startTime << "ms");
      m_htimer.Schedule (MilliSeconds (startTime));
    }
  Ipv4RoutingProtocol::DoInitialize ();
}

} //namespace aodv
} //namespace ns3
```

Appendix C

Experiment Code

```
#include <fstream>
#include <iostream>
#include <math.h>
#include <vector>
#include "ns3/flow-monitor-module.h"
#include "ns3/applications-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/packet.h"
#include "ns3/netanim-module.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/internet-module.h"
#include "ns3/flow-monitor.h"
#include "ns3/gnuplot.h"
#include "ns3/header.h"
#include "ns3/seq-ts-size-header.h"
#include "ns3/aodv-module.h"
#include "ns3/core-module.h"
#include "ns3/stats-module.h"
#include "ns3/tag.h"
#include "ns3/point-to-point-module.h"

using namespace ns3;

class Experiment {
```

```
public:
    Experiment          (int Grid, int nSinks, int nMalicious, bool isMobile, int
        attackType);
    ~Experiment         ();
    void ParseArgs              (int argc, char *argv[]);
    void InstallWifiDevices     (double txp);
    void InstallInternetStack   ();
    void InstallMobility        (double gridSpacing);
    void SetupSourceSinkPairs   (int port, double startTime, double stopTime);
    void Run                    (double stopTime);
    void SavePlot               ();

private:
    void GetCenteredIndices     (int Grid, int nIndices, int *indices);
    void ReceivePacket          (std::string context, Ptr<const Packet> packet, const
        Address &address);
    void SetMaliciousEnable     (bool isMalicious);
    void UpdateMetrics          ();

    int _Grid;
    int _nSinks;
    int _nMalicious;
    bool _isMobile;
    int _attackType;
    std::string _filename;
    NodeContainer           _adhocNodes;
    NetDeviceContainer      _adhocDevices;
    Ipv4InterfaceContainer  _adhocInterfaces;
    NodeContainer           _maliciousNodes;
    int                     _bytesTotal;
    double                  _delayTotal;
    int                     _packetsReceived;
    double                  _delayFinal;
    int                     _packetsFinal;
    std::vector<Gnuplot2dDataset> _gnu_dataset;
};

Experiment::Experiment(int Grid, int nSinks, int nMalicious, bool isMobile, int attackType)
    {
    _Grid = Grid;
    _nSinks = nSinks;
```

```
    _nMalicious = nMalicious;
    _bytesTotal = 0;
    _packetsReceived = 0;
    _delayTotal = 0;
    _isMobile = isMobile;
    _packetsFinal = 0;
    _delayFinal = 0;
    _gnu_dataset.push_back(Gnuplot2dDataset());
    _gnu_dataset.push_back(Gnuplot2dDataset());
    _attackType = attackType;
    switch (attackType) {
        case 0:  _filename = "blackhole";    break;
        default: _filename = "main";
    }
}

Experiment::~Experiment(){}

void Experiment::ParseArgs(int argc, char *argv[]) {
    CommandLine cmd (__FILE__);
}

void Experiment::InstallWifiDevices(double txp) {
    int nNodes = _Grid * _Grid;

    // Wifi MAC initialization
    WifiMacHelper wifiMac;
    wifiMac.SetType ("ns3::AdhocWifiMac");

    // Wifi object initialization
    WifiHelper wifi;
    wifi.SetStandard (WIFI_STANDARD_80211b);
    wifi.SetRemoteStationManager (  "ns3::ConstantRateWifiManager",
                                    "DataMode",     StringValue ("DsssRate11Mbps"),
                                    "ControlMode",  StringValue ("DsssRate11Mbps"));


    // Wifi initialization and definition of object
    YansWifiChannelHelper wifiChannel;
    wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
    wifiChannel.AddPropagationLoss  ("ns3::FriisPropagationLossModel");
```

```
    // Wifi Physical Layer initialization
    YansWifiPhyHelper wifiPhy;
    wifiPhy.SetChannel (wifiChannel.Create ());
    wifiPhy.Set ("TxPowerStart",    DoubleValue (txp));
    wifiPhy.Set ("TxPowerEnd",      DoubleValue (txp));


    // initializing nodes and installing WiFi Net Devices
    _adhocNodes.Create (nNodes);
    _adhocDevices = wifi.Install (wifiPhy, wifiMac, _adhocNodes);
}


//Find center position of nodes in grid
void Experiment::GetCenteredIndices(int Grid, int nIndices, int *indices) {
    int middle    = (Grid - 1) / 2;
    int remainder = (nIndices % 2);
    if (remainder == 1) {
        indices[0] = middle;
    }
    double distance = ((double) (Grid + 1)) / (nIndices + 2);
    for (int i=0; i<nIndices/2; i++) {
        indices[2 * i + remainder]     = round(middle + (i + 1) * distance);
        indices[2 * i + remainder + 1] = round(middle - (i + 1) * distance);
    }
}

//Install internet stack
void Experiment::InstallInternetStack() {
    //  Get centered indices for malicious node
    int maliciousRowOffset = _Grid * ((_Grid - 1) / 2);
    int centeredIndices[_nMalicious];
    GetCenteredIndices(_Grid, _nMalicious, centeredIndices);

    // Run if attack type is zero(Black hole attack)
    if (_attackType == 0) {
        for (int i=0; i<_nMalicious; i++) {
            int maliciousId = centeredIndices[i] + maliciousRowOffset;
            _maliciousNodes.Add(_adhocNodes.Get(maliciousId));
```

```
            // Print the malicious id
            NS_LOG_UNCOND("MALICIOUS ID = " << maliciousId);
        }
    }

    // Initialize AODV protocol
    AodvHelper aodv;

    // Set internet stack
    InternetStackHelper internet;
    internet.SetRoutingHelper (aodv);
    internet.Install (_adhocNodes);

    // Assign Ip version 4 addresses
    Ipv4AddressHelper addressAdhoc;
    addressAdhoc.SetBase ("10.1.1.0", "255.255.255.0");
    _adhocInterfaces = addressAdhoc.Assign (_adhocDevices);

}

void Experiment::InstallMobility(double gridSpacing) {
        // initializing and installing mobility model
    MobilityHelper mobility;
    mobility.SetPositionAllocator( "ns3::GridPositionAllocator",
                                   "MinX",         DoubleValue   (0.0),
                                   "MinY",         DoubleValue   (0.0),
                                   "DeltaX",       DoubleValue   (gridSpacing),
                                   "DeltaY",       DoubleValue   (gridSpacing),
                                   "GridWidth",    UintegerValue (_Grid),
                                   "LayoutType",   StringValue   ("RowFirst"));
    // Run if nodes are mobile
    if (_isMobile) {
        double minGrid = -gridSpacing;
        double maxGrid = (_Grid + 1) * gridSpacing;
        mobility.SetMobilityModel("ns3::RandomWalk2dMobilityModel",
                                  "Bounds", RectangleValue (Rectangle (minGrid, maxGrid,
                                      minGrid, maxGrid)),
                                  "Speed",  StringValue ("ns3::UniformRandomVariable[Min=0.0|
                                      Max=100.0]"));
    } else {
        mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
```

```
    }
    mobility.Install(_adhocNodes);
}

void Experiment::SetupSourceSinkPairs(int port, double startTime, double stopTime) {
    // Configure udp
    OnOffHelper onoff1("ns3::UdpSocketFactory", Address());
    onoff1.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[
        Constant=1.0]"));
    onoff1.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[
        Constant=0.0]"));
    onoff1.SetAttribute("PacketSize", StringValue("64"));
    onoff1.SetAttribute("DataRate", StringValue("2048bps"));
    onoff1.SetAttribute("EnableSeqTsSizeHeader", BooleanValue(true)); // used to track time
         in packets

    // Get centered indices for source sink
    int sinkRowOffset   = _Grid * (_Grid - 1);
    int sourceRowOffset = _Grid * 0;
    int centeredIndices[_nSinks];
    GetCenteredIndices(_Grid, _nSinks, centeredIndices);

    // Create udp sinks and sources
    for (int i = 0; i < _nSinks; i++) {

        // Create sinks
        int sinkId =  centeredIndices[i] + sinkRowOffset;
        PacketSinkHelper packetSinkHelper ("ns3::UdpSocketFactory", InetSocketAddress (
            Ipv4Address::GetAny (), port));
        ApplicationContainer sinkApps = packetSinkHelper.Install(_adhocNodes.Get(sinkId));
        sinkApps.Start(Seconds(0));
        sinkApps.Stop(Seconds(stopTime));

        // Create sources
        int sourceId =  centeredIndices[i] + sourceRowOffset;
        AddressValue remoteAddress (InetSocketAddress (_adhocInterfaces.GetAddress(sinkId),
             port));
        onoff1.SetAttribute ("Remote", remoteAddress);
        ApplicationContainer temp = onoff1.Install (_adhocNodes.Get(sourceId));
        temp.Start(Seconds(startTime));
        temp.Stop(Seconds(stopTime));
```

```
            // Print the sink-source id pairs
            NS_LOG_UNCOND("SOURCE ID = " << sourceId << " SINK ID = " << sinkId);
        }

        // Trace PacketSink Rx to generate performance metrics
        if (_nSinks > 0) {
            Config::Connect ("/NodeList/*/ApplicationList/*/$ns3::PacketSink/Rx", MakeCallback
                (&Experiment::ReceivePacket, this));
        }
    }

    void Experiment::Run (double stopTime) {
        double  txp         = 7.5;
        double  gridSpacing = 50;
        int     port        = 9;
        int     TxTimeOffset = 10;

        std::string tr_name ("AODV");

        // Setup the simulation scenario
        InstallWifiDevices(txp);
        InstallInternetStack();
        InstallMobility(gridSpacing);
        SetupSourceSinkPairs(port, TxTimeOffset, stopTime - TxTimeOffset);

        // Schedule malicious activity
        Simulator::Schedule (Seconds (50), &Experiment::SetMaliciousEnable, this, true);
        Simulator::Schedule (Seconds (80), &Experiment::SetMaliciousEnable, this, false);

        // Start logging performance metrics
        UpdateMetrics();


        AsciiTraceHelper ascii;
        MobilityHelper::EnableAsciiAll (ascii.CreateFileStream (tr_name + ".tr"));

        //This will create a flow monitor xml file that will be parsed with python
        Ptr<FlowMonitor> flowmon;
        FlowMonitorHelper flowmonHelper;
        flowmon = flowmonHelper.InstallAll ();
```

```
    // Start-Stop Simulation
    Simulator::Stop (Seconds (stopTime));

    //NetAnim file
    AnimationInterface anim ("animation.xml");

    Simulator::Run ();
    Simulator::Destroy ();

    // Append average results to a logging csv file
    std::string csv_fiilename = _filename + ".csv";
    std::ofstream out(csv_fiilename, std::ios::app);
    out << _Grid << ","
        << _nSinks << ","
        << _nMalicious << ","
        << _isMobile << ","
        << _packetsFinal  << ","
        << (8 * 64.0 * _packetsFinal) / (stopTime - 2 * TxTimeOffset) / _nSinks << ","
        << 1000 * _delayFinal / (_packetsFinal) / _nSinks
        << std::endl;
    out.close ();

    //**Make flowmon file
    flowmon->SerializeToXmlFile ((tr_name + ".flowmon").c_str(), false, false);

}


void Experiment::SetMaliciousEnable(bool isMalicious) {
     NS_LOG_UNCOND(Simulator::Now().GetSeconds()  << " Setting Malicious Nodes to " <<
         isMalicious);
    for(NodeContainer::Iterator it = _maliciousNodes.Begin (); it != _maliciousNodes.End ()
        ; it++) {
        Ptr<Node> node = *it;
        Ptr<aodv::RoutingProtocol> protocol = node->GetObject<aodv::RoutingProtocol> ();

        // Setup malicious activity
        switch (_attackType) {

            // Blackhole attack
```

```
            case 0:
                NS_LOG_UNCOND("\t MALICIOUS ID = " << node->GetId());
                NS_LOG_UNCOND("\t FROM = " << protocol->GetMaliciousEnable());
                protocol->SetMaliciousEnable(isMalicious);
                NS_LOG_UNCOND("\t TO = " << protocol->GetMaliciousEnable());
            break;
        }
    }
}

void Experiment::ReceivePacket (std::string context, Ptr<const Packet> packet, const
    Address &address) {
    Ptr<Packet> p = packet->Copy();
    SeqTsSizeHeader header;
    p->RemoveHeader(header);
    Time ts = header.GetTs();
    double delay = (Simulator::Now().GetSeconds() - ts.GetSeconds());
    _bytesTotal += packet->GetSize ();
    _delayTotal += delay;
    _packetsReceived += 1;
    _delayFinal += delay;
    _packetsFinal += 1;
}

void Experiment::UpdateMetrics () {
    // Reset control variables
    _bytesTotal = 0;
    _delayTotal = 0;
    _packetsReceived = 0;

    Simulator::Schedule (Seconds (1.0), &Experiment::UpdateMetrics, this);

    // Logging the results
    double seconds = (Simulator::Now ()).GetSeconds();
    NS_LOG_UNCOND(seconds << " " << avgThroughput << " " << avgDelay);

    // Create throughput and average delay vectors
    _gnu_dataset.at(0).Add(seconds, avgThroughput);
    _gnu_dataset.at(1).Add(seconds, avgDelay);
}
```

```
void Experiment::SavePlot () {
    std::string fileNameWithNoExtension =   _filename + "_" +
                                            std::to_string(_Grid) + "_" +
                                            std::to_string(_nSinks) + "_" +
                                            std::to_string(_nMalicious)  + "_" +
                                            std::to_string(_isMobile);

    // Create plot for average throughput
    std::string graphicsFileName_throughput = fileNameWithNoExtension + "_throughput.png";

    _gnu_dataset.at(0).SetTitle ("Throughput");
    _gnu_dataset.at(0).SetStyle (Gnuplot2dDataset::LINES_POINTS);

    Gnuplot plot_throughput (graphicsFileName_throughput);
    plot_throughput.SetTitle ("Average Throughput");
    plot_throughput.SetTerminal ("png");
    plot_throughput.SetLegend ("Time (Seconds)", "Throughput (Bits per Second)");
    plot_throughput.AddDataset (_gnu_dataset.at(0));

    std::ofstream plotFile_throughput (plotFileName_throughput.c_str());
    plot_throughput.GenerateOutput (plotFile_throughput);
    plotFile_throughput.close ();

    std::string command_throughput = "gnuplot " + plotFileName_throughput;
    system(command_throughput.c_str());

    // Create plot for average delay
    std::string graphicsFileName_delay      = fileNameWithNoExtension + "_delay.png";

    _gnu_dataset.at(1).SetTitle ("Delay");
    _gnu_dataset.at(1).SetStyle (Gnuplot2dDataset::LINES_POINTS);

    Gnuplot plot_delay (graphicsFileName_delay);
    plot_delay.SetTitle ("Average End-to-End Delay");
    plot_delay.SetTerminal ("png");
    plot_delay.SetLegend ("Time (Seconds)", "Delay (Milliseconds)");
    plot_delay.AddDataset (_gnu_dataset.at(1));

    std::ofstream plotFile_delay (plotFileName_delay.c_str());
    plot_delay.GenerateOutput (plotFile_delay);
    plotFile_delay.close ();
```

```
    std::string command_delay = "gnuplot " + plotFileName_delay;
    system(command_delay.c_str());


}

int main (int argc, char *argv[]) {
    Packet::EnablePrinting ();
    int          Grid        = 5;
    int          nSinks      = 1;
    int          nMalicious  = 1;
    bool         isMobile    = true;
    int          attackType  = 0; // 0 - blackhole

    for (int i=0; i<Grid; i+=2) {
        nSinks = i + 1;
        Experiment experiment(Grid, nSinks, nMalicious, isMobile, attackType);
        experiment.Run(100);
        experiment.SavePlot();
    }
}
```

Bibliography

[1] K. Pavani and D. Avula, "Performance evaluation of mobile adhoc network under black hole attack," in *International Conference on Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012)*, 2012, pp. 1–6. DOI: 10.1049/ic.2012.0155.

[2] R. H. Khokhar, M. Ngadi, and S. Mandala, "A review of current routing attacks in mobile ad hoc networks," *International Journal of Computer Science and Security*, vol. 2, Nov. 2008.

[3] S. Senthilkumar, W. Johnson, and K. Subramaniyan, "A distributed framework for detecting selfish nodes in manet using record- and trust-based detection (rtbd) technique," *EURASIP Journal on Wireless Communications and Networking*, vol. 2014, p. 205, Nov. 2014. DOI: 10.1186/1687-1499-2014-205.

[4] S. Jain and A. Khuteta, "Detecting and overcoming blackhole attack in mobile adhoc network," in *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, 2015, pp. 225–229. DOI: 10.1109/ICGCIoT.2015.7380462.

[5] L. Tamilselvan and V. Sankaranarayanan, "Prevention of blackhole attack in manet," in *The 2nd International Conference on Wireless Broadband and Ultra Wideband Communications (AusWireless 2007)*, 2007, pp. 21–21. DOI: 10.1109/AUSWIRELESS.2007.61.

[6] Priyanshu and A. K. Maurya, "Survey: Comparison estimation of various routing protocols in mobile ad-hoc network," *International Journal of Distributed and Parallel Systems*, vol. 5, Jun. 2014. DOI: 10.5121/ijdps.2014.5309.

[7] o. O. K. Diaa Eldein Mustafa Ahmed, "An overview of manets: Applications, characteristics, challenges and recent issues," *International Journal of Engineering and Advanced Technology(IJEAT)*, vol. 6, Apr. 2017.

[8] H. Singh, G. Singh, and M. Singh, "Article: Performance evaluation of mobile ad hoc network routing protocols under black hole attack," *International Journal of Computer Applications*, vol. 42, no. 18, pp. 1–6, 2012, Full text available.

[9] O. Sbai and M. Elboukhari, "Simulation of manet's single and multiple blackhole attack with ns-3," in *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)*, 2018, pp. 612–617. DOI: 10.1109/CIST.2018.8596606.

[10] g. Pankajini Panda Khitish Ku. and P. Niranjan, "Manet attacks and their countermeasures: A survey," *International Journal of Computer Science and Mobile Computing*, vol. 2, pp. 319–330, Nov. 2013.

[11] M. Ichaba, "Security threats and solutions in mobile ad hoc networks; a review," *Universal Journal of Communications and Network*, vol. 6, pp. 7–17, Jan. 2019. DOI: 10.13189/ujcn.2018.060201.

[12] C. Jamadagni, "[ns-3] blackhole attack simulation in ns-3." [Online]. Available: https://mohittahiliani.blogspot.com/2014/12/ns-3-blackhole-attack-simulation-in-ns-3.html.