MAC LAYER MISBEHAVIOR DETECTION AND REACTION IN WIRELESS NETWORKS

A Thesis by

Anbarasan Shenbagaraj

Bachelor of Engineering, Madras Institute of Technology, 2006

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

July 2012

©Copyright 2012 by Anbarasan Shenbagaraj

All Rights Reserved

MAC LAYER MISBEHAVIOR DETECTION AND REACTION IN WIRELESS NETWORKS

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science with a major in Electrical Engineering.

____

Vinod Namboodiri, Committee Chair

____

Neeraj Jaggi, Committee Member

____

Krishna Krishnan, Committee Member

DEDICATION

To my family and friends

ACKNOWLEDGEMENTS

I would like to thank Dr. Neeraj Jaggi for his valuable guidance throughout my thesis. He was very supportive and inspiring and I learned a lot while working with him. I am very thankful to my parents and my brother for their continuous support. I also thank Dr. Vinod Namboodiri and Dr. Krishna Krishnan for their valuable comments.

ABSTRACT

IEEE 802.11 wireless LAN medium access control (MAC) provides transmission fairness to nodes in a network. A misbehaving node can modify its MAC and operate in a selfish manner to improve its own performance (throughput) at the expense of other nodes' performance. In this thesis, new misbehavior detection and reaction mechanisms are proposed and evaluated, and their effect on throughput and fairness is studied. This thesis proposes a new criteria for misbehavior detection: the inter-packet transmission (IPT) time. Nodes maintain a moving average of their own IPT time and also maintain the moving average of neighboring nodes' IPT times. The ratio of a node's own IPT time to its neighbors' IPT time is calculated, and if this ratio exceeds a predetermined threshold, then the presence of misbehavior is detected. When a misbehavior is detected in the network, genuine nodes react by collectively misbehaving, based on the extent of misbehavior in the network, in order to decrease the performance of the misbehaving nodes and also to improve their own performance. It has been shown using extensive simulations that the newly proposed metric of inter-packet transmission time is very effective in designing misbehavior detection and reaction schemes. Reaction effectiveness greater than 85% is achieved in all scenarios considered.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION 1

1.1 IEEE 802.11 Wireless Networks 1
1.2 IEEE 802.11 Distributed Coordination Function 2
1.3 MAC Layer Misbehavior 3
1.4 Contribution of This Thesis 4
1.5 Thesis Organization 5

2 RELATED WORK 6

2.1 Misbehavior Detection 6
2.2 Reaction Mechanism 6
2.3 Motivation 7

3 DETECTION SCHEME 9

3.1 Introduction 9
3.2 Estimating Inter-Packet Transmission Time 9
3.2.1 Node's Own IPT Time 9
3.2.2 Neighbor's IPT Time Calculation 11
3.3 Simulation Details 12
3.4 Proposed Detection Scheme 13
3.5 Summary 16

4 REACTION SCHEME 17

4.1 Introduction 17
4.2 Desired Properties of Reaction Scheme 17
4.3 Design of Reaction Scheme 18
4.4 Summary 22

5 SIMULATION RESULTS 23

5.1 Introduction 23
5.2 NS-2 Simulator 23
5.3 Performance Metrics 23
5.4 Simulation Details 24
5.5 Results 24
5.6 Discussion 29
5.7 Summary 29

Chapter Page

6 CONCLUSIONS AND FUTURE WORK 30

6.1 Summary and Contributions 30
6.2 Conclusions 30
6.3 Future Work 31

REFERENCES 32

APPENDIX 34

A. NS-2 C++ Implementation 35
B. NS-2 TCL Implementation 45
C. Helper Scripts 51

LIST OF TABLES

Table | Page

3.1 Node Parameters 13

3.2 Average of Worst Case IPT Time Ratios 14

4.1 Optimal CW 19

4.2 Desired CW_{fix} 20

LIST OF FIGURES

Figure Page

1.1 IEEE 802.11 basic access method 2

3.1 Node's Own IPT time calculation 10

3.2 Neighbor's IPT time calculation 11

3.3 Worst case ratio of node's own IPT time to neighbor's IPT time (no misbehavior) . . . 14

5.1 Throughput with N=5 nodes 25

5.2 Throughput with N=10 nodes 26

5.3 Throughput with N=15 nodes 26

5.4 Throughput with N=20 nodes 27

5.5 Fairness among genuine nodes 27

5.6 Reaction effectiveness 28

5.7 Overall network throughput 28

LIST OF ABBREVIATIONS

ACK	Acknowledgement
ADC	Anomaly Detection Component
BEB	Binary Exponential Backoff
CA	Collision Avoidance
CSMA	Carrier Sense Multiple Access
CTS	Clear to Send
CW	Contention Window
CW_{fix}	Fixed Contention Window
DCF	Distributed Coordination Function
DEC	Deviation Estimation Component
DIFS	Distributed Inter-Frame Space
DMC	Decision Making Component
IPT	Inter-Packet Transmission
MAC	Medium Access Control
NAV	Network Allocation Vector
NS-2	Network Simulator-2
PCF	Point Coordination Function
PHY	Physical Layer
RTS	Request to Send
SIFS	Short Inter-Frame Space

LIST OF ABBREVIATIONS (continued)

SN	Suspect Node
SPRT	Sequential Probability Ratio Test
TCP	Transmission Control Protocol
UN	Untrusted Nodes
UDP	User Datagram Protocol

LIST OF SYMBOLS

α Misbehavior Parameter

γ Misbehavior Strength

CHAPTER 1

INTRODUCTION

The IEEE 802.11 standard for wireless networks provides fair access to the medium for all users in the network. But nodes can deviate from the standard by modifying the medium access control (MAC) layer parameters. Such nodes get increased access to the medium, resulting in increased throughput while reducing the throughput of other nodes in the network. This behavior is referred to as MAC misbehavior. In this thesis, schemes are proposed to detect MAC layer misbehavior in the network and to react to reduce the throughput of misbehaving nodes.

1.1 IEEE 802.11 Wireless Networks

Wireless local area networks (WLANs) are computer networks formed by connecting devices, usually computers, using a wireless medium. These are infrastructure-based networks, where the wireless devices are connected to an access point that is connected to a wired local area network. WLANs are seen in the work place, educational institutions, home, and public hot spots. Wireless ad hoc networks are networks formed on demand by connecting devices or computers using a wireless medium without connecting to an infrastructure. The wireless ad hoc network mode is used in military battle fields, in natural disaster responses and in home gaming systems.

Both WLAN and ad hoc networks follow the IEEE 802.11 standard [1] for wireless networks. IEEE 802.11 compliant wireless devices operate in either of the following two modes: distributed coordination function (DCF) or point coordination function (PCF). These modes dictate the procedure to be followed by nodes in the network in order to access the shared medium. The IEEE 802.11 PCF uses a master node that polls devices in the network to send and receive data. The IEEE 802.11 DCF is the fundamental access method of IEEE 802.11 MAC and uses carrier sense multiple access with collision avoidance (CSMA/CA) and random backoff time.

1.2 IEEE 802.11 Distributed Coordination Function

The basic access scheme, shown in Figure 1.1, and binary exponential backoff (BEB) algorithm provide fair access to wireless medium. The binary exponential backoff (BEB) algorithm used by the basic access scheme is a distributed medium-access procedure. This component of the IEEE 802.11 standard ensures randomness, which in turn guarantees fair throughput, among all nodes accessing the medium.

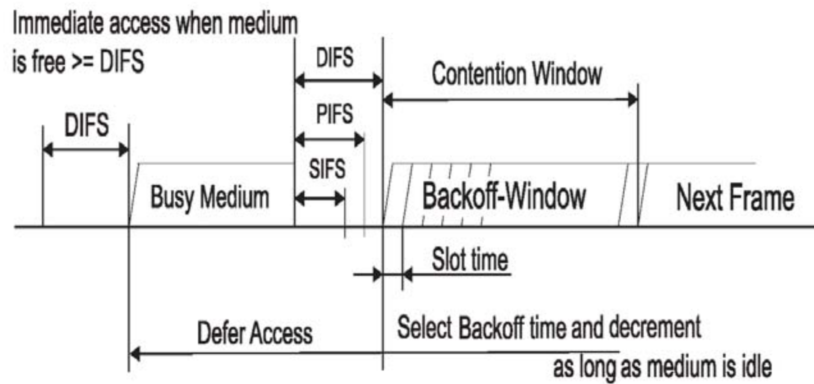


Figure 1.1: IEEE 802.11 basic access method

IEEE 802.11 basic access scheme operates in the following way: A node that wants to transmit packets listens to the medium for any transmission. If the medium is idle for a distributed inter-frame space (DIFS) duration, then the node transmits the packet. If the medium is busy, the node uniformly chooses a backoff value in the range (0, CW-1), where CW is the contention window, and waits for a backoff value times slot time (20 μs) by counting down to zero. The contention window (CW) is initialized to CW_{min}, which is dependent on the physical (PHY) layer and its default value is 31. Backoff counting occurs only when the medium is idle, i.e., when there are no transmissions in the network. If the medium is busy, the backoff timer is frozen and resumes counting down when the medium becomes idle. If a collision occurs during transmission, the node doubles its contention window and uniformly chooses a value from the new contention window. The process of counting down to zero and doubling the contention window repeats until the packet is transmitted. The maximum value of the contention window is CW_{max}, and its default value is

1023.

In addition to the basic access method, an optional request to send - clear to send (RTS-CTS) mechanism is also defined by the standard. In basic access scheme, when large size frames are transmitted, the probability of collision increases. The RTS-CTS scheme is aimed at reducing the collision of large frames. In RTS-CTS scheme, the sending node first transmits a small RTS frame addressed to the receiving node. The receiving node detects the RTS frame and responds with a CTS frame after short inter-frame space (SIFS) duration. The sending node, after receiving the CTS frame, transmits the large data frame. The RTS and CTS frames exchanged by the sending and receiving nodes contain information about the length of the packet to be transmitted. The RTS and CTS frames are received by the neighboring nodes as well and based on the length of the packet to be transmitted, the neighboring nodes update their network allocation vector (NAV). The NAV value indicates the period of time during which the medium will be busy and the nodes refrain from transmitting during this time and freeze their backoff counters. RTS-CTS mechanism is considered in this thesis.

1.3 MAC Layer Misbehavior

The IEEE 802.11 BEB algorithm provides nodes fair access to the medium. Greedy nodes deviate from the standard BEB to gain more frequent access to the medium, thereby gaining more throughput from the network. This greedy behavior is called a MAC misbehavior, and these nodes are referred to as **misbehaving nodes**. Nodes that follow the standard BEB are referred to as **genuine nodes**. Deviating from the standard BEB is not the only way to misbehave. Nodes can modify MAC parameters such as short inter-frame space (SIFS) and DIFS, and nodes can choose values smaller than those specified in the standard. Giri and Jaggi [2] classify different types of misbehavior and study their effectiveness. Two types of misbehaviors, which they define in [2], are used in this thesis: α-misbehavior and CW_{fix}-misbehavior.

- α-misbehavior : Instead of choosing the backoff b uniformly at random from the interval $[0 \ldots CW-1]$, the selfish node chooses b uniformly at random from the interval $[0 \ldots \alpha(CW-$

$1)]$, where $0 < \alpha < 1$. Thus, the node ends up choosing a smaller backoff interval than it is supposed to and increases its chances of accessing the channel next.

- Fixed contention window (CW_{fix})-misbehavior: The selfish node sets its contention window to a small, fixed size CW_{fix} and always chooses its backoff interval uniformly at random from the interval $[0 \ldots CW_{fix}]$.

According to Giri and Jaggi [2], misbehavior effectiveness is defined as the percentage of improvement in throughput when a node misbehaves. This definition of effectiveness is followed in this thesis, where misbehaving nodes are considered to misbehave according to α-misbehavior with α = 0.1.

1.4 Contribution of This Thesis

In this thesis, a new detection scheme using the inter-packet transmission (IPT) time and a collective reaction strategy are proposed. The contributions of this thesis can be summarized as follows:

- A method to calculate the inter-packet transmission time of a node and its neighbors is presented.
- A detection threshold is calculated for a varying number of nodes based on the ratio of a node's own IPT time and its neighbor's IPT time.
- A detection rule based on the inter-packet transmission time and the detection threshold is proposed.
- A collective reaction strategy based on the level of misbehavior in the network is proposed.
- Results from the Network Simulator-2 (NS-2) simulations are used to evaluate the proposed detection and reaction schemes. Fairness among the reacting genuine nodes is also studied.

- Simulations results indicate that the proposed detection and reaction schemes are effective and perform well even with varying number of misbehaving and genuine nodes in the network.

- By detecting the misbehavior and reacting dynamically, the genuine nodes are able to improve their own throughputs, and also succeed in inflicted throughput losses on misbehaving nodes.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents a literary overview of the work done in this area of research. Chapter 3 discusses the proposed detection mechanism. Chapter 4 talks about the proposed misbehavior reaction strategy and derives an equation used by the reacting genuine nodes. Chapter 5 presents the NS-2 results of the detection and reaction mechanisms discussed. Finally, Chapter 6 summarizes the thesis, and discusses conclusions and future directions.

CHAPTER 2

RELATED WORK

Much research in wireless ad hoc networks has focused on MAC-layer misbehavior. The literature related to MAC misbehavior is presented here.

2.1 Misbehavior Detection

Radosavac et al. [3] observe neighboring nodes' backoff intervals and apply the sequential probability ratio test (SPRT) on the backoff values to detect misbehavior. Rong et al. [4] also use the SPRT on packet inter-arrival times and throughput degradation to detect misbehavior.

Raya et al. in [5] propose a mechanism, named DOMINO, which is deployed in access points, to detect misbehavior. This mechanism uses a modular architecture that consists of individual tests and a decision-making component (DMC). Tests are run on parameters such as shorter DIFS, oversized network allocation vector (NAV), backoff, and scrambled frames. Each test consists of a deviation estimation component (DEC) and anomaly detection component (ADC). The test results are given as input to the decision-making component where results are aggregated using functions and weighted results are generated. This result is used in the decision of misbehavior detection.

2.2 Reaction Mechanism

Kyasanur and Vaidya [6] present modifications to the IEEE 802.11 protocol to detect misbehavior and to penalize selfish misbehavior. They introduce the concept of receiver-assigned backoff to detect misbehavior. The receiver sends a new backoff value to the sender in its clear to send (CTS) or acknowledgement (ACK) frame which the sender uses in its next transmission. By listening to the frames of the sender and based on its backoff values, the receiver classifies a sending node as misbehaving or genuine. The penalty scheme developed by the authors penalizes deviating hosts by assigning larger backoff values to them than those assigned to well-behaved hosts.

Guang et al. [7] propose a detection and reaction scheme for a new class of malicious misbehaviors. This proposed scheme, named DREAM (a system for detection and reaction to a timeout MAC-layer misbehavior), tackles the problem of transmission timeouts of MAC frames. The DREAM system uses two stages to detect and react to misbehavior. In the first stage, a bad credit value of a node is used in the detection. If the credit crosses a threshold, then the corresponding node is declared as a suspect node (SN). In the second stage, the SN is assigned a trust level, which is increased or decreased based on misbehavior. If the SN node misbehaves, the trust level is reduced and if the trust level falls below a threshold, the node is declared as untrusted node (UN). The trust level is propagated to the upper layers and prevents traffic via the UNs.

Cardenas et al. [8] compare the two popular detection algorithms SPRT [3] and DOMINO [5], both theoretically and by using simulations. Guang et al. [9] propose modifications to the BEB algorithm in order to facilitate easy detection and penalization of a misbehaving sender. These authors propose a predictable random backoff algorithm to detect misbehavior in the network.

Jaggi et al. [10] use throughput degradation to detect misbehavior. They propose two reaction methods that follow a collective misbehavior strategy. In the non-adaptive reaction scheme, all genuine nodes, upon detection of misbehavior, use a constant contention window size that is not varied. In the adaptive reaction scheme, the contention window size of the genuine nodes are dynamically computed based on the throughput degradation experienced by the individual nodes. Both strategies are distributed in nature and rely upon local information available to the genuine nodes.

2.3 Motivation

Most of the MAC-layer misbehavior research consider access point based wireless network where misbehavior detection and reaction happen in the access points. This thesis aims to propose a detection and reaction scheme that can applied to ad hoc networks as well. The detection schemes proposed in other misbehavior research consider some known type of misbehavior. This thesis, does not consider specific type of misbehavior but aims to address any type of misbehavior. An

important goal of this thesis is to propose a new parameter or criteria for misbehavior detection. Hence a new method using inter-packet transmission (IPT) time for detection of misbehavior is proposed in this thesis. The collective reaction strategy proposed by Jaggi et al [10] evaluate their scheme only for fixed number of genuine and misbehaving nodes. A good detection and reaction scheme should be able to scale with number of nodes in the network. The detection and reaction schemes proposed in this thesis are designed to work with varying number of nodes in the network.

CHAPTER 3

DETECTION SCHEME

3.1 Introduction

This chapter discusses the proposed misbehavior detection scheme, which is based on estimating the average inter-packet transmission times of neighboring nodes and comparing them with the node's own average IPT time. This scheme has many advantages over existing schemes, which are discussed in detail here.

3.2 Estimating Inter-Packet Transmission Time

The IPT time is defined as the time between two successful transmissions of data (DATA) packets (under the basic access mechanism) or request to send (RTS) packets (under the RTS-CTS mechanism). The following sections describe a node's own IPT time calculation and its neighbor's IPT time calculation.

3.2.1 Node's Own IPT Time

As mentioned earlier, the IPT time is the time between transmissions of DATA packets or RTS packets. The individual or instantaneous IPT time values observed by a node are varying and random due to the binary exponential backoff algorithm. Therefore, individual IPT time values do not help in detecting misbehavior. Hence, a simple moving average of the IPT time is maintained. A simple running average is not chosen to calculate the IPT time but rather a simple moving average is used and it assists in discarding older IPT time values and considering newer IPT time values. Thus, using simple moving average to calculate IPT time reflects the current state of nodes in the network (genuine/misbehaving).

For a node's own IPT time calculation, the time between CTS packets is used. Even though this differs from the definition of IPT time, an IPT time calculation using CTS packets yields the

correct result. This will be explained later in this section.

Calculation of IPT time is as follows: When a CTS packet is successfully received, the current time is noted. When the next CTS packet is received, the previously noted time is subtracted from the current time. The subtracted value is the recent IPT time and is added with the running sum of previously calculated IPT time values. When the new IPT time is added to the running sum, the oldest IPT time is subtracted from the running sum (according to definition of simple moving average). The number of IPT time values added is 250, and the sum is divided by 250 to obtain the node's own average IPT time. The reason for choosing 250 as simple moving average period is explained later in this section. This simple moving average IPT time is used in misbehavior detection. The node's own IPT time calculation is shown in Figure 3.1.

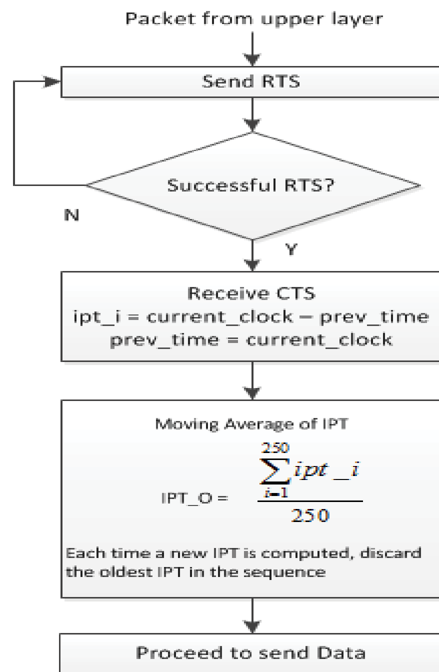


Figure 3.1: Node's Own IPT time calculation

According to the definition, the IPT time is the time between two RTS packets. In NS-2, calculating the time between two successful RTS packets is difficult since it involves adjusting for the retransmission of RTS packets. Hence, the time between the two CTS packets, instead of the RTS packets, is used in the IPT time calculation. In spite of the difference between the definition of IPT time and the implementation of IPT time, the value calculated by the using the CTS packets

results in an accurate calculation of the IPT time. This is verified using NS-2 simulation, whereby packets are sent at a slow rate (1 packet per second), and the node's own IPT time is verified to be close to 1 (0.999) which is inline with the transmission rate (1 packet per second).

3.2.2 Neighbor's IPT Time Calculation

Like the node's own IPT time, each node maintains a list of neighboring nodes, and for each of the nodes, a simple moving average of the IPT time is maintained. This calculation is shown in Figure 3.2.

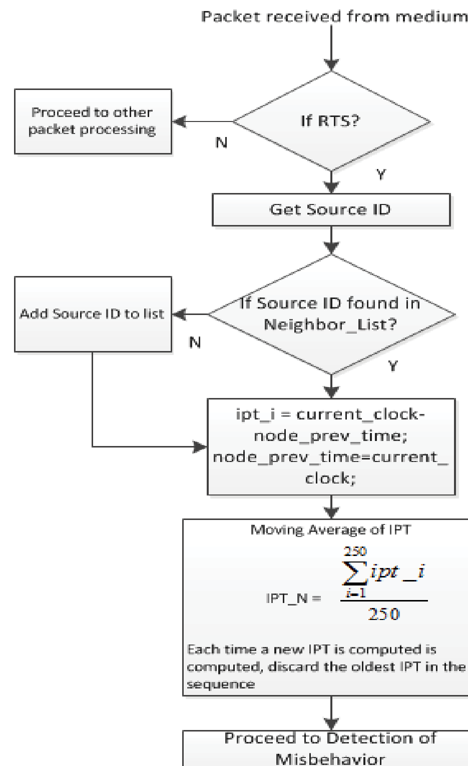


Figure 3.2: Neighbor's IPT time calculation

The neighbor's IPT time is estimated by listening to the medium and calculating the time between RTS packets sent by the neighbor. A linked-list data structure is used in the neighbor's IPT time calculation and maintainence. The linked-list is indexed using node IDs. Each node in the network maintains the linked-list, and each node in the linked-list contains the neighbor's node ID and the moving average of IPT corresponding to the node. The neighbor's IPT time for a node

is calculated based on the time between reception of two RTS packets from the same node. Similar to the node's own IPT time calculation, simple moving average of the IPT time values for each of the neighboring nodes is maintained.

Both IPT times (a node's own IPT time and a neighbor's IPT time) are simple moving averages of individual or instantenous IPT times. The moving average is calculated over a period of 250 packets. The value of 250 is chosen for the moving average period after several trials with larger and smaller values. If the moving average period value is large, say 1000, then the node takes more time to detect misbehavior, and if the period is very small, then the node may falsely conclude misbehavior even for small variations in the network. Therefore, a value of 250 is chosen.

Note that in a node's own IPT time calculation, we used the time difference between two successfully received CTS frames. But for neighbor's IPT time calculation, successful reception of RTS frames is only considered. This may lead to the conclusion that there could be a difference in number of IPT time data counts among the sending and receiving nodes due to collision of CTS frames. But this scenario does not cause any problem to the IPT time calculation. In the scenario considered for this thesis, once a successful reception of RTS is confirmed, it is guaranteed that the corresponding CTS frame will be received by the sending node since no new node in the network may suddenly appear that may cause collision of CTS frames. Moreover, in the simulation, nodes are stationary and this will not cause any loss of CTS or any other packet due to change in physical characteristics of the medium. Hence using CTS frames for own IPT time calculation and RTS frames for neighbor IPT time calculation does not cause any difference in IPT time count.

3.3 Simulation Details

NS-2 simulation is used in finding the detection threshold discussed in the following Section 3.4. NS-2 version 2.34 is used in the evaluation of the detection and reaction schemes. The topology consists of varying the number of sending nodes and using one receiving node. The number of sending nodes is varied with the values of 4, 9, 14 and 19. The topology is set in an area of 100x100m. Sending nodes transmit 512 bytes of user datagram protocol (UDP) packets at a

constant bit rate of 100 packets per second. The simulation is run for 6,000 seconds. When a node is configured as a misbehaving node, it follows α-misbehavior with $\alpha = 0.1$. The misbehaving node(s) misbehave(s) for the entire duration of the simulation. The node parameters are captured in Table 3.1.

Table 3.1

Node Parameters

MAC	802.11b
Data Rate	2 Mbps
RTS Threshold	128 bytes
CW_{Min}	31
CW_{Max}	1023
Slot Time	20 μs
SIFS	10 μs

3.4 Proposed Detection Scheme

In the detection scheme proposed, the presence of misbehavior in an ad hoc network is detected using IPT time. As mentioned earlier, each node maintains its own IPT time (IPT_O) and a list of neighbors' IPT times ($(IPT_N)^i$). In addition to neighbors' IPT times, a ratio of a node's own IPT time and each of its neighbor's IPT time, denoted R_i is also maintained, as shown in equation (3.1).

$$R_i = \frac{IPT_O}{(IPT_N)^i} \tag{3.1}$$

Equation (3.1) is used in the detection of misbehavior. To define a misbehavior detection rule, a parameter called detection threshold (D_{th}) is introduced. The misbehavior detection threshold of a node is the maximum value of the ratio in equation (3.1) for a given number of nodes in the network. To find the threshold value, NS-2 simulations are performed when no misbehavior is

present in the network. During the simulation, with four sending nodes (nodes 1, 2, 3 and 4) and a receiving node (node 5), the highest value of the ratio of IPT times for node number 1 is noted. The simulation is repeated with 9, 14 and 19 sending nodes. This is captured in Figure 3.3 which represents the worst case $\frac{IPT_O}{(IPT_N)^i}$ when no misbehavior is present in network.

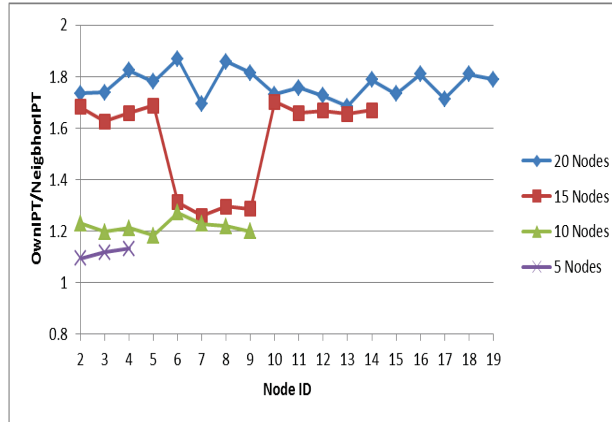


Figure 3.3: Worst case ratio of node's own IPT time to neighbor's IPT time (no misbehavior)

If this ratio exceeds the detection threshold, then the node concludes presence of misbehavior in the network. The average values of the worst case IPT time ratio for different network sizes are captured in Table 3.2. These IPT time ratios are the detection threshold (${D_{th}}^N$) for N nodes in the network.

Table 3.2

Average of Worst Case IPT Time Ratios

Nodes	IPT Time Ratio
5	1.15
10	1.25
15	1.55
20	1.75

An example of a detection threshold for ten nodes is discussed here. Based on the value in

Table 3.2, the detection threshold (${D_{th}}^{10}$) of 1.25 is chosen for ten nodes, i.e., if $\frac{IPT_O}{(IPT_N)^i}$ exceeds 1.25, then the corresponding neighbor is declared as misbehaving. Similar approach can be used to set appropriate thresholds for N = 5,15,20 nodes as well.

The inverse of $\frac{IPT_O}{(IPT_N)^i}$ is the misbehavior strength of node i (γ_i), and it lies between 0 and 1. More precisely, $0 < \frac{1}{\frac{IPT_O}{(IPT_N)^i}} \leq 1$. The misbehavior strength (γ) is used during the reaction mechanism. The reaction mechanism is discussed in detail in the Chapter 4. In the case of ten nodes, if the largest of $\frac{IPT_O}{(IPT_N)^i}$ is found to be 3.5, then the misbehavior strength is 0.28. This detection scheme is presented in Algorithm 1.

Input: $IPT_O\ and\ (IPT_N)^i$
m=0
i=1
while $i <= N$ **do**
  if $\frac{IPT_O}{(IPT_N)^i} > {D_{th}}^N$ **then**
    $node_i\ is\ misbehaving$
    if $m < \frac{IPT_O}{(IPT_N)^i}\ ratio$ **then**
      $m = \frac{IPT_O}{(IPT_N)^i}$
    end if
  end if
  $i++$
end while
if $m > 1$ **then**
  $m = \frac{1}{\frac{IPT_O}{(IPT_N)^i}}$
  return m
else
  return 1
end if

Algorithm 1: Detection Algorithm

The detection algorithm is run by each of the genuine nodes in the network. Since misbehavior detection is based on the inter-packet transmission time, the detection algorithm is invoked whenever neighbor IPT time is updated, i.e., when a packet is received from a neighbor. The detection algorithm has access to both the node's own IPT time and the node's linked list, which maintains the neighbor's IPT time.

The detection algorithm traverses through the linked list and calculates the ratio of its own IPT

time and each neighbor's IPT time. If the ratio exceeds the detection threshold, then the corresponding node is marked as misbehaving, and the variable m indicating misbehavior is updated with the $\frac{IPT_O}{(IPT_N)^i}$. If more than one neighboring node is misbehaving, then the greatest $\frac{IPT_O}{(IPT_N)^i}$ is considered, and m is updated with that value. The inverse of $\frac{IPT_O}{(IPT_N)^i}$ is returned by the detection algorithm as the misbehavior strength (γ) present in the network. The detection scheme returns 1, if no misbehavior is detected. The detection scheme is used only by genuine nodes; misbehaving nodes do not use any detection scheme.

The detection scheme proposed uses a new parameter namely the IPT time. This method has not been proposed by any literature in this field of research. The results shown in Chapter 5 indicate that the proposed scheme is successful in detecting misbehaviors. The detection scheme proposed, unlike [6], does not require any protocol modifications. The information collected by the genuine nodes is collected locally and the detection scheme is easily scalable. An important characteristic of this detection scheme is that it is dynamic in nature and can detect if the nodes have stopped misbehaving and returned to normal BEB algorithm.

3.5 Summary

This chapter explains the estimation of a node's own IPT time and also its neighbor's IPT times. It also discusses the choice of detection threshold and how it varies with the number of nodes. A detection scheme based on the ratio of a node's own IPT time and neighbor's IPT time is presented.

CHAPTER 4

REACTION SCHEME

4.1 Introduction

The reaction scheme is the strategy or methodology used by the genuine nodes to react against misbehavior in order to improve their own throughputs and to penalize the misbehaving node(s). This chapter describes the design of the proposed reaction strategy used by the genuine nodes. CW_{fix}-misbehavior, defined in Section 1.3, is used by the genuine nodes for a reaction scheme. An equation to compute an appropriate value to use for CW_{fix} is also presented in this chapter.

4.2 Desired Properties of Reaction Scheme

The reaction scheme employed by the genuine nodes should be simple and distributed, and should result in minimal or no protocol change. Jaggi et al. [10] used a collective misbehavior reaction strategy. This thesis also uses, a collective misbehavior reaction strategy. In a collective reaction strategy, when the genuine nodes detect misbehavior in the network, they also start misbehaving to improve their throughput. All genuine nodes follow the same reaction scheme. The collective reaction strategy used here is distributed and requires minimal protocol modification.

The reaction scheme proposed uses a fixed contention window (CW_{fix}) type of misbehavior. Genuine nodes choose a random backoff from this fixed contention window and do not double the contention window upon collision, as in normal operation of the IEEE 802.11 BEB algorithm. The value of the CW_{fix} used by the genuine nodes should be smaller (nodes back off for a smaller duration) in order to provide an advantage to the genuine nodes, i.e., more frequent access to the medium. If the CW_{fix} is too small, then genuine nodes will spend most of the time in collision and less time in successful transmission of packets. Thus, a tradeoff is involved in the choice of the CW_{fix} value. Also if the misbehavior is mild, a larger CW_{fix} may suffice but if the misbehavior is severe, a smaller CW_{fix} might be needed. This tradeoff and a proposed scheme to compute an effective value for CW_{fix} are discussed here.

4.3 Design of Reaction Scheme

Three parameters are used in the reaction scheme:

- Reaction Type - The type of misbehavior used by the genuine nodes to react against misbehavior.
- Reaction Factor - The strength of misbehavior used by the genuine nodes. In the case of the CW_{fix}-misbehavior type, the reaction factor is CW_{fix}. If α-misbehavior is used for the reaction, then the reaction factor is α.
- Reaction Count - This count is used to compensate for the time spent until misbehavior is detected. Note that genuine nodes do not react until misbehavior is detected. This count is incremented when misbehavior is detected and decremented when no misbehavior is detected. If reaction count is greater than zero, reaction is applied. If the count is zero and no misbehavior is detected, then the genuine nodes do not react.

Misbehavior is detected as described in Chapter 3. The detection scheme returns the presence of misbehavior in the network. If misbehavior is present, then the strength of misbehavior (γ) is returned. This is used in calculating the reaction factor. As discussed in the previous section, the reaction scheme used in the experiments conducted in this thesis is CW_{fix} misbehavior. Hence, the reaction factor mentioned earlier is CW_{fix}. Note that other reaction schemes are equally suitable and could also be used. The CW_{fix} used by reacting genuine nodes should address the tradeoff, discussed earlier in Section 4.2, about achieving the maximum gain while minimizing collisions in the network. If CW_{fix} is smaller for a large number of nodes, then the number of collisions will increase, thus leading to less throughput. Therefore, CW_{fix} should be directly proportional to number of nodes in the network.

Bianchi et al. [11] propose a method for increasing the network saturated throughput by using an optimal contention window instead of the IEEE802.11 BEB algorithm. It derives optimal CW equations for the basic access scheme and the RTS-CTS mechanism. By combining an optimal

CW value and the misbehavior strength, CW_{fix} is derived. The $CW_{optimal}$ equation in the case of the RTS-CTS mechanism presented by Bianchi et al. [11] is

$$CW_{optimal} = n * \sqrt{2K} \tag{4.2}$$

where n is number of nodes contending for the channel, $K = (T_{PH} + T_{RTS} + DIFS + \tau)/slottime$, T_{PH} is the PHY header transmission time, and T_{RTS} is the RTS transmission time, and τ is the propagation delay. Note that K is expressed in slot times. In this simulation, the values used are $T_{PH} + T_{RTS} = 352\mu s$, $DIFS = 50\mu s$, $\tau = 2\mu s$, $slottime = 20\mu s$. Substituting these values, the value of $\sqrt{2K}$ is found to be 6.356099.

Note that according to equation (4.2), n is number of nodes contending for the channel. Hence when number of nodes in the network is five, there are four nodes contending for the channel. Using equation (4.2), the $CW_{optimal}$ values for different number of nodes are presented in Table 4.1.

Table 4.1

Optimal CW

Nodes	Contention Window
5	25
10	57
15	89
20	121

As discussed earlier, the detection scheme returns the strength of the misbehavior present in the network. This value is actually the ratio of the neighbor's IPT time and the node's own IPT time. Various values of CW_{fix} for the reaction is tried to see which value performs the best. The best value found using experimentation for different number of nodes is shown in Table 4.2. The objective of the reaction scheme is to express and compute the desired best CW_{fix} value in terms of

known and computed system parameters such as number of nodes (N), $CW_{optimal}$ and misbehavior strength (γ), as returned by detection algorithm.

Table 4.2

Desired CW_{fix}

Nodes	Best value of CW_{fix} for reaction
5	2
10	4
15	6
20	8

Using the $CW_{optimal}$ and misbehavior strength (γ), and the number of nodes, equation (4.3) is proposed to calculate the CW_{fix} which is used by the reacting genuine nodes. Note that if there are multiple misbehaving nodes, the misbehavior strength corresponds to the most severe misbehaving node. Let N_c denote the neighbor node count.

$$CW_{fix} = CW_{optimal} * (N_c)^2 * (\gamma)^2 * 0.005 \tag{4.3}$$

Note that this value may change for every RTS packet received by the node. According to equation (4.3), CW_{fix} increases as the number of nodes increases as described in Section 4.3. Also CW_{fix} decreases if misbehavior strength increases (i.e., γ decreases).

The reaction scheme used by the genuine nodes is presented in Algorithm 2. This algorithm uses misbehavior strength (γ) as the input. This is returned by the detection rule described in Algorithm 1. The aim of the reaction algorithm is to set the parameters $misbhType$ and $misbhValue$ to zero (if no misbehavior is present) and to an appropriate value (if misbehavior is detected). In Algorithm 2, $misbhType$ is CW_{fix}-misbehavior. When misbehavior is detected ($\gamma < 1$), $misbhValue$ takes the value calculated according to equation (4.3).

Input: γ
if $\gamma < 1$ **then**
  $CW_{fix} = CW_{optimal} * (N_c)^2 * (\gamma)^2 * 0.005$
  $CW_{fix} = (CW_{fix} > 3) \,?\, CW_{fix} : 3$
  {Presence of Misbehavior and reacting}
  $misbhReactCount = misbhReactCount + 2$
  $misbhType = MISBH_REACT_TYPE$
  {$MISBH_REACT_TYPE$ indicates CW_{fix}-misbehavior}
  $misbhValue = CW_{fix}$
else if $misbhReactCount$ **then**
  {No Misbehavior but reacting}
  $misbhReactCount = misbhReactCount - 1$
  $misbhType = MISBH_REACT_TYPE$
  $misbhValue = \frac{CW_{min}}{2}$
else
  {No Misbehavior and not reacting}
  $misbhType = 0$
  $misbhValue = 0$
end if

Algorithm 2: Reaction Scheme

Genuine nodes react, i.e., misbehave only after misbehavior has occurred in the network. This may take some time to get reflected in the simple moving average of the IPT calculation discussed in Chapter 3. To compensate for the misbehavior that has already occurred, genuine nodes continue to react even after detecting that no misbehavior is present in the network. This is accomplished by maintaining a counter $misbhReactCount$. This counter is incremented (by two) when the node has detected misbehavior (and nodes are reacting) and it is decremented (by one) when there is no misbehavior but continue to react until $misbhReactCount$ reaches zero. When $misbhReactCount$ is decremented, a mild reaction is required and hence the CW_{fix} used during this period is $\frac{CW_{min}}{2}$. When no misbehavior is detected and $misbhReactCount$ is zero, then the genuine nodes do not react and follow the IEEE 802.11 BEB algorithm. The simulation results of the detection scheme and the reaction scheme are presented in Chapter 5.

The proposed reaction scheme uses a novel method of controlling reaction strength based on IPT time. The scheme considers factors like number of nodes in the network, the misbehavior strength and adjusts the reaction dynamically. A distinctive feature of this scheme is that it is scalable to any network size and the scheme can be extended to other reaction types (like α misbehavior

instead of CW_{fix} misbehavior.) Scheme is adaptive as in if misbehaving nodes stops misbehaving or returns to normal behavior, the genuine nodes also return to normal BEB behavior. The reaction requires a minimal protocol modification and the decisions made by the genuine nodes are totally based on the local information available to the nodes.

4.4 Summary

This chapter presents the proposed reaction scheme used by the genuine nodes. The genuine nodes use CW_{fix}-misbehavior. Based on the strength of the misbehavior present and the number of nodes in the network, the reaction scheme calculates the appropriate CW_{fix} values to be used by the genuine nodes. The reaction scheme dynamically adjusts according to the strength of misbehavior present in the network.

CHAPTER 5

SIMULATION RESULTS

5.1 Introduction

This chapter presents the simulation results of detection and reaction mechanisms performed using the Network Simulator-2. Network topology, simulation parameter details, and metrics used in the experiments conducted are presented here. Results are captured by varying number of nodes, reaction or no reaction and number of misbehaving nodes.

5.2 NS-2 Simulator

NS-2 is used for the simulation study. This discrete event simulator is written in OTCL and C++. OTCL serves as the front end of the simulator, and actual simulation is implemented in C++. NS-2 is used because it is open source and free to use. Moreover, it supports a variety of protocols, and most of the wireless network research all over the world is simulated in NS-2. Detection and reaction schemes presented in Chapters 3 and 4 are implemented in C++ module of NS-2. C++ code modifications that are required to implement the proposed detection and reaction schemes and TCL topology script are presented in Appendices A and B respectively.

5.3 Performance Metrics

Following are the metrics considered in this thesis:

- Inter-packet transmission time is used in detection of misbehavior, as described in Chapter 3. It is defined as the time between two successfully transmitted data packets by a node. Each node maintains its own IPT time and the IPT time of its neighboring nodes. A node listens to the medium, and based on the difference between the time of consecutive RTS arrivals, the node calculates its neighbor's IPT time. In addition to this, each node maintains a ratio of its own IPT time and each of its neighbor's IPT time. The nodes in NS-2 calculate the IPT time

at the run time. The calculation of inter-packet transmission time is performed dynamically by the nodes and is used in misbehavior detection.

- Throughput is used in the evaluation of the effectiveness of detection and reaction schemes. Once the simulation is complete, using a PERL script, the trace output of the simulation is analyzed and throughput of nodes is computed. Based on the improvement of genuine nodes' throughput, it can be concluded if the detection and reaction mechanisms are effective in dealing with the presence of misbehavior in the network.

- Fairness in throughput among genuine nodes is also calculated. Jain's fairness index [12] is used in the throughput fairness calculation.

5.4 Simulation Details

The simulation details are the same as mentioned in Section 3.3.

5.5 Results

Simulation is run by varying the following:

- Number of nodes in the network (5, 10, 15, 20)

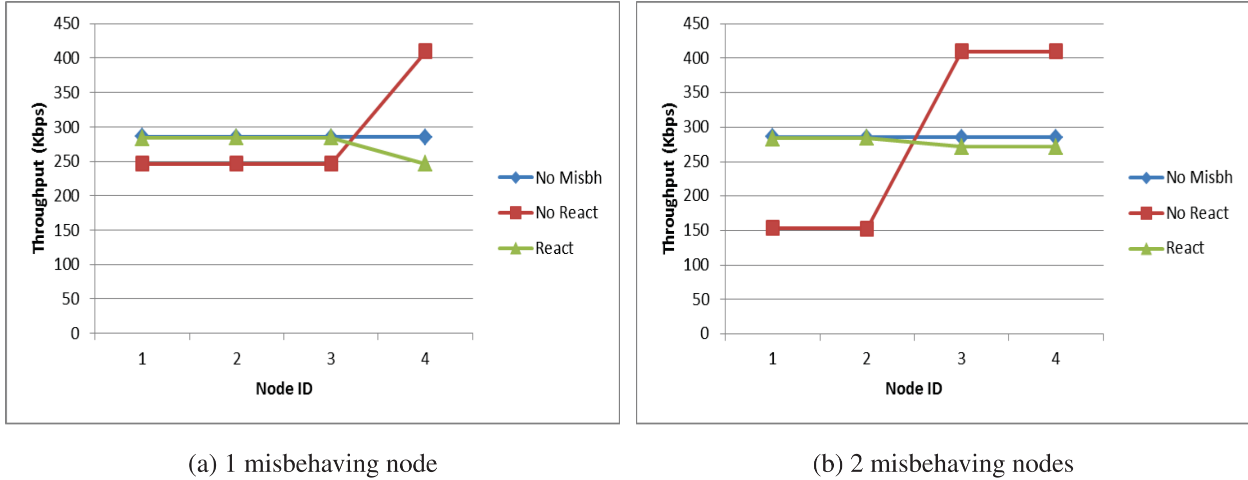
- Number of misbehaving nodes (1, 2)

- Genuine nodes' reaction or no reaction.

The three different throughput information (in absence of misbehavior, in presence of misbehavior and when genuine nodes react) are captured in Figures 5.1 to 5.4. The largest numbered node in each figure is the misbehaving node and the other nodes are the genuine nodes. The throughput values shown are the individual genuine node's throughput at the end of simulation. Figure 5.1a depicts the following:

- Throughput of four nodes in the absence of misbehavior

- Throughput of four nodes when one node is misbehaving
- Throughput of four nodes when genuine nodes react using the reaction scheme discussed in Chapter 4.



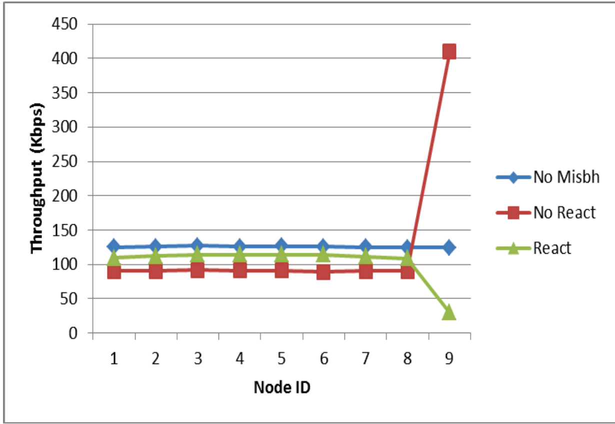
(a) 1 misbehaving node

(b) 2 misbehaving nodes

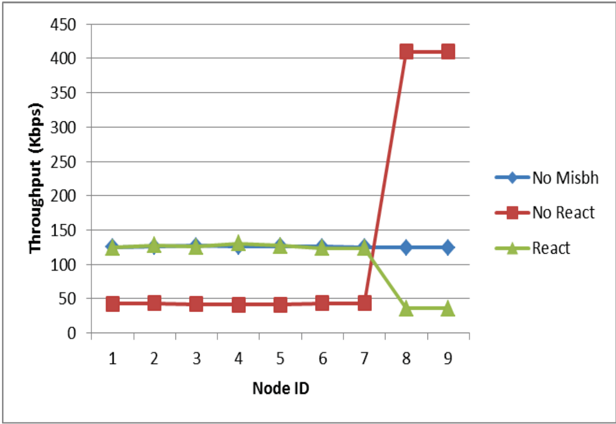
Figure 5.1: Throughput with N=5 nodes

In the absence of misbehavior, the throughput of all nodes is $\approx$ 285 Kbps. It is clear from the Figures 5.1a and 5.1b that when the genuine nodes react in the presence of misbehavior, the throughput of the misbehaving nodes drops significantly. The genuine nodes' throughput, after reaction, is nearly the same ($\approx$ 284 Kbps) as the no-misbehavior scenario. According to the reaction scheme discussed in Section 4.3, the genuine nodes, upon detecting presence of misbehavior in the network, choose a small CW_{fix} value instead of following the IEEE 802.11 BEB algorithm. Since the genuine nodes themselves misbehave, the genuine nodes see an increase in throughput because of their increased access to the medium. The CW_{fix} value chosen by the genuine nodes vary dynamically throughout the simulation in accordance with the strength of misbehavior present in the network. Similar behavior is observed when two nodes are misbehaving, as shown in Figure 5.1b.

Figure 5.2a depicts the throughput of nine nodes in the absence of misbehavior, when one node is misbehaving and the throughput increase of the genuine nodes when the genuine nodes react.



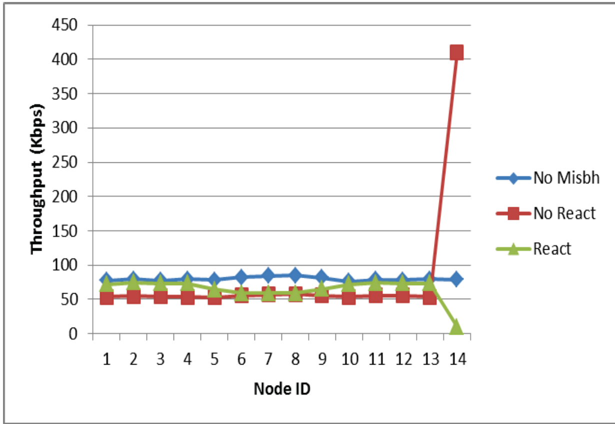
(a) 1 misbehaving node



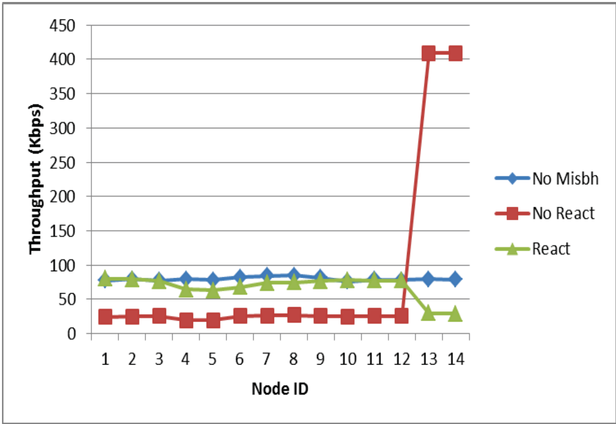
(b) 2 misbehaving nodes

Figure 5.2: Throughput with N=10 nodes

Throughput of the nodes in the absence of misbehavior is $\approx$ 125 Kbps. Figure 5.2a shows when a node misbehaves, throughput of the genuine nodes falls by approximately 28% to 90 Kbps. When the genuine nodes react, throughput of the individual genuine nodes increase by 24% to 112 Kbps. Figure 5.2b depicts the scenario when two nodes are misbehaving. Due to the two misbehaving nodes, the genuine nodes suffer a greater throughput loss (dropping to 43 Kbps) and the misbehavior strength is larger i.e., γ is smaller than in 5.2a.

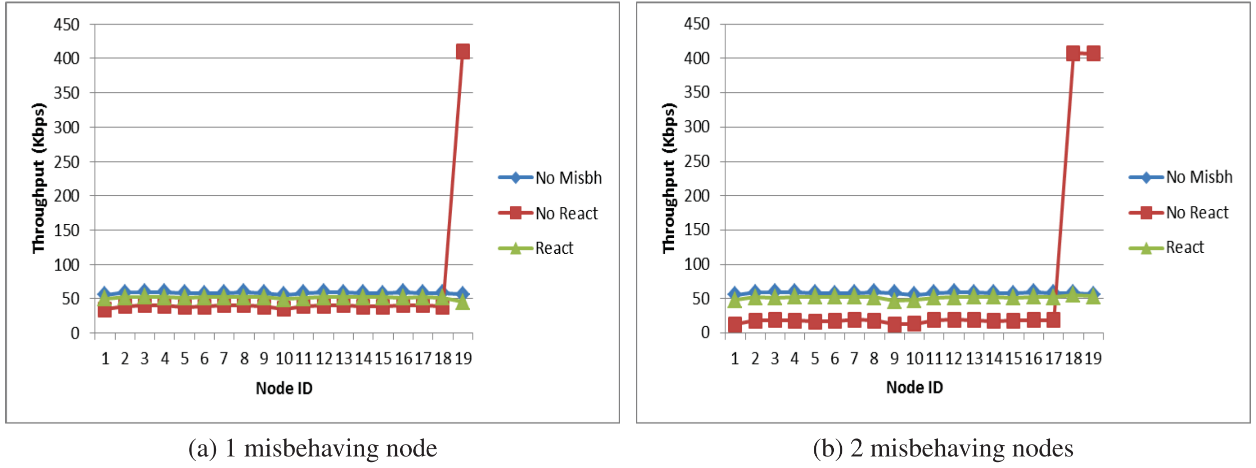


(a) 1 misbehaving node



(b) 2 misbehaving nodes

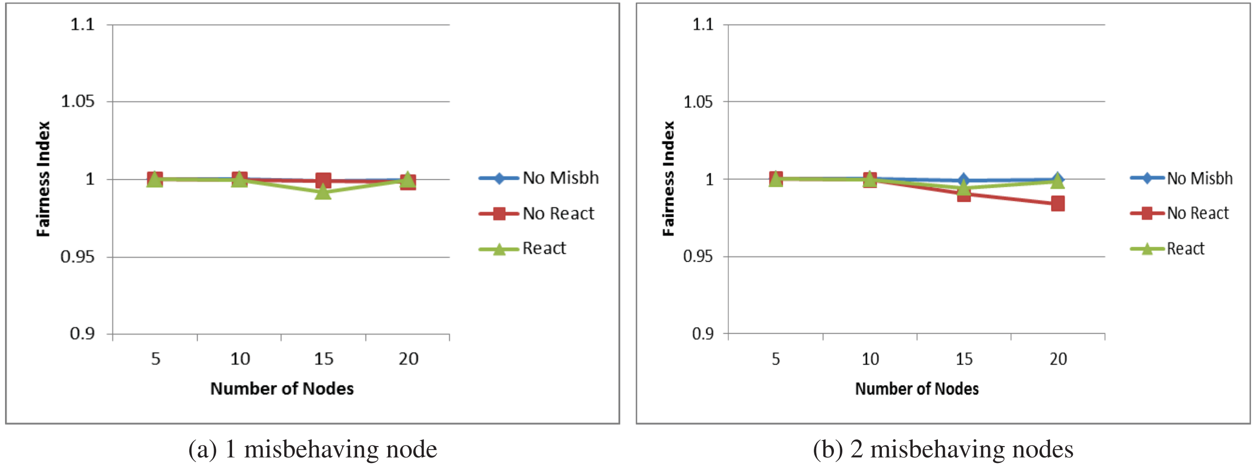
Figure 5.3: Throughput with N=15 nodes



(a) 1 misbehaving node

(b) 2 misbehaving nodes

Figure 5.4: Throughput with N=20 nodes



(a) 1 misbehaving node

(b) 2 misbehaving nodes

Figure 5.5: Fairness among genuine nodes

From the perspective of the genuine nodes, the misbehavior is very strong in the network, and nodes aggressively react, resulting in an increase in throughput to levels greater than 125 Kbps. A similar explanation holds for the 14 and 19 sending nodes, and results are depicted in Figures 5.3a, 5.3b, 5.4a, and 5.4b. In all of these scenarios, the fairness among the throughput achieved by genuine nodes is close to optimal and is around 0.99 in all cases. This is depicted in the Figures 5.5a and 5.5b. To measure the reaction effectiveness of the reaction scheme, a metric called reaction effectiveness (denoted R_{eff}) is defined as

$$R_{eff} = \frac{genuine\ nodes'\ average\ throughput\ with\ reaction}{genuine\ nodes'\ average\ throughput\ with\ no\ misbehavior} * 100 \qquad (5.4)$$

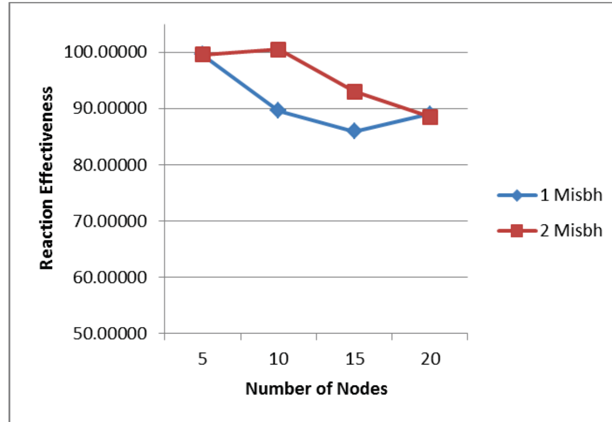
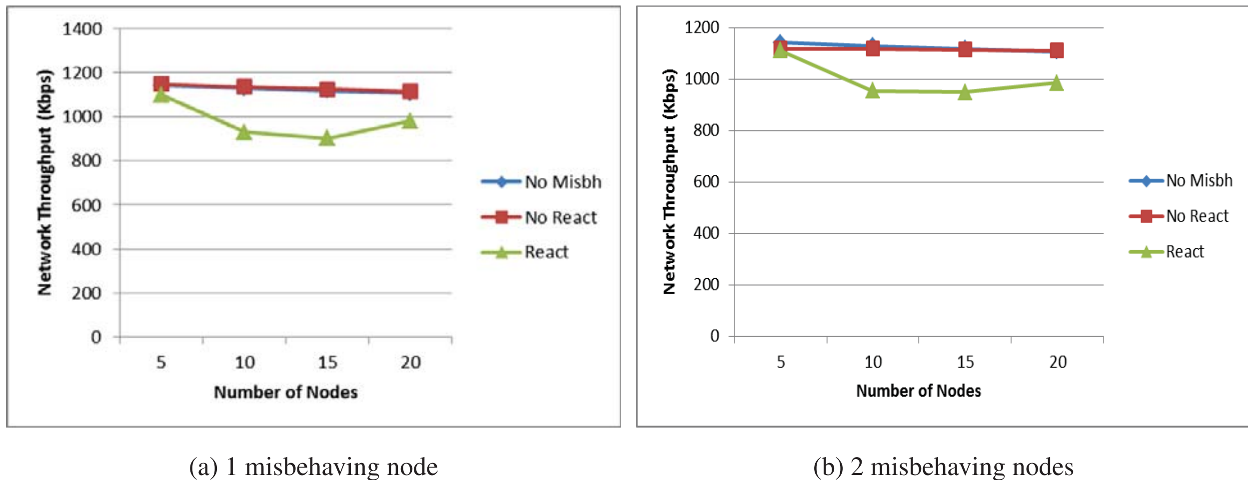


Figure 5.6: Reaction effectiveness

Figure 5.6 depicts the reaction effectiveness. Reaction effectiveness (R_{eff}) values closer to 100% indicate that the genuine nodes, after reaction, were able to achieve as much throughput as they would have achieved if there was no misbehavior. From the results, reaction is more effective when number of misbehaving nodes is larger and when the total number of nodes in the network is smaller. As a result of applying reaction, the overall network throughput decreases slightly. This is expected, since all the genuine nodes are misbehaving under the reaction scheme. However, most of the loss in throughput is shared by misbehaving nodes, since the reaction effectiveness is close to 100% for many scenarios and is greater than 85% in all scenarios evaluated.



(a) 1 misbehaving node

(b) 2 misbehaving nodes

Figure 5.7: Overall network throughput

5.6 Discussion

The collective reaction strategy using the fixed contention window (CW_{fix}) is shown to be effective for MAC-layer misbehavior. The throughput of the genuine nodes almost reaches the original levels and fairness is also maintained among the genuine nodes. As the number of nodes become larger in the network, a drop in reaction effectiveness is observed. In this scenario of increasing number of nodes, the genuine nodes do not suffer greater throughput loss and hence do not react strongly, but reaction is strong enough to achieve throughput close to the level observed when no misbehavior is present. When genuine nodes react, that is collectively misbehave, all the nodes try to access the medium very frequently. This results in increased collision in the network and the overall network throughput drops. This side effect of the collective reaction strategy can be seen in Figures 5.7a and 5.7b.

5.7 Summary

This chapter analyzes the misbehavior reaction with a varying number of nodes and varying number of misbehaving nodes. Reaction effectiveness, throughput of nodes and fairness among the genuine nodes are also discussed. The effect of reaction strategy on the overall network throughput is also discussed.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the work done in this thesis and provides conclusions of the results. Suggestions for future work are also provided.

6.1 Summary and Contributions

In this thesis, a novel method for detecting misbehavior in a wireless ad hoc network is presented. The inter-packet transmission times of a node and its neighbors are used to detect misbehaviors in the network. A collective misbehavior reaction strategy is used to suppress the misbehaving node and allow genuine nodes to regain their own throughput. Important characteristics of both the detection and reaction mechanisms are that they are distributed in nature and rely only on a node's local information. The detection and reaction mechanisms require minimal protocol change and thus are easily implementable in practice. NS-2 simulations are performed to evaluate the proposed detection and reaction mechanisms. Results indicate significant improvement in throughput of the genuine nodes and reduction in the misbehaving nodes' throughput when the proposed reaction scheme is applied. Scalability of the detection and reaction strategies are also studied, and the results indicate that detection and reaction mechanisms are effective with a varying number of nodes in the network.

6.2 Conclusions

Detection of misbehavior based on the ratio of inter-packet transmission is very effective, and the detection threshold is designed to vary with the number of nodes in the network. The proposed collective reaction strategy is effective in restoring the node's original throughput and in penalizing the misbehaving node. The distributed nature of the reaction scheme facilitates the genuine nodes' local decision making capability and eliminating the need for central authority to monitor the network and penalize the misbehaving nodes. During reaction, the fairness among

the throughput achieved by genuine nodes is close to optimal and is around 0.99 in all cases. As expected, due to collective misbehavior strategy, the overall network throughput of the network decreases slightly but the benefit of the scheme is achieved in the penalization of the misbehaving nodes. Due to the collective misbehavior reaction strategy, for the same misbehavior strength, the reaction effectiveness tends to decrease with the increase in number of nodes in the network.

6.3 Future Work

In the future, a mathematical model for the detection threshold could be derived instead of the empirical method proposed in the thesis. Another direction of future work includes evaluating the reaction scheme with transmission control protocol (TCP) traffic and for multi-hop ad-hoc networks. In case of multi-hop networks, there are additional factors to consider such as routing protocols, routing decisions etc. Also, additional reaction methods become applicable. For eg. if node A determines node B to be misbehaving, node A can react such that it will not forward any packets originating from node B.

REFERENCES

REFERENCES

[1] IEEE Computer Society LAN MAN Standards Committee. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.ll-1999. 1999.

[2] V. R. Giri and N. Jaggi. “MAC Layer Misbehavior Effectiveness and Collective Aggressive Reaction Approach”. In: *Proc. 33rd IEEE Sarnoff Symposium*. Princeton, NJ, Apr. 2010, pp. 1–5.

[3] S. Radosavac, J. S. Baras, and I. Koutsopoulos. “A Framework for MAC Protocol Misbehavior Detection in Wireless Networks”. In: *4th ACM Workshop on Wireless Security*. Cologne, Germany, 2005, pp. 33–42.

[4] Y. Rong, S-K. Lee, and H-A. Choi. “Detecting Stations Cheating on Backoff Rules in 802.11 Networks Using Sequential Analysis”. In: *Proc. 25th IEEE International Conference on Computer Communications (INFOCOM)*. Barcelona, Spain, Apr. 2006, pp. 1–13.

[5] M. Raya et al. “DOMINO: Detecting MAC Layer Greedy Behavior in IEEE 802.11 Hotspots”. In: *IEEE Transactions on Mobile Computing* 5.12 (Dec. 2006), pp. 1691–1705.

[6] P. Kyasanur and N. H. Vaidya. “Selfish MAC layer Misbehavior in Wireless Networks”. In: *IEEE Transactions on Mobile Computing* 4.5 (Sept. 2005), pp. 502–516.

[7] L. Guang, C. Assi, and Y. Ye. “DREAM: A System for Detection and Reaction Against MAC Layer Misbehavior in Ad Hoc Networks”. In: *Elsevier Computer Communications* 30.8 (June 2007), pp. 1841–1853.

[8] A. A. Cardenas, S. Radosavac, and J. S. Baras. “Evaluation of Detection Algorithms for MAC Layer Misbehavior: Theory and Experiments”. In: *IEEE/ACM Transactions on Networking* 17.2 (Apr. 2009), pp. 607–617.

[9] L. Guang, C. Assi, and A. Benslimane. “Modeling and Analysis of Predictable Random Backoff in Selfish Environments”. In: *Proc. 9th ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems*. Terromolinos, Spain, 2006, pp. 86–90.

[10] N. Jaggi V. R. Giri and Vinod Namboodiri. “Distributed Reaction Mechanisms to Prevent Selfish Misbehavior in Wireless Ad Hoc Networks”. In: *IEEE Globecom 2011*. Houston, TX, Apr. 2011, pp. 1–5.

[11] G. Bianchi, L. Fratta, and M. Oliveri. “Performance Evaluation and Enhancement of the CSMA/CA MAC Protocol for 802.11 Wireless LANs”. In: *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)* 2 (Oct. 1996), pp. 392–396.

[12] D-M. Chiu and R. Jain. “Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks”. In: *Computer Networks and ISDN Systems* 17.1 (June 1989), pp. 1–14.

APPENDICES

APPENDIX A

NS-2 C++ Implementation

mac-timers.cc

File used in implementation of different types of misbehavior.

```
void
BackoffTimer::start(int cw, int idle, double difs)
{
        Scheduler &s = Scheduler::instance();

        assert(busy_ == 0);

        busy_ = 1;
        paused_ = 0;
        stime = s.clock();

        if(mac->misbhType_ == 1){
                //If Type 1 Misbehavior, then Alpha Misbehavior.
                cw = (int)(mac->misbhValue_ * cw);
                if(cw == 0) {
                        cw = 1;
                }
                rtime = (Random::random() % cw) * mac->phymib_.getSlotTime();
        } else if(mac->misbhType_ == 2){
                //If Type 2 Misbehavior, then deterministic Misbehavior. No
                    random selection of backoff timer.
                rtime = (mac->misbhValue_) * (mac->phymib_.getSlotTime());
        } else if(mac->misbhType_ == 3){
                //If Type 3 Misbehavior, then Beta Misbehavior.
                cw = max(mac->phymib_.getCWMin(),min((u_int32_t)(mac->
                    misbhValue_ * cw),mac->phymib_.getCWMax()));
                if(cw == 0) {
                        cw = 1;
                }
                rtime = (Random::random() % cw) * mac->phymib_.getSlotTime();
        } else if(mac->misbhType_ == 4){
                //If Type 4 Misbehavior, the adjustment is done in TCL Script.
                     So do the regular work.
                rtime = (Random::random() % cw) * mac->phymib_.getSlotTime();
        } else if(mac->misbhType_ == 5){
                //If Type 5 Misbehavior (fixed CW), then fix the CW to the
                    user provided CW value.
                cw = (int)mac->misbhValue_;
                if(cw == 0) {
                        cw = 1;
                }
                rtime = (Random::random() % cw) * mac->phymib_.getSlotTime();
        } else {
                //For no misbehavior, do the regular rtime calculation.
                //printf("\ncw %d misbhType:%d misbhValue:%f phymisbhType:%d
                    phymisbhValue:%d",cw, mac->misbhType_,mac->misbhValue_,mac
```

```
                ->phymib_.getMisbhType(),mac->phymib_.getMisbhValue());
            rtime = (Random::random() % cw) * mac->phymib_.getSlotTime();
    }

#ifdef USE_SLOT_TIME
    ROUND_TIME();
#endif
    difs_wait = difs;

    if(idle == 0)
            paused_ = 1;
    else {
            assert(rtime + difs_wait >= 0.0);
            s.schedule(this, &intr, rtime + difs_wait);
    }
}
```

mac-802_11.h

File defines the macros, class variables.

```
#define MAC_Subtype_ProbeReq   0x04
#define MAC_Subtype_ProbeRep   0x05
#define MISBH_SMA_PERIOD   500
#define MISBH_REACT_TYPE   5
#define MISBH_REACT_VALUE_DEFAULT ((phymib_.getCWMin())/2) //For CWFix
    misbehavior reaction
//####//
struct priority_queue {
  int frame_priority;
  struct priority_queue *next;
};

/* structure for maintaing neighbor IPT Time. Neighbors are represented by
    nodes of linked list.
ema_ratio is Exponential Moving Average is used for detection. */
struct misbh_neighborll {
  int index;
  double sma_packet_count;
  double sma_previous_pkt_rx_time;
  double sma_ipttime;
  double sma_sum_ipttime;
  double sma_ipttime_temp;
  double sma_ipttime_ratio;
  double sma_ratio;
  double sma_difference;
  double sma_var_sum;
  double sma_var;
  double sma_sd;
  double SmaTemp[MISBH_SMA_PERIOD];
  int misbehavior;
  struct misbh_neighborll *next;
};
//####//
```

```
class PHY_MIB {
public:
  PHY_MIB(Mac802_11 *parent);

  inline u_int32_t getCWMin() { return(CWMin); }
  inline u_int32_t getCWMax() { return(CWMax); }
  inline int getMisbhType() { return(MisbhType); }
  inline double getMisbhValue() { return(MisbhValue); }
  inline int getMisbhTime() { return(MisbhTime); }
  inline int getMisbhReact() { return(MisbhReact); }
//####//
  u_int32_t PreambleLength;
  u_int32_t PLCPHeaderLength;
  double     PLCPDataRate;
  int    MisbhType;
  double     MisbhValue;
  double     MisbhTime;
  int    MisbhReact;
};
//####//
  inline void set_nav(u_int16_t us) {
    double now = Scheduler::instance().clock();
    double t = us * 1e-6;
    if((now + t) > nav_) {
      nav_ = now + t;
      if(mhNav_.busy())
        mhNav_.stop();
      mhNav_.start(t);
    }
  }
  void calc_neighbor_sma_ipttime(int);
  double sma_detect_misbehavior();
  void print_neighborllist();
//####//
  struct ap_table *ap_list;
  struct priority_queue *queue_head;
  struct misbh_neighborll *neighborll_head;
//####//
  u_int32_t cw_;     // Contention Window
  u_int32_t ssrc_;     // STA Short Retry Count
  u_int32_t slrc_;     // STA Long Retry Count
  int    misbhType_; //Misbehavior Type
  double     misbhValue_;  //Misbehavior Value
  double     misbhTime_; //Misbehavior Time
  int    misbhReact_;  //Should the node react?
  double             misbhSmaPacketCount_; //Successfully sent packet. Counted
     based on the recevied ACK.
  double             misbhSmaPrevTime_; //Time at which previous ACK was received
  double             misbhSmaIpttime_; //Moving Average Inter packet transmission
      time
  double             misbhSmaSumIpttime_; //Cumulative Inter packet transmission
     time for moving average
  double             misbhSmaVarSum_; //Variance Sum
  double             misbhSmaVar_; //Variance
```

```
  double            misbhSmaSD_;         //Standard Deviation
  double            misbhSmaIpttimeTemp[MISBH_SMA_PERIOD];
  double            misbhRxPacketCount_; //RX Packet count
  double            misbhReactCount_; //React count
  double            misbhReactionFactor_; //Reaction Factor
  int               misbhNeighborCount_; //Number of neighbors
  double            misbhMaxIptRatio_; //Maximum IPT Ratio
  double            misbhMinIptRatio_; //Minimum IPT Ratio
//####//
```

mac-802_11.cc

File used to implement detection and reaction mechanisms.

Node's own parameters such as Misbehavior type, value, IPT moving average, reaction parameters are member variables of class PHY_MIB. Reaction parameters are applied to nodes in $Mac802_11 :: recv_timer$ function. Node's own moving average of IPT is calculated in $Mac802_11 :: recvCTS$. New function $Mac802_11 :: calc_neighbor_sma_ipttime$ is defined and used to calculate neighbor's moving average of IPT. This function updates the neighbor IPT in a linked list ($misbh_neighborll$). New function $Mac802_11 :: sma_detect_misbehavior$ implements the detection rule and the calculates the strength of the reaction.

```
//####//
  parent->bind("BeaconInterval_", &BeaconInterval);
  //Setting value in from TCL
  parent->bind("MisbhType_", &MisbhType);
  parent->bind("MisbhValue_", &MisbhValue);
  parent->bind("MisbhTime_", &MisbhTime);
  parent->bind("MisbhReact_", &MisbhReact);
  //####//
  infra_mode_ = 0;
  cw_ = phymib_.getCWMin();
  //Initializing variables
  misbhType_ = phymib_.getMisbhType();
  misbhValue_ = phymib_.getMisbhValue();
  misbhTime_ = phymib_.getMisbhTime();
  misbhReact_ = phymib_.getMisbhReact();
  misbhSmaPacketCount_ = 0;
  misbhSmaPrevTime_ = 0;
  misbhSmaIpttime_ = 0;
  misbhSmaVarSum_ = 0;
  misbhSmaVar_ = 0;
  misbhSmaSD_ = 0;
  misbhRxPacketCount_ = 0;
  misbhReactCount_ = 0;
  misbhReactionFactor_ = 0;
  misbhMaxIptRatio_ = 0.0;
  misbhMinIptRatio_ = 100.0;
//####//
  cache_node_count_ = 0;
  client_list = NULL;
  ap_list = NULL;
  queue_head = NULL;
  neighborll_head = NULL;
```

```
//####//
void
Mac802_11::recv_timer()
{
  u_int32_t src;
  hdr_cmn *ch = HDR_CMN(pktRx_);
  hdr_mac802_11 *mh = HDR_MAC802_11(pktRx_);
  u_int32_t dst = ETHER_ADDR(mh->dh_ra);
  u_int32_t ap_dst = ETHER_ADDR(mh->dh_3a);
  u_int8_t  type = mh->dh_fc.fc_type;
  u_int8_t  subtype = mh->dh_fc.fc_subtype;
  double misbehavior=0;
.
.
.
  if (netif_->node()->energy_model() &&
      netif_->node()->energy_model()->adaptivefidelity()) {
    src = ETHER_ADDR(mh->dh_ta);
    netif_->node()->energy_model()->add_neighbor(src);
  }
  /* This code is added only to control the duration of misbehavior.
     If this feature is not needed then this code is not necessary. */
  if((phymib_.getMisbhType())) {
    if((Scheduler::instance().clock() > 0) && (Scheduler::instance().clock() <
        misbhTime_)) {
      misbhType_ = phymib_.getMisbhType();
      misbhValue_ = phymib_.getMisbhValue();
    } else {
      misbhType_ = 0;
      misbhValue_ = 0;
    }
  }
  /* Following used by genuine nodes */
  if((!(phymib_.getMisbhType())) && (subtype == MAC_Subtype_RTS)) {
    src = ETHER_ADDR(mh->dh_ta);
    if((src) && (Address::instance().get_nodeaddr(((MobileNode*)(netif_->node
        ()))->nodeid()))) {
      /* packets from node 0 are not used for calculation */
      //if(((Address::instance().get_nodeaddr(((MobileNode*)(netif_->node()))
          ->nodeid())) != 2) && (src == 2)){
      misbhRxPacketCount_++;
      //}
      calc_neighbor_sma_ipttime(src);
      //print_neighborllist();
      if(((Address::instance().get_nodeaddr(((MobileNode*)(netif_->node()))->
          nodeid())) != 2) && (src == 2)){
        //   printf("\nNodeID: %d RxRTSCount: %f",(Address::instance().
             get_nodeaddr(((MobileNode*)(netif_->node()))->nodeid())),
             misbhRxPacketCount_);
      }
      //if((fmod(misbhRxPacketCount_,100)) == 0) {
      if(1) {
        if(Scheduler::instance().clock() > 50) {
          misbehavior = sma_detect_misbehavior();
```

```
        }
        if((misbhReact_)) {
          if(misbehavior) {
            //printf("\nClock: %f NodeID: %d misbehavior: %f",Scheduler::
                instance().clock(),(Address::instance().get_nodeaddr(((
                MobileNode*)(netif_->node()))->nodeid())),misbehavior);
            misbhReactionFactor_ = misbehavior;
            misbhType_ = MISBH_REACT_TYPE;
            misbhValue_ = (int)misbhReactionFactor_;
            misbhReactCount_ = misbhReactCount_ + 2;
          } else if(misbhReactCount_) {
            misbhType_ = MISBH_REACT_TYPE;
            misbhValue_ = (int)(MISBH_REACT_VALUE_DEFAULT/2);
            misbhReactCount_ = misbhReactCount_ - 1;
          } else {
            misbhType_ = 0;
            misbhValue_ = 0;
          }
        }
      }
    }
  }
  /*
   * Address Filtering
   */
//####//
void
Mac802_11::recvCTS(Packet *p)
{
  double ipttime=0.0;
.
.
.
  ssrc_ = 0;

  misbhSmaPacketCount_++;
  ipttime = (Scheduler::instance().clock() - misbhSmaPrevTime_);
  if(misbhSmaPacketCount_ > 1) {
    if((misbhSmaPacketCount_-1) < (MISBH_SMA_PERIOD)) {
      misbhSmaIpttimeTemp[((int)(misbhSmaPacketCount_-1)) - 1] = ipttime;
      misbhSmaSumIpttime_ = misbhSmaSumIpttime_ + ipttime;
    } else if((misbhSmaPacketCount_-1) == (MISBH_SMA_PERIOD)) {
      misbhSmaIpttimeTemp[(MISBH_SMA_PERIOD) - 1] = ipttime;
      misbhSmaSumIpttime_ = misbhSmaSumIpttime_ + ipttime;
      misbhSmaIpttime_ = misbhSmaSumIpttime_ / MISBH_SMA_PERIOD;
    } else if((misbhSmaPacketCount_-1) > (MISBH_SMA_PERIOD)) {
      if(((int)fmod((misbhSmaPacketCount_-1),(MISBH_SMA_PERIOD))) == 0) {
        misbhSmaSumIpttime_ = misbhSmaSumIpttime_ - misbhSmaIpttimeTemp[((int)
            fmod((misbhSmaPacketCount_-1),(MISBH_SMA_PERIOD))) + (
            MISBH_SMA_PERIOD- 1)];
        misbhSmaIpttimeTemp[((int)fmod((misbhSmaPacketCount_-1),(
            MISBH_SMA_PERIOD))) + (MISBH_SMA_PERIOD - 1)] = (Scheduler::
            instance().clock() - misbhSmaPrevTime_);
      } else {
```

```
        misbhSmaSumIpttime_ = misbhSmaSumIpttime_ - misbhSmaIpttimeTemp[((int)
            fmod((misbhSmaPacketCount_-1),(MISBH_SMA_PERIOD))) - 1];
        misbhSmaIpttimeTemp[((int)fmod((misbhSmaPacketCount_-1),(
            MISBH_SMA_PERIOD))) - 1] = ipttime;
      }
      misbhSmaSumIpttime_ = misbhSmaSumIpttime_ + ipttime;
      misbhSmaIpttime_ = misbhSmaSumIpttime_ / MISBH_SMA_PERIOD;
    }
  }
  misbhSmaPrevTime_ = Scheduler::instance().clock();

  tx_resume();
//####//
//At the end of the file
void Mac802_11::calc_neighbor_sma_ipttime(int src)
{
  struct misbh_neighborll *current_node=NULL,*temp_node=NULL;
  int node_found=0;
  current_node = neighborll_head;
  if(current_node == NULL) {
    node_found = 1;
    current_node = (struct misbh_neighborll *)malloc(sizeof(struct
        misbh_neighborll));
    current_node->index = src;
    current_node->sma_packet_count=1;
    current_node->sma_previous_pkt_rx_time = Scheduler::instance().clock();
    current_node->sma_sum_ipttime = 0.0;
    current_node->sma_ipttime = 0.0;
    current_node->sma_ratio = 0.0;
    current_node->misbehavior = 0;
    current_node->sma_difference=0;
    current_node->sma_var_sum=0;
    current_node->sma_var=0;
    current_node->sma_sd=0;

    current_node->next = NULL;
    neighborll_head = current_node;
    return;
  }
  while(current_node != NULL) {
    if(src == current_node->index) {
      node_found = 1;
      current_node->sma_packet_count++;
      if((current_node->sma_packet_count -1) < (MISBH_SMA_PERIOD)) {
        current_node->SmaTemp[((int)(current_node->sma_packet_count-1)) - 1] =
             (Scheduler::instance().clock() - current_node->
            sma_previous_pkt_rx_time);
        current_node->sma_sum_ipttime = current_node->sma_sum_ipttime + (
            Scheduler::instance().clock() - current_node->
            sma_previous_pkt_rx_time);
      } else if(((current_node->sma_packet_count-1)) == (MISBH_SMA_PERIOD)) {
        current_node->SmaTemp[(MISBH_SMA_PERIOD) - 1] = (Scheduler::instance()
            .clock() - current_node->sma_previous_pkt_rx_time);
```

```
      current_node->sma_sum_ipttime = current_node->sma_sum_ipttime + (
          Scheduler::instance().clock() - current_node->
          sma_previous_pkt_rx_time);
      current_node->sma_ipttime = current_node->sma_sum_ipttime /
          MISBH_SMA_PERIOD;
    } else if(((current_node->sma_packet_count-1)) > (MISBH_SMA_PERIOD)) {
      if(((int)fmod(((current_node->sma_packet_count-1)),(MISBH_SMA_PERIOD))
          ) == 0) {
        current_node->sma_sum_ipttime = current_node->sma_sum_ipttime -
            current_node->SmaTemp[((int)fmod(((current_node->
            sma_packet_count-1)),(MISBH_SMA_PERIOD))) + (MISBH_SMA_PERIOD-
            1)];
        current_node->SmaTemp[((int)fmod(((current_node->sma_packet_count-1)
            ),(MISBH_SMA_PERIOD))) + (MISBH_SMA_PERIOD - 1)] = (Scheduler::
            instance().clock() - current_node->sma_previous_pkt_rx_time);
      } else {
        current_node->sma_sum_ipttime = current_node->sma_sum_ipttime -
            current_node->SmaTemp[((int)fmod(((current_node->
            sma_packet_count-1)),(MISBH_SMA_PERIOD))) - 1];
        current_node->SmaTemp[((int)fmod(((current_node->sma_packet_count-1)
            ),(MISBH_SMA_PERIOD))) - 1] = (Scheduler::instance().clock() -
            current_node->sma_previous_pkt_rx_time);
      }
      current_node->sma_sum_ipttime = current_node->sma_sum_ipttime + (
          Scheduler::instance().clock() - current_node->
          sma_previous_pkt_rx_time);
      current_node->sma_ipttime = current_node->sma_sum_ipttime /
          MISBH_SMA_PERIOD;
    }
    current_node->sma_previous_pkt_rx_time = Scheduler::instance().clock();
    if(misbhSmaIpttime_ && current_node->sma_ipttime) {
      current_node->sma_ratio = misbhSmaIpttime_ / current_node->sma_ipttime
          ;
      misbhMaxIptRatio_ = current_node->sma_ratio > misbhMaxIptRatio_ ?
          current_node->sma_ratio : misbhMaxIptRatio_;
      misbhMinIptRatio_ = current_node->sma_ratio < misbhMinIptRatio_ ?
          current_node->sma_ratio : misbhMinIptRatio_;
    }
    break;
  }
  if(current_node->next != NULL) {
    current_node = current_node->next;
  } else {
    break;
  }
}
if(node_found == 0) {
  temp_node = (struct misbh_neighborll *)malloc(sizeof(struct
      misbh_neighborll));
  temp_node->index = src;
  temp_node->sma_packet_count=1;
  temp_node->sma_previous_pkt_rx_time = Scheduler::instance().clock();
  temp_node->sma_sum_ipttime = 0.0;
  temp_node->sma_ipttime = 0.0;
```

```
    temp_node->sma_ratio = 0.0;
    temp_node->misbehavior = 0;
    temp_node->sma_difference=0;
    temp_node->sma_var_sum=0;
    temp_node->sma_var=0;
    temp_node->sma_sd=0;
    temp_node->next = NULL;

    current_node->next = temp_node;
  }
}

double Mac802_11::sma_detect_misbehavior()
{
  int node_count=0,misbh_count=0;
  double misbehavior=0,kvalue=0.0,cwoptimal=0.0;
  struct misbh_neighborll *current_node=NULL;

  current_node = neighborll_head;
  if(current_node == NULL) {
    return 0;
  }
  while(current_node != NULL) {
    node_count++;
    if(((current_node->sma_ipttime * 1.25) < (misbhSmaIpttime_)) && (
        misbhSmaIpttime_) && (current_node->sma_ipttime)){
      current_node->misbehavior=1;
      misbehavior = (misbhSmaIpttime_ / current_node->sma_ipttime) >
          misbehavior ? (misbhSmaIpttime_ / current_node->sma_ipttime) :
          misbehavior;
      misbh_count++;
    }
    current_node = current_node->next;
  }
  misbehavior = misbehavior > 1 ? (1/misbehavior) : 0;
  if(!misbehavior) {
    return 0; //can be changed to 1 to indicate no misbehavior
  }
  //code below is part of reaction and can be moved to the function where this
       is function is called.
  kvalue = sqrt(2 * ((txtime(phymib_.getRTSlen(),basicRate_) +
      DSSS_MaxPropagationDelay + phymib_.getDIFS())/(phymib_.getSlotTime())));
  //printf("\n t_rts: %f tau: %f DIFS %f kvalue %f",(txtime(phymib_.getRTSlen
      (),basicRate_)),DSSS_MaxPropagationDelay,phymib_.getDIFS(),kvalue);
  cwoptimal = ((node_count + 1)) * kvalue;
  misbehavior = ((cwoptimal * (node_count*node_count) * (misbehavior *
      misbehavior))) / (200);
  misbehavior = misbehavior < 3 ? 3 : misbehavior;

  return misbehavior;
}
void Mac802_11::print_neighborllist()
{
  struct misbh_neighborll *current_node=NULL;
```

```
  current_node = neighborll_head;

  while(current_node != NULL){
    if((Address::instance().get_nodeaddr(((MobileNode*)(netif_->node()))->
        nodeid())) == 2) {
      printf("\nNodeID: %d ownipt: %f neighborid: %d neighboript: %f",(Address
          ::instance().get_nodeaddr(((MobileNode*)(netif_->node()))->nodeid())
          ),misbhSmaIpttime_, current_node->index, current_node->
          sma_ipttime_temp);
    }
    if(current_node->next != NULL) {
      current_node = current_node->next;
    } else {
      break;
    }
  }

}
//####//
```

APPENDIX B

NS-2 TCL Implementation

ns-default.tcl

File used to set default values misbehavior simulation parameters.

```
Mac/802_11 set BeaconInterval_        0.1      ;# 100ms
Mac/802_11 set ScanType_ PASSIVE
Mac/802_11 set ProbeDelay_ 0.0001     ;# 0.1 ms
Mac/802_11 set MaxChannelTime_ 0.011    ;# 11 ms
Mac/802_11 set MinChannelTime_ 0.005    ; # 5 ms
Mac/802_11 set ChannelTime_ 0.12    ;# 120 ms
#Following are the values used by all nodes. They are overwritten (configured)
    by the TCL script
Mac/802_11 set MisbhType_ 0       ;# No misbehavior
Mac/802_11 set MisbhValue_ 0.0      ;# No misbehavior and so value does not
    matter
Mac/802_11 set MisbhTime_ 0       ;# Zero misbehavior time
Mac/802_11 set MisbhReact_ 0      ;# No misbehavior reaction
```

misbh_udp_cntrl.tcl

File used to create the simulation topology.

```
# Initialize some parameters
Mac/802_11 set dataRate_ 2Mb
# rts thrshold is in bytes
Mac/802_11 set RTSThreshold_ 128

# ======================================================================
# Define options
# ======================================================================

set val(chan)    Channel/WirelessChannel;  # channel type
set val(prop)    Propagation/TwoRayGround; # radio-propagation model
set val(netif)  Phy/WirelessPhy;     # network interface type
set val(mac)  Mac/802_11;       # MAC type
set val(ifq)  Queue/DropTail/PriQueue;  # interface queue type
set val(ll)   LL;         # link layer type
set val(ant)  Antenna/OmniAntenna;     # antenna model
set val(x)       100;          # X dimension of the topography
set val(y)       100;          # Y dimension of the topography
set val(ifqlen) 50;        # max packet in ifq
set val(adhocRouting)   DumbAgent;     # routing protocol
# ======================================
# Main Program
# ======================================
if { $argc != 9 } {
        puts "Wrong no. of cmdline args."
        puts "Usage: ns misbh_udp_cntrl.tcl <nodes> <simulation time> <rate> <
            misbhnodes> <misbhtype> <misbhvalue> <misbhtime> <misbhreact> <
            outfile>"
```

```
        exit 0
}

set nodes [lindex $argv 0]
set stime [lindex $argv 1]
set rate [lindex $argv 2]
set misbhnodes [lindex $argv 3]
set misbhtype [lindex $argv 4]
set misbhvalue [lindex $argv 5]
set misbhtime [lindex $argv 6]
set misbhreact [lindex $argv 7]
set outfile [lindex $argv 8]

set  $misbhtime [expr {$misbhtime + 200}]

puts "Misbhnodes is $misbhnodes"
puts "Misbhtime is $misbhtime"
puts "Misbhreact is $misbhreact"

# Initialize Global Variables
set cwmin 31
set cwmax 1023
set packetsize 512
set trace_file trace_misbh_udp_cntrl.tr
#set nam_file nam_misbh_udp_cntrl.nam

# create simulator instance
set ns_          [new Simulator]

global defaultRNG
$defaultRNG seed 0

# setup topography object
set topo         [new Topography]

#Deleting Trace file
#exec rm $trace_file
#exec rm $nam_file

# create trace object for ns and nam
set tracefd   [open $trace_file w]
#set namfd  [open $nam_file w]

$ns_ trace-all $tracefd
#$ns_ namtrace-all-wireless $namfd $val(x) $val(y)
$ns_ use-newtrace

# define topology
$topo load_flatgrid $val(x) $val(y)

#
# Create God
#
create-god $nodes
```

```
# configure node
#

#$val(mac) set bandwidth_ 1.0e6
#$val(mac) set CWMin_ 31
#$val(mac) set CWMax_ 1023

#Node 0 is receiving. So niether misbehavior nor reaction.
$val(mac) set MisbhType_ 0
$val(mac) set MisbhValue_ 0.0
$val(mac) set MisbhTime_ 0
$val(mac) set MisbhReact_ 0

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

set node_(0) [$ns_ node]

#Create the specified number of mobilenodes [$nodes] and "attach" them to the
    channel.
#Here nine nodes are created-nodes 1 through 8
#Nodes can react only. They cannot misbehave.
$val(mac) set MisbhType_ 0
$val(mac) set MisbhValue_ 0
$val(mac) set MisbhTime_ 0
$val(mac) set MisbhReact_ $misbhreact

$ns_ node-config -macType $val(mac)

for {set i 1} {$i < [expr {$nodes-$misbhnodes}] } {incr i} {
  set node_($i) [$ns_ node]
  $node_($i) random-motion 0;# disable random motion
}

#Misbehaving node is has different configuration. After setting configuration
    parameters, misbehaving node is created.
#Misbehaving node cannot react.
proc set_misbh {misbhtype misbhvalue} {
  global val cwmin cwmax misbhtime
  if {$misbhtype == 0} {
    puts "No Misbehavior."
```

```
    puts "CW Min is $cwmin"
    puts "CW Max is $cwmax"
    $val(mac) set CWMin_ $cwmin
    $val(mac) set CWMax_ $cwmax
    $val(mac) set MisbhType_ 0
    $val(mac) set MisbhValue_ 0
    $val(mac) set MisbhReact_ 0
    puts "MisbhValue_ is 0"
  } elseif {$misbhtype == 1} {
    puts "Type 1 Misbehavior. Alpha behavior."
    puts "CW Min is $cwmin"
    puts "CW Max is $cwmax"
    $val(mac) set CWMin_ $cwmin
    $val(mac) set CWMax_ $cwmax
    $val(mac) set MisbhType_ 1
    $val(mac) set MisbhValue_ $misbhvalue
    $val(mac) set MisbhReact_ 0
    puts "MisbhValue_ is $misbhvalue."
  } elseif {$misbhtype == 2} {
    puts "Type 2 Misbehavior. Deterministic behavior."
    puts "CW Min is $cwmin"
    puts "CW Max is $cwmax"
    $val(mac) set CWMin_ $cwmin
    $val(mac) set CWMax_ $cwmax
    $val(mac) set MisbhType_ 2
    $val(mac) set MisbhValue_ $misbhvalue
    $val(mac) set MisbhReact_ 0
    puts "MisbhValue_ is $misbhvalue."
  } elseif {$misbhtype == 3} {
    puts "Type 3 Misbehavior. Beta behavior."
    if {$misbhvalue < 31} {
      set cwmin [expr {$misbhvalue*31}]
    }
    puts "CW Min is $cwmin"
    puts "CW Max is $cwmax"
    $val(mac) set CWMin_ $cwmin
    $val(mac) set CWMax_ $cwmax
    $val(mac) set MisbhType_ 3
    $val(mac) set MisbhValue_ $misbhvalue
    $val(mac) set MisbhReact_ 0
    puts "MisbhValue_ is $misbhvalue."
  } elseif {$misbhtype == 4} {
    puts "Type 4 Misbehavior. Fixed Maximum Contention Window behavior."
    if {$misbhvalue < 31} {
      set cwmin $misbhvalue
    }
    puts "CW Min is $cwmin"
    puts "CW Max is $misbhvalue"
    $val(mac) set CWMin_ $cwmin
    $val(mac) set CWMax_ $misbhvalue
    $val(mac) set MisbhType_ 4
    $val(mac) set MisbhValue_ 0.0
    $val(mac) set MisbhReact_ 0
    puts "MisbhValue_ is $misbhvalue."
```

```
  } elseif {$misbhtype == 5} {
    puts "Type 5 Misbehavior. Fixed Contention Window behavior."
    $val(mac) set MisbhType_ 5
    $val(mac) set MisbhValue_ $misbhvalue
    $val(mac) set MisbhReact_ 0
    puts "MisbhValue_ is $misbhvalue."
  } else {
    puts "Invalid Misbehavior Type. Considering it as no misbehavior."
    $val(mac) set MisbhType_ 0
    $val(mac) set MisbhValue_ 0.0
    $val(mac) set MisbhReact_ 0
  }
  $val(mac) set MisbhTime_ $misbhtime
}

#Creating Misbehaving nodes
for {set count 0} {$count < $misbhnodes} {incr count} {
  #If all misbehaving need to use same misbhvalue, multiply count by zero in
     below line.
  set_misbh $misbhtype [expr {$misbhvalue + ($count * 0)}]
  $ns_ node-config -macType $val(mac)
  set node_($i) [$ns_ node]
  $node_($i) random-motion 0;# disable random motion
  incr i
}

#Provide initial (X,Y, for now Z=0) co-ordinates for mobilenodes
$node_(0) set X_ 50.0
$node_(0) set Y_ 50.0
$node_(0) set Z_ 0.0

##nodes in a circular way
set pi 3.1416
set r 50
set a [expr {2*$pi*0.111111111}]
for {set i 1} {$i < $nodes } {incr i} {
  $node_($i) set X_ [ expr { 50 + ($r * cos($i * $a))}]
  $node_($i) set Y_ [ expr { 50 + ($r * sin($i * $a))}]
  $node_($i) set Z_ 0.0
  $ns_ initial_node_pos $node_($i) 2
}

$ns_ initial_node_pos $node_(0) 2

#traffic between nodes
set sink [new Agent/Null]
$ns_ attach-agent $node_(0) $sink

for {set i 1} {$i < $nodes } {incr i} {
  set udp($i) [new Agent/UDP]
  $udp($i) set class_ 2
  set cbr($i) [new Application/Traffic/CBR]
  $cbr($i) attach-agent $udp($i)
  $cbr($i) set packetSize_ $packetsize
```

```
        $cbr($i) set interval_ $rate
        #$cbr($i) set rate_ $rate
        $cbr($i) set random_ false

  $ns_ attach-agent $node_($i) $udp($i)

  $ns_ connect $udp($i) $sink

  $ns_ at 0.0 "$cbr($i) start"
}

# Tell nodes when the simulation ends
for {set i 0} {$i < $nodes } {incr i} {
  $ns_ at $stime "$node_($i) reset";
}
#$ns_ at $stime "puts \"NS Exiting...\" "
$ns_ at $stime "stop"

#Define a 'stop' procedure
proc stop {} {
  #global ns_ tracefd namfd
  global ns_ tracefd nodes trace_file outfile misbhnodes
  #global nam_file
  $ns_ flush-trace
  #close $namfd
  close $tracefd
  #exec nam $nam_file &
  $ns_ halt
  puts "Analyzing trace file..."
  set tmp [ exec perl throughput_cntrl.pl $trace_file $nodes $misbhnodes | tee
      ./run/$outfile]
  puts stdout $tmp

  #Deleting Trace file
  #exec rm $trace_file
  #exec rm $nam_file

  exit 0
}

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

APPENDIX C
Helper Scripts

throughput_cntrl.pl
File used to calculate throughput by analyzing trace file.

```
#!/bin/perl

use strict;
use warnings;

#Input trace file
my($infile) =$ARGV[0];

#Keep track of variables
my(@x);
my($nodes) = $ARGV[1];
my($misbhnodes) = $ARGV[2];
my($packet_size) = 512;
my($total_tx_count) = 0;
my($total_rx_count) = 0;
my($simulation_time) = 0;
my($system_throughput) = 0;
my($system_pdr) = 0;
my(%hash_table_send) = ();
my(%hash_table_receive) = ();
my($hash_key) = 0;
my($hash_value) = 0;
my($sip) = 0;
my($node_send_count) = 0;
my($node_receive_count) = 0;
my($node_throughput) = 0;
my($result) = 0;
my($sum) = 0;
my($sum_misbh) = 0;
my($sqr_sum) = 0;
my($ratio) = 0;
my($fairness) = 0;


open(DATA,"$infile") || die "could't open $infile$!";
while(<DATA>)
{
  @x=split(' ');
  if((defined($x[44])) && ($x[44] eq 'cbr') && ($x[18] eq 'MAC') && ($x[0] eq
      's')) {
    $hash_key = $x[30];
    $hash_value = $hash_table_send{"$hash_key"};
    $hash_value++;
    $hash_table_send{"$hash_key"} = $hash_value;
    #Counting send events.
    $total_tx_count++;
```

```
  } elsif((defined($x[44])) && ($x[44] eq 'cbr') && ($x[18] eq 'MAC') && ($x
      [0] eq 'r')) {
    #Source IP is the key for the hash table.
    $hash_key = $x[30];
    $hash_value = $hash_table_receive{"$hash_key"};
    $hash_value++;
    #Per source receive count is maintained.
    $hash_table_receive{"$hash_key"} = $hash_value;
    $total_rx_count++;
  }
}
$simulation_time = $x[2];
#$packet_size = $x[36];

print("\n *** Trace File Analysis output ***\n");
#print("\n Packet Size = $packet_size bytes");
print("\n Total Transmit Count = $total_tx_count");
print("\n Total Receive Count = $total_rx_count");
print("\n Simulation Time = $simulation_time seconds");

if($total_rx_count != 0) {
  $system_throughput = ($total_rx_count) / $simulation_time;
        #print("\n System Throughput = $system_throughput packets per second
            .");
  $system_throughput = ($total_rx_count * $packet_size * 8) / $simulation_time
      ;
  $system_throughput = $system_throughput / 1000;
  $result = sprintf("%.3f", $system_throughput);
        print("\n System Throughput = $result Kbps");
} else {
  print("\n No packet reached!");
}
print("\n");
for($sip = 1;$sip < $nodes;$sip++) {
  $hash_key = $sip.".0";
        $node_send_count = $hash_table_send{"$hash_key"};
        $node_receive_count = $hash_table_receive{"$hash_key"};
        $node_throughput = ($node_receive_count * $packet_size * 8) /
            $simulation_time;
        $node_throughput = $node_throughput / 1000;
  $result = sprintf("%.3f", $node_throughput);
        print("\n Node $sip  Send = $node_send_count Receive =
            $node_receive_count Throughput = $result Kbps");

  if($sip < ($nodes - $misbhnodes)) {
    $sum = $sum + $result;
    $sqr_sum = $sqr_sum + ($result * $result);
  } else {
    $sum_misbh = $sum_misbh + $result;
  }
}
print("\n");
if($misbhnodes == 0) {
  $ratio = 0;
```

```
} else {
  $ratio = ($sum_misbh/$misbhnodes)/($sum/(($nodes-1) - $misbhnodes));
}
print("\n Misbh/Normal Throughput Ratio = $ratio");

$fairness = ($sum * $sum) / ((($nodes - 1) - $misbhnodes) * $sqr_sum);
print("\n Fairness Index = $fairness");

print("\n");
print("\n");
close DATA;
exit(0);
```

run.sh

File used to configure varying number of nodes, misbehavior strength, react or no react.

```
#/bin/bash
SIM_TIME=6000
DATA_RATE=0.01
MISBH_TIME=6000
MISBH_TYPE=1
for n in 5 10 15 20
do
  echo "ns misbh_udp_cntrl.tcl $n ${SIM_TIME} ${DATA_RATE} 0 0 0 0 0 file_${n}
      _${SIM_TIME}_${DATA_RATE}_0_0_0"
  ns misbh_udp_cntrl.tcl $n ${SIM_TIME} ${DATA_RATE} 0 0 0 0 0 file_${n}_${
      SIM_TIME}_${DATA_RATE}_0_0_0
  for m in 1 2
  do
    for v in 0.1
    do
      for re in 0 1
      do
        echo "ns misbh_udp_cntrl.tcl $n ${SIM_TIME} ${DATA_RATE} $m ${
            MISBH_TYPE} $v ${MISBH_TIME} $re file_${n}_${m}_${v}_${re}"
        ns misbh_udp_cntrl.tcl $n ${SIM_TIME} ${DATA_RATE} $m ${MISBH_TYPE} $v
             ${MISBH_TIME} $re file_${n}_${m}_${v}_${re}
      done
    done
  done
done
```