

DISSERTATION

TOWARDS EMULATION OF LARGE-SCALE IP NETWORKS USING END-TO-
END PACKET DELAY CHARACTERISTICS

Submitted by

Daniel A. Vivanco

Department of Electrical & Computer Engineering

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2008

UMI Number: 3332753

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3332753

Copyright 2008 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.


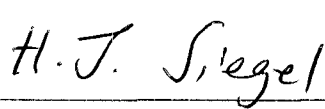
ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

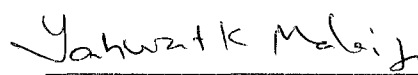
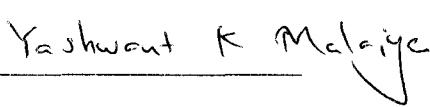
COLORADO STATE UNIVERSITY

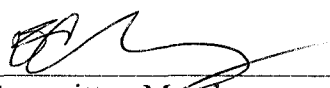
May 6, 2008

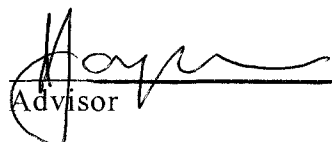
WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY DANIEL A. VIVANCO ENTITLED TOWARDS EMULATION OF LARGE-SCALE IP NETWORKS USING END-TO-END PACKET DELAY CHARACTERISTICS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

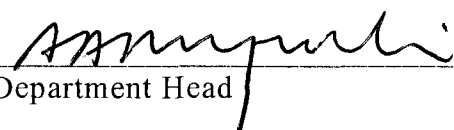
Committee on Graduate Work

 
Committee Member

 
Committee Member

 Edwin Chong
Committee Member

 Anura Jayasumana,
Advisor

 A. A. MACIEJEWSKI
Department Head

ABSTRACT OF DISSERTATION

TOWARDS EMULATION OF LARGE-SCALE IP NETWORKS USING END-TO-END PACKET DELAY CHARACTERISTICS

Network emulation combines concepts from network simulation and measurements and provides an emulated network testbed over which application and protocol software can be evaluated. Network emulators allow the investigation of the interaction of network and protocols and applications in a controllable and repeatable manner. Existing network emulators are not scalable due to the limitations of available computer hardware infrastructure and the reliance on one-to-one packet mapping and modeling schemes.

This research proposes a measurement-based modeling methodology for the design of a network-in-a-box emulator. The methodology aims at overcoming the limitation of computational overhead and end-to-end network system characterization complexity. A comprehensive study of end-to-end packet delay dynamics, in the context of network system modeling, is presented.

A framework for large scale IP network emulation, named Overall Trend Replicating Network Emulator Tool (OTRENET), is presented. OTRENET intercepts data packet streams and modifies them, based on network system

models, in real-time. The complexity and overhead is reduced over those of packet-by-packet mapping and modeling, while producing results consistent with measurements by means of a traffic sampling algorithm. The algorithm monitors traffic metrics at a per-packet level, to dynamically separate sequences of packets into frames. Traffic behavior is then characterized by the average response of each time frame. The proposed Average Traffic Sampler by Time Frame Segmentation Algorithm captures significant trends of the traffic metrics while not being sensitive to instantaneous fluctuations. Design, implementation and performance of the proposed algorithm and the emulator are described in detail. Experimental results are used to demonstrate the effectiveness of OTRENET in replicating realistic conditions imposed by modeled environments.

A comprehensive study of end-to-end packet delay dynamics, in the context of network system modeling, is presented. Theoretical basis, techniques and measurements for network packet delay dynamics characterization for various sending rate conditions and network stages have been developed. Modeling network systems by means of modeling of end-to-end packet delay dynamics is performed with emphasis on the effect due to cross traffic, sending rate and packet size. Measurements of packet delays over the Internet under various conditions indicate that packet autocorrelation dynamics change according to the sending rate and packet size of the probes. Moreover, under weakly-stationary network conditions, traditional ARMA and ARIMA time series techniques can be used to model packet delay and IPG processes. Under these conditions, goodness-of-fit results demonstrate the modeling accuracy for both packet delay

and IPG processes for cases where sending bit rate is relatively small compared to the link capacity. However, as the sending bit rate increases, as a fraction of the bandwidth, IPG becomes a better alternative for network system modeling.

Measurement based analysis of packet streams has also demonstrated that packet autocorrelation, along with other packet delay characteristics, tends to vary in time in a non-stationary manner. A novel approach for online modeling end-to-end packet delay dynamics is proposed to address this. Proposed methodology models and captures the network system characteristics taking into account the non-stationarity of the packet delay samples by identifying time frames during which the trace can be considered to be weakly- stationary, while keeping computational and storage requirements low. Experiment results demonstrate the potential for online packet delay classification with the proposed algorithm, while keeping computational and storage requirements low. In general, results presented show that analyzing packet delay processes by modeling the segmented traces yield a better understanding of the network system dynamics.

Daniel A. Vivanco
Department of Electrical & Computer Engineering
Colorado State University
Fort Collins, Colorado 80523
Summer, 2008

TABLE OF CONTENTS

CHAPTER 1.	INTRODUCTION.....	2
1.1	Evolution of the Internet	4
1.2	Modeling and prediction techniques for network and application performance and development.....	5
1.3	Challenges in end-to-end network modeling and prediction	6
1.4	Dissertation outline	10
CHAPTER 2.	OVERVIEW OF NETWORK EMULATOR TOOLS.....	13
2.1	Prior approaches for implementing network emulators	14
2.2	NISTNET	17
2.2.1	NISTNET architecture	18
2.2.2	Applying NISTNET rules to incoming traffic	19
2.3	NSE.....	20
2.2.3	Real-time scheduler	22
2.2.4	Tap agent.....	23
2.2.5	Network objects	23
2.4	Advantages and limitations of NISTNET and NSE	23
CHAPTER 3.	REVIEW OF END-TO-END PACKET DELAY.....	25
3.1	Packet delay	26
3.2	Components of packet delay	27
3.3	Packet delay modeling	28
3.4	Challenges on measuring packet delay over the internet	33
3.5	Packet delay variation.....	34
3.5.1	IPDV: Inter-packet delay variation	35
3.5.2	PDV: Packet delay variation	36
CHAPTER 4.	DESIGN OF A LARGE SCALE IP NETWORK EMULATOR TOOL.....	38
4.1	Overview of network emulation.....	39
4.2	OTRENET: Overall Trend Replicating Network Emulator Tool	41
4.3	OTRENET UNITS: Design, Implementation, and Synchronization	44
4.3.1	Input traffic sampler unit	44
4.3.2	Network model unit	44
4.3.3	Traffic adjuster unit	45
4.3.4	Synchronization among units	46
CHAPTER 5.	TRAFFIC SAMPLER BY TIME FRAME SEGMENTATION ALGORITHM. .	49
5.1	Algorithm description.....	50
5.2	Evaluation of threshold function for ASAF algorithm.....	58
5.3	Performance analysis and comparison of traffic sampler algorithm ...	60
CHAPTER 6.	OTRENET PERFORMANCE ANALYSIS.....	69
6.1	Results	69
6.2	Remarks	75
CHAPTER 7.	A MEASUREMENT-BASED MODELING APPROACH FOR NETWORK-INDUCED PACKET DELAY	77
7.1	Modeling end-to-end packet delay using time series techniques	78
7.1.1	ACF and PACF analysis for end-to-end packet delay.....	79
7.1.2	ARMA and ARIMA model selection for end-to-end packet delay processes.....	82
7.1.3	Optimization criteria and fitting procedures for ARMA Models.....	84
7.1.4	Diagnostic checking for ARMA and ARIMA models.....	86
7.2	Analysis of measurment data	87

7.2.1	Methodology for fitting ARMA and ARIMA models into packet delay series	87
7.2.2	Modeling results.....	89
7.2.3	Goodness of ARMA model fitting vs. time series dynamics	96
7.3	Remarks	108
CHAPTER 8.	AN ONLINE METHODOLOGY FOR MODELING NON-STATIONARY END-TO-END PACKET DELAY	109
8.1	Impact of non-stationarity on network system dynamics.....	110
8.2	Exploring LRD on packet delay series.....	113
8.3	Statistical methodologies for modeling non-stationary end-to-end packet delta series.	119
8.4	A computational efficient method for segmenting non-stationary packet delay series.....	123
8.4.1	An online packet delay segmentation algorithm.....	127
8.4.2	Memory storage savings vs. computational overhead.....	131
8.4.3	Segmentation algorithm settings	134
8.5	Performance Results	139
8.5.1	Memory storage savings vs. non-stationarity	153
8.6	Remarks	156
CHAPTER 9.	CONCLUSIONS	158
REFERENCES.	161	
APENDIX A.	Script for traffic sampler by time frame segmentation algorithm.....	169
A.1.	Matlab script for traffic sampler by time frame segmentation	169
APENDIX B.	Script for OTRENET Emulator	173
B.1.	Tcl script for NS network simulation.....	173
B.2.	Awk script for measuring simulated traffic metrics and online traffic sampler by time frame segmentation.....	176
B.3.	Perl script that executes and synchronizes otrenet units.....	182
B.4.	Modified version of knistnet.c	184
APENDIX C.	Script for online packet delay segmentation algorithm.....	207
C.1.	Matlab script for implementing packet delay segmentation	207

TABLES AND FIGURES

FIGURES:

2.1. NSE objects infrastructure.	22
4.1. OTRENET architecture.....	42
4.2. Flow diagram of the main OTRENET code and module synchronization.	47
5.1. Time frame segmentation according to traffic variation, (a) Traffic metric vs. time, (b) Cumulative metrics and exponential threshold vs. time	53
5.2. Time frame segmentation for simulated delay, (a) Instantaneous simulated delay and averaged simulated delay using time frame segmentation vs. time, (b) Variation of simulated cumulative delay and exponential threshold vs. time	57
5.3. Comparison of metric threshold functions.	59
5.4. Comparison of the estimated average traffic sampler by time frame segmentation algorithm against C-MA, (a) Analyzed time 0-500 sec, (b) Analyzed time 180-350 sec.....	64
5.5. Comparison analysis of the estimated average traffic sampler by time frame segmentation algorithm and modified MA, (a) Reduction of number of frames comparison analysis, (b) MSE comparison analysis.....	66
6.1. Emulation configuration and test environment.....	71
6.2. Input bit rate variation vs. time.	72
6.3. Error sampling vs. rate of increase of bit rate.....	73
6.4. Error in delay emulation vs. rate of increase of bit rate.	74
7.1. ACF function for lag 1-20 of packet delay values for varied sending bit rates (0.5 – 80Mbps) using 64 bytes packet size	90
7.2. PACF function for lag 1-20 of packet delay values for varied sending bit rates(0.25,1,30 and 70Mbps) using 64 bytes packet size.....	90
7.3. ACF function for lag 1-20 of packet delay values for varied sending bit rates (0.5 - 80Mbps) using 256 bytes packet size.....	93
7.4. PACF function for lag 1-20 of packet delay values for varied sending bit rates (0.5 - 80Mbps) using 256 bytes packet size.....	94
7.5. Hurst parameter of the packet delay values for varied sending bit rates (0.5-80Mbps) using 64 and 256 bytes packet size.....	98
7.6. IDI for blocks of k consecutives packet delay samples for varied sending bit rates, (a) using 64 bytes packet size, (b) using 256 bytes packet size.....	100
7.7. Normal Q-Q plot residual of fitted ARMA models for packet delay and IPG traces for varied sending bit rates using 64 bytes packet size.	104
7.8. Normal Q-Q Plot Residual R^2 values for test of randomness for residual of fitted ARMA models for packet delay and IPG traces for varied sending bit rates and packet sizes.	105
7.9. p-values of Ljung-Box test of randomness for residual of fitted ARMA models for packet delay and IPG traces for varied sending bit rates and packet sizes.	106
8.1. Packet delay analysis for scenario; 130.206.163.166 → 157.181.172.103, (8.1.a) Packet delay trace, (8.1.b) ACF distribution of packet delay trace, (8.1.c) Recurrence plot of packet delay trace.	115
8.2. Packet delay analysis for scenario; 130.206.163.166 → 132.65.240.106, (8.2.a) Packet delay trace, (8.2.b) ACF distribution of packet delay trace, (8.2.c) Recurrence plot of packet delay trace.	115

8.3. Location of Φ_0 and Φ_1 models for the divergence test method.	125
8.4. Packet Delay Series Segmentation Methodology.	129
8.5. Flow diagram of online packet delay segmentation algorithm.	139
8.6. Segmentation analysis of Packet Delay trace for Run1 HUJI→ UNAV experiment using 100pps, (8.6.a) Packet delay trace, (8.6.b) Segmentation process of the packet delay trace, (8.6.c) ACF distribution of packet delay.	143
8.7. Segmentation analysis of packet delay trace for HUJI→ UNAV experiment, (8.7.1.x) Run 4 using 2000pps, (8.7.2.x) Run 4 using 100pps, (8.7.3.x) Run 10 using 1000pps.	146
8.8. Performance of the proposed algorithm under different values of λ , (8.8.a) Number of segments vs. λ , (8.8.b) Average entropy vs. λ	149
8.9. Evolution in time of two AR coefficients used for online modeling of packet delay trace shown on Figure 8.7.3.a.	152
8.10. Index of Dispersion of Intervals (IDI) for blocks of k consecutive packet delay samples obtained from multiple traces.	154

TABLES:

7.1. End-to-end packet delay model fitting for different sending bit rate scenarios using 64 bytes packet size.	92
7.2. End-to-end packet delay model fitting for different sending bit rate scenarios using 256 bytes packet size.	95
8.1. Algorithm settings used on segmentation analysis.	138
8.2. Packet Delay Characteristics and UTC times for the HUJI→ UNAV set of experiment results.	141
8.3. Percentage of memory storage savings using segmentation algorithm as a function of λ	156

CHAPTER 1. INTRODUCTION

Network research and development generally requires a simulation, emulation, or testbed environment to test and evaluate the performance of protocols, algorithms, services, and applications for both wireless and wired networks. When the target network is sufficiently large, simulation can consume a large amount of memory, and results are mostly based on traffic and network modeling assumptions. Network emulators, like network simulators, allow for investigation the interaction between networks and protocols in a controllable and repeatable manner. In addition, compared to testeds, their construction is less labor intensive and costly.

Despite multiple studies on network emulation, existing emulator modules are still no-scalable due to the limitation of available physical infrastructure and the one-to-one packet mapping and modeling scheme. This research proposes a measurement-based modeling methodology for the design of a network-in-a-box emulator, which aims to overcome the limitation of computational overhead and network system modeling. A framework for large scale IP network emulation, named Overall Trend Replicating Network Emulator Tool (OTRENET), is formally introduced in this research. OTRENET overcomes the overhead of packet-by-packet mapping and modeling, while keeping track of the consistency of results, by means of a proposed Average Traffic Sampler by Time Frame Segmentation Algorithm.

Complementarily, a comprehensive study of end-to-end packet delay dynamics, in the context of network system modeling, is presented in this dissertation. Modeling network systems by means of end-to-end packet delay characterization is performed with emphasis on the effect due to cross traffic, sending rate, and packet size. The impact of non-stationarity on modeling network system dynamics is also analyzed in this research. A computer network system is considered non-stationary when its statistic properties vary over time. In the context of this analysis, a computationally efficient methodology for online segmentation and modeling of packet delay series based on an adaptive AR model, Kalman Filtering algorithm, and a modified version of the Divergence-Test is proposed. Experimental results demonstrate the online packet delay classification capability of the proposed algorithm based on the non-stationarity of the observations, while keeping computational and storage requirements low.

A brief overview of the evolution of the Internet is provided in Section 1.1. In Section 1.2 alternatives for modeling and prediction of network and applications performance and development are presented. Section 1.3 highlights the challenges associated with current approaches for modeling and prediction of network and applications. In addition, in Section 1.3 the motivation of this research is discussed. Finally in Section 1.4, the outline of the dissertation is presented.

1.1 EVOLUTION OF THE INTERNET

In 1973, the U.S. Defense Advanced Research Projects Agency (DARPA) started a research program to investigate the development of communication protocols, which would allow networked computers to communicate transparently across multiple, linked packet networks [80]. This project was named the Internetting project and the system of networks that emerged from this research are now known as the Internet. TCP/IP protocol suite was developed over the course of this research. In 1986, the U.S. National Science Foundation (NSF) initiated the development of the NSFNET project, which today provides a major backbone communication service for the Internet. The National Aeronautics and Space Administration (NASA) and the U.S. Department of Energy contributed additional backbone facilities in the form of the NSINET and ESNET respectively.

Today Internet has revolutionized the computer and communications world like nothing before. Internet is at once a world-wide broadcasting capability, a mechanism for information dissemination, and a medium for collaboration and interaction between individuals and their computers without regard for geographic location. A large number of applications, ranging from voice, broadcast and on-demand video, gaming, data transfer, peer-to-peer, among others, are carried over the Internet daily.

1.2 MODELING AND PREDICTION TECHNIQUES FOR NETWORK AND APPLICATION PERFORMANCE AND DEVELOPMENT

Understanding the nature of end-to-end packet dynamics is crucial to several areas of application development, routing and transport protocol design, and congestion and flow control algorithm development. For instance, on the development and testing of congestion control algorithms [34], routing protocols [48], and real-time applications [72], a deep understanding of packet flow characteristics, i.e. one-way delay and jitter, is crucial. Application software developed for emerging complex distributed systems such as Collaborative Adaptive Sensing of the Atmosphere [8] also need to be evaluated and tested under a wide variety of network conditions.

It can be difficult to study network protocols and distributed applications in real networks because of the complexity of the network, the randomness associated with queuing and processing interaction of packet flows, and load variations on different links. Several alternatives have been presented for accurate representation of end-to-end packet dynamics. For instance, queuing theory has been used as a powerful tool for modeling packet flow dynamics. However, such an approach requires knowledge of the inter-arrival and inter-departure traffic distribution at every single link, resulting in tremendous computational demands, thus rendering it infeasible for large-scale networks [70]. The use of actual packet traces, i.e. packet delay traces, has also been proposed. Although this approach entirely captures traffic and network dynamics in a granular manner, its main limitation results from the large amount of data

needed to be collected and stored, depending on the duration of the observations. Network simulation approaches are also suggested alternatives. Network simulations are modeled representations of a network system that allow the researcher to create network topologies and conditions that are difficult to reproducibly achieve [27]. Packet level simulations do not generate real network traffic, but rather model traffic and major network components internally. Modeled traffic, in general, assumes well-defined distributions, i.e. Poisson or Pareto, throughout the entire simulation. These assumptions can obscure the understanding of behavior in real world situations by concealing their random nature [35]. Network testbed-based approaches are also proposed for studying network and application performance and development. However, while they closely mimic realistic characteristics, constructing testbeds is labor intensive and costly.

Conversely, network emulators combine real world and modeled network components to provide an emulated network testbed over which application and protocol software may be evaluated. Network emulators alter real network traffic between nodes in a physical network based on various modeled network configurations. In general, such network models can be either packet level simulation or any other end-to-end packet model.

1.3 CHALLENGES IN END-TO-END NETWORK MODELING AND PREDICTION

Modeling behavior of end-to-end packet dynamics over the Internet is not without major challenges. Complexity of network topologies, together with

randomness and non-stationarity of packet data streams [35], make it difficult to capture end-to-end packet dynamics over the Internet.

Modeling end-to-end packet dynamics by means of network emulator tools also comes with challenges; among them, computational overhead and network system modeling. Several network emulators have emerged in an attempt to come up with accurate and inexpensive network emulator solutions. In general, incoming real packet data streams are altered within a network emulator tool based on modeled end-to-end packet dynamics. Two main approaches are typically used. The first approach is to capture each incoming data packet and translate it into synthetic replicates, which are used in an embedded packet level simulation model of the network. Simulated packets are then converted into real packet data [26] [32]. Such an approach yields to granular analysis of packet flow dynamics and its interaction with the network components; however, the computational overhead involved in these processes can rise considerably for high loads and complex network topologies under standard computer station conditions [26].

Conversely, the second alternative aims to reduce computational overhead by providing an environment for evaluation of end-to-end packet dynamics as the network system is abstracted to a simple router with specific packet handling operations [3] [19] [71]. However, topology related protocols, such as routing protocols and queuing metrics, cannot be evaluated. In addition, packet-handling

operations need to be dictated by a pre-defined modeled representation of the network system, which has to be obtained by other means.

This research proposes a dynamic sampling algorithm that collects the average-sampled stream characteristics of an incoming packet stream without affecting its behavior, while maintaining lower computational overhead than a packet-by-packet capturing approach. In search of generic models, we approached the problem of end-to-end packet dynamics characterization and prediction by proposing novel modeling techniques based on time series analysis analyses for the design of a large scale IP network emulator tool. Modeling network end-to-end packet dynamics is achieved by characterizing packet delay with emphasis on the effect due to cross traffic, sending rate, and packet size under weakly-stationary and non-stationary network conditions [17].

This work is supplemented by a detailed study on end-to-end packet delay modeling. This research also develops a theoretical foundation, techniques and measurements for characterization of network packet delay dynamics under various sending rate conditions and network stages. Using abundant measurements of packet delay over the Internet under various conditions, we found that traditional ARMA and ARIMA time series techniques [17] can be used to model packet delay and IPG processes under weakly-stationary network conditions. Under these conditions, model goodness-of-fit results demonstrate modeling accuracy for both packet delay and IPG processes under small sending

bit rate conditions. However, as the sending bit rate increases as a fraction of the bandwidth, IPG becomes a better alternative for network system modeling [69].

In the context of modeling end-to-end packet dynamics, an in-depth study of packet delay under various network conditions is also performed. This analysis concludes that using packet delay characteristics (such as mean and higher moments) for network system modeling does not always result in a complete system model [59] [69] due to the fact that correlation of packets belonging to the same stream is not considered. However, autocorrelation of sample packet delay is an effective way for faithfully modeling associated characteristics [59] [69]. Measurement analysis of various packet streams demonstrates that packet autocorrelation, along with other packet delay characteristics, tends to vary in time under non-stationary network conditions [35]. Thus modeling techniques based on this metric need to be adjusted according to changes in the system dynamics. A methodology for online modeling of end-to-end packet delay characteristics induced by non-stationary network conditions is introduced. Effect of network dynamics induced into a packet flow is captured by means of Adaptive AR statistical models. Test statistics techniques are used sequentially to detect significant changes on the model parameters. This methodology separates, in real-time, non-stationary packet delay traces into stationary segments, in which a segment is generated only when a significant change on the system dynamics is detected, and each segment is represented by a different statistical model. Results demonstrate the potential online packet delay classification capability of the proposed algorithm

based on the no-stationarity of the observations. False sense of LRD on packet delay is also studied in the context of the proposed algorithm, and the importance of distinguishing it when modeling packet delay processes is highlighted

1.4 DISSERTATION OUTLINE

The reminder of this research is structured as follows. In CHAPTER 2, an overview of previous studies on network emulation is presented. Advantages and limitations of previous studies related to this research are listed and discussed.

In CHAPTER 3 a review of end-to-end packet delay is presented. Techniques on how to mitigate their impact on the quality of experience are presented. This chapter focuses on the study end-to-end packet delay characteristics, and it is also supplemented by the theoretical foundation, metrics and techniques for network modeling based on end-to-end packet delay. Multiple alternatives for packet delay modeling are presented and critiqued in CHAPTER 3 .

In CHAPTER 4 the design of a scalable emulator tool capable of recreating large networks in real-time is presented. The proposed tool, Overall Trend Replicating Network Emulator Tool (OTRENET), intercepts and alters incoming real packet data streams based on modeled end-to-end packet dynamics. OTRENET architecture is presented in Section 4.2. In addition, its functionalities are described, and compared to the limitations of previous related studies. In CHAPTER 4 an alternative for packet-by-packet capture and translation emulation approach is introduced. This algorithm is named Average Traffic

Sampler by Time Frame Segmentation (ASAF), and it is designed to be embedded into the proposed OTRENET emulator tool.

In CHAPTER 5 the Average Traffic Sampler by Time Frame Segmentation Algorithm (ASAF) introduced in CHAPTER 4 is explained in detail. ASAF is focused on minimizing the overhead delay caused by packet-by-packet capture and translation approaches done by previous network emulator modules. ASAF algorithm is mathematically described in detail in CHAPTER 5. A performance comparison analysis of ASAF against other methodologies for traffic sampling and trend detection is presented. This analysis consists of determining its ability to generate estimated sampled traffic that resembles the original traffic, and its ability to reduce the number of time frames generated on the estimated traffic. In CHAPTER 6 the performance of the current stage of the proposed OTRENET module, as described in CHAPTER 4 and CHAPTER 5, is analyzed. CHAPTER 6 describes in detail the experiment setup, performance metrics to be monitored and evaluated, and the emulation outcomes. Performance analysis of OTRENET on replicating realistic conditions imposed by simulated environments is tested in different manners in Section 6.1.

In CHAPTER 7 approaches for packet delay modeling and characterization are discussed. Using measurements performed over the Internet, end-to-end packet delay dynamics are modeled using time series techniques under weakly-stationary network conditions. The impact of sending rate and packet size of probes is investigated on the modeled results. Impact of Auto Correlation

Function (ACF) and Partial Autocorrelation Function (PACF) distributions on the packet delay modeling is also studied. CHAPTER 7 is complemented by testing and comparing the goodness of the fitted packet delay and IPG time series models under varied sending conditions.

CHAPTER 8 extends the research presented in CHAPTER 7 to network systems under non-stationary conditions. The impact of non-stationarity when modeling network system dynamics is analyzed in CHAPTER 8. In addition, CHAPTER 8 proposes a computationally efficient methodology for online segmentation and modeling of packet delay series based on adaptive AR model, Kalman Filtering algorithm, and a modified version of the Divergence-Test. This method is based on the non-stationarity of the packet delay observations. Experimental results demonstrate the potential online packet delay classification capability of the proposed algorithm based on the piecewise-stationary of the observations, while keeping computational and storage requirements low. False sense of the Long Range of Dependency on packet delay is also studied in the context of the proposed algorithm, and the importance of identifying it when modeling packet delay processes is highlighted in CHAPTER 8. CHAPTER 9 concludes the dissertation.

CHAPTER 2. OVERVIEW OF NETWORK EMULATOR TOOLS

This chapter provides an overview of available network emulator tools. Well known and novel approaches for implementing network emulation are described. NISTNET and NSE emulators are described in detail, and their benefits and limitations are remarked addressed.

In Section 2.1 previous studies related to network emulation are presented and described. Advantages and limitations of prior approaches are considered.

NISTNET and NSE are selected for description in Sections 2.2 and 2.3, respectively due to the fact that they are directly related to our network emulation analysis. NISTNET architecture is explained in detail in Section 2.2.1. It's functionalities will be borrowed for the development of a large scale IP network emulator tool, which is presented in CHAPTER 4.

In section 2.4, the most relevant advantages and limitations of NISTNET and NSE are listed and discussed. Alternative approaches that overcome these limitations are discussed in the context of the development of a large scale IP network emulator tool, and presented in CHAPTER 5, CHAPTER 7, and CHAPTER 8.

2.1 PRIOR APPROACHES FOR IMPLEMENTING NETWORK EMULATORS

In recent past, several network emulators have emerged trying to come up with an accurate, inexpensive network emulator tool [1]. For instance, ENDE: An End-to-end Network Delay Emulator Tool for Multimedia Protocol Development [71] and the Ohio Network Emulator (ONE) [2] were designed focused to imitate queuing, propagation and end-to-end delays for local and wide networks.

ENDE was designed to emulate end-to-end delays between two hosts machines in a single machine. ENDE enables the user to test new multimedia protocols realistically. ENDE has two modes; the delay-observing mode, and the delay-impacting mode. In the former mode, ENDE can generate accurate traces of one-way delays between two hosts on the network. In the later mode, ENDE can be used to simulate the functioning of a protocol or an application as if it were running on the network.

Additionally, ONE enables researchers the emulation of a network between a pair of interfaces on a single Solaris-based workstation. Various features of wired networks, such as propagation delay, queueing characteristics and bandwidth can be controllable by the user. In addition, tool offers the capability of emulating variable propagation delays on satellite networks, based on the orbits of satellites.

In [40] an emulator for performance evaluation of TCP/IP applications over a mixed network of wired and wireless elements is presented. Emulator presented

in [40] is a Linux based framework, which separates the simulated topology into two parts: wired network simulation and wireless network emulation. Complex topology of wired network is simulated in a Linux processor. Network traffic is then redirected from the simulated host and connected with the real wireless network.

RAMON (Rapid-Mobility Network Emulator) is presented in [30]. RAMON is tailored to mimic the realistic characteristics of wireless networks. The main focus of RAMON is studying how mobile protocols handle high vehicular speed.

Dedicated boxes for IP network emulators can also be found. For instance, PacketStorm communications [75] provides dynamic IP network emulators, and other bandwidth emulator products. Such products are a combination of dedicated hardware and software in a single box, which are focused on replicating the unfavorable conditions of IP networks and WANs in a controllable and repeatable lab setting.

Well-known network simulators have also been extended to include emulator modules. As an example, OPNET Technologies has presented the system-in-the-loop (SITL) module[84]. SITL provides a simple plug and play interface that connects live applications or network devices to OPNET discrete event simulations. SITL extends OPNET's simulation technology to support application and network device testing for equipment manufacturers. OPNET consist of several models Internet protocols and architectures, which can be

modeled using discrete event simulation, flow analysis, or in a hybrid manner (combination of the two previous modes).

In addition the Berkeley Network Simulator (NS) [26] has also been extended to include an emulator module (NSE) [26] over the well known NS simulator. NS is a discrete-event simulator used widely in the networking research community [66] [67], it includes several models of common and novel Internet protocols and architectures. NSE captures incoming real world packet streams, convert them into simulator compatible packets, and then inject them into NS simulator. NSE has the support of NS simulator for accurate network/traffic metrics generation, and its scalability can be improved with Parallel Discrete Event Simulation (PDES) techniques [28]. However, NSE is planned to be a real-time emulator, and thus all computational tasks involved in the emulation process need to be accomplished in real-time. Conversely, such a condition may fail in case of high loads and complex systems, in which computation delays related to the emulation process can rise considerably. As an alternative, EMPOWER [73] retains the packet-by-packet emulation process but attempts to reduce computation overhead by creating an IP network emulator cluster of workstations in a local-area network. Thus, computation overhead is just distributed among workstations, as opposed to being centralized in one system.

On the other hand, the NISTNET emulator [19], developed by the National Institute of Standards and Technology [74] alleviates the problem of real-time

execution. This system allows an inexpensive standard PC-based router to imitate numerous complex performance scenarios, such as tunable packet delay distributions, congestion and background loss, without incurring unnecessary time overheads. NISTNET neither captures nor transforms incoming streams. It just buffers or drops incoming packets depending on parameters specified by the user and the random nature of the process. However, these parameters have to be pre-specified, and there is no in-built support for real-time emulation of dynamic network conditions. While NISTNET has many advantages, it is focused more to mimic the behavior of a router than a computer network. In the following subsections a complete explanation of NISTNET and NSE is presented. NISTNET and NSE are selected due to their relation to our network emulation analysis, NISTNET architecture is explained in detail, since its functionalities will be borrowed for the development of a large scale IP network emulator tool, which is presented in CHAPTER 4.

2.2 NISTNET

NISTNET is a simple Linux PC based network emulator that works as a “network-in-a-box”, NISTNET provides capabilities of applying network characteristics to an IP packet stream going to or through the Linux PC it runs on. Network characteristics such as packet loss, delay, duplication, congestion, bandwidth limitation and reordering are included in NISTNET. NISTNET requires as its input the network effects to be applied and a flow specification to identify the IP datagrams. This flow is the target for applying those specific effects. The basic form of a flow specification is the source IP address and the

destination IP address of the IP packets. However, other IP header fields such as higher-level protocol (such as UDP, TCP, ICMP, IGMP etc.) and type of service (ToS) can be provided for more refined packet selection. Some higher level protocol header fields such as source and destination port numbers (for UDP and TCP), type or code field (for ICMP or IGMP) and multicast group (for IGMP) can also be provided to get even more explicit flow specification [19].

NISTNET allows the user to input flow specifications, either at the beginning of the emulation or during the emulation. In general, network dynamic can be reproduced in real-time when using NISTNET if accurate up-to-date flow information is available.

2.2.1 NISTNET ARCHITECTURE

Inside a Linux kernel, an IP packet is stored and moved in the form of a socket buffer (SKB). A socket buffer is designed such that a packet can be queued and transferred easily using pointers or references without copying the contents of the packet repetitively. A socket buffer is neither same as nor related to send and receive buffers of a BSD socket. An instance of C structure called `SK_BUFF` is associated with each socket buffer. This structure holds the information about the socket buffer such as time and device it arrived at, length, checksum, priority, etc [19]. The structure also has a field called the packet type of the socket buffer, which has a value `ETH_P_IP` for IP packets. The packet type determines the packet handler (in the IP stack) that will process the socket

buffer. Linux kernel provides a number of kernel level functions to manage the socket buffers [19].

NISTNET is basically a Linux Loadable Kernel Module (LKM). It utilizes hooks provided in the Linux kernel to modify the procedure of processing a socket buffers in the networking stack. NISTNET takes control of the IP packet type (ETH_P_IP) handler to intercept the socket buffer of an incoming packet. This forces the network device driver to forward the socket buffer to NISTNET rather than to the kernel's own packet handler. After NISTNET is finished with identifying an IP datagram using the flow specifications and applying corresponding network effects (if any) to it, the socket buffer is passed on to the kernel's packet handler for actual IP stack processing. For timing computations required when applying real time delays, NISTNET modifies the frequency of the real time clock (RTC) and uses it as the source of time. NISTNET makes use of the highest frequency that RTC can provide (8192 MHz) to obtain a good tick granularity (122 us). This allows for computing delays with sufficient accuracy for most practical applications [78]. NISTNET replaces the interrupt service handler for RTC device by a simple interrupt handler, ISR. All the RTC related operations are controlled by NISTNET fastRTC module.

2.2.2 APPLYING NISTNET RULES TO INCOMING TRAFFIC

Packet loss rate can be provided as a simple percentage loss or as DRD (Derivative Random Drop) parameters (drdmax and drdmin). If NISTNET decides to drop a packet using these parameters, it simply frees up the socket

buffer as if it never arrived at that host. The kernel IP stack never sees the socket buffer after this. Duplication rate can be supplied as the probability of duplicating a packet with an optional correlation factor. If a packet is to be duplicated, NISTNET clones the socket buffer and sends both the original packet and the cloned packet, back to back to give an effect of duplication. In real streams, however, if duplication occurs, the two copies, original and duplicate, may arrive with some gap (or other packets) between them. However, NISTNET does not implement it that way for simplicity of operation.

Delay specifications can be provided as mean delay with optional parameters of standard deviation and delay correlation. After the delay for a packet is decided, an entry is added to the delayed packets linked list. This entry holds the socket buffer and an expiration time. As described earlier, NISTNET replaces the ISR for real time clock, i.e., RTC. At every interrupt generated by RTC, the handler checks if the delay of any of the packets in the delayed-packets linked list has expired. If it finds one, it takes the socket buffer out of the linked list and calls the kernel's IP level code for further processing of the packet. This is achieved by calling kernel's packet handler for packet type ETH_P_IP. This is the same kernel function that is called by the network device driver if NISTNET module is not present.

2.3 NSE

NSE (NS Emulator) is a logical extension of NS, which aims to provide an interaction between simulated and real-world components. NS is a discrete-event

simulator used widely in the networking research community [66] [67]. It includes several modules of common and novel Internet protocols and architectures. NSE incorporates special objects within the NS simulator to make it capable of introducing live traffic into the simulator and injecting traffic from the simulator into live networks. Injecting real word traffic in real-time into the simulator is done by first capturing the incoming packets through the Berkeley Packet Filter (BPF), and then converting them into simulator compatible packets.

When using NSE, a special version of the system scheduler, Real-Time Scheduler, is used. This scheduler uses the same underlying structure as the standard calendar-queue based scheduler, but ties the execution of events to real-time. Calendar-queue based scheduler uses a data structure analogous to a one-year desk calendar, in which events on the same month/day of multiple years can be recorded in one day [18].

The interface between the simulator and live network is provided by a collection of objects including tap agents and network objects. Tap agents embed live network data into simulated packets and vice-versa. Network objects are installed in tap agents and provide an entry point for the sending and receipt of live data. Figure 2.1 illustrates how these objects are used on NSE [26].

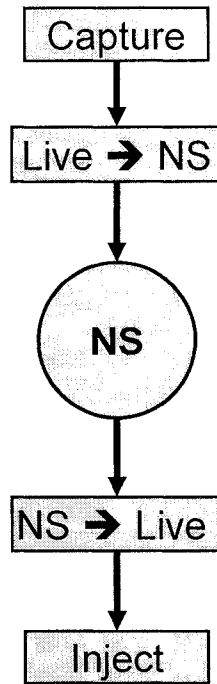


Figure 2.1. NSE objects infrastructure.

2.2.3 REAL-TIME SCHEDULER

The real-time scheduler implements a soft real-time scheduler, which ties the event execution within the simulator to real time [26]. Sufficient CPU horsepower is needed to keep up with arriving packets, thus the simulator virtual time should closely track real-time. If the simulator becomes too slow to keep up with elapsing real time, a warning is continually produced.

The real-time scheduler should always be used with an emulation facility. Failure to do so may easily result in the simulator running faster than real-time. In such cases, traffic passing through the simulated network will not be affected by the proper characteristics specified by the simulator at the right time.

2.2.4 TAP AGENT

The class TAP AGENT is a simple class derived from the base agent class. As such, it is able to generate simulator packets containing arbitrarily assigned values within the NS common header. TAP AGENT handles the setting of the common header packet size field and the type field. It uses the packet type PT_LIVE for packets injected into the simulator. Each tap agent can have at most one associated network object, although more than one tap agent may be instantiated on a single simulator node. Tap agents are able to send and receive packets to/from an associated network object.

2.2.5 NETWORK OBJECTS

Network objects provide access to a live network (or to a trace file of captured network packets). In general, network objects provide an entry point into the live network at a particular protocol layer (e.g. link, raw IP, UDP, etc.) and with a particular access mode (read-only, write-only, or read-write). Some network objects provide specialized facilities such as filtering or promiscuous access (i.e., the pcap/bpf network object) or group membership (i.e. UDP/IP multicast). The C++ class Network is provided as a base class from which specific network objects are derived. Three network objects are currently supported: pcap/bpf, raw IP, and UDP/IP [26].

2.4 ADVANTAGES AND LIMITATIONS OF NISTNET AND NSE

Although NSE is still under construction some limitations can already been identified and foreseen, among them the computational overhead generated by

the packet-by-packet emulation process is the most relevant. NSE captures incoming real-world packets, which will then be converted into simulator compatible packets. After these packets are altered within NS simulator engine, based on the modeled network system, these simulator packets are converted back into real world packets and sent back to the live network. This procedure is done in a packet-by-packet manner and must be accomplished in real-time, thereby computation overhead delays related to the emulation process can rise considerably for high loads and complex systems, which can compromise the real-time emulation process promises by NSE.

Conversely, NISTNET overcomes the limitation of packet-by-packet capturing, simulation and transformation. NISTNET does not capture nor transform incoming streams. It just buffers or drops incoming packets depending on parameters specified by the user and the random nature of the process. However, these parameters have to be pre-specified, and there is no in-built support for real-time emulation of dynamic network conditions. While NISTNET has many advantages, it is focused more to mimic the behavior of a router than a computer network. When a computer network is emulated using NISTNET, dynamics of the network need to be injected into the emulator module. However, such dynamics need to be generated in real time using external models of the network.

CHAPTER 3. REVIEW OF END-TO-END PACKET DELAY

This research focuses on the study of end-to-end packet delay and its application on network system modeling. This chapter is supplemented by the theoretical foundation, metrics and techniques for network modeling based on end-to-end packet delay.

In Section 3.1, an introduction to packet delay is provided. Importance of end-to-end packet delay and jitter on the application performance is also discussed in this section, in addition with techniques on how to mitigate their impact on the quality of experience. Packet delay components are presented in Section 3.2. In Section 3.3 multiple alternatives for packet delay modeling are presented and critiqued.

Section 3.4 presents the challenges associated with measuring packet delay parameter, mainly clock synchronization and clock skew issues are presented, along with proposed methods to counter these effects.

In Section 3.5 packet delay variation metric is presented as an alternative for packet delay measuring. Packet delay variation is defined as the difference between the delays of two IP packets. This parameter has a great influence on the behavior of streaming applications and real-time protocols. Section 3.5 presents two different formulations for packet delay variation widely used in the context of active measurements.

3.1 PACKET DELAY

End-to-end packet delay refers to the time taken by a packet to transit through the network from source to destination. End-to-end packet delay has been extensively used in the past to study several areas of network application performance and development. Several congestion control algorithms [32], routing protocols [47], and real-time applications [71], are designed based on one-way delay and jitter characteristics. For instance, it is well known that increase on packet delay can affect significant the TCP protocol behavior [32]. TCP relies on synchronized acknowledgments to detect congestion in the network and avoid further congestion by reducing the sending bit rate.

Moreover, monitoring delay characteristics provides a good understanding about the network state and allows Internet Service Providers make engineering traffic decisions based on the expected quality of service of different types of applications. For example, for voice applications G.114 standard of ITU (International Telecommunication Union) recommends 150 milliseconds as the maximum one-way delay for voice traffic, as anything above this value will degrades the quality of the voice [71]. In general, time susceptible applications such as voice and video need to be handle differently than the other types of applications. For these types of applications end-to-end delay and the second order of the delay, also known as the jitter delay, play an important role. Higher jitter indicates more buffering leading to a decrease of the voice or video quality, and also may lead to other undesirable effects such as packet reordering [60] and packet loss [49]. Vendors of routers handle delay and jitter in different ways.

Some routers provide load-balancing options to alleviate network link congestion. On the other hand, jitter buffer management is also widely used for voice and video application, in which packets are buffered in such a way that they can be then played out by the application at a constant rate.

3.2 COMPONENTS OF PACKET DELAY

A packet can be delayed within the network due to determinist or stochastic factors. Deterministic components of packet delay depend on packet delay, link capacity and physical distance, and are independent of the network congestion. Transmission delay and propagation delay are considered deterministic components of packet delay. The former represents the time to transmit an entire packet, from first bit to last bit over a communication link. The later represents the time to propagate a bit through the communication link, which is determined by the time of electromagnetic wave through a physical channel of communication path.

Conversely, stochastic components of packet delay change accordingly to the network congestion and the traffic scheduling policies applied. Queuing delay and processing delay are considered stochastic components of packet delay. The former is represented by the waiting time of a packet in a buffer, either in a switch or in a router. The later, represents the time needed to process a packet at each network element. In general, stochastic components of packet delay vary not only based on the traffic intensity but also based on the presence of packets from other streams, also known as cross-traffic.

3.3 PACKET DELAY MODELING

Capturing accurately the end-to-end packet delay characteristics is crucial for understanding the dynamics of the end-to-end delivery process and for the development of accurate models. Such models can also have a great impact on operation and management of networks. For instance, with both real-time [9] and non-real time congestion control mechanisms [15][32], delay based alternatives to packet-loss based approaches aim at preventing network congestion at early stages.

Furthermore, end-to-end packet delay models can be applied to estimate the behavior of live streams under particular network conditions. A direct application for accurate delay models can be found on network emulators [19][68], in which incoming real packet data streams are altered based on traces capturing end-to-end behavior or analytical/simulation models representing such behavior. Such emulators are useful for investigating the performance and behavior of distributed applications and application software under different network conditions. In general end-to-end packet delay and inter-packet gap (IPG) models characterize the effects induced by cross traffic and the probe's sending rate conditions [59] [69].

Multiple studies have been performed, mostly using the measurements over the Internet, to explain the end-to-end packet delay dynamics. We have selected few research works that have drawn varied conclusions according to the analysis conducted.

Several alternatives have been presented for accurate representation of packet delay, among them queuing theory has been used as a powerful tool for modeling packet delay. However, such an approach requires knowledge of the inter-arrival and inter-departure traffic distribution at every single link, which requires tremendous computational demands and thus not feasible for large scale networks [70]. The use of actual delay traces has also been proposed for this matter. Although this approach capture entirely the traffic and network dynamics in a granular manner, its main limitation relies on the fact that large amount of data need to be collected and stored, depending on the duration of the observations. Black box models have also been proposed for modeling packet delay. Such approaches are based on the analysis of the system output observations, and rarely the system input information. Among them, time series is one the most popular ones used by the research community. Time series is a collection of observations made sequentially in time [17]. Packet delay traces, when collected accurately, are considered typical time series data. Modeling techniques based on time series captures the dynamics of the process by analyzing the collected samples. After process is modeled, samples can be discarded. Several modeling research based on time series have been conducted on the past. In [10], predictive models for video packet delay using autoregressive models are presented. In [53] a variable bit rate (VBR) flow of probes and an ARX model (Auto-Regressive eXogenous) are applied for modeling the end-to-end delay, in which the packet delay process is captured based not only

on the observed end-to-end packet delays, but also on the probe's inter-departure packet gap distribution.

A measurement-based tool for traffic modeling and queuing analysis is developed in [41], which uses CMPP (Circulant Modulated Poisson Process) for a traffic model. A comparative analysis of network traffic prediction based on both ARMA and MMPP (Markov-Modulated Poisson Process) models is presented in [61].

End-to-end packet delay over the Internet has also been modeled using system identification techniques[53]. In [53] end-to-end is modeled as SISO (Single-Input and Single-Output) system. However, due to the varying network conditions, a SIMO (Single-Input and Multiple-Output) system is recommended.

Mathematically, delay has been modeled using different distributions in previous studies. Exponential, Weibull, and Pareto distributions, as well as, time varying exponentials are some of the distributions used for this matter. In [14] delay distribution is modeled using Gamma-like distributions with heavy tails of sub-exponential. Although, analysis is promising, network condition information is missing, and results seem to be generalized from a set of measurements. In [31] packet delay is modeled as time varying exponential. In this research one-way delay is modeled by the composition of states, each of them modeled as a shifted exponential distribution with varying parameters. In [59] packet delay on a data stream was fitted into several distributions for different probe's sending rate and packet size conditions. Measurements indicate that the delay distribution

follow a spectrum of distributions ranging from gamma to beta. It is also pointed in this study the existence of correlation among delay values of consecutives packets. Such correlation tends to get stronger for high sending rates than for low ones.

In [59] was also shown that the distribution of end-to-end packet delay by itself does not always result in a complete model, due to the fact that the correlation of packets belonging to the same stream is not considered. In [69], a system approach is used to characterize the network system dynamics by means of modeling end-to-end packet delay. Findings of this study reveal that the behavior of end-to-end packet delay and IPG sequences can be captured effectively by ARMA and ARIMA models, when CBR probe flows are used. Effects of sending bit rate, packet size, and available link capacity are analyzed on the context of system modeling. Results indicate that modeling system dynamics using IPG traces yields to better goodness-of-fit than packet delay, for the same probe stream bit rate, as the combination of data rates and packet size increases. Packet auto correlation have been study in both [59] [69], as a function of probe's sending rate and packet size.

The impact of packet autocorrelation on traffic modeling has also been previously study. For instance, in [41], time series approaches are used to study the impact of packet autocorrelation on the queue response. In [6][39] methodologies for modeling autocorrelation functions for Long-Range-Dependent (LRD) and Short-Range-Dependent (SRD) traffic are presented.

Tools for replicating actual network conditions in controllable environments, such as network emulators [17][68], are impaired by the necessity of capturing and injecting packet autocorrelation from measurements into the emulated network traffic. This is crucial to reproduce the actual observed network conditions for experimental purposes. For instance, NISTNET [19], uses correlation coefficients, to a limited extent, to generate delay values for data streams.

Packet delay, among many Internet traffic metrics, is considered to be a non-stationary process by nature, even under light congested link scenarios [35]. A process is considered to be non-stationary when its statistical properties change in time. Although network link congestion is considered as the main reason for non-stationarity and LRD on packet delay observations [6], it has previously been demonstrated that other network conditions, such as link failure, routing table updates, and routing flapping [35], can also be responsible for this phenomenon. Thus, in practice it is common to observe patterns of periodic spikes, bursty behavior, and level shifting in packet delay traces.

One of the most important factors that LRD introduces into time series is non-stationarity [17][34] [35]. However random spikes and irregular events on the network system can indeed create a false sense of LRD on the observations.

When modeling non-stationary time series, traditional time series methodologies rely on transforming them into stationary ones by means of differencing techniques [17]. However such an approach may fail to distinguish

uneven events responsible of creating false sense of LRD on the packet traffic, since it only captures the overall behavior of the system during the observation period. Consequently, segmenting the observation's trace into groups of stationary time series has been proposed as an alternative solution. Time series segmentation is considered a useful approach for quantifying a non-stationary packet delay series, since it represents the observed trace as a number of time series that are themselves stationary [26][56].

Time series modeling techniques is used on this research to model network system dynamics by means of packet delay observations. CHAPTER 7 presents methodology, results, and remarks for system modeling based on packet delay observations under weakly-stationary network conditions. CHAPTER 8 presents a novel approach for similar analysis of system modeling based on packet delay observations under non-stationary network conditions. The approach presented in CHAPTER 8 relies on the segmentation of packet delay traces. Experiment results indicate that the segmented stationary packet delay series yields to a better understanding of the network system dynamics and lead to more accurate modeling and prediction analysis.

3.4 CHALLENGES ON MEASURING PACKET DELAY OVER THE INTERNET

Provide an unbiased and quantitative measure of packet delay is a crucial task on research and network system monitoring. However, measuring packet delay comes with some challenges. Challenges on measuring packet delay

Clock synchronization between sender and receiver is critical for reliable packet delay measurement. If the timing of arrival or transmission is off between the sender and the receiver, then packet delay information will be distorted. Network Time Protocol (NTP), has mostly been used to synchronize clocks of systems to high precision atomic clocks located in different parts of the world.

Conversely, clock skew is also considered an essential reason of uncertainties when measuring packet delay. Clock skew refers to the phenomena in which two clocks involved in the measurement, sender and receiver for instance, run at different frequencies. Numerous techniques for reducing the effect of clock skew issues can be found on the literature [57]. However, such a phenomenon can not be totally eliminated. Clock skew is usually in the range of microseconds in an interval of few seconds. Thus, if the measurements are performed over networks within few seconds and the delays are in milliseconds range, most of these errors can be neglected.

3.5 PACKET DELAY VARIATION

Contrary to packet delay, packet delay variation metric does not present the mentioned challenges when measured over the Internet. In general packet delay variation metrics are derived metrics, thus their definition rely on another metric [23]. The fundamental of this metric is the one-way delay, variations are computed by taking the difference between two individual one-way delay measurements. This intrinsic property of the packet delay variation metrics, makes them unsusceptible to the challenges packet delay measurements are

exposed, which were mentioned in Section 3.4, as notion of packet sending time is not needed.

In general, packet delay variation metrics have a great impact on the behavior of streaming applications and real-time protocols. Typical cases for such applications are voice-over-IP, video applications like video conferencing, internet radio or other multimedia applications [72]. In such cases, the use of a buffer to smooth out the delay variations encountered on the path from source to destination is needed. Buffer needs to be dimensioned in such a way to accommodate most of the expected variation, otherwise packet loss will result. However, if buffer is too large, large delays will be experienced on the communication and thus conversational dynamics will be affected [72]. In addition, Internet Service Providers usually monitor these metrics and compare them against numerical objective from Service Level Agreement, to ensure quality of real-time applications.

There are many ways to formulate delay variation metrics for packet networks. However, two main formulations are preferred [23], the Inter-Packet Delay Variation (IPDV), and the Packet Delay Variation (PDV). Both formulations are explained and compared in detail below.

3.5.1 IPDV: INTER-PACKET DELAY VARIATION

IPDV is the abbreviation for IP packet delay variation, also known as jitter delay. IPDV is defined as the difference between the delays of two consecutive

IP packets [23][51]. The reference packet in the pair is always the previous packet in the sending sequence. IPDV is defined below;

$$IPDV(i) = D(i) - D(i - 1) \quad (3.1)$$

where $D(i)$ and $D(i - 1)$ are the end-to-end delay of two consecutive packets, i and $i - 1$, respectively. In general, IPDV can be considered as a measure of the network's ability to preserve the spacing between packets [23].

3.5.2 PDV: PACKET DELAY VARIATION

Packet delay variation, also known as PDV, is defined as the difference of a packet delay observation minus the minimum one-way-packet delay sample within the specified interval of the observations [51]. Using the same nomenclature introduced in Section 3.5.1, PDV is defined below;

$$PDV(i) = D(i) - D(\min) \quad (3.2)$$

where $D(\min)$ is the minimum one-way-packet delay observation within the specified interval of the observations.

IPDV and PDV differ from each other on the reference term. Comparison of these metrics has previously been done [22]. The most significant conclusions reached from such comparison are listed below;

1. Distribution of IPDV is usually symmetrical about the origin, with a zero mean value.
2. IPDV distinguishes quick delay variations, from longer term variations.
3. IPDV places reduced demands on the stability and skew of measurement clocks.
4. PDV does not distinguish quick variation from variation over the complete test interval.
5. Location of PDV distribution is very sensitive to the reference delay, minimum packet within the specified interval of the observations.
6. Shape of the PDV distribution is identical to the delay distribution, but shifted by the reference delay.

CHAPTER 4. DESIGN OF A LARGE SCALE IP NETWORK EMULATOR TOOL.

Network research requires accurate environments for protocols and services analysis and performance evaluation. In this chapter the design of a scalable emulator tool capable of recreating large networks in real-time is presented. The proposed tool, Overall Trend Replicating Network Emulator Tool (OTRENET), intercepts and alters incoming real packet data streams based on modeled end-to-end packet dynamics. The current version of OTRENET, presented in this section, uses a network simulator to provide the modeled end-to-end packet dynamics. However, OTRENET is flexible enough to handle other types of models. CHAPTER 7 and CHAPTER 8 present alternatives means of network modeling, which can be used on the context of network emulation.

In Section 4.1 an overview of network emulation is provided. Classic architecture of previous studies is presented and critiqued. The proposed network emulator tool is also outlined in Section 4.1, aiming at overcome limitations of previous efforts. In Section 4.2 OTRENET is formally introduced. OTRENET architecture is presented in Section 4.2. In addition, its functionalities are described in this chapter, and compared to previous related studies.

In Section 4.3 the units that form OTRENET are described. An explanation on how they work individually and collectively is provided. Synchronization of OTRENET units is explored in 4.3.4.

4.1 OVERVIEW OF NETWORK EMULATION

Precise and efficient tools for network analysis and performance evaluation are critical for testing new distributed applications, protocols and technologies. Network simulators have been widely used for this purpose; while this is a repeatable and manageable experimental approach, it requires the use of simplifying assumptions (i.e., traffic patterns, dropping probability, etc.). These assumptions can obscure understanding of behavior in real world situations, and often conceal the random nature of real systems [55] [56]. Application software developed for emerging complex distributed systems such as Collaborative Adaptive Sensing of the Atmosphere [8] need to be evaluated and tested in a wide variety of network conditions. The approach presented in this research, can be used to evaluate the end-to-end performance of such applications by providing an emulated network environment that directly connects to nodes running the application. Network emulation combines concepts from network simulation and measurements and provides an emulated network testbed over which application and protocol software may be evaluated. Compared to simulation, a network emulation approach could lead faster and more accurate results. This approach is more practical to implement, more versatile and significantly less expensive than a real testbeds [7].

A network emulation system can be conceptualized using three units. The first unit captures real incoming traffic. The second unit uses the captured traffic to inject simulated traffic into a modeled environment. In the third unit, the output of the simulated network modulates the real traffic, captured in the first

unit, prior to its release from the emulator. Prior approaches suggest meeting the terms of the three mentioned units in a packet-by-packet level [26]. Although that yields to accurate emulation, it also may yields to computation overhead when the simulated network is sufficiently large. Several existing network emulators attempt to reduce the computation overhead on the simulation unit by providing environments for end-to-end protocol evaluation as they abstract a network cloud to a simple router with specific packet handling operations [3][71][26]. Therefore topology related protocols such as routing protocols cannot be evaluated.

Our approach receives real traffic stream characteristics of incoming packets and uses them as input for a network model to regulate characteristics of real traffic streams. The proposed approach differs from previous emulators in two aspects. On one hand, it obtains average information of a sample group of incoming data packet, as apposed to packet-by-packet capture. On the other hand, it regulates the actual stream based on simulator/model generated characteristics, since no packet capture is attained. Thus, although the terms of the three mentioned emulator units are met, computation overhead on the first and third unit is greatly reduced.

As an alternative to the packet-by-packet capture and translation approach, OTRENET uses the embedded Average Traffic Sampler by Time Frame Segmentation (ASAF) algorithm to sample incoming traffic and inject parameters corresponding to dynamically determined time-frames into the simulator/model.

By including this algorithm in the system, the proposed emulator module mimics the overall behavior of real network scenarios with significantly less computation/time overhead. Thus, the proposed emulator maintains repeatability and ease of configuration while using real-world interaction to minimize deficiencies of simulated approaches. OTRENET, as its name suggests, was designed to mimic the overall behavior of real network scenarios rather than the instantaneous packet-by-packet responses of the system.

4.2 OTRENET: OVERALL TREND REPLICATING NETWORK EMULATOR TOOL

OTRENET modifies the characteristics of a stream of packets mimicking the effect on it when passing through a given network. The stream enters the emulator via a network interface card (NIC), Eth1, and leaves via a second NIC, Eth2, with the delay, losses, and throughput etc. of the stream changed according to the results from a network model.

OTRENET overcomes the need for packet-by-packet capture and injection of packets to the network model; yet it allows the simulation model to depend on stream traffic, and end-to-end traffic in emulated network. Our approach is to collect the average input stream information from incoming real-time traffic flow as an alternative to injecting it directly into the simulator model packet-by-packet, as NSE proposes to do it. The statistical characteristics gathering process is done by employing a dynamic sampling algorithm at the input, sampling real time streams without affecting their behavior.

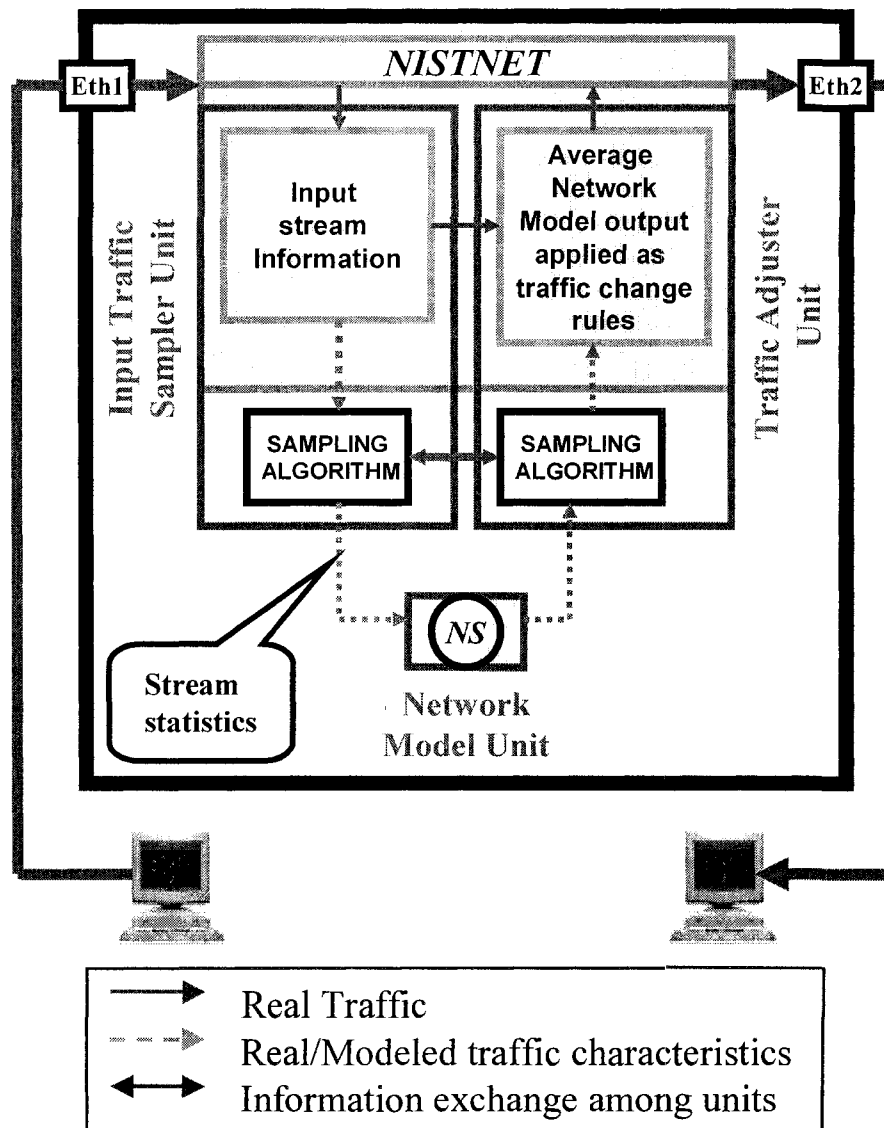


Figure 4.1. OTRENET architecture

This way, OTRENET will not cause unnecessary packet loss at the entry at any given time, even when the module gets overloaded due to excessive traffic. After the average-sampled stream characteristics are obtained from the sampling algorithm, those values are introduced into the network model (NS), see Figure

4.1. In the current version of OTRENET input stream information is collected directly from the NISTNET kernel, as shown in Figure 4.1. This approach has been found to be less computational expensive than collecting it from the Eth1 NIC.

The output of the network model is used to extract characteristics of traffic passing through it, such as burst packet losses, transmission delay and duplication created by the simulated environment set up. These are features that in turn act upon traffic routed through it in order to trigger and emulate the congestion behavior of real networks. Thus output stream features are collected from the network model unit, and these parameters represent the particular behavior of each real-world stream inside the modeled network environment.

Finally, the statistics of these output values are collected from the sampling scene and injected into the traffic adjuster unit (NISTNET), which is responsible for fine-tuning the real-traffic stream that is currently passing through the module as shown in Figure 4.1. In the following subsections the units that form the emulator are described, also an explanation on how they work individually and collectively is provided.

Notice that even though OTRENET currently uses NS and NISNET for network modeling and traffic adjusting, respectively, it is able to accept other tools capable to provide similar functions.

4.3 OTRENET UNITS: DESIGN, IMPLEMENTATION, AND SYNCHRONIZATION

4.3.1 INPUT TRAFFIC SAMPLER UNIT

This unit is in charge of sampling the input traffic coming into the emulator, via the network interface card Eth1. Besides filtering, classifying and collecting the stream information, this unit is also responsible for adjusting the sampling process dynamically depending on the input fluctuations and the availability of the simulator unit. Thus the input stream information is not only used as a feeder to the simulator, but its characteristic distributions are used to decide how often the sampling has to be done. This process is achieved by employing an embedded Average Traffic Sampler by Time Frame Segmentation algorithm (ASAF) within the input traffic sampler unit, as shown in Figure 4.1. CHAPTER 5 explains in detail the algorithm utilized in this unit, the metrics that it requires and the dynamic thresholds selection.

4.3.2 NETWORK MODEL UNIT

This unit consists on a model of the emulated IP network. This could be an analytical model, a trace table, packet-by-packet simulation or even a scaled simulation version. Its selection depends on the performance accuracy and the execution speed needed. A customized version of NS has been chosen as a network model for the current version of OTRENET.

Since the simulated network characteristics (available bandwidth, queue length, network congestion, etc.) constantly change during the time, it is not

possible to turn this component on and off every time new input traffic characteristics are collected and injected. Therefore it is essential to keep the simulator running all the time that the emulator is on. NS has been designed as a discrete event simulator in which the traffic stream characteristics have to be specified before it is executed, and the system output characteristics can only be collected and analyzed at the end of the simulation. Thus NS was modified to support these features.

Two main modifications were done on the NS functionalities. The first one consists on a periodic feeding mechanism of real-traffic characteristics into the simulator. The second one consists on a “on the fly” analysis of each real stream which traverses the simulated network. The latter approach consists of examining periodically the end-to-end real stream behavior such as throughput, end-to-end delay, jitter, and packet-loss rate. End-to-end delay and packet-loss rate were found appropriate to accomplish the input-output mapping based on network simulation.

Note that the architecture lends itself for replacement of NS simulator with other appropriate simulation or analytical models as well.

4.3.3 TRAFFIC ADJUSTER UNIT

This unit is in charge of the traffic adjustment, and triggers NISTNET based on the output responses (end-to-end delay, packet loss rate, etc) of the simulator unit captured through the sampling-algorithm within the traffic adjuster unit, as shown in Figure 4.1.

The same traffic sampling algorithm that is used to capture the characteristics of the input stream is used to capture the characteristics of the simulator output, which in turn is used to control the outgoing stream. The periodicity of the real traffic adjuster triggering depends primarily on the simulated response variations. Also since the input sampler unit collects IP addresses/port number information of the source and destination of each incoming stream, this unit is able to provide sufficient information to the traffic adjuster unit in order to accurately match the simulated stream responses with the real traffic passing through the emulator.

4.3.4 SYNCHRONIZATION AMONG UNITS

Synchronization among the three units is crucial for the emulation processes. Each of its units must work with tight dependence to the others. Furthermore, to accurately recreate the behavior of real word scenario and to take advantage of inherent parallelism while minimizing the total execution time of the whole process, we execute these three units in parallel. Thus each unit has to be executed only at specific times and just the amount of time that it is expected to work.

Time boundaries are calculated during the emulation processing to prevent system inaccuracies or collapses. The scripts used in each unit are linked together by a main script, in which three procedures were created; traffic collector, network simulator/traffic adjuster and system scheduler. These procedures are executed in parallel, as shown in Figure 4.2.

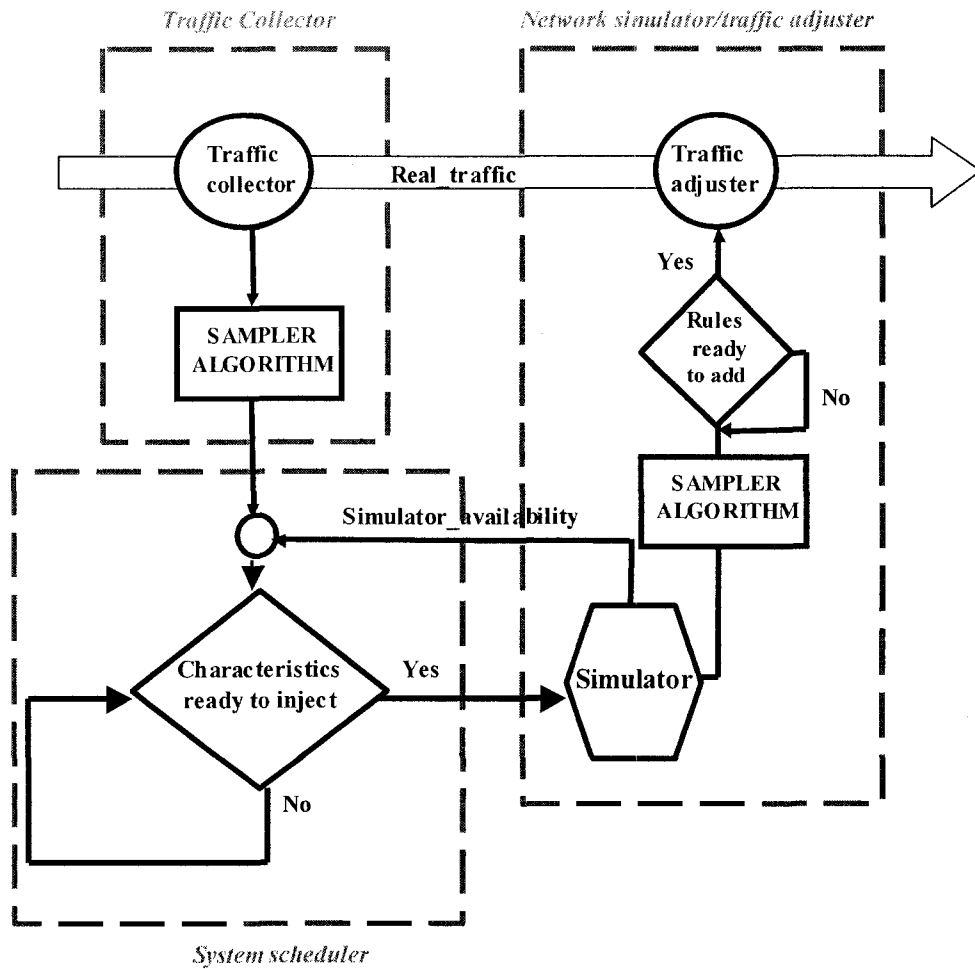


Figure 4.2. Flow diagram of the main OTRENET code and module synchronization.

The first procedure is responsible for collecting the stream characteristics. The second one combines the network model and traffic adjuster emulator units. Finally the third procedure is responsible of synchronizing and interconnecting the previous two threads. So the scheduler procedure will transfer the collected information on incoming packet stream from the first procedure to the second one only when the network model unit is idle (i.e., it is done simulating the

previous transferred information). Also in the case that the network-model/traffic-adjuster process goes faster than the collecting process the scheduler procedure will force the simulator-model/traffic process to wait preventing this way the starvation of other processes.

The script used to execute and synchronize OTRENET units in the manner described above is presented on appendix B.3.

CHAPTER 5. TRAFFIC SAMPLER BY TIME FRAME SEGMENTATION ALGORITHM.

In this chapter the Average Traffic Sampler by Time Frame Segmentation Algorithm (ASAF) introduced in CHAPTER 4 is explained in detail. ASAF is focused on minimize the overhead delay caused by packet-by-packet mapping approach, done by previous network emulator modules.

In Section 5.1, ASAF algorithm is mathematically described in detail. Metrics to be monitored are specified, as well as algorithm settings are fine tuned according to the expected sensitivity. In Section 5.1, ASAF algorithm performance is tested. Experiment was conducted by comparing a simulated end-to-end delay response against the average delay per time frame generated by the proposed algorithm. Results indicate that the average delay per time frame follows closely the variability of the instantaneous delay response, even when it changes severely.

In Section 5.2 several functions are evaluated as alternative candidates for the threshold decay function embedded in the proposed ASAF algorithm.

In Section 5.3 a rigorous performance comparison analysis of the proposed ASAF algorithm against other methodologies for traffic sampling and trend detection is presented. Analysis is done to measure the algorithm ability on

replicating the original sample series, while keeping the number of time frames low, to the extent possible.

5.1 ALGORITHM DESCRIPTION

As was mentioned in CHAPTER 4, the main disadvantage of network emulator modules such as NSE is the packet-by-packet capture and translation approach, in which each incoming real packet is captured, translated into simulated one, and injected into a network model embedded in the emulator module, in altered based on modeled network end-to-end packet dynamics. Although by this approach accurate packet-by-packet emulation is guaranteed, heavy CPU resources are required to accomplish these tasks in real-time. However, if this cannot be achieved, which is likely to be the case with complex networks, additional computation overhead delay is added into the results. In cases where hundreds of thousands of packets are emulated, e.g., with Mbps and Gbps links, this becomes a very significant limitation.

In this section the ASAF algorithm is proposed as an alternative to the packet-by-packet capturing and translating approach. This algorithm is used to report significant changes and not instantaneous fluctuations, for both real entry traffic and output modeled traffic response, as was mentioned in CHAPTER 4. ASAF aims at reducing the computation overhead delay, while keeping track on the consistency of the results.

On the current version of OTRENET the input sampler unit monitors packet size and inter-arrival packet gap, while in traffic adjuster unit the utilized metrics

are end-to-end delay and packet loss rate. In this section we explain in detail only the use of the algorithm as utilized in the traffic adjuster unit. However, its use in the input sampler is very similar.

The main objective of the ASAF algorithm is to detect significant changes in the metrics of interest. After this, a new time frame is created and the short-term averages of the metric during the generated frame are reported. The output responses of the network simulator-model unit may change rapidly within short periods of time during network transients. By analyzing the output trace of the simulator offline, one can actually observe how harshly the per-packet response fluctuates. In such situation a sensitive algorithm that reports changes in metrics too frequently can trigger the traffic adjuster excessively, and in many cases unnecessarily, making it impossible for the outgoing real stream to adjust itself in real time according to the traffic adjuster rules imposed by the algorithm. As a result, incongruity between modeled traffic and emulator output traffic can be expected, see CHAPTER 6 for details. As an alternative, the proposed ASAF algorithm is triggered by the change of the accumulated difference of the metrics. In this case the simulated end-to-end delay and packet loss rate are calculated periodically for each real-time stream within the simulator. These metrics are subtracted from their previous respective values and absolute values of this difference are accumulated during time, $abs_acc_delay_{(t)}$, and $abs_acc_drop_{(t)}$, respectively. The value of $abs_acc_delay_{(t)}$ is given by equation (5.1).

$$abs_acc_delay_{(t)} = \left| delay_{(t)} - delay_{(t-1)} \right| + abs_acc_delay_{(t-1)} \quad (5.1)$$

where $delay_{(t)}$ and $delay_{(t-1)}$ represents the end-to-end delay at times t and $t-1$, respectively. t and $t-1$ represent the present and the previous simulated time, respectively. The absolute value was chosen for this task in order to keep track of the changes regardless whether they are positive (increasing change) or negative (decreasing change). Small fluctuations or slow changes on the metrics will take longer time to trigger the algorithm than drastic fluctuations. Thus when the accumulated difference of the metric exceeds a threshold value ($thr_delay_{(t, frame_t)}$, $thr_drop_{(t, frame_t)}$) a *system-trigger* alarm is activated and a new time frame is created. The outgoing real traffic is then adjusted with the average end-to-end delay and packet loss of the last time frame (see Figure 5.1).

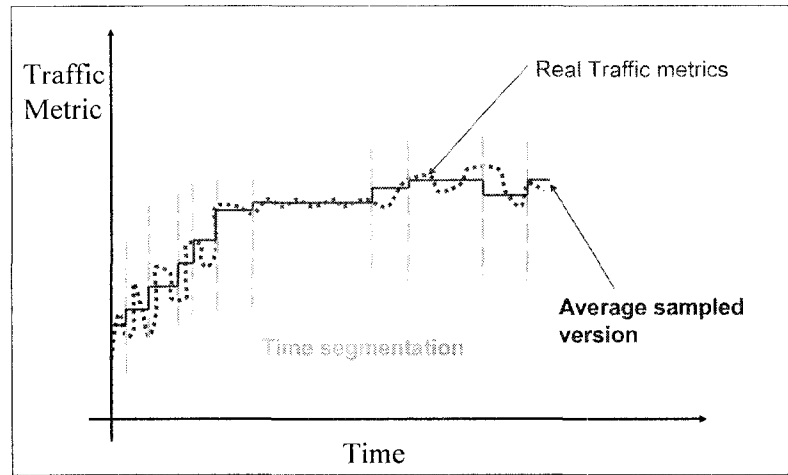


Figure 5.1.a

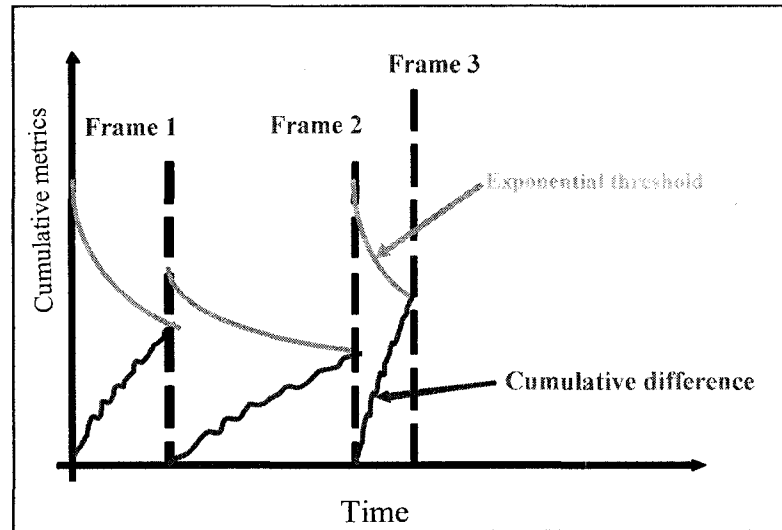


Figure 5.1.b

Figure 5.1. Time frame segmentation according to traffic variation, (a) Traffic metric vs. time, (b) Cumulative metrics and exponential threshold vs. time

Threshold values utilized in the proposed algorithm need to change according to actual time of the analyzed frame. This way, after a change has been detected and a new time frame is about to begin, the threshold value is raised to a

predefined peak value (see Figure 5.1). Then the threshold has to decrease in a smooth controlled decay fashion. The curve chosen for this threshold decay function can follow many shapes such as linear, quadratic, polynomial, and exponential. However, exponential decay threshold function was found to be a more convenient alternative as explained in section 5.2.

Segmentation of the simulated time into frames depends upon the variation of the simulated response. For instance, long frames are associated with traffic that varies slowly. Finally the traffic adjuster unit (NISTNET) receives the average value of the metrics for the current time frame, which represents the average behavior of the traffic within it.

The proposed ASAF algorithm starts calculating the threshold distribution of the analyzed metrics as a function of the simulated time t and frame i . The exponential threshold for end-to-end delay is shown in equation (5.2).

$$thr_delay_{(relative_t, frame_i)} = \left(ampl_thr_delay_{(frame_i)} \right) * e^{-\left(\frac{relative_t}{\tau_delay_{(frame_i)}} \right)} \quad (5.2)$$

where $\tau_delay_{(frame_i)}$ represents the speed decay of the exponential threshold curve of frame i , and $ampl_thr_delay_{(frame_i)}$ the initial peak amplitude of the exponential functions on frame i . Also, $relative_t$ represents the current time on frame i . Note that each frame will start with $relative_t=0$. Moreover the increment of this parameter will cause the thresholds amplitude to decrease in an

exponential manner. Thus when $abs_acc_delay_{(t)}$ and/or $abs_acc_drop_{(t)}$ exceed their respective thresholds, a *system-trigger* alarm will be activated and the frame average for each metric will be inserted into the NISTNET rules.

After counters are reset, new values for $\tau_delay_{(frame_i)}$ and $ampl_thr_delay_{(frame_i)}$ are calculated for the next new frame, using equation (5.4).

$$\begin{cases} \tau_delay_{(frame_i)} = \frac{B}{T_{i-1}} \\ ampl_thr_delay_{(frame_i)} = A * thr_delay_{(T_{i-1}, frame_i-1)} \end{cases} \quad (5.3)$$

where T_{i-1} is the duration of frame $i-1$, (or the largest *relative_t* on frame $i-1$). Thus, $thr_delay_{(T_{i-1}, frame_i-1)}$ is the lowest point of the exponential threshold function on frame $i-1$. From equation (5.4), it can be seen that $ampl_thr_delay_{(frame_i)}$ depends directly on the lowest point that the previous exponential threshold function reached. Also it can be seen that $\tau_delay_{(frame_i)}$, depends inversely on the previous frame duration. Thus a small duration frame (rapid traffic fluctuation) will force next frame to quickly respond to traffic variations, if it appears. A and B have been chosen as constants. Their selection has been done by rule of thumb, such as $1 \leq A \leq 2$ and $B \geq 30$.

$$\begin{cases} \tau_delay(frame_i) = \frac{B}{T_{i-1}} \\ ampl_thr_delay(frame_i) = A * thr_delay(T_{i-1}, frame_i-1) \end{cases} \quad (5.4)$$

The result of this process is an accurate algorithm that will divide the total simulated time into frames of variable sizes. These frames are generated every time a significant change in the metrics is detected. At the same time the average of the metric of interest during frame time, rather than instantaneous change of the metric, is used as input for the NISNET rules.

Sensitivity of the ASAF algorithm can be fine-tuned. With this algorithm OTRENET can trade-off fidelity and computation time.

Figure 5.2 shows the time frame segmentation and the exponential threshold adjustment for a simulated end-to-end delay response case. A and B (see equation (5.4)) have been chosen to be 1.3 and 50, respectively, for this case and for the next scenarios presented in CHAPTER 6. Figure 5.2.b shows how the $abs_acc_delay_{(t)}$ changes with the simulated end-to-end delay responses (see Figure 5.2.a). When the cumulative value exceeds the corresponding exponential threshold value, a new time frame is generated, the exponential threshold is raised to a predefined value, and the cumulative value of delay is reset.

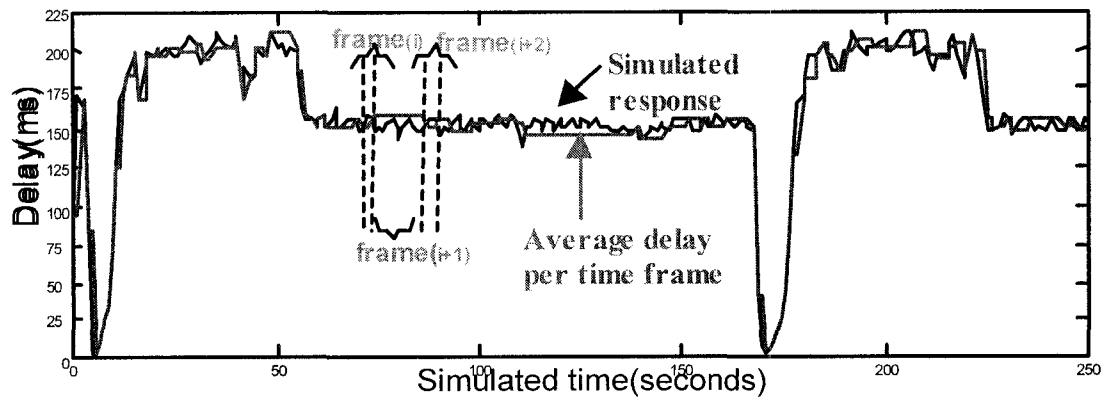


Figure 5.2(a).

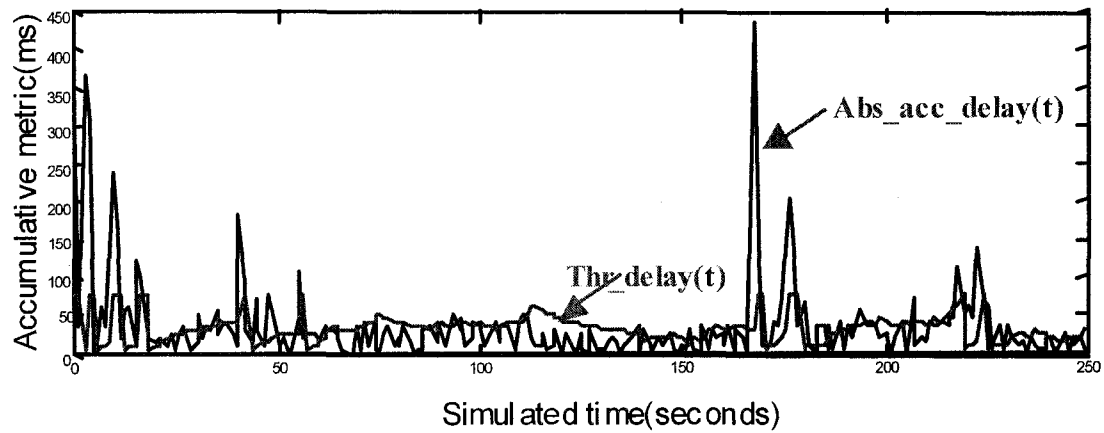


Figure 5.2(b).

Figure 5.2. Time frame segmentation for simulated delay, (a) Instantaneous simulated delay and averaged simulated delay using time frame segmentation vs. time, (b) Variation of simulated cumulative delay and exponential threshold vs. time

Figure 5.2.a shows the time frame generation and compares the simulated end-to-end delay response against the average delay per time frame generated by the proposed algorithm. From this figure, it can be seen that the average delay per time frame follows very well the variability of the simulated delay response, even when it change severely. Note that in the input sampler unit, the packet size and the inter-arrival packet gap are used as the monitored metrics in the same manner as shown in equations (5.1), (5.2) and (5.4). The A and B values selected for this unit were also 1.3 and 50, respectively.

5.2 EVALUATION OF THRESHOLD FUNCTION FOR ASAF ALGORITHM

In the section several functions are evaluated as alternative candidates for the threshold decay function embedded in the proposed ASAF algorithm. The criteria for the function selection rely not only on its ability to rapidly detect changes in the traffic variation but also in its simplicity of implementation. Performance results of the proposed ASAF algorithm were generated using the source code presented on appendix C.1.

Figure 5.3 shows a cumulative metrics and threshold functions versus time, t , in the same manner as was presented in Figure 5.2.b. Three functions are shown; linear, polynomial (2nd order) and exponential. These functions start at an arbitrary point A and decay along t .

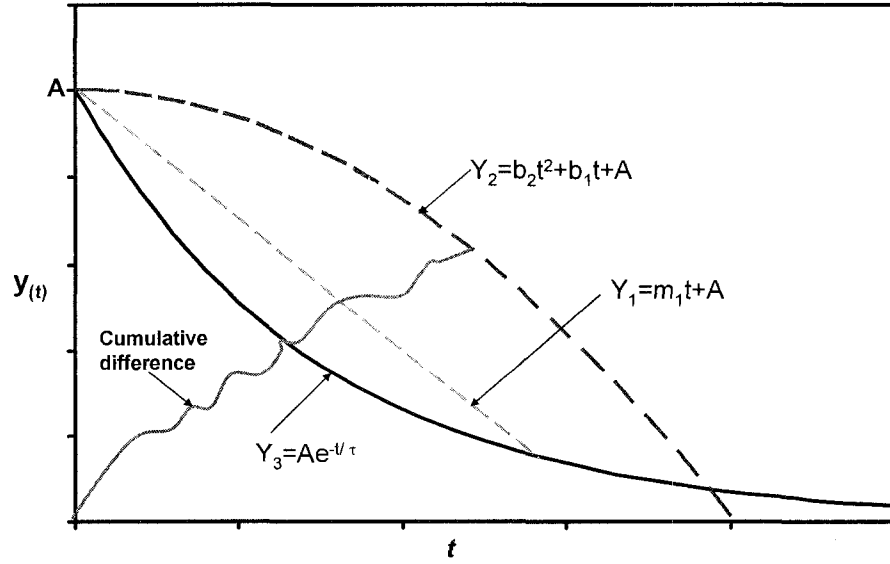


Figure 5.3. Comparison of metric threshold functions.

Their mathematical expressions with their respective parameters are shown in equation (5.5), as y_1 , y_2 and y_3 , respectively.

$$\begin{cases} y_{1(t)} = m_{(t)} * t + A \\ y_{2(t)} = \sum_{i=1}^n \left(b_{(i,t)} * t^i \right) + A \\ y_{3(t)} = A * e^{-\frac{t}{\tau}} \end{cases} \quad (5.5)$$

From Figure 5.3 it can be seen that the decay speed of the selected threshold function plays an important role on reporting traffic variations. I.e., a threshold function that decays too fast could report traffic variations too frequently and sometimes unnecessarily. Even though the parameters of the linear and polynomial functions can be adjusted every time frame to produce longer or shorter tail thresholds that can match traffic variability, this will require prior knowledge of the traffic behavior, which is not possible. Another alternative could be to vary these parameters not only every time frame but also as a function of the simulated time t , as shown in equation (5.5) (i.e., $m_{(t)}$, $b_{(t)}$). Although possible, this approach will add additional complexity to the algorithm.

As an alternative, exponential threshold requires the change of only two parameters for each time frame, and keeps them constant during the frame. The concave and long-tail variation of the exponential function makes it a more suitable threshold function for reporting significant changes on the traffic variability. Thus, the exponential function has been chosen as the most convenient option for this task.

5.3 PERFORMANCE ANALYSIS AND COMPARISON OF TRAFFIC SAMPLER ALGORITHM

In this section the performance of the proposed ASAF algorithm is compared against two methodologies for traffic sampling and trend detection; Moving Average (MA) and a customized version of MA, referred to as C-MA.

As mentioned in CHAPTER 4, the performance of the sampler algorithm does not only depend on its ability of generating estimated sampled traffic that can resemble the original one, but also, on the reduction of the number of time frames generated on the estimated traffic, to the extent possible. It was concluded in Section 5.2 that the length of a time frames is associated with the corresponding traffic variability. The computational requirements of the traffic adjuster unit and the traffic sampler unit can be lessened significantly by reducing the number of times frames. Note that the term time frame used in this section comes from Section 5.2. Invariant portions of the traffic will have similar metrics and will be clustered in the same frame, and each metric will be represented by the average of its samples belonging to that particular frame.

Moving averages, MA, are one of the most popular and easy to use tools available for trend detection. MA smooth data series and make it simpler to spot tendencies by flattening out rapid fluctuations [76]. The two most popular types of moving averages are the Simple Moving Average (SMA) and the Exponential Moving Average (EMA). SMA is explained below, and its performance is compared against the proposed time frame segmentation algorithm. Similar analysis has been done previously with EMA, and can be found on [68].

Given a sequence $\{X_i\}_{i=1}^N$, an MA order n , n -SMA, is defined as a new sequence $\{Y_i\}_{i=1}^{N-n+1}$ defined from the x_i term by taking the average of the previous n terms:

$$y_i = \left(\frac{1}{n}\right) \sum_{j=1}^{i-n+1} x_j \quad (5.6)$$

The number of time frames of the estimated sequence, Y , referred to here as P_y , represents the length of the estimated sequence. Thus $P_y = N - n + 1$. Note that the reduction of sequence length from the original to the estimated sequence, $n - 1$, is only due to the estimated sequence Y starts from the x_{n-1} term. Hence using n -SMA will provide almost no improvement on time frame reduction. On the other hand a customized version of MA, $C\text{-}MA_{(n,m)}$, can address this problem better and it is presented in equation (5.7).

$$\left\{ \begin{array}{l} y_1 = \left(\frac{1}{n}\right) \sum_{j=1}^n x_j; y_1 = y_2 = y_3 = \dots = y_m \\ y_{m+1} = \left(\frac{1}{n}\right) \sum_{j=m-n+2}^{m+1} x_j; y_{m+1} = y_{m+2} = \dots = y_{2m} \\ y_{2m+1} = \left(\frac{1}{n}\right) \sum_{j=2m-n+2}^{2m+1} x_j; y_{2m+1} = y_{2m+2} \dots = y_{3m} \\ \vdots \\ \vdots \\ \vdots \end{array} \right. \quad (5.7)$$

$C\text{-}MA_{(n,m)}$ depends on two parameters, n and m . The former represents the number of previous x_i terms averaged to obtain an estimated term y_i , the latter represents the number of times the estimated term will be kept invariant. Since n and m are chosen off-line without knowing the behavior of original sequence, their selection will play an important role on the $C\text{-}MA$ performance. On one

hand, n and m will assure the accuracy of Y . On the other hand, P_y will be driven by the value of m selected. Note that $C-MA_{(n,0)} = n-SMA$, and also in this scenario n and m are both defined in seconds.

Performance of the proposed ASAF algorithm is compared against those of SMA and C-MA next. Figure 5.4.a shows an end-to-end delay trace obtained through simulation and two estimated traces of this delay, one generated using the proposed traffic sampler algorithm, and the other generated using $C-MA_{(n,m)}$, for n and m are 5 and 7 seconds, respectively. Figure 5.4.b shows a particular region of Figure 5.4.a (from 180 to 350 seconds) for a more granular analysis.

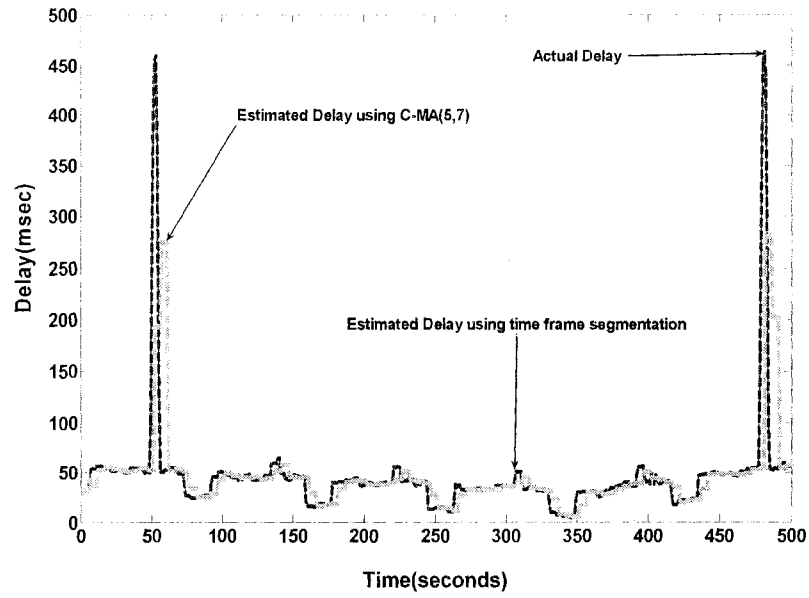


Figure
5.4.a

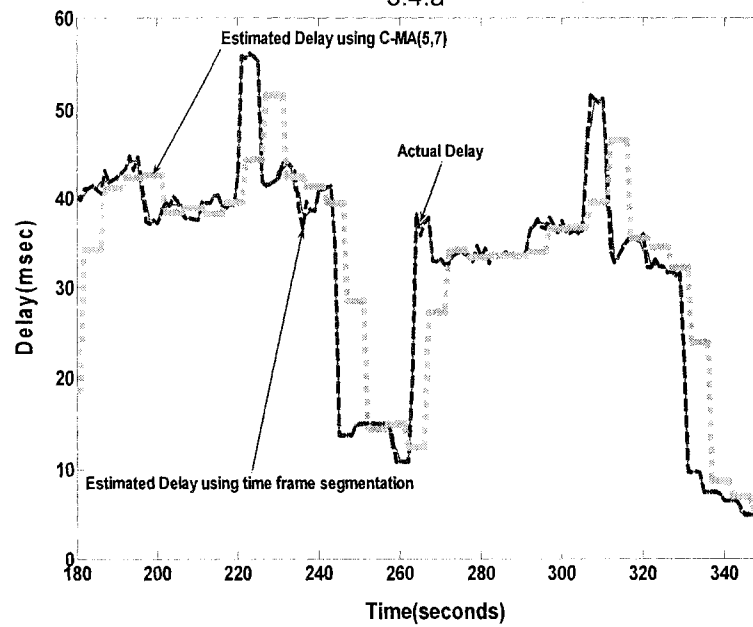


Figure
5.4.b.

Figure 5.4. Comparison of the estimated average traffic sampler by time frame segmentation algorithm against C-MA, (a) Analyzed time 0-500 sec, (b) Analyzed time 180-350 sec.

From these figure, the superiority of the proposed ASAF algorithm over C-MA can be observed. Note that since only the traffic sampler algorithm is tested in this section, the simulated scenario for this analysis is not relevant at this point, thus it is omitted on the description.

The time frame reduction and the algorithms accuracy are analyzed in Figures 5.5.a and 5.5.b, respectively. Figures 5.5.a shows the ratio of P_y obtained using C-MA $_{(n,m)}$ over the one obtained using the proposed traffic sampler algorithm for different values of m . From this figure, it can be seen that C-MA $_{(n,0)}$, provides no improvement on the time frame reduction. On the other hand, C-MA $_{(n,m)}$ with m values greater than 0 deploys smaller P_y than n -SM. Also for this particular scenario C-MA $_{(n,m)}$ with $m>2$ deploys smaller P_y than the proposed time frame segmentation algorithm. Time frame segmentation done in C-MA $_{(n,m)}$ is done independently to the traffic variability, thus the reduction of this metric is a trade-off with the algorithm accuracy, which can be appreciated in Figure 5.5.b.

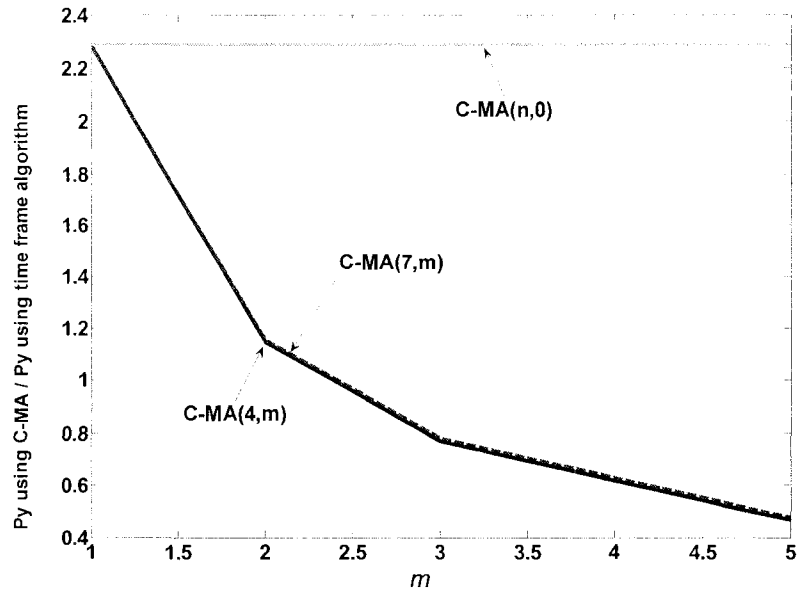


Figure 5.5.a.

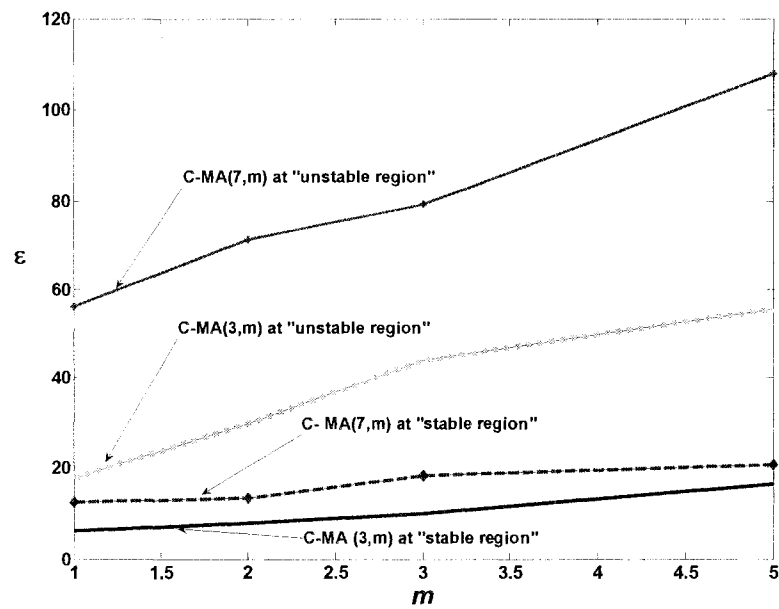


Figure 5.5.b.

Figure 5.5. Comparison analysis of the estimated average traffic sampler by time frame segmentation algorithm and modified MA, (a) Reduction of number of frames comparison analysis, (b) MSE comparison analysis.

Figure 5.5.b shows the accuracy of the algorithms replicating the original sequence. This has been done by calculating the Mean Square Error (MSE) between the original metric and the estimated ones, using equation (5.8).

$$MSE = \left(\frac{X - Y}{X} \right)^2 \quad (5.8)$$

where X is the original sequence and Y corresponds to the predicted response calculated using the scheme. We denote by $MSE(MA)$ and $MSE(ASAF)$ for C- $MA_{(n,m)}$, and the proposed traffic sampler algorithm respectively. Finally the ratio of this parameters, ϵ , is calculated as $\epsilon = \frac{MSE(MA)}{MSE(ASAF)}$ and plotted in Figure 5.5.b. Two regions were analyzed in this figure; a stable region and an un-stable region. The former was selected from 120 to 135 seconds, the latter from 135 to 150 seconds, see Figure 5.4.a. This was done to show the dependency of ϵ with the traffic variability. Figure 5.5.b indicates that MSE for C- $MA_{(n,m)}$ changes according to the n and m values selected and the variability of the sequence. Also from this figure, it can be seen that $MSE(ASAF)$ always shows smaller than $MSE(MA)$. This analysis demonstrates that the performance superiority of the proposed ASAF algorithm over SMA and C- $MA_{(n,m)}$ is due to its ability of dynamically select accurate n and m values according to the traffic variability, i.e., its ability for detecting small and large changes of the average behavior. Thus when using the proposed sampler algorithm, heavy concentration of small

time frames can be found only in areas where the traffic changes rapidly. Note that the intent of the algorithm is not just to reproduce the behavior of the actual signal, but also to do this with the smallest number of frames.

CHAPTER 6. OTRENET PERFORMANCE ANALYSIS.

In this chapter the performance of the proposed OTRENET module, as described in CHAPTER 4 and CHAPTER 5, is analyzed.

Section 6.1 describes in detail the experiment setup, performance metrics to be monitored and evaluated, and the emulation outcomes. Performance analysis of OTRENET on replicating realistic conditions imposed by simulated environments is tested in Section 6.1.

6.1 RESULTS

In this section the performance of the current stage of the proposed OTRENET module, as described in CHAPTER 4 and CHAPTER 5, is tested and evaluated. The network model unit is based on a customized version of NS-simulator, which was described in Section 4.3. ASAF algorithm, which was described and tested in CHAPTER 5, is used to report significant changes and not instantaneous fluctuations, for both real entry traffic and output modeled traffic response, as was explained in CHAPTER 4.

Notice that in this chapter the OTRENET architecture description used in Section 4.3 is used to illustrate the units of the emulator.

In this section the performance of OTRENET is tested and evaluated. The network model unit is based on a customized version of NS-simulator, which was described in Section 4.3. In this section we measure two aspects of the

performance of the emulator system. First, the accuracy of the ASAF algorithm used in the input traffic sampler unit is evaluated. Second, the emulator system response is compared against the response of a pure network simulation (NS) under the same conditions. The first set of performance measurements indicate how an error in input sampling produces inaccuracy in the simulation results, which in turn is reflected in the traffic adjuster and in the actual real output traffic. The second set of performance measurements demonstrate that errors between the emulator response and pure network simulation can be attributed to two factors; error in the input traffic sampler and inaccuracy of the traffic adjuster unit. The emulator environment implementation is shown in Figure 6.1.

The simulated topology within the emulator box consists of a simple network with a 100kbps bottleneck link. For simplicity, no simulated background traffic was utilized. The emulator module was installed on a Pentium III-861Mhz dual processor computer with two Ethernet cards, allowing the traffic from one card (Eth1 NIC) to pass to the other (Eth2 NIC) after being modified by the module. Performance results of the proposed OTRENET module were generated using the source codes presented on appendix B.

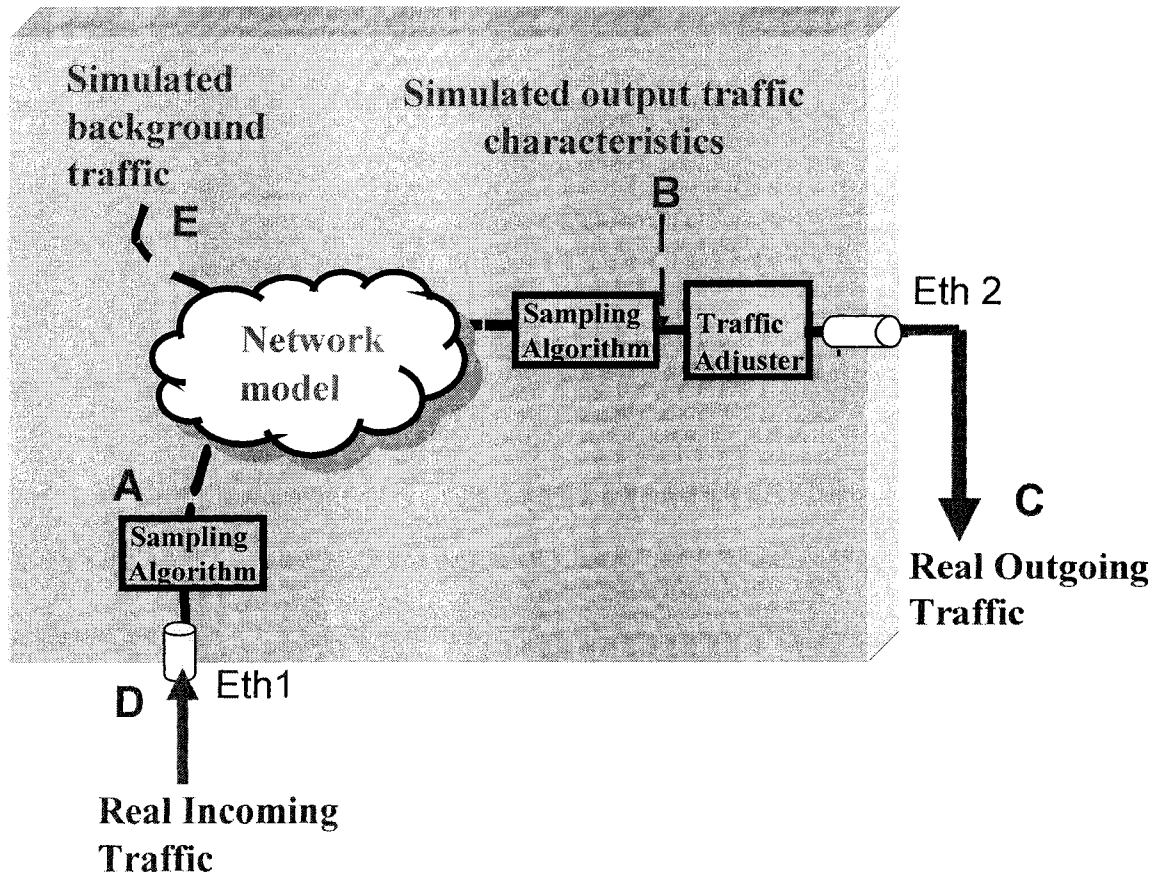


Figure 6.1. Emulation configuration and test environment.

The *real incoming traffic* shown on point D of Figure 6.1 was generated with an IXIA Traffic Generator 1600 Performance Analyzer device [77] and its variation in time is shown in Figure 6.2.

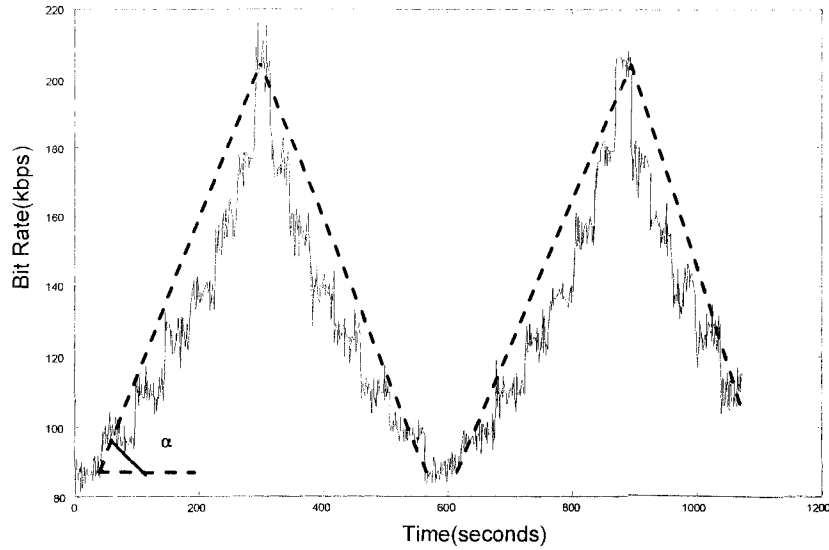


Figure 6.2. Input bit rate variation vs. time.

In this particular experiment the variability of the input traffic, represented as the bit rate slope (α), was varied to cover smooth to steep changes in traffic. After this traffic is analyzed and sampled with proposed ASAF algorithm, the sampled characteristics are inserted into the simulator unit (point A of Figure 6.1). The traffic at this point is named *sampled real traffic characteristics*. The simulated *output traffic characteristics* (point B, Figure 6.1) represent the simulated output response of the sampled version of the incoming traffic injected into the simulator unit. Traffic at point B is inserted as NISTNET rules to regulate the real traffic passing through the emulator box according to the simulated response. The modified real output traffic is named *real outgoing traffic* (point C, Figure 6.1).

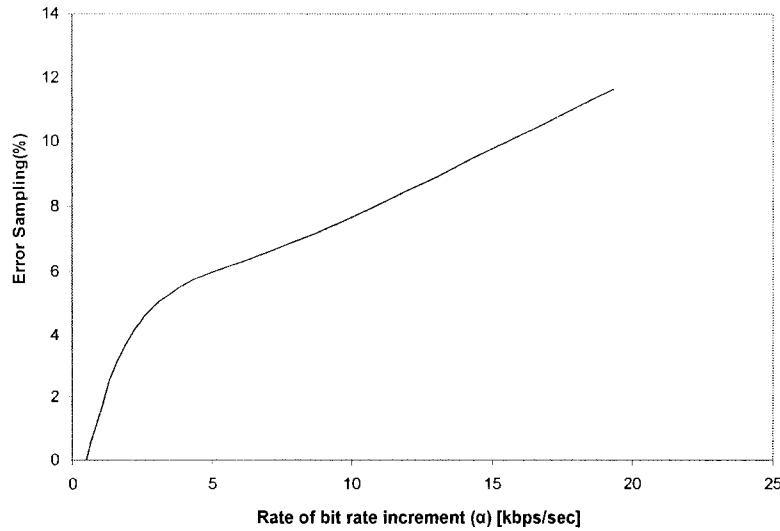


Figure 6.3. Error sampling vs. rate of increase of bit rate.

In this experiment, the error in the input traffic sampling is identified and shown in Figure 6.3. Error in the input traffic sampling was identified by comparing the *sampled real traffic characteristics* against the real input traffic collected at the module entrance. Figure 6.3 shows the percentage sampling error calculated using equation (5.8), against a range of rate of increase of bit rate (Kbps/sec), α . Notations X and Y represent the input real traffic before and after the sampling traffic algorithm, points D and A of Figure 6.1, respectively. This figure demonstrates that the error in sampling gets larger for steeper traffic (big α values). The maximum error percentage obtained from the previous analysis is 12% for the steepest case (20Kbps/sec). To measure the performance of the proposed emulator module, its outputs were compared to the output based on a pure simulation. The pure simulation was carried out using the same network topology used for the emulation; but the real explicit input (Figure 6.2)

was used instead of a sampled version of the input. Since the emulator module (Figure 6.1) uses a sampled version of the input bit rate, the outcomes depend on the quality of the input traffic sampling. Two error traces were calculated using equation (5.8) as functions of α . For both cases, X is the end-to-end delay of the pure simulation. The first and second error trace use Y_1 and Y_2 as their respective inputs, where Y_1 and Y_2 are the end-to-end delay obtained from the emulator module at points B and C, of Figure 6.1, respectively.

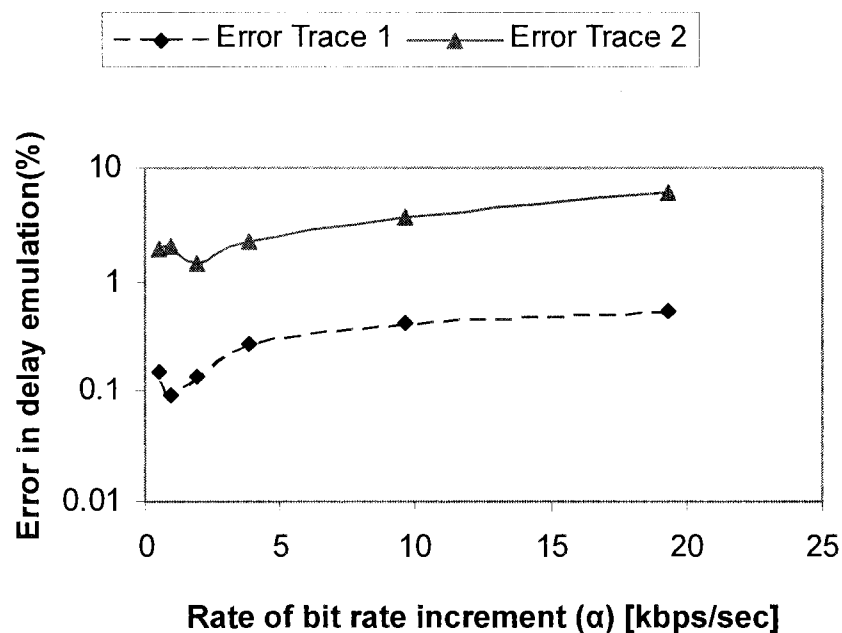


Figure 6.4. Error in delay emulation vs. rate of increase of bit rate.

Figure 6.4 shows these two error traces for different values of rate of increase of bit rate, α (see Figure 6.2). The first error trace is due to input traffic sampling, while the second error trace is due to the input traffic sampling and the error introduced from the traffic adjuster unit (see Figure 6.1). Figure 6.4 indicates that in both cases the error is increased when rate of increase of bit rate increases. This is understandable since both traces are driven by the input sampling, the error that increases with the rate of increase of bit rate (see Figure 6.2). It is also shown in Figure 6.4 that the second error trace is higher than the first one; a phenomenon attributed to two factors. On one hand, when a change in the simulated metrics response is detected and the NISTNET rules are applied right away, the change in the outgoing traffic is not seen instantaneously. This change is observable in the outgoing traffic after a small period of time, generating small time shifts between the outgoing traffic and its correspondent simulated version. On the other hand, since NISTNET was built to accurately reproduce network conditions, it automatically introduces randomness into the analyzed outgoing traffic together with the input rules. More performance analysis results of OTRENET can be found [68].

6.2 REMARKS

Performance analysis results shown in Section 6.1 indicate that the proposed OTRENET module fulfills our expectations of mimicking the overall behavior of a real network scenario.

The proposed ASAF algorithm, described in CHAPTER 5, has proven to reduce the computational overhead required by the packet-by-packet capturing and translating technique. This algorithm not only reproduces the behavior of the actual series, but also does this with the smallest number of frames.

The results presented demonstrate the effectiveness of OTRENET on replicating realistic conditions imposed by simulated environments.

On-going work on OTRENET includes the evaluation of alternatives methodologies to reduce the computation overhead associated with the packet-by-packet simulation employed by simulators like NS. Alternatives ways of end-to-end packet modeling are explained in CHAPTER 7 and CHAPTER 8.

CHAPTER 7. A MEASUREMENT-BASED MODELING APPROACH FOR NETWORK-INDUCED PACKET DELAY

In CHAPTER 3, we discussed various studies on packet delay modeling and characterization. In this chapter, using measurements performed over the Internet, end-to-end packet delay dynamics are modeled using time series techniques under weakly stationary network conditions. Impact of sending rate and packet size of the probes are investigated on the modeling results. Section 7.1 describes time series techniques utilized for packet delay modeling. The impact of ACF and PACF distributions on the packet delay modeling is presented in Section 7.1.1. Sections 7.1.2 and 7.1.3 present methodologies for time series model selection and optimization, respectively. In Section 7.1.4, criteria for evaluating the model goodness of fit and its impact on system modeling is explained.

Section 7.2 presents the experiment setup, methodology and results of end-to-end packet delay and Inter-Packet Gap (IPG) modeling based on the measurement data. Section 7.2.3 complements the analysis by testing the goodness of the fitted packet delay and IPG time series models. In addition, Section 7.2.3 compares the model goodness of fitting results against the characteristics of the packet streams that originate each modeled process.

7.1 MODELING END-TO-END PACKET DELAY USING TIME SERIES TECHNIQUES

A time series $\{X(t)\}$ is defined as a set of observations ordered sequentially in time [47]. A series of n observations can be viewed as a random process of the variables X_1, X_2, \dots, X_n , sampled at, often equidistant, time intervals t_1, t_2, \dots, t_n . Time series can be considered as the output of a dynamic system of which external input can not be observed [43]. There are two main goals of time series analysis; prediction and modeling. The former aims at forecasting future system output values. However, we are interested in latter, in which the properties of the series are summarized and its salient features characterized.

In general, time series modeling focuses on series that are not deterministic but contains a random component. If this random component is stationary, powerful techniques for modeling can be developed. ARMA models are widely used for this purpose. However, most time series data on the Internet are non-stationary or weakly stationary. For such cases there are methods which transform a non-stationary series into a stationary one. In most cases first and second-order differencing are sufficient to remove any kind of trend existing in a time series [47]. ARIMA methodology is based on such an idea [15].

An ARMA model can be viewed as a special case of ARIMA models. ARMA and ARIMA are considered passive black box approaches in which model identification relies solely on the data, without prior information of the system that generated the data. ARMA and ARIMA models fit very well into the study

of Internet data packets, since very little information is known to build up the state of the system, due to the complexity of the networks [70]. Identification and developing of ARMA and ARIMA models rely on ACF and PACF distribution and coefficients. These concepts are presented and associated with the Internet end-to-end packet delay processes next.

7.1.1 ACF AND PACF ANALYSIS FOR END-TO-END PACKET DELAY

Auto Correlation Function (ACF) and Partial Auto Correlation Function (PACF) play an important role on time series modeling and prediction, since they provide useful measures on the degree of dependence between the sampled values at different times.

ACF of a random process describes the correlation between the process at different points in time. Informally, ACF is a measure of how well a series matches a time-shifted version of itself, as a function of the amount of time shift.

Sample ACF, $\hat{\rho}_h$, is defined as $\hat{\rho}_h = \frac{\hat{\gamma}_h}{\hat{\gamma}_0}$. Where $\hat{\gamma}_h$ is the sample auto-covariance

function at lag h , which is presented on equation (7.1):

$$\hat{\gamma}_h = \frac{\sum_{t=1}^{n-|h|} \left(X_{t+|h|} - \bar{X}_n \right) \left(X_t - \bar{X}_n \right)}{n} \quad (7.1)$$

where $-n < h < n$, \bar{X}_n is the mean of the observed time series $\{X(t)\}$, and n is the number of data samples. ACF for end-to-end packet delay process denotes

the amount of dependency the delay of the current packet has to previous packets.

Contrary to ACF, PACF is used to measure the degree of association between the current sample of the series, X_t , and a previous sample, X_{t-k} , when the effect of the other $k-1$ time lags is removed [61]. PACF can be considered as the amount of correlation between a variable and a lag of itself that is not explained by correlations at all lower-order-lags. In practice, ACF of a end-to-end packet delay time series, $\{D(t)\}$, at lag 1 is the coefficient of correlation between D_t and D_{t-1} , which is also most likely to be the correlation between D_{t-1} and D_{t-2} . However, if D_t is correlated with D_{t-1} , and D_{t-1} is equally correlated with D_{t-2} , it has to result in a correlation between D_t and D_{t-2} . Thus, the correlation at lag 1 propagates to lag 2 and most probably to higher-order lags. The PACF at lag 2 is therefore the difference between the actual correlation at lag 2 and the expected correlation due to the propagation of correlation at lag 1 [17].

Previous analysis have observed the relationship between the probe's sending rate, as a fraction of the available link capacity, and the end-to-end packet delay ACF [59] and PACF [10] distributions. Degree of link congestion itself depends on the sending bit rate and the available link capacity, and only sending bit rate can be controlled in an experiment. So when it is increased up to a point at which link congestion is perceived, packets get closer to each other and thus their correlation can be expected to become stronger. This effect

manifests as a slower decay of the ACF function as the sending bit rate increases. Contrary to ACF, the PACF function decays towards zero faster as the sending bit rate increases [10]. Relationship between PACF and ACF is presented in equation (7.2):

$$\Phi = R_p^{-1} \Gamma_p \quad (7.2)$$

where Φ is the vector of the *PACF* coefficients, and R_p and Γ_p are presented in equations (7.3) and (7.4) respectively [17]. Note that the number of PACF coefficients, p , obtained through equation (7.2) depends on the number of ACF coefficients used.

$$R_p = \begin{bmatrix} 1 & \rho_1 & \rho_2 & \rho_3 & \cdot & \cdot & \rho_{p-1} \\ \rho_1 & 1 & \rho_1 & \rho_2 & \cdot & \cdot & \rho_{p-2} \\ \rho_2 & \rho_1 & 1 & \rho_1 & \cdot & \cdot & \rho_{p-3} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \rho_{p-1} & \rho_{p-2} & \rho_{p-3} & \rho_{p-4} & \cdot & \cdot & 1 \end{bmatrix} \quad (7.3)$$

$$\Gamma_p = (\rho_1 \quad \rho_2 \quad \rho_3 \quad \cdot \quad \cdot \quad \cdot \quad \rho_p)^T \quad (7.4)$$

For a CBR flow of probes, the scenario considered in this research, end-to-end packet delay has a direct relationship to Inter Packet Gap (IPG), G_i , as shown in equation (7.5).

$$G_i = D_{i+1} - D_i \quad (7.5)$$

Thus a relationship between ACF and PACF distributions of IPG and end-to-end packet delay can also be expected. This fact plays an important role on end-to-end packet delay modeling, since measuring IPG is less complex and more accurate. This alternative will be explained in detail in the Section 7.2.1.

7.1.2 ARMA AND ARIMA MODEL SELECTION FOR END-TO-END PACKET DELAY PROCESSES

The main goal of Internet traffic modeling is to develop powerful models that represent closely the behavior and characteristics of the observed data values. The black-box modeling approach obtained through ARMA and ARIMA models is very appropriate for characterizing the impact of network on Internet traffic streams. Selection of the best model that represents the observed process depends intrinsically on ACF and PACF characteristics. On one hand, a stationary series can be identified straightforwardly from an ACF distribution, as their autocorrelation coefficients die out quickly. If this is not the case, the observed series has to be considered to be in the range of weakly stationary to non-stationary, depending on its degree of autocorrelation. In the former case, as well as for the stationary series, ARMA is suitable for representing the observed process. However, for non-stationary series, ARIMA models are better alternatives. Order of ARIMA(p, q, d) models are represented by their indexes. Where p and q indicate the order of the embedded AR(p) and MA(q) models, respectively. However d indicates the number of times the process has to be differentiated before it becomes a stationary one. ARMA(p, q) for modeling a $\{X(t)\}$ process is presented in equation (7.6).

$$\hat{X}_t - \sum_{i=1}^p \phi_i X_{t-i} = Z_t + \sum_{j=1}^q \theta_j Z_{t-j} \quad (7.6)$$

where \hat{X}_t is the best linear mean-square predictor of X_t based on the data up to time $t-1$, Z_t is assumed to be a sequence of independent and normal distributed random variables with zero and variance of σ^2 ($i.d. \sim N(0, \sigma^2)$), and ϕ_i and θ_j are the AR and MA coefficients, respectively. Note that AR(p), MA(q) models can be obtained from equation (7.6) when taking $q=0$ and $p=0$, respectively. Equation (7.6) can also be written as $\phi(B)X_t = \theta(B)Z_t$, where $\phi(\bullet)$ and $\theta(\bullet)$ are the p^{th} and q^{th} degree polynomials,

$$\theta(B) = 1 + \theta_1 B + \dots + \theta_q B^q \quad (7.7)$$

$$\phi(B) = 1 - \phi_1 B - \dots - \phi_p B^p \quad (7.8)$$

and B is the backward shift operator ($B^j X_t = X_{t-j}$) [17]. Following this methodology, ARIMA(p, q, d) models for non-stationary process are shown in equation (7.9).

$$\phi^*(B)\hat{X}_t \equiv \phi(B)(1-B)^d \hat{X}_t = \theta(B)Z_t \quad (7.9)$$

Note that $AR(p)$ may be considered as a way of differentiation if model coefficients are close to unity [17], see equation (7.9); thus it is possible that $ARMA(p, q)$ or $AR(p)$ can represent non-stationary processes when coefficients are accurately selected. Note also that since IPG at the receiver side denotes the first differentiation, $d=1$, of the packet delay time series, $\{D(t)\}$, $ARMA(p, q)$ of $\{IPG(t)\}$ is equivalent to $ARIMA(p, q, 1)$ of $\{D(t)\}$.

In practice $MA(q)$ models are called q -correlated processes, as their ACF is reduce to zero for all lags greater than q . Hence, the ACF is a good indication of the $MA(q)$ order. However for a strongly correlated series, ACF tails off but never approaches zero for any q values. In such cases, it is difficult to characterize the process based on ACF only. For these cases, $AR(p)$ models are better alternatives. $AR(p)$ models apply similar methodology as $MA(q)$ models for identifying the model order but, on the contrary, use the PACF function [17]. However, in general $ARMA(p, q)$ and $ARIMA(p, q, d)$ models are often used in time series modeling, combining the benefits of the two previous models and often providing lower order models.

Notice that in general ACF and PACF functions are assumed to reach zero, or cut off, when lying within the 95% Confidence Interval [17].

7.1.3 OPTIMIZATION CRITERIA AND FITTING PROCEDURES FOR ARMA MODELS

Although ARMA and ARIMA model order can go as high as the number of available data samples, n , over-specified models may fail to distinguish the

systematic effects of the data from its random effects [17] [43]. Thus, it is our goal to find a model that fits accurately the observed sampled data values with the smallest number of parameters. Scoring methods have been developed to quantify the relative goodness-of-fit of statistical models for a given data. These methods add a penalty factor to the negative log-likelihood for each parameter of the fitted model. One of the widely used methods is the Akaike's Information Corrected Criterion (AICC). For an ARMA(p, q) process, the AICC score is computed by [17];

$$AICC = -2\ln(L(\tilde{s})) + \frac{2n \times (p + q + 1)}{n - p - q - 2} \quad (7.10)$$

where n is the sample size and $L(\tilde{s})$ is the Gaussian Likelihood of an ARMA process with n observations.

$$L(\tilde{s}) = \frac{1}{\sqrt{\left(2\pi\sigma^2\right)^n r_0 \dots r_{n-1}}} \exp\left\{-\frac{1}{2\sigma^2} \sum_{j=1}^n \frac{(x_j - \hat{x}_j)^2}{r_{j-1}}\right\} \quad (7.11)$$

where $r_{t-1} = E(X_t - \hat{X}_t)^2 / \sigma^2$ and σ^2 is the white noise variance of the fitted model. \hat{x}_t and x_t are the modeled and actual data samples at time t , respectively. The values p , q and σ^2 that maximize equation (7.11), $L(\tilde{s})_{\max}$, is called the Maximum Likelihood Estimator, which is interpreted as the ARMA parameter values most likely to be responsible for the observed data values. As a

result, the model that has a lower AICC score is a better representation of the process than those with higher scores. After selecting the right order of the ARMA or ARIMA process, estimation of its parameters has to be done. Several techniques exist for this. Yule-Walker and Burg procedures apply to the fitting of pure autoregressive models, although the former can be adapted to models with $q > 0$ its performance is less efficient than when $q=0$. On the other hand, Innovation and Hannan-Rissanen algorithms are used also to provide preliminary estimates of ARMA parameters when $q > 0$. For pure autoregressive models Burg's algorithm usually gives higher likelihoods than the Yule-Walker equations. For pure moving average models the Innovation algorithm often gives slightly higher likelihoods than the Hannan-Rissanen algorithm. For mixed models the Hannan-Rissanen algorithm usually gives better fitting. Detailed information on the above mentioned techniques can be found in [17].

7.1.4 DIAGNOSTIC CHECKING FOR ARMA AND ARIMA MODELS

Prediction and modeling analysis using ARMA and ARIMA models typically judges the goodness of fit of a statistical model to a set of data by comparing the observed values with the corresponding predicted values obtained from the fitted model. It is known that if the fitted model is appropriate, then the residual should have properties consistent with those of a white noise sequence [17][43]. Residuals, \hat{w}_t , are defined to be the rescaled one-step predictor errors [17];

$$\hat{w}_t = (X_t - \hat{X}_t) / \sqrt{r_{t-1}} \quad (7.12)$$

To check the appropriateness of the model we can therefore examine the residual series and check that it resembles a $WN(0,1/n)$ sequence [17]. ACF/PACF distribution, histogram and data plot generated from the model residual can be compared to the expected generated by a $WN(0,1)$ sequence when evaluating the correctness of the model [53].

7.2 ANALYSIS OF MEASUREMENT DATA

In this section, experiment setup, methodology and results of end-to-end packet delay modeling are presented. The data used for this analysis is from a previous study described in detail in [59]. Here CBR UDP traffic streams of 20,000 packets each, corresponding to 64 and 256 bytes packet size, were sent from California Polytechnic to Colorado State University, using Ixia 1600T chassis [77] at both sides. Average one-way delay was found to be 22 milliseconds. Experiments were run on consecutive days at the same time to maintain consistency. Non-peak times of the days were chosen when running the experiments to keep cross traffic within a narrow range. End-to-end packet delay and IPG values were collected for a variety of sending rates and packet sizes.

7.2.1 METHODOLOGY FOR FITTING ARMA AND ARIMA MODELS INTO PACKET DELAY SERIES

The research presented in this section focuses on capturing the effect of the network induced on a CBR flow of UDP probes, by means of finding an optimum

ARMA/ARIMA model that best represents the dynamics of the probe's packet trace. In practice, this is mostly done by collecting consecutive packet delays samples [59]. However, such an approach requires clock synchronization techniques on both sides to prevent clock skew issues. Conversely, collecting IGP samples is presented here as an alternative. IPG not only avoids the synchronization dilemma between the sender and the receiver, but also represents an alternative for modeling non-stationary packet delay processes, as was explained in Section 7.1.2. Packet delay model can be obtained afterwards, integrating the IPG model. Note that higher order of differentiation may be needed in some cases.

For both packet delay and IPG modeling, AR and MA model order can be estimated by observing the ACF and PACF distributions, respectively. These in turn depend on probe's sending rate, as a fraction of the available link capacity, and probe's packet size, as will be seen in Section 7.2.2. However, ARMA and ARIMA represent a mixture of AR and MA models, and their orders are calculated through scoring methods. Thus, although a change of the model order is expected according to changes on the probe's sending conditions, a clear relationship of the model orders to the probe's sending conditions can not be expected, as it is on AR and MA models.

The analysis presented in this section studies the effect of the network induced on a CBR flow of probes, by finding the most optimum ARMA/ARIMA model which captures the dynamics of the observed probe's packet delay or IPG

series. A comparison of packet delay and IPG modeling approaches is given for varied bit rate and packet size scenarios in the following subsections. Note that for very low sending bit rate scenarios, as a fraction of the available link capacity, packet delay autocorrelation is very weak and thus its distribution may be enough to represent the process. However, this conclusion can only be reached after examining the ACF/PACF distributions of the series.

7.2.2 MODELING RESULTS

Figure 7.1 shows the sample ACF of the packet delay series, for different sending bit rates, using 64 bytes for packet size. ACF distributions for low sending rates decay faster than ones coming from medium or high sending rates. PACF distributions are shown in Figure 7.2 for four packet delay traces, generated with four different sending bit rates; 0.25, 1, 30 and 70Mbps, also using 64 bytes for packet size. It can be seen from here that, as the lag increases all PACF coefficients diminish to zero faster than their corresponding ACF, as was anticipated in Section 7.1.1. However, we note that it takes a larger number of lags for the PACF to die off for the one generated by the smallest sending bit rate (0.25Mbps).

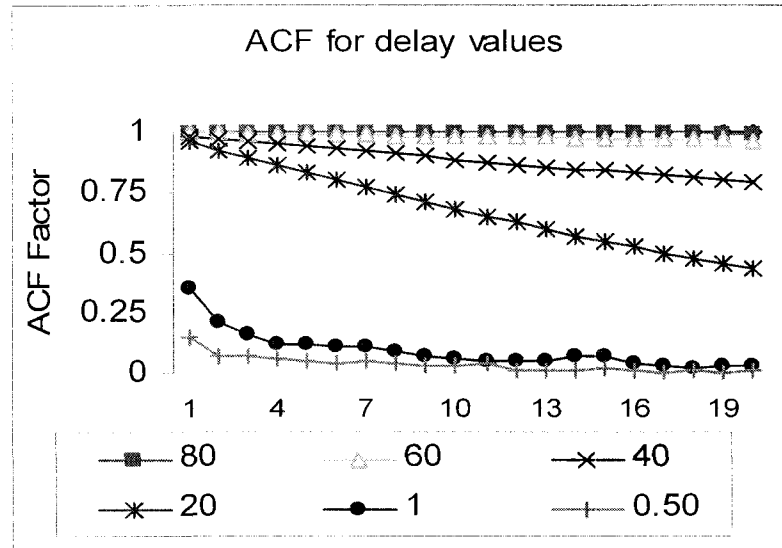


Figure 7.1. ACF function for lag 1-20 of packet delay values for varied sending bit rates (0.5 – 80Mbps) using 64 bytes packet size

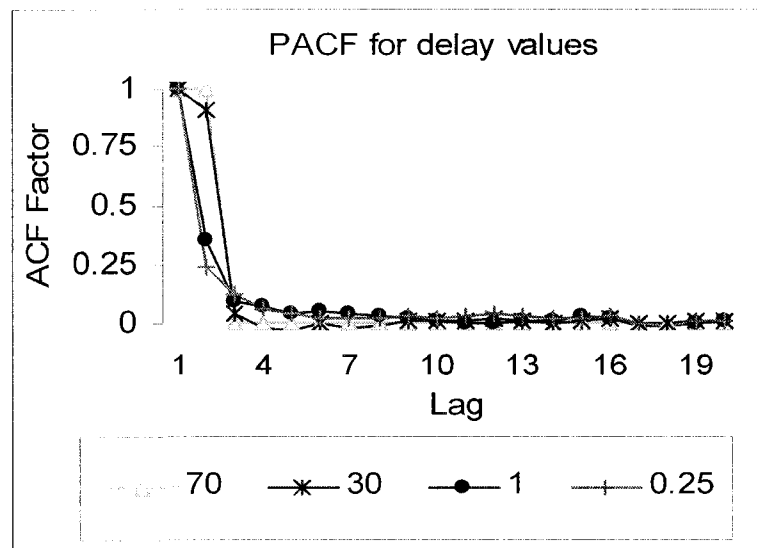


Figure 7.2. PACF function for lag 1-20 of packet delay values for varied sending bit rates(0.25,1,30 and 70Mbps) using 64 bytes packet size.

This can be explained since as sending bit rate increases, IPG's tend to decrease [59] and thus adjacent packets get closer to each other and more likely

to be aligned together on the same buffer [10]. As a result correlation between adjacent delay samples becomes stronger. When applying PACF, this chain of dependency is broken (strong influence of intermediate samples is removed), consequently PACF coefficients will decay faster than their corresponding ACF ones. Since packet delay chain of dependency for high sending rates streams is stronger than for low ones, it can be expected that *PACF* coefficients tend to decay faster, as was anticipated in Section 7.1.1 , and previously observed on [10]. The inverse relationship between *ACF* and *PACF* coefficients (Φ and Γ_p) can also be seen mathematically from equations (7.2), (7.3), and (7.4).

Table 7.1 shows results of the model fitting done for a set of sending bit rates scenarios for the experiment set up described above. Order of the ARMA models, as well as model parameters were obtained using the methodologies presented in Section 7.1 and equation (7.10). ITSM package is used for the model fitting [78]. Table 7.1 shows also the negative log-maximum likelihood estimator, $-2 \ln(\hat{L}(\hat{\xi})_{\max})$, which represents the goodness-of-fit for the modeled series [17]. Model that has lower $-2 \ln(\hat{L}(\hat{\xi})_{\max})$ is considered better fit for the analyzed series, as explained in Section 7.1.3. From Table 7.1, it can be seen that *ARMA* order selection for packet delay and *IPG* series show no clear connection to the sending bit rate, as was anticipated in Section 7.2.1. Also as the sending bit rate increases, greater than 70Mbps for this experiment set up, model order for both series tend to decrease. This can be understood due to the

fact that *PACF* for packet delay and *IPG* decays faster as sending bit rate increases, see Figure 7.2.

End-to-end Packet Delay and IPG Modeling Fitting						
Sending Bit Rate	<i>ARMA</i> model for packet delay series			<i>ARMA</i> model for <i>IPG</i> series		
	<i>p</i>	<i>q</i>	$-2\ln(L(\hat{s})_{\max})$	<i>p</i>	<i>q</i>	$-2\ln(L(\hat{s})_{\max})$
0.5 Mbps	3	4	6.06E+03	4	5	6.12E+03
1 Mbps	6	6	4.35E+03	3	3	4.47E+03
10 Mbps	2	7	-2.33E+04	7	1	-2.32E+04
20 Mbps	1	7	-3.96E+04	7	5	-3.95E+04
30 Mbps	1	6	-4.27E+04	1	7	-4.26E+04
40 Mbps	7	5	-4.86477E+05	7	7	-4.85815E+05
50 Mbps	1	6	-5.35692E+05	4	4	-5.35711E+05
70 Mbps	1	0	-7.99783E+04	1	0	-8.00136E+04
80 Mbps	1	0	-1.25418E+05	1	0	-1.25485E+05

Table 7.1. End-to-end packet delay model fitting for different sending bit rate scenarios using 64 bytes packet size.

Figure 7.3 and 7.4 show the ACF and PACF distributions for packet delays for varied sending bit rate scenarios for 256 bytes packet size. From here, it can be seen that autocorrelation of the process at different points in time is weaker than the one observed using 64 bytes packet size for the same probe sending rate. This is expected since sender IPG for these scenarios are 4 times bigger than with corresponding 64 packet size cases for the same probe stream bit rate.

From Figure 7.4 it can be seen that PACF coefficients die off abruptly after a small number of lags, in fact, for sending bit rates higher than 10 Mbps PACF distribution cuts off after the second lag. This phenomenon tells us that the chain of dependency of intermediate samples is easily broken, see Section 7.1.1.

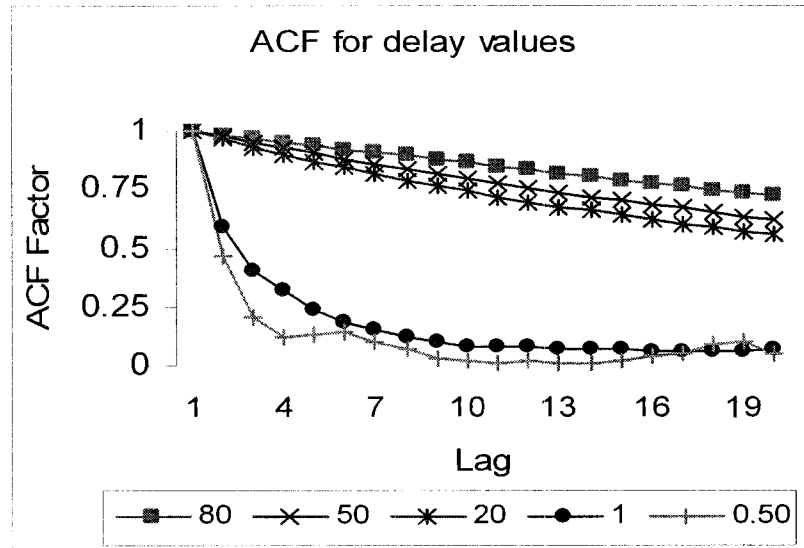


Figure 7.3. ACF function for lag 1-20 of packet delay values for varied sending bit rates (0.5 - 80Mbps) using 256 bytes packet size.

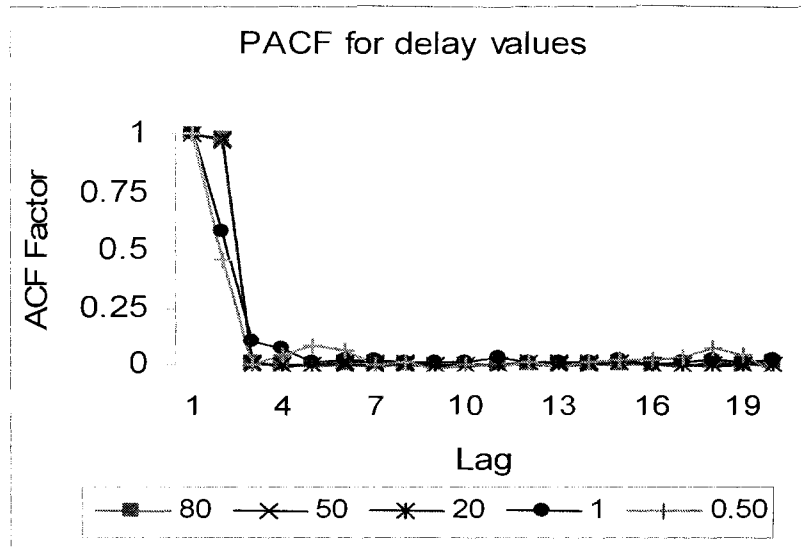


Figure 7.4. PACF function for lag 1-20 of packet delay values for varied sending bit rates (0.5 - 80Mbps) using 256 bytes packet size.

Comparing Figure 7.4 to Figure 7.2, it can be seen that the chain of dependency of intermediate packet samples does not only depend on probe's sending bit rate but also on probe's packet size. Thus, it can be expected that auto-regressive models of packet delay and IPG series will vary according these two parameters, and thus both have to be considered when extracting the effect of the network on the captured packet stream.

End-to-end Packet Delay and IPG Modeling Fitting						
Sending Bit Rate	ARMA model for packet delay series			ARMA model for IPG series		
	p	q	$-2 \ln(\hat{L}(\hat{s})_{\max})$	p	q	$-2 \ln(\hat{L}(\hat{s})_{\max})$
0.5 Mbps	5	7	1.99E+05	5	7	2.00E+05
1 Mbps	4	1	1.80292E+05	3	1	1.80492E+05
10 Mbps	6	4	1.43592E+05	1	2	1.43871E+05
20 Mbps	1	0	1.6921E+05	1	2	1.67212E+05
30 Mbps	1	0	1.2275E+05	1	1	1.22264E+05
40 Mbps	1	0	1.4591E+05	1	1	1.4517E+05
50 Mbps	1	0	1.41872E+05	1	1	1.41851E+05
70 Mbps	1	0	1.37721E+05	2	1	1.37531E+05
80 Mbps	1	0	1.3579E+05	2	1	1.3521E+05
100 Mbps	1	0	1.3991E+05	2	1	1.3972E+05

Table 7.2. End-to-end packet delay model fitting for different sending bit rate scenarios using 256 bytes packet size.

Table 7.2 shows the results of the model fitting done for a set of sending bit rates scenarios for 256 bytes packet size. Results were obtained in a similar manner to those in Table 7.1. From Table 7.2 it can be seen that ARMA model orders, for both packet delay and IPG series, decreases rapidly for sending bit rates higher than 10Mbps. This can be expected from the PACF distribution observed in Figure 7.4. By comparing results obtained from Table 7.2 to the ones observed in Table 7.1, it can be concluded that the packet delay model becomes

an AR process at smaller sending data rates for the 256 bytes packet size experiment than for the 64 bytes packet size one. Also, in general ARMA packet delay and IPG models show lower orders for the 256 bytes packet size cases than their corresponding 64 bytes packet size cases. This can be explained since both ACF and PACF distributions decay faster for 256 bytes packet size than for 64 bytes packet size. From here it can be concluded that ACF and PACF distributions and therefore ARMA/ARIMA models depends not only on the probe's sending data rate but also on the probe's packet size.

Note that since the above experiments were conducted at non peak times of the day, network cross traffic remained within a narrow range. Thus non-stationarity of the observed packet delay samples was modeled successfully using $ARIMA(p, q, d)$, where $d=\{0-1\}$, and not higher orders of d were needed. $d=0$ represent the ARMA model of packet delay and $d=1$ the ARMA process of IPG. Modeling efforts presented in this research aim at generating ARMA models which characterize the overall behavior of a packet delay or IPG trace, and not to obtain a perfect match of these ones at any given time. In the next subsection goodness of fitting for these ARMA models is tested.

7.2.3 GOODNESS OF ARMA MODEL FITTING VS. TIME SERIES DYNAMICS

In this subsection the goodness of the fitted packet delay and IPG ARMA models presented in Section 7.2.2 is tested. Goodness of fitting results is related to the characteristics of the packet streams that originate each of them. In this research characteristics of packet stream dynamics are captured by two

complementary means; range of dependency of packet samples, and degree of non-stationarity of packet trace. The former is measured by the ACF distribution of the collected data packet trace; packet delay and IPG series. Range of dependency of trace samples can be categorized by fitting the ACF distribution into a Zipf function, as shown below:

$$\rho_h \sim h^{-2+2H} \quad (7.13)$$

where ρ_h is the autocorrelation coefficient at lag h , and $H \in \{1/2, 1\}$ is the Hurst parameter [11]. Range of dependency of trace samples can be analyzed by means of the H value. $H \approx 0.5$ indicates a true *random walk*, which denotes no correlation between samples [17]. Short-Range Dependent (*SRD*) and Long-Range Dependent (*LRD*) traffic, follow onto the $0.5 \leq H \leq 1$ range. However, *SRD* traffic exhibit H values closer to the lower boundary, in which *ACF* distribution shows an exponential rate of decay. While H values for *LRD* traffic get closer to the higher boundary as traffic gets more autocorrelated. In general, the essential contrast between *SRD* and *LRD* traffic lies on the fact that the former represents a Poisson-like traffic, while the latter represents traffic bursty in nature [6].

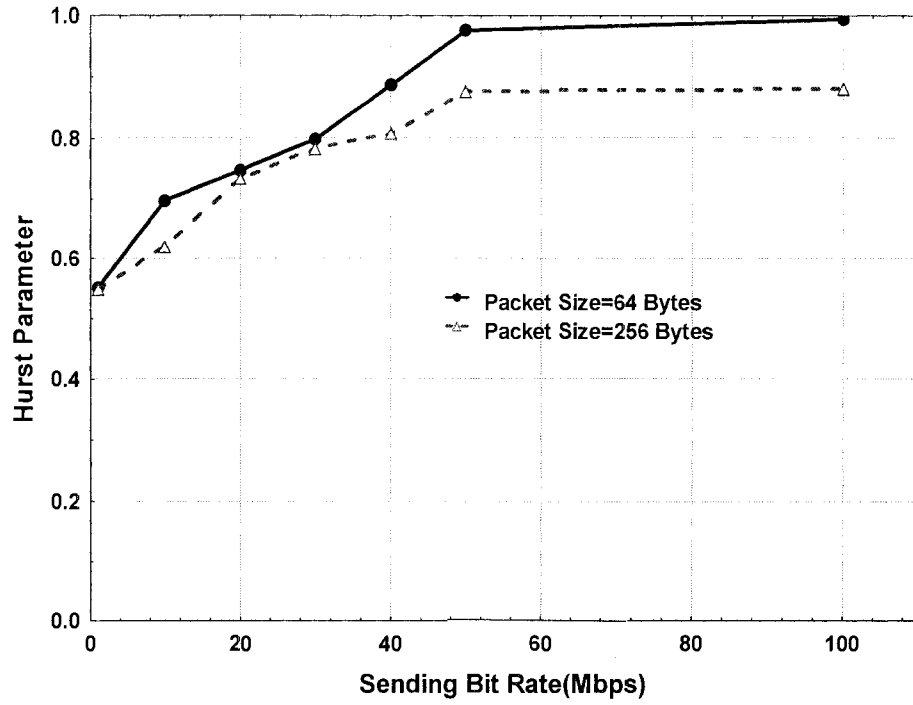


Figure 7.5. Hurst parameter of the packet delay values for varied sending bit rates (0.5-80Mbps) using 64 and 256 bytes packet size.

ACF distributions of packet delay traces obtained from the experiment described in Section 7.2 were fitted into a Zipf distribution using Matlab Curve Fitting Toolbox [82]. Figure 7.5 shows the H parameters for packet delay samples for varied sending bit rates using 64 and 256 bytes packet size. From here it can be seen that H increases asymptotically along with the sending bit rate, as a traffic changes from SRD to LRD. Low sending bit rate, compared to the available link capacity, exhibit $H \approx 0.5$, which denotes traffic dynamics resemble a random walk process [6]. From Figure 7.5 it can be seen that H rises faster and higher for the 64 bytes packet size than for the 256 bytes packet size scenario. Peak values of H

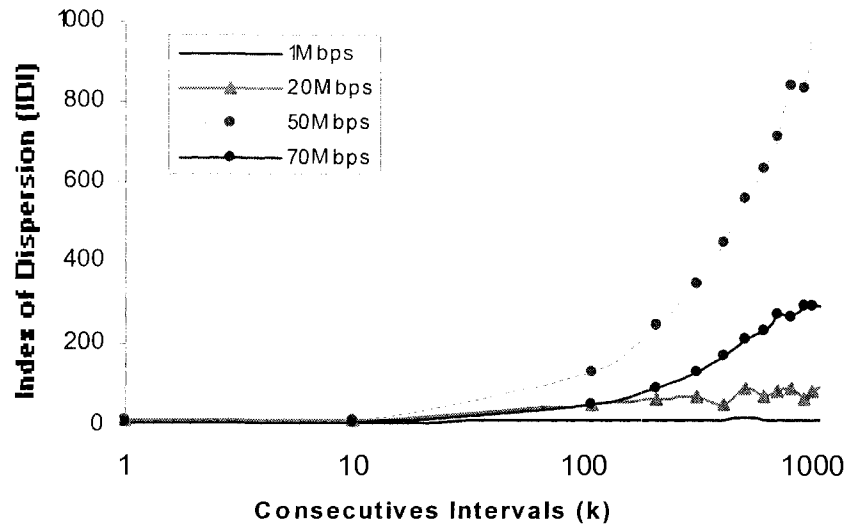
are reached at sending bit rates greater than 50Mbps, for both 64 and 256 bytes packet size.

Conversely, to describe the degree of non-stationarity on a packet delay trace, the Index of Dispersion of Intervals (IDI) is used. In general IDI measures the dependence between consecutives samples on a trace, and it is often used to describe the burstiness of a signal [65]. IDI is defined as a sequence $\{c_k^2\}$, $k \geq 1$, where;

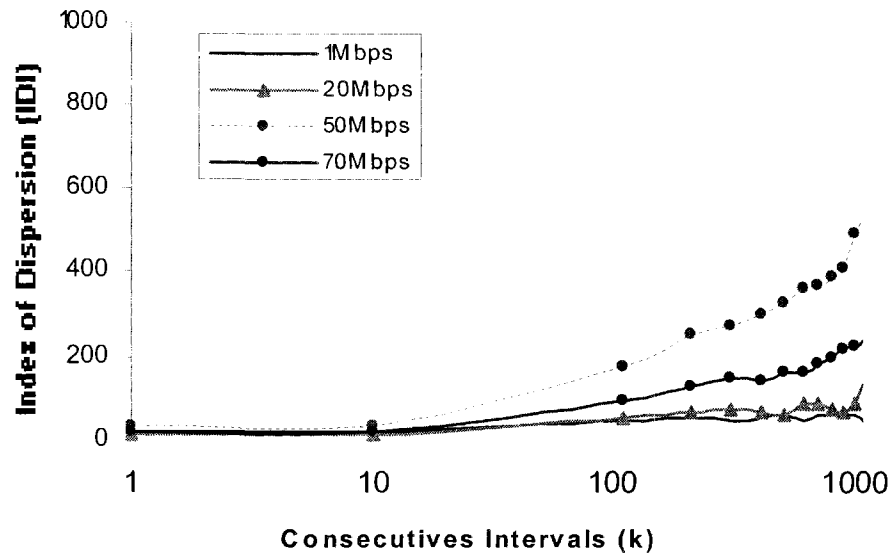
$$c_k^2 = \frac{kVar(S_k)}{[E(S_k)]^2} \quad (7.14)$$

and the random variable S_k is the sum of k consecutives samples on a trace.

If the trace represents a Poisson process, then $c_k^2 \equiv 1$ for every k . However if process has higher variance at some time scale, then c_k^2 will tend to increase as a function of k [35].



(7.6.a)



(7.6.b)

Figure 7.6. IDI for blocks of k consecutive packet delay samples for varied sending bit rates, (a) using 64 bytes packet size, (b) using 256 bytes packet size.

Figures 7.6.a and 7.6.b show the IDI for the packet delay traces obtained from the experiment described in Section 7.2, using 64 and 256 bytes, respectively. From Figure 7.6 it can be seen that packet delay distribution can be fitted as stationary Poisson process for low sending bit rates, compared to the available link capacity, for both 64 and 256 bytes packet size cases. However, as sending bit rates increases processes tend to get more non-stationary. This phenomenon gets more notorious for 64 bytes packet size than for 256 bytes packet size. In general, results of Figure 7.6 agree and complement the ones observed in Figure 7.5.

Goodness of fitted ARMA models is measured by analyzing the randomness of the residual, as was explained in Section 7.1.4. In this research test for checking the hypothesis that residuals are independent and identical distributed (iid) sequence are performed by two means, by analyzing the normality of the residuals and by measuring the amount of autocorrelation on the residual samples.

Normality of the residuals is checked by means of the Normal Quantile-Quantile plots (q-q plots). Normal q-q plots are a graphical method for diagnosing differences between the probability distribution of a statistical population from which a random sample has been taken, and a comparison normal distribution [17]. If samples belongs to a normal distribution, points of the normal q-q plots will straggle about the line $y = x$. In general normality of a trace is measured by the R^2 value obtained by comparing the points of the q-q

plots against the $y = x$ line. If R^2 is closer to one, then the residuals can safely be assumed to be normally distributed, however assumption of normality is rejected if R^2 is sufficient small [17].

Conversely, the Ljung-Box Test is used to measure the amount of autocorrelation on the residual samples. The Ljung-Box test is based on the autocorrelation plots, which are commonly used to test the range of dependency on the samples. However, instead of testing randomness at each distinct lag, it tests the overall randomness based on a number of lags [17]. The Ljung-Box test statistic is calculated as:

$$Q = n(n+2) \sum_{l=1}^k \left(\frac{\rho_l^2}{n-l} \right) \quad (7.15)$$

where n is the sample size, ρ_l is the autocorrelation at lag l , and k is the number of lags being tested. The hypothesis of randomness is rejected if $Q > \chi_{1-\alpha;k}^2$, where $\chi_{1-\alpha;k}^2$ is the critical value of a chi-square distribution with k degrees of freedom [17]. In general, critical values are cut-off values that define regions where the test statistic is unlikely to lie [17]. Critical values for a hypothesis test depend upon a test statistic, and the significance level, α , which defines the test sensitivity [83]. It is a common practice to use $\alpha = 0.05$, which implies that the null hypothesis is rejected 5% of the time when it is in fact true [83]. Another measurement of testing the mentioned hypothesis is by means of

the p -value. The p -value is formally defined as the probability of the chi-square test statistic being at least as extreme as the one observed given that the null hypothesis is true [83]. In general, a small p -value is an indication that the null hypothesis is false. In addition, p -value is analogous to the significance level for the test, α . For instance, rejecting the null hypothesis for $\alpha=0.05$, it is equivalent of rejecting the null hypothesis for p -value smaller than 0.05 [83]. In this research degree of autocorrelation of the residual samples is measured, in the context of the Ljung-Box Test, by analyzing the p -value for a $\alpha=0.05$ condition.

Figure 7.7 shows the normal Q-Q plot residual samples of fitted ARMA models for packet delay and IPG traces for varied sending bit rates using 64 bytes packet size. As it can be seen from here, points of the normal q-q plots differ from the $y=x$ line as the sending bit rate increases. As a result, R^2 value for both packet delay and IPG normal residual analysis decreases as the sending bit rate increase. From Figure 7.7 also it can be seen that for high sending bit rate scenarios, IPG sample residual analysis shows better resembles to a normal distribution than the corresponding packet delay model.

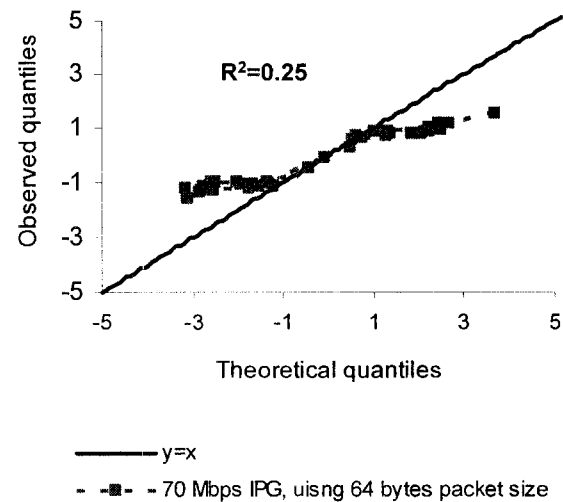
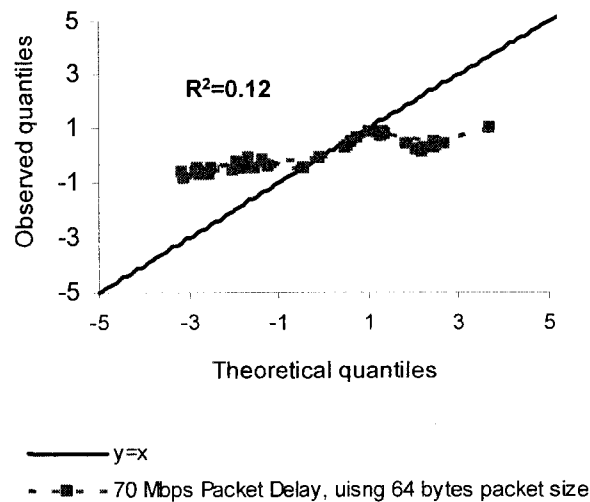
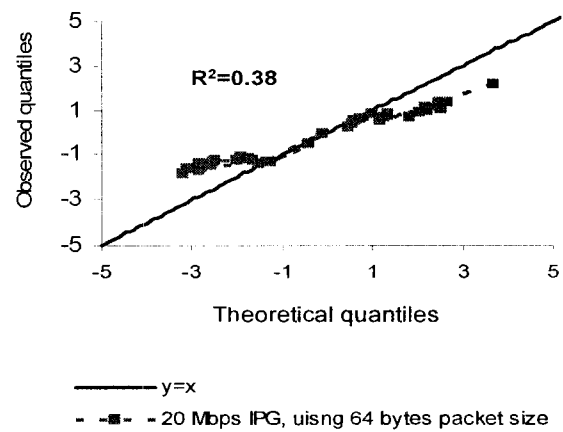
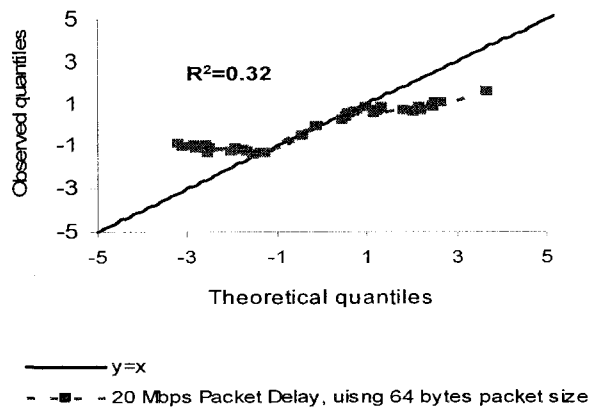
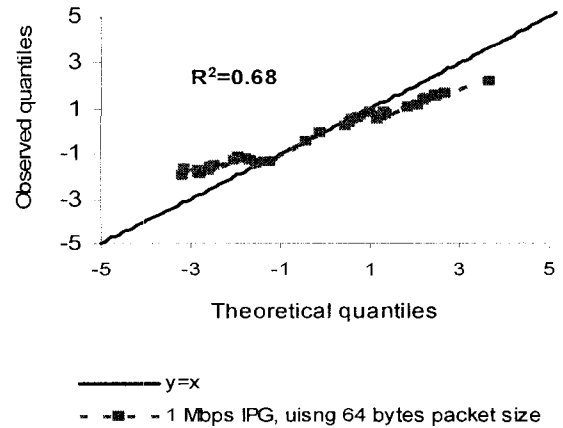
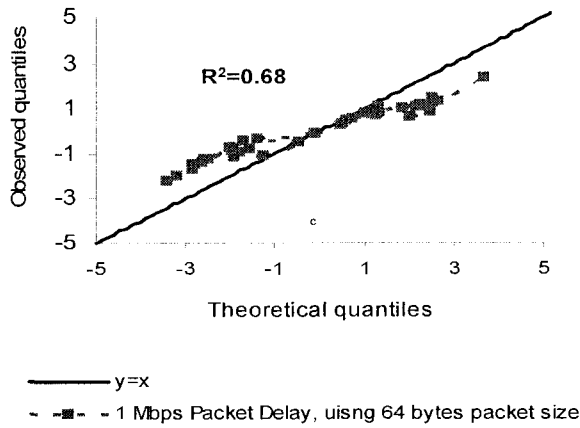


Figure 7.7. Normal Q-Q plot residual of fitted ARMA models for packet delay and IPG traces for varied sending bit rates using 64 bytes packet size.

In Figure 7.8 the normal Q-Q Plot Residual R^2 values for test of randomness for residual of fitted ARMA models for packet delay and IPG traces for varied sending bit rates and packet sizes is shown. As it can be seen from Figure 7.8, both packet delay and IPG model residual series differ from a normal distribution, within a narrow range, as the sending bit rate increases. In general, Figures 7.8 and 7.7 show that model residual for both, packet delay and IPG processes, can be assumed to be normally distributed. However, residual sample distribution differs from normal distribution, within a narrow range, as sending bit rate increases.

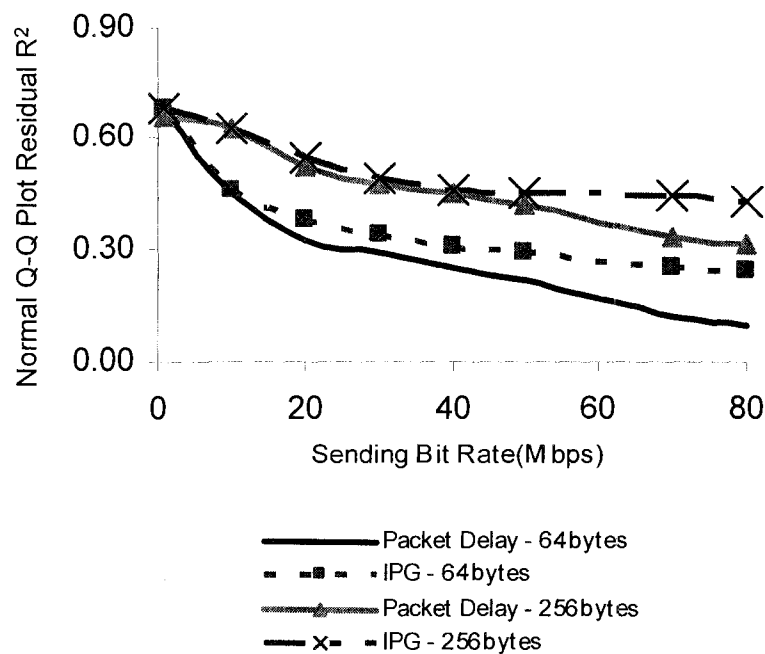


Figure 7.8. Normal Q-Q Plot Residual R^2 values for test of randomness for residual of fitted ARMA models for packet delay and IPG traces for varied sending bit rates and packet sizes.

From Figure 7.8 it can be concluded that IPG is less susceptible than the corresponding packet delay processes, for a given sending rate and packet size scenario, to this phenomenon. Also that model residuals generated with 256 bytes packet size show better resembles to a normal distribution than the corresponding 64 bytes packet size scenarios.

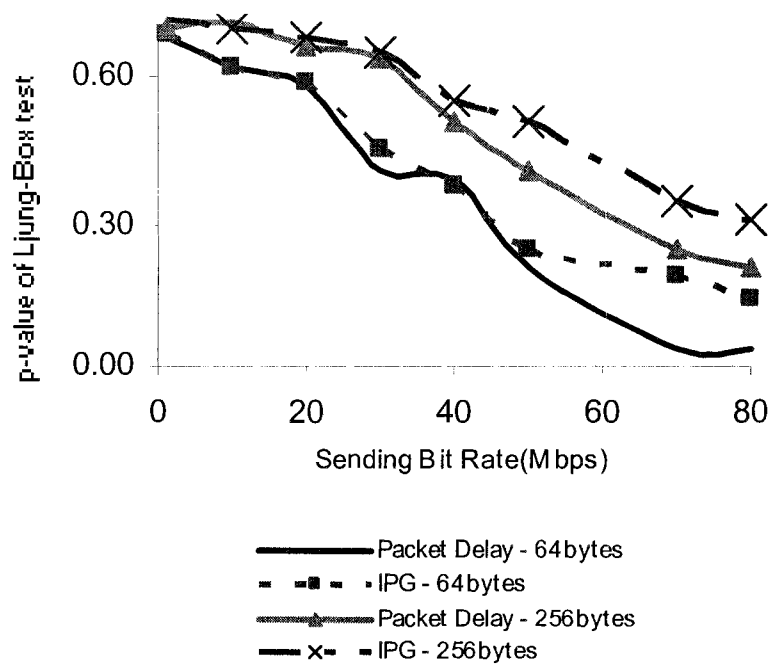


Figure 7.9. p -values of Ljung-Box test of randomness for residual of fitted ARMA models for packet delay and IPG traces for varied sending bit rates and packet sizes.

In Figure 7.9 the p -values of the Ljung-Box test of randomness for residual of fitted ARMA models for packet delay and IPG samples for varied sending bit rates and packet sizes is shown. From here it can be seen that the p -value of the Ljung-Box

Test for packet delay series generated with sending bit rates greater than 70Mbps and 64 bytes packet size, are lower than 0.05, which implies that the null hypothesis is rejected for the residual samples of these cases. However, for all the other cases the p -value of the Ljung-Box Test shows values greater than 0.05, which implies that residual samples have low dependency. In general, results of Figure 7.9 reach similar conclusion that the ones observed in Figure 7.8.

In a nutshell, it can be stated that the analyzed packet delay and IPG traces show accurate goodness of ARMA model fit for all the analyzed sending conditions. Goodness of ARMA model fit deteriorates as the sending bit rate increases, and shows stronger robustness for the 256 bytes scenario than the 64 bytes scenarios. In general, goodness of ARMA model fit for packet delay and IPG processes is similar for small sending bit rate. However, as sending bit rate increase IPG shows a better alternative for network system modeling. This can be understood due to the fact that sending bit rate increment creates non-stationarity and autocorrelation on the sample trace. ARIMA models are more suitable when modeling such a series, and it was previously mentioned in Section 7.1.2, ARMA for IPG traces represents an $ARIMA(p, q, 1)$ for packet delay traces. Finally, it can be concluded that goodness of ARMA model fit agrees with the corresponding characteristics of the analyzed packet traces dynamics.

7.3 REMARKS

This chapter discussed the impact of packet delay and IPG on the network system modeling. Methodology, results, and remarks for network system

modeling, by means of time series techniques, based on packet delay and IPG observations under weakly-stationary network conditions has been presented in this chapter. The impact on packet autocorrelation on capturing the network system dynamics has also been investigated here.

Finding presented in this chapter concludes that the behavior of end-to-end packet delay and IPG sequences can be captured effectively by ARMA and ARIMA models, under weakly-stationary network conditions and using CBR probe flows. Effects of sending bit rate, packet size, and available link capacity on network system modeling has been analyzed. Under these conditions, model goodness-of-fit results demonstrate modeling accuracy for both packet delay and IPG processes under small sending bit rate conditions. However, as sending bit rate increases, as a fraction of the bandwidth, IPG becomes better alternative for network system modeling.

CHAPTER 8. AN ONLINE METHODOLOGY FOR MODELING NON-STATIONARY END-TO-END PACKET DELAY

In CHAPTER 7 end-to-end packet delay dynamics were modeled using time series techniques under weakly stationary network conditions. In this chapter this analysis is extended to network systems under non-stationary conditions.

In Section 8.1 the impact of non-stationarity when modeling network system dynamics is analyzed. Factors that create non-stationarity and Long Range of Dependency (LRD) on packet sample observations are presented in Section 8.1. In addition, traditional methods for modeling non-stationarity network systems are critiqued and a novel approach is proposed.

In Section 8.2 the effect of non-stationarity on packet delay is explored and real evidence of false sense of LRD on packet delay is presented. In Section 8.3 a methodology for modeling time variant packet delay series is presented based on adaptive AR model and Kalman Filtering algorithm.

In Section 8.4 a modified version of the Divergence-Test [9] is proposed for online segmentation of packet delay traces. Such method is based on the non-stationary of the packet delay observations.

Section 8.5 presents the experiment setup, methodology and results of the proposed methodology for segmenting non-stationary packet delay traces.

Abundant measurements of packet delay over the Internet under various conditions are used for testing the proposed methodology. Experiment results demonstrate a potential online packet delay classification capability of the proposed algorithm based on the non-stationary of the observations, while keeping computational and storage requirements low. In general, results shows that analyzing packet delay processes by modeling the segmented stationary traces yield to a better understanding of the network system dynamics.

8.1 IMPACT OF NON-STATIONARITY ON NETWORK SYSTEM DYNAMICS

In CHAPTER 3 the theoretical foundation for network system modeling based on packet delay have been presented. In addition, system characterization based on real measurements has been presented in CHAPTER 7. Modeling efforts shown in CHAPTER 7 assumes weakly stationary network conditions. However it is well known that packet delay, among many Internet traffic metrics, may change in time due to several factors [35], and thus system can be become non-stationary.

Although network link congestion is considered as the main reason for non-stationarity and strong autocorrelation, also known as LRD, on packet delay observations [6], it has previously been demonstrated that other network conditions, such as link failure, routing table updates, and routing flapping [35], can also be responsible for this phenomenon. In practice it is common to observe patterns of periodic spikes, bursty behavior, and level shifting in packet delay traces. One of the most important factors that LRD introduces into time series is

non-stationarity [17] [35]. However random spikes and irregular events on the network system can indeed create a false sense of LRD on the observations.

When modeling non-stationary systems, traditional time series methodologies rely on transforming them into stationary ones by means of differencing techniques [17]. However, in the context of packet delay modeling, such an approach may fail of distinguishing uneven events responsible of creating false sense of LRD on the packet traffic since it only captures the overall behavior of the system during the observation period.

Consequently, segmenting the observation's trace into groups of stationary time series has been proposed as an alternative solution. Time series segmentation is considered a useful approach for quantifying a piecewise-stationary series [35], since it represents the observed trace as a number of time series that are themselves stationary [29] [58]. Analyzing packet delay observations by modeling the segmented stationary series yields a better understanding of the network system dynamics and lead to more accurate modeling and prediction analysis.

Segmentation of a packet delay trace based on its observed non-stationarity is a complex task. In general, obtaining an exact segmentation of a non-stationary time series demands tremendous computation requirements that scales as $O(N^N)$, where N is the number of packet delay [29]. In addition, packet delay observations may require enormous storage requirements depending on the length of the experiment and the probe's sending conditions employed, when

analyzed offline. Moreover, collecting and storing packet-by-packet data may be impractical and unnecessary for offline analysis; such an approach can also be considered to be extremely computationally expensive for modeling, prediction, and flow control mechanisms when conducted in an online manner.

In this chapter we propose a novel approach for online packet delay segmentation and modeling based on the non-stationarity of the samples. The proposed methodology aims at modeling, in real-time, the effect of the network dynamics induced into a packet flow traversing it, while keeping computational and storage requirements low. This methodology, which is based on the Divergence-Test [9], separates the observed packet delay trace into segments in which each segment is represented by a different statistical model (AR model) [17]. A segment is generated only when a significant change on the packet delay dynamics is detected. Each segment represents a stationary process, which is uncorrelated to the others. The proposed methodology employs an embedded Kalman Filtering algorithm to recursively update the system statistical model based on the current observation and the past modeled network system stage. The online segmentation methodology compares the distance between the updated system model and the model of the previous stationary segment detected [9], when deciding if segmentation is needed.

Low memory storage requirement is one of the main benefits of the proposed mechanisms. This is achieved, due to the fact that only key statistical model parameters of the detected segments are stored in memory, as opposed of

storing every given observed packet delay sample. Notice that the proposed modeling and segmentation mechanism does not attempt to detect instantaneous changes of the packet delay observations; on the contrary, its segmentation process is triggered by the non-stationarity of the observations. In addition, the adaptive behavior of the proposed methodology offers the capability of understanding the dynamics of the network system online, which can be used for real-time decision making. Moreover, the proposed mechanism accomplishes a tradeoff between the complexity of the calculations and the desired precision of the results. The proposed segmentation methodology is explained in detail in Section 8.4.

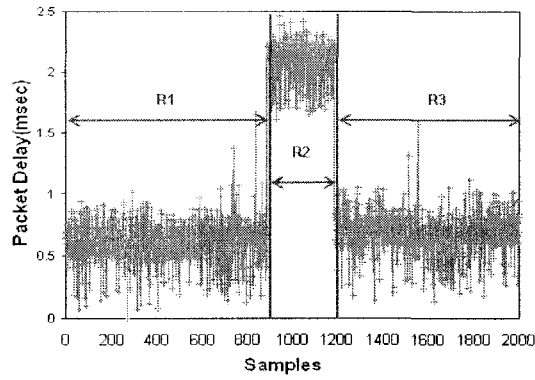
8.2 EXPLORING LRD ON PACKET DELAY SERIES

Packet delay dynamics varies in time according to network conditions, such as link congestions, link failure, routing table updates, and routing flapping (which may appear at link failure situations), among others [35]. Such phenomenon tends to vary packet delay characteristics in time, among them, packet delay autocorrelation, which is a key factor for modeling the network dynamics [59] [69].

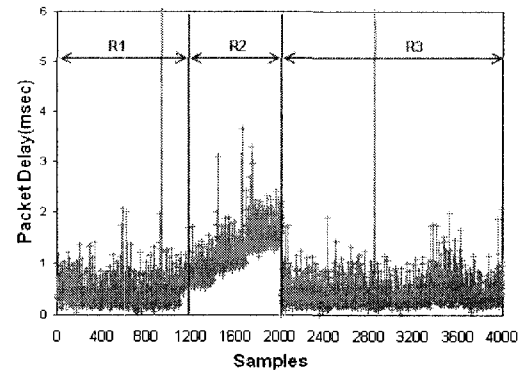
Although network link congestion is considered to be one of the main reasons for LRD on traffic samples [6], network spikes can also create a false sense of LRD on packet delay samples [35]. Failing to distinguish and separate irregular events like these may lead to misinterpreted results [35].

To illustrate the effects discussed above, real packet delay values in terms of one-way delay (OWD) traces were collected and are analyzed below. These traces were collected from the Evergrow Traffic Observatory Measurement Infrastructure (ETOMIC) European project [80]. ETOMIC is a measurement infrastructure, which is focused on realizing a pan-european measurement infrastructure, consisting of a number of measurement nodes deployed at selected European locations [50]. ETOMIC offers to the scientific community a measurement platform for conducting multiple types of traffic and network measurement experiments, among them, analysis of packet delay measurements between nodes. In addition, ETOMIC conducts periodic measurement experiments, results of which are provided to the scientific community as open repositories [80].

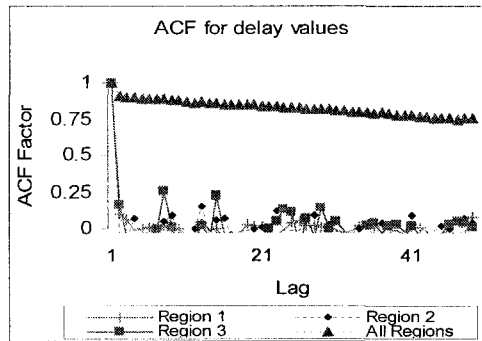
ETOMIC offers a high-precision infrastructure, based on hardware specially created and modified according to the project requirements; such as for example, packet trains may be transmitted with strict timing, with resolutions in the range of nanoseconds. Furthermore, a GPS system is incorporated to the measurement nodes to synchronize the infrastructure to the same reference clock. For further information refer to [50] [80].



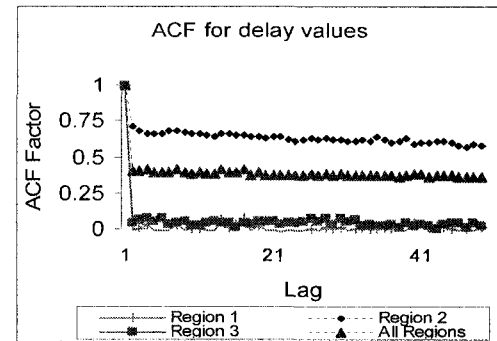
(8.1.a)



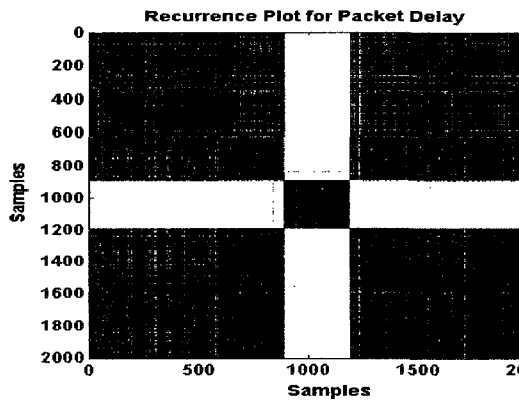
(8.2.a)



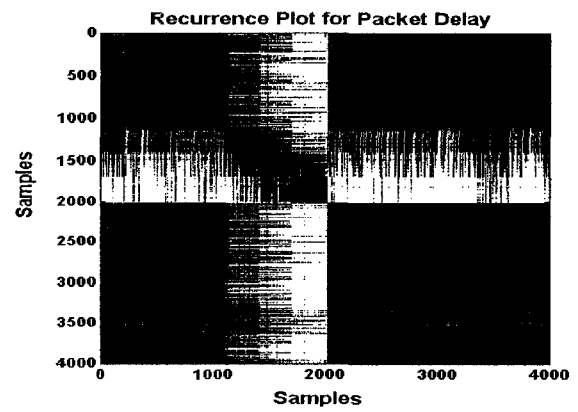
(8.1.b)



(8.2.b)



(8.1.c)



(8.2.c)

Figure 8.1. Packet delay analysis for scenario; 130.206.163.166 → 157.181.172.103, (8.1.a) Packet delay trace, (8.1.b) ACF distribution of packet delay trace, (8.1.c) Recurrence plot of packet delay trace.

Figure 8.2. Packet delay analysis for scenario; 130.206.163.166 → 132.65.240.106, (8.2.a) Packet delay trace, (8.2.b) ACF distribution of packet delay trace, (8.2.c) Recurrence plot of packet delay trace.

Figure 8.1.a shows 2000 OWD samples collected from the ETOMIC repositories. Samples represent an experiment measurement conducted between two nodes in the ETOMIC infrastructure conducted on March, 2006. Packet probes were sent from 130.206.163.166 \rightarrow 157.181.172.103, traversing 18 hops, at 1 pps rate and using 46 byte packet size (entire packet length). Figure 8.2.a shows 4000 OWD samples also collected from the ETOMIC repositories. Packet probes were sent from 130.206.163.166 \rightarrow 132.65.240.106 in August, 2006, traversing 15 hops, at 1 pps and using 46 bytes packet size. Average OWD of the collected samples were 35.33 and 45.44 msec respectively, for the first and second experiments. Note that the offset, minimum value, of the OWD traces has already been removed in Figures 8.1.a and 8.2.a. From Figures 8.1.a and 8.2.a it can be seen that packet delay changes in time immensely due to one or many of reasons explained before. Packet delay traces have been segmented manually in regions according to their observed nature. Each trace has been segmented to three regions and shown in Figures 8.1.a and 8.2.a. It is apparent that region R2 in Figure 8.1.a is the product of a level shift, possibly caused by routing flapping. However, the gradual ramping behavior of region R2 shown in Figure 8.2.a is likely to be due to incremental packet queue congestion. In both cases the statistical properties of the packet delay series are different in each region, and thus different from each the entire trace.

Figures 8.1.b and 8.2.b show the Auto Correlation Function (ACF) distribution of the corresponding packet delay for the entire trace and for each segmented region. In general ACF distribution describes how well a series

matches a time-shifted version of itself as a function of the amount of time shift [17]. Previous studies [59][69] have analyzed the relationship between ACF distributions and probe's sending conditions. For further information on ACF calculation and ACF properties refer to [17].

From Figures 8.1.a and 8.2.a and 8.1.b and 8.2.b. it can be seen that ACF changes drastically due to the presence of region R2 on both traces. ACF distributions of Figure 8.1.b for all three regions denote weak sample autocorrelation. However the entire trace shows clearly LRD of its samples, due to the level shifting. In Figure 8.2.b only region R2 shows clear LRD among its samples. Consequently, the ACF of the entire trace also exhibits LRD, just due to the network stage change observed on region R2.

Figures 8.1.c and 8.2.c show the Recurrence Plots (RP) for the packet delay samples shown in Figures 8.1.a and 8.2.a, respectively, and were obtained using [45]. RP were first introduced in [24].

RP is a two dimensional representation, which denotes all those times at which a state of the dynamical system recurs [24], or in other words, reveals all the times when the phase space trajectory visits roughly the same area in the phase space [45]. Recurrence is a fundamental feature of many non-linear dynamic systems, and has been previously used to study the non-stationarity of processes [24]. Interpretation of RP can be summarized for the study of non-stationarity processes as follows [45].

- ❖ Homogeneity; The process is stationary.
- ❖ Fading to the upper left and lower right corners; Non-stationarity. The process contains a trend or drift.
- ❖ Disruptions (white bands); Non-stationarity. Some states are rare or far from the normal behavior. Transitions may have occurred.
- ❖ Single isolated points; Heavy fluctuation in the process. If only single isolated points occur, the process may be an uncorrelated random or even anti-correlated.

Based on the RP interpretations and by inspecting both Figures 8.1 and 8.2 it can be seen that in fact the white bands shown on both Figures 8.1.c and 8.2.c denote transition stages which require the creation of a new series segment. Also, it can be seen that regions are filled uniformly, which also indicates segments are indeed stationary, with the exception of region R2 of Figure 8.2 in which a packet queue congestion trend is observed.

From the results presented above it can be concluded that, although ACF is believed to be mainly driven by queuing delay [6], and thus it is often used as a measurement of network link congestion [59], this metric may change drastically in time due to other stimulus, such as link failures or routing table update, which may disappear rapidly. Also, it can be seen that contribution of small abnormal events on the observations may alter considerably the ACF distributions of the

entire packet delay trace. Consequently, a false sense of LRD on packet delay samples can be created, which may lead to misinterpreted model results.

8.3 STATISTICAL METHODOLOGIES FOR MODELING NON-STATIONARY END-TO-END PACKET DELTA SERIES.

Auto-Regressive (AR) models is one of the most widely used methods for statistical time series modeling and prediction [10][17]. $AR(p)$ models represent a process in which the observation at time t is a weighted average of the most recent p previous observations in the series [17]. Selection between AR and ARMA models depends on the computational complexity. In general, AR model is much simpler to handle [33] and its performance has been proven to be adequate for many applications [10].

Considering an end-to-end packet delay time series, $\{D(t)\}, d_t$ can be estimated as a linear summation of its previous observations by use of a scalar AR process of order p , $AR(p)$, [17]. and it is given by equation (8.1).

$$d_t = \sum_{i=1}^p a_i d_{t-i} + e_t \quad (8.1)$$

where, e_t is assumed to be a sequence of independent and normal distributed random variables with zero mean and variance σ^2 ($i.d. \sim N(0, \sigma^2)$) [17]. Traditional time series models, such as AR models, focus on modeling systems in which their statistical properties change slightly with time or do not change at

all. Such systems are known as stationary or weakly stationary, and are rare to find or just occur for moderate periods of time on real-life observations.

Nevertheless, dynamics of real systems, such as Internet traffic, tend to change in time according to many factors. Traditional AR models need to be updated recursively in time to reflect such effects. Such models are known as adaptive AR (AAR), in which the autoregressive coefficients, a_k , change in time to capture the dynamics of the system. In general, AAR offers a solution for modeling time variant systems. AAR has been used extensively for modeling time variant systems [5][24][33]. In this context, equation (8.1) can be re-written as;

$$d_t = \mathbf{D}_t^T \mathbf{A}_t + e_t \quad (8.2)$$

where \mathbf{A}_t and \mathbf{D}_t are shown in equations (8.3) and (8.4), respectively. Note that all boldface variables are vectors or matrices.

$$\mathbf{A}_t = (a_{1,t}, a_{2,t}, \dots, a_{p,t})^T \quad (8.3)$$

$$\mathbf{D}_t = (d_{t-1}, d_t, \dots, d_{t-p})^T \quad (8.4)$$

Vector of AR coefficients, \mathbf{A}_t , is no longer time invariant, on the contrary, it changes in time reflecting the system dynamics. It has been previously

demonstrated [30] that in general the evolution of \mathbf{A}_t can be characterized as a first order Markov process, with small changes in the state, as shown below [5][24];

$$\mathbf{A}_{t+1} = \mathbf{A}_t + \mathbf{w}_t \quad (8.5)$$

where \mathbf{w}_t is a zero-mean white noise. Equation (8.5) indicates that the AR coefficients, $a_{k,t}$, change in time in a random walk manner and assume small changes in the state. An alternative is to estimate \mathbf{A}_t by means of prediction techniques based on the system observations. Among them, Kalman Filtering algorithm has been widely used. Kalman Filtering algorithm is an efficient recursive technique which estimates the state of a dynamic system from a series of incomplete and noisy measurements [30]. A basic condition for using the Kalman Filtering algorithm is that the series model has a representation in state-space form, consisting of two joined linear equations; the state equation and the observation equation [30]. For the end-to-end packet delay process, the space equation is represented by equation (8.2) and the state equation is represented by equation (8.5).

In the context of the Kalman Filtering algorithm, the system stage and the AR coefficient's vector, $\hat{\mathbf{A}}_t$, can be recursively estimated as follows [30];

$$\mathbf{P}_{\hat{\mathbf{A}}_{t|t-1}} = \mathbf{P}_{\hat{\mathbf{A}}_{t-1}} + \mathbf{C}_{w_{t-1}} \quad (8.6)$$

$$\mathbf{K}_t = \mathbf{P}_{\hat{\mathbf{A}}_{t|t-1}} \mathbf{D}_t^T \left(\mathbf{D}_t^T \mathbf{P}_{\hat{\mathbf{A}}_{t|t-1}} \mathbf{D}_t + \mathbf{C}_{e_t} \right)^{-1} \quad (8.7)$$

$$\mathbf{P}_{\hat{\mathbf{A}}_t} = \left(\mathbf{I} - \mathbf{K}_t \mathbf{D}_t^T \right) \mathbf{P}_{\hat{\mathbf{A}}_{t|t-1}} \quad (8.8)$$

$$\hat{\mathbf{A}}_{t+1} = \hat{\mathbf{A}}_t + \mathbf{K}_t \left(d_t - \mathbf{D}_t \hat{\mathbf{A}}_t \right) \quad (8.9)$$

where $\bar{\mathbf{A}}_t = \mathbf{A}_t - \hat{\mathbf{A}}_t$ is the estimation error. κ_t is the Kalman Filter gain vector, and C_{e_t} and C_{w_t} denote the covariance of e_t and w_t , respectively. $\mathbf{P}_{\hat{\mathbf{A}}_{t|t-1}}$ is the so called error covariance matrix, which is used to measure the stability of the system. C_{w_t} is estimated as shown in equation (8.10) by means of an updated coefficient, UC , and knowledge of the system's previous state [24]. For further information on C_{w_t} estimation refer to [24].

$$C_{w_t} = \frac{UC * \text{trace}(\mathbf{P}_{\hat{\mathbf{A}}_{t-1}})}{p} \quad (8.10)$$

In [63], a methodology for selecting UC and the model order, p , is proposed, such as the estimated AAR model best describe the analyzed process. For further information on UC estimation refer to [63]. Finally, it can be seen that the

estimation of the state noise covariance matrix, C_{w_t} , presented in Equation (8.10) is based on the prediction error from a single prediction and therefore statistically unreliable [21]. For C_{w_t} to be reliably estimated, it is necessary to calculate the prediction error over a window of samples [21]. An alternative approach is to smooth C_{w_t} , as shown below, where α is the smoothing parameter.

$$C_{w_t} = \alpha * C_{w_{t-1}} + (1 - \alpha) * C_{w_t} \quad (8.11)$$

8.4 A COMPUTATIONAL EFFICIENT METHOD FOR SEGMENTING NON-STATIONARY PACKET DELAY SERIES

Modeling packet delay by means of Kalman Filtering algorithm requires to update the recursive equations presented in Section 8.3 for any given packet delay sample. Although this approach yields an accurate up-to-date model of the trace dynamics, it can be seen that it produces $p+1$ parameters for any given sample. Collecting and storing this amount of data at a packet-by-packet granularity may be impractical and unnecessary for offline analysis, and extremely computational demanding for online modeling, and online network congestion control mechanisms.

However, considering that it is the goal of this analysis to fit the packet delay dynamics of the samples within each stationary segment into AR model, and not to model extensively the packet-by-packet delay behavior, it is considered a more efficient approach to report and store only AR models that

represent significant changes on the packet delay dynamics. In this chapter we propose to achieve this by employing an online modeling and segmentation algorithm, which compares the distance [9] of the current AR model against the AR model that represents the previous stationary segment detected. If distance is small, the current sample belongs to the previous stationary AR model, if not a new segment needs to be created. Notice that the AR model of each segment governs the dynamics of the packet delay samples within the segment.

Stationarity of the observations depends on several factors, such probe's traffic conditions, network link congestion status, and network changes, among others. It is expected that weakly stationary packet delay traces need few segments, or maybe just one. However, non-stationary packet delay traces may need many more.

In the past, previous efforts on online [36] and offline [58] segmentation of time series have been conducted. Techniques based on the statistical properties of time series [29][36][44][58], power spectral analysis [1], and tracking the roots of ARMA processes [54], among others, have been proposed. Many of them are used in application such digital signal processing, voice recognition [4], and biometric related analysis [54], in which level shifting and peak detection is key for the application.

However on the study of Internet end-to-end packet delay, the goal of segmentation is to reflect significant changes on the probe's packet delay dynamics, which indeed reflect variations on the network stage, and not to detect

instantaneous changes on the sample trace. Thus, in this chapter we propose an online methodology for packet delay series segmentation which requires storing low amount of data and low computation overhead. Memory storage capacity requirements and computational overhead produced by the proposed algorithm are analyzed and quantified in Section 8.4.2.

The proposed method uses the Divergence-Test [9], which is based on measuring and monitoring the distance between two models, Φ_0 and Φ_1 . Figure 8.3 shows Φ_0 and Φ_1 . Φ_0 and Φ_1 are the two AR models to be compared, and are composed by n and L samples, respectively, where $n \leq L$. In the context of the proposed algorithm Φ_1 represents a subset of the first n samples of Φ_0 , as shown on Figure 8.3.

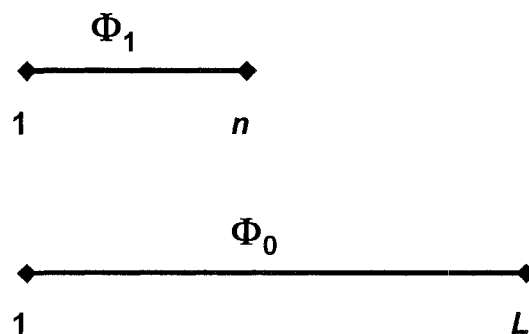


Figure 8.3. Location of Φ_0 and Φ_1 models for the divergence test method.

The distance measure is derived from the cross entropy between the conditional distributions of Φ_0 and Φ_1, ζ_k , which in the Gaussian case is given by [4][9];

$$\zeta_k = \frac{1}{2} \left(2 \frac{e_{0,k} e_{1,k}}{\sigma_1^2} - \left[1 + \frac{\sigma_0^2}{\sigma_1^2} \right] \frac{e_{0,k}^2}{\sigma_0^2} + \left[1 - \frac{\sigma_0^2}{\sigma_1^2} \right] \right) \quad (8.12)$$

where $n < k \leq L$, σ_0^2 and σ_1^2 are the variance of Φ_0 and Φ_1 , respectively. $e_{0,k}$ and $e_{1,k}$ are the forward prediction errors of the packet delay series using the Φ_0 and Φ_1 model, respectively, (see Equation (8.1)). Under the hypothesis that Φ_0 and Φ_1 represent the same process, ζ_k has zero conditional drift. However, if a change is detected, ζ_k will show a strong negative conditional drift [9]. Change detection is identified when the long term and short term models, Φ_0 and Φ_1 respectively, disagree in the sense of the cumulative sum statistics [4], as shown below;

$$\hat{\zeta}_L = \sum_{k=n}^L (\zeta_k + \delta) \quad (8.13)$$

where δ is a positive bias, used to generate a positive drift on the modified cumulative sum. However when changes occur, a strong negative drift on $\hat{\zeta}_L$ is expected. For further information refer to [4]. In the following subsection a methodology for online segmentation, in the context of packet delay analysis, is presented.

8.4.1 AN ONLINE PACKET DELAY SEGMENTATION ALGORITHM

Considering an end-to-end packet delay time series, $\{D(t)\}$, the proposed algorithm starts by collecting the first n samples. Using the initial n samples the first optimal AR model is calculated, $AR_{\{n,p_1\}}^1$, using classic time series techniques [17][69].

Where p_1 is the order of the fitted first AR model. $AR_{\{n,p_1\}}^1$ is the statistical model that represents Φ_1 , see Figure 8.4, and it is assumed to be a stationary process. The non-stationary case is studied in Section 8.4.3. After that, the buffered n packet delay samples are flushed out of memory. Next, the recursive Kalman Filtering algorithm presented in Section 8.3, is used for estimating the AR model that governs the system for samples k , $k > n$, which is denoted by $AR_{\{k,p_e\}}^e$. $AR_{\{k,p_e\}}^e$ represents the evolution in time of $AR_{\{n,p_1\}}^1$, and it is the statistical model that represents Φ_0 , see Figure 8.4. Stationarity of $AR_{\{k,p_e\}}^e$ is assured by the stationarity of Φ_1 , and as long as a process change is not detected. Subsequently, $\hat{\xi}_k$ is calculated recursively with every observed sample to measure the distance between Φ_0 and Φ_1 . Notice that in the context of the described packet delay segmentation algorithm, as shown in Figure 8.4, Φ_0 and Φ_1 are conformed by n and k samples, respectively. Thus $\hat{\xi}_k$ is the modified cumulative sum from sample n to k .

Equation (8.14) is then used for identifying the initial change point, r , where the two stationary AR models diverge from each other, and thus a new segment is created [4], see Figure 8.4.

$$\max_{\substack{r > n \\ m > n}} (\hat{\zeta}_r - \hat{\zeta}_m) \geq \lambda \quad (8.14)$$

where λ is the selected threshold for detecting the beginning of a new segment, $1 \leq m < r$, and $\hat{\zeta}_m$ is the previous maximum peak value of $\hat{\zeta}_k$ occurred before $\hat{\zeta}_r$. After r is identified, the first segment is created in which $AR_{\{n, p_1\}}^1$ governs the behavior of all k samples, $1 \leq k < r$. Notice that, $\hat{\zeta}_m$ has to belong to the segment which is analyzed [4]. The second segment is then commenced, which has r as its starting point. Considering that the first and second segments are uncorrelated to each other [4], n new samples need to be collected starting from sample r in order to create the initial optimal AR model for the second segment, Φ_3 . Φ_2 is then created and updated in the same recursive manner done for the first segment, see Figure 8.4. Distance between the new two models, Φ_3 and Φ_2 , is again compared online using $\hat{\zeta}_k$ to identify the ending of the second stationary segment. This process continues for the duration of the experiment. Notice that regardless of the degree of sample auto-correlation within each stationary segment, it can be seen that any segment is uncorrelated to its adjacent ones, since this is basically the underlying reason for the series

segmentation [4]. Figure 8.4 shows graphically the proposed packet delay segmentation methodology.

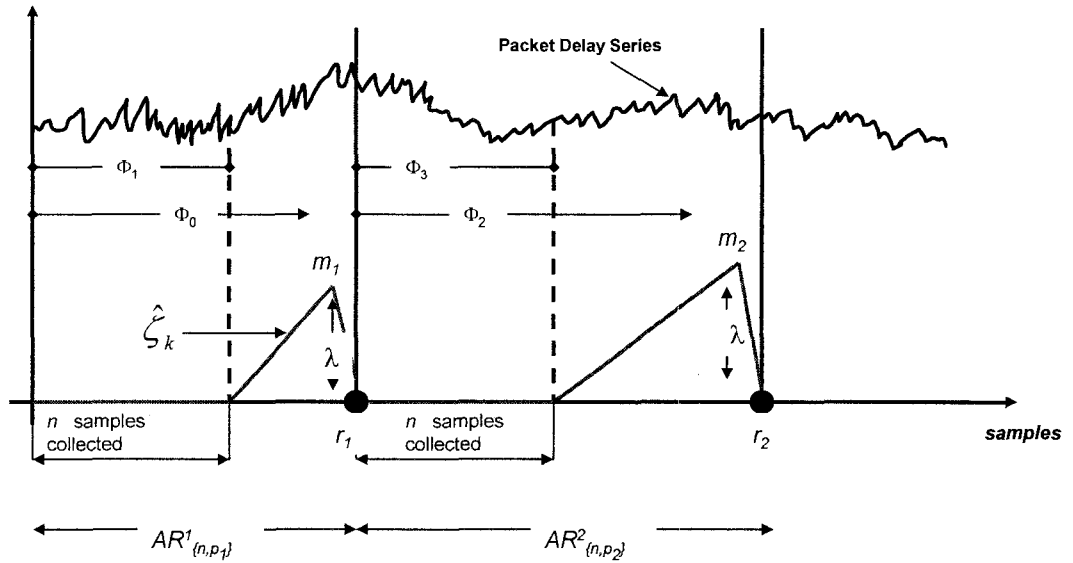


Figure 8.4. Packet Delay Series Segmentation Methodology.

Notice that since $\hat{\zeta}_k$ is related to the change in time of the AR model which governs the packet delay dynamics of a specific segment, it can be expected that this one change according to the degree of non-stationarity of the samples. However, only significant changes on the network system dynamics should be responsible of triggering the segmentation algorithm. Although equation (8.14) specifies the threshold for negative drifts responsible of detecting network system dynamics changes, it can be noticed that this threshold can be reached either by; sudden spikes on the trace, or significant changes on the network

system dynamics, either abruptly or gradually. Sudden spikes should be avoided by the algorithm, to the extent possible. Moreover spikes occurring apart of each other within the same segment can make equation (8.14) reaches λ , due to the accumulative behavior of $\hat{\zeta}_k$ and the condition of equation (8.14), and it should also be avoided. The former phenomena can be evaded by analyzing the negative slope of the drift when threshold of equation (8.14) is reached. Sudden spikes are associated with large negative slopes, and can be quantified by the small number of samples between r and m , see Figure 8.4. Such condition can be expressed as; $\mathcal{E} \leq (m-r)$, where \mathcal{E} is a threshold selected to avoid algorithm detecting sudden isolated spikes. Conversely, the latter phenomena can be evaded by selecting m as the previous maximum peak value of $\hat{\zeta}_k$ occurred before r , see Figure 8.4, and thus eliminating the contributions of previous spikes observed on the delay trace.

It can be seen that the proposed online segmentation algorithm only adds small additional computation overhead on top of the recursive fitting model procedure explained in Section 8.3. This can be explained since the updated model, Φ_0 , see Figure 8.3 and first segment of Figure 8.4, is already generated recursively by the Kalman Filtering algorithm. Thus no additional computation overhead is needed when finding the parameters related to Φ_0 needed for equation (8.12). Note also that none of these parameters are kept in memory after condition of equation (8.14) is satisfied. However, parameters related to the initial model, Φ_1 , needed in equation (8.14) requires an additional small

computation overhead. For instance, σ_1^2 for Φ_1 is calculated when the initial n samples are fitted into the $AR_{\{n,p_1\}}^1$ model. In addition, Φ_1 model has to be used against the incoming packet delay samples, k ($k > n$), when executing the online calculation of $e_{1,k}$. Calculation of $e_{1,k}$ is done by adding equation (8.1), containing the Φ_1 model, into the recursive Kalman Filter equations. Storage capacity requirements and computational overhead produced by the proposed algorithm are analyzed and quantified in Section 8.4.2.

Note that Φ_0 and Φ_1 have to be both stationary to be used on equation (8.12). Since Φ_0 represents the evolution in time of Φ_1 , its stationarity is conditional to the one of Φ_1 , as long as a process change is not detected by $\hat{\zeta}_k$. However stationarity of Φ_1 will depend on the starting point of the segment, the probe's sending condition, and the value of n , among other factors. Section 8.4.3 will explain this phenomenon in detail. In addition, effectiveness of the proposed algorithm depends on its settings. Therefore a tradeoff exists among accuracy, computational overhead and memory storage requirements according to the settings employed. This will be explained in more detail in Section 8.4.3.

8.4.2 MEMORY STORAGE SAVINGS VS. COMPUTATIONAL OVERHEAD

In this subsection the memory storage savings achieved by using the proposed packet delay segmentation algorithm are quantified and compared against the computational overhead this generates.

In the context of packet delay analysis, offline modeling techniques require storing the entire packet delay trace prior to modeling the dynamics of the network system, either by segmenting the series or modeling it entirely. Such an approach may require enormous storage requirements depending on the length of the experiment and the probe's sending conditions employed. However, the proposed segmentation algorithm reduces memory storage requirements compared to such an approach.

To quantify this remark, let's considering the packet delay trace shown on Figure 8.4 and explained on Section 8.4.1. First segment starts at sample 1 and ends at sample r_1 , also considering that the first n samples of the first segment are used to determine the initial system model, Φ_1 . After Φ_1 is determined, and characterized by p_1+1 parameters, the initial n samples are flushed out of memory. The embedded Kalman Filtering algorithm is then used to update the state of Φ_1 recursively at any given iteration, k ($n < k \leq r_1$), such an evolution is represented by Φ_0 . Following the algorithm condition presented on Section 8.4.1, the distance between Φ_0 and Φ_1 is measured for any given iteration, k . Due to the first segment ends at sample r_1 , see Figure 8.4, it can be safely assumed that the distance between Φ_0 to Φ_1 is minimum, see Section 8.4.1. Thus it also saved to conclude that Φ_1 can characterize all the r_1 initial samples of the trace, and therefore only the p_1+1 coefficients need to be kept in memory.

This process continues in the same manner for the consecutives segments for the duration of the observations. In general, memory storage savings

compared to storing the entire packet delay trace can be quantified by equation (8.15);

$$perc_memory_storage_savings = 1 - \left(\frac{\sum_{i=1}^s (p_i + 1)}{\sum_{i=1}^s (r_i)} \right) \quad (8.15)$$

where s is the total number of segments in which the delay traces is fragmented to, and r_i is the number of packet delay samples on segment i . Notice that $\sum_{i=1}^s (r_i)$ represents the entire packet delay sample trace. As it can be seen, memory storage savings depends on the total number segments generated, which indeed is based on the degree of non-stationarity of the packet delay trace and the algorithm settings employed.

The proposed algorithm creates an initial system model, Φ_1 , at each segment with the first n samples of the segment. Creation of Φ_1 using traditional $AR(p)$ fitting methods, such Yuke Walker, need to be done at every single new segment [17]. Such fitting methods can demand a significant amount of computational overhead compared to the Kalman Filtering recursive equations [17]. For instance, Yuke Walker equations need to calculate set of $(p+1)$ correlation factors that then are used to solve $(p+1)$ linear equations, when fitting a system model [17]. Moreover, the proposed segmentation algorithm adds additional computational overhead from calculations used on equations

(8.12), (8.13), and (8.14). Although these equations demand low computational overhead, they need to be executed $\sum_{i=1}^s (r_i - n - 1)$ times within segment i .

In general, it can be concluded, that the proposed packet delay segmentation algorithm trades off accuracy with computational overhead and memory storage requirements. For stationary packet delay trace scenarios, memory storage savings can be considerable, and computational overhead can be considered to be small. Conversely, non-stationarity packet delay traces scenarios may have the opposite effect depending on the degree of non-stationarity. Although computational overhead and memory storage savings are sacrificed when non-stationary traces are analyzed, this is needed to accomplish accurate representation of the network system dynamics.

8.4.3 SEGMENTATION ALGORITHM SETTINGS

In this subsection, the settings for the proposed segmentation algorithm are specified according to the probe's sending conditions and the desired granularity of the results.

Previous efforts have analyzed the effect of probe's sending rate and probe's packet size on packet delay autocorrelation [59][69]. In general, higher-rate probe flows tend to generate stronger autocorrelation on packet delay samples than lower-rate ones, as they occupy a higher fraction of link bandwidth [69], and thus generate larger packet queuing delays. However this phenomenon can also be attributed to the fact that high rate probes monitor the stages of the

network in at a more granular level than do low bit rate ones, regardless of the fraction of bandwidth they occupy on the network links. Thus high rate probes are more likely to capture sudden spikes and irregularities that occur in the network, and as was mentioned on Section 8.2 these events may cause false sense of LRD on packet delay samples. In both cases, the algorithm proposed in Section 8.4.1 segments the packet delay trace into as many stationary segments as needed, in which each segment represents a different state of the network system. However, in the context of the segmentation methodology proposed in this paper, probe's sending rate may affect the creation of segments Φ_0 and Φ_1 . This is due to the fact that Φ_1 is formed by the first n packet delay samples collected right after a new segmented is detected. Classic statistical methods require n to be large enough for Φ_1 to reflect accurately the stage of the network system and to generate proper sample distributions. However, collecting large amount of packet delay samples can take different amounts of time depending on the probe's sending rate and probe's packet size. For instance, consider the case of small packet size probes sent at high rates; it can take a few seconds to collect the required n samples. On the contrary, the same amount of samples can take several minutes to be collected for low sending rates probes, under the same network circumstances and using the same probe's packet size. Thus, it can be seen that granularity of the segmentation depends on the probe's sending conditions, and these have to be selected according to the expected results. However, probe's sending conditions settings are not suggested in this paper and

are left to be chosen by the researcher. Moreover, experiment results employing different sending conditions are presented and analyzed in Section 8.5.

Conversely, settings of the segmentation algorithm, such as selection of $\delta, \lambda, \varepsilon$, and n values, needs to be specified in fine ranges to assured its performance results. Selection of δ and λ values are critical for the algorithm performance. In [4] it has been suggested to select δ on the $\{0-1\}$ range. δ has to be large enough, such as $\hat{\zeta}_k$ shows always a positive drift when both Φ_0 and Φ_1 represent similar processes. However, choosing δ too large can cause the drift of $\hat{\zeta}_k$ to be too positive and thus equation (8.14) may fail detecting significant divergence between Φ_0 and Φ_1 . λ has also to be chosen in such a way that only significant changes trigger equation (8.14). Considering that the goal of the packet delay segmentation algorithm is to reflect significant changes on the probe's packet flow dynamics and not detect instantaneous changes on the trace, it is suggested to choose a large enough value of λ , as oppose to ones used on more rigorous trace segmentation studies, such as that in [4]. In addition ε has also to be selected in such a way that the algorithm is not triggered, to the extent possible, by isolated spikes on the packet delay trace. Large values of ε are suggested for this matter, however a relationship between ε and λ can be foreseen to avoid large negative slopes on $\hat{\zeta}_k$, see Figure 8.4. From experiment results it has been found that better performance of the algorithm is reached when selecting ε values in the range; $40^\circ \leq \tan^{-1} (\lambda/\varepsilon) \leq 50^\circ$.

Assuring the stationarity of Φ_1 is also a crucial condition for the algorithm. However, by following the segmentation algorithm steps presented on Section 8.4.1, Φ_1 may represent two or more uncorrelated sample regions, and thus stationarity of Φ_1 may not be achieved.

To clarify the previous statement, let's apply the algorithm steps presented on Section 8.4.1 into the measurement showed on Figure 8.1.a. Let's consider $n=1000$, and Φ_1 starting at sample 1. It can be seen that Φ_1 in fact is a non-stationary model, due to the shift level occurred at sample 900, and therefore it can not be used by the proposed segmentation algorithm. To overcome this and the previously mentioned phenomenon, we propose to use $n = 1000$ for any probes' sending condition, however stationarity of Φ_1 has to be tested before using it in the proposed algorithm. This test will consist on finding the change point(s), r , ($1 \leq r \leq n$) in the same manner as explained in Section 8.4.1, but in a static manner. As shown on Figure 8.3, the mentioned test will create a large and a small model, formed by n and $n/2$ samples, respectively. Note the small model is a subset of the large one as explained on Section 8.4. The distance of these models is measured as described on Section 8.4.1. If a change point(s) are found, Φ_1 stationarity test fails, and two or more stationary sub-segments are created, which replace the original Φ_1 segment.

Next, the algorithm is forced to collect the following n packet delay samples to create a new Φ_1 , and its stationary tested before using it in the proposed algorithm. Based on previous analysis of the nature of packed delay

process [35], it can be concluded that irregularities in packet delay samples, such as the one observed at sample 900 in Figure 8.1.a, are unlikely to happen consecutively in the small time windows, such as the time window used for collecting n packet delay samples. Thus it can be concluded that if Φ_1 is found to be non-stationary, only few uncorrelated stationary segments will be obtained from the sub-segmentation of Φ_1 . Notice that, testing the stationarity of Φ_1 , may add some additional computational overhead time into the algorithm, and the online behavior of the algorithm can be delayed temporarily. However, the proposed algorithm recovers from this stage rapidly. Φ_2 does not need to be tested for stationarity if Φ_1 is stationary.

Algorithm Settings					
δ	λ	ε	α	n	UC
0.25	150	$[0.84\lambda - 1.19\lambda]$	0.0001	1000	0.05

Table 8.1. Algorithm settings used on segmentation analysis.

Based on this, Table 8.1 shows the algorithm settings chosen and employed for the analysis done in this paper. In Section 8.5 performance results of the algorithm using these settings is presented. Section 8.5 also presents a performance comparison analysis for different sets of settings.

Figure 8.5 summarizes the procedure of the proposed online packet delay segmentation algorithm. Figure 8.5 presents a flow diagram of the mentioned algorithm.

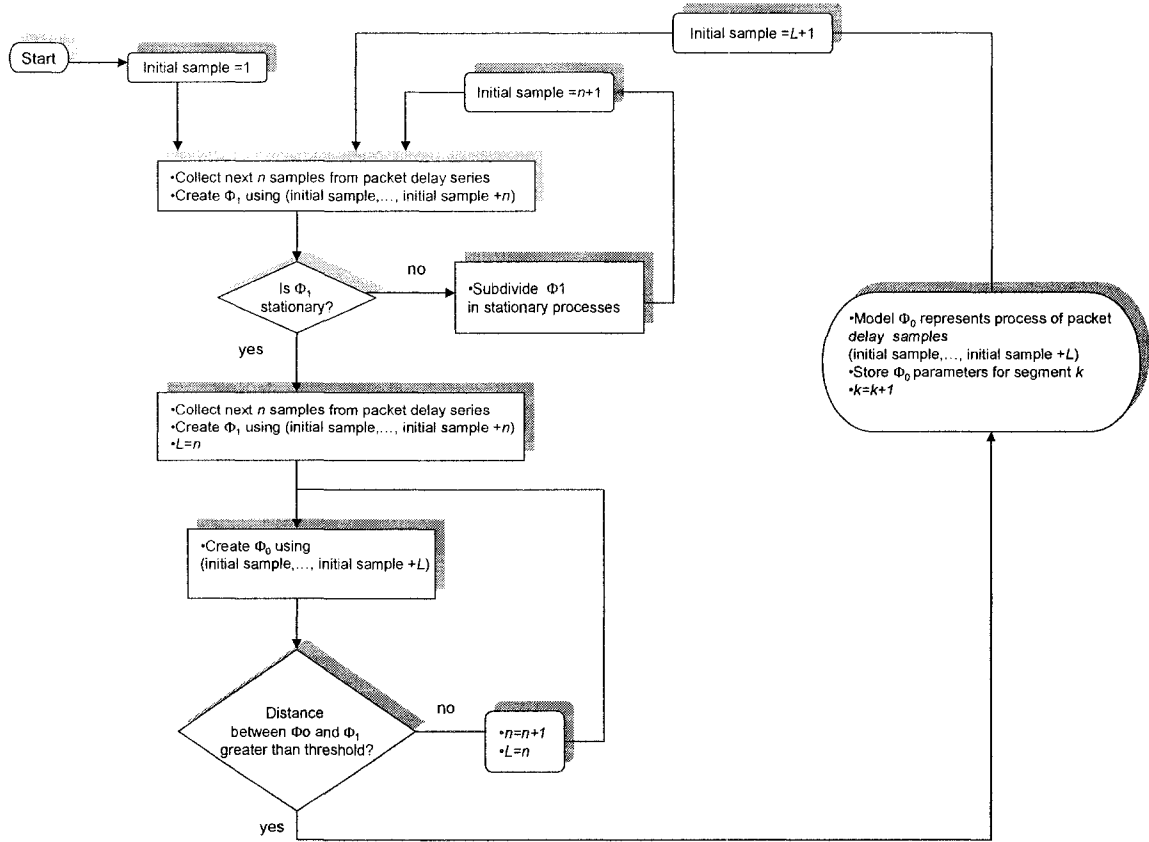


Figure 8.5. Flow diagram of online packet delay segmentation algorithm.

8.5 PERFORMANCE RESULTS

In this section performance of the proposed online modeling and segmentation methodology is tested against real end-to-end packet delay traces. Packet delay traces were collected by conducting customized experiments using the ETOMIC infrastructure [80]. ETOMIC offers to the research community a vast flexibility for setting network traffic experiments over its infrastructure,

however experiments are constrained to one hour duration and probe's sending rate up to 1Mbps, to avoid network link congestion and allow multiple concurrent user experiments [50] [80]. Based on this, a set of experiments were conducted using two ETOMIC nodes under different traffic sending conditions. Pamplona-Spain(UNAV-130.206.163.166) and Jerusalem-Israel(HUJI-132.65.240.106) were the two selected nodes. Packet flows were sent in both direction, UNAV→HUJI and HUJI→UNAV, traversing 14 hops on both cases. Flows were generated using 64bytes packet size and a set of three sending rates were used; 100packets per second (pps), 1000pps and 2000pps. Each scenario was run several times a day and at different times of the day, aiming of capturing different packet delay characteristics induced by various degrees of network link utilization. Experiments were conducted during consecutive days between July-August, 2007, to achieve consistency on the results. Performance results of the modeling and segmentation algorithm were generated using the source code presented on appendix C.1

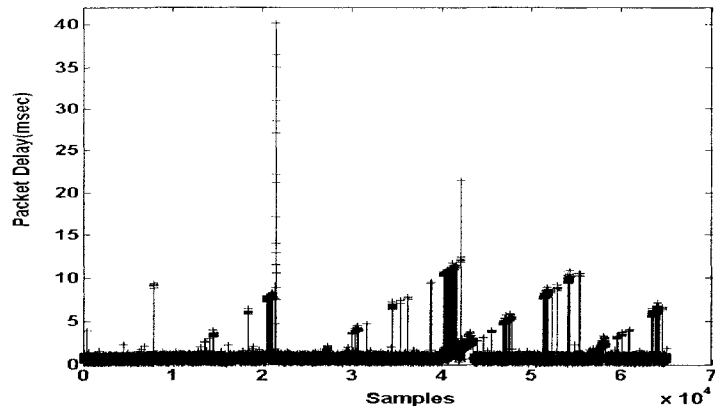
Packet Delay Characteristics and UTC time for the HUJI→ UNAV scenar				
	UTC Time	Average Delay(msec) / Delay Variance		
		100pps	1000pps	2000pps
Run 1	19:00 UTC 2007	49.21/1.25	49.11/ 0.04	48.99/ 0.09
Run 2	21:00 UTC 2007	49.11/0.14	49.10/ 0.04	49.01/ 0.08
Run 3	23:00 UTC 2007	49.10/0.08	49.25/0.05	49.14/ 0.11
Run 4	01:00 UTC 2007	49.17/0.05	49.27/0.07	49.06/ 0.12
Run 5	03:00 UTC 2007	49.28/0.06	49.25/0.04	49.07/ 0.10
Run 6	05:00 UTC 2007	49.29/0.05	49.24/0.14	49.03/ 0.10
Run 7	07:00 UTC 2007	49.12/0.08	49.14/0.51	50.67/ 0.22
Run 8	09:00 UTC 2007	49.23/ 0.05	49.01/0.08	48.63/ 0.30
Run 9	11:00 UTC 2007	49.22/0.09	49.05/0.16	48.60/ 0.13
Run 10	13:00 UTC 2007	49.06/0.13	49.35/1.12	48.69/ 0.07

Table 8.2. Packet Delay Characteristics and UTC times for the HUJI→ UNAV set of experiment results.

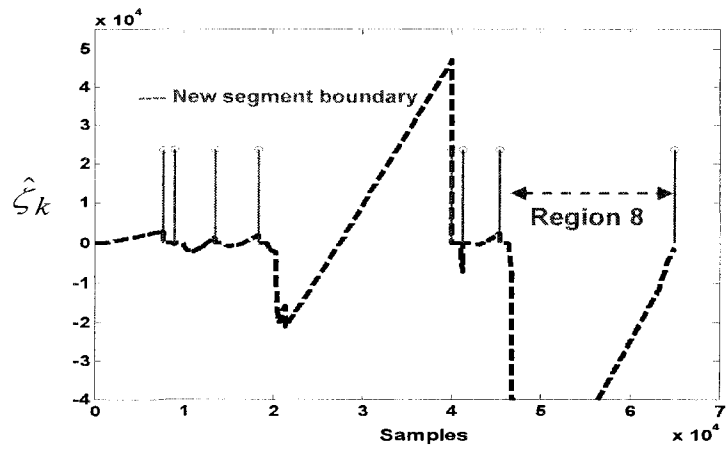
Table 8.2 shows the approximated UTC time for each scenario run on the HUJI→UNAV experiment. In addition Table 8.2 shows the observed average and variance of each packet delay trace collected in each scenario. From Table 8.2 no clear peak delay hour can be identified, however analysis of individuals traces show that high variance packet delay traces are mainly encountered on runs{1,8,9 and 10}. Although a large amount of data was collected through the set of mentioned experiments, only selected packet delay traces were chosen on this research to test the performance of the proposed mechanism. Criterion used for choosing packet delay traces is based on demonstrating the algorithm performance under different packet delay dynamics, induced by various degrees of link congestion.

In Figure 8.6, performance of the modeling and segmentation algorithm is analyzed by using a trace of packet delay samples from the HUJI→ UNAV

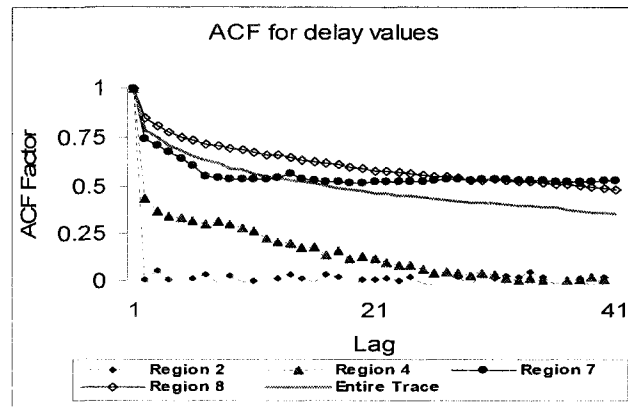
experiment under a sending rate condition of 100pps. Figure 8.6.a shows 65,000 packet delay samples collected from Run 1. Notice that the offset, minimum value, of the packet delay trace has already been removed on Figures 8.6.a, as well as on the rest of packet delays traces shown in this section. As it can be seen from Figure 8.6.a, dynamics of selected packet delay trace vary in time, showing stationary regions followed by sudden spikes.



(8.6.a)



(8.6.b)



(8.6.c)

Figure 8.6. Segmentation analysis of Packet Delay trace for Run1 HUJI→ UNAV experiment using 100pps, (8.6.a) Packet delay trace, (8.6.b) Segmentation process of the packet delay trace, (8.6.c) ACF distribution of packet delay.

Figure 8.6.b shows $\hat{\zeta}_k$, in dotted lines, for the corresponding packet delay traces (see equation (8.13)). Figure 8.6.b also shows the segments created using the proposed algorithm; solid lines represent the end of a segment and the beginning of the next one. As it can be seen, the proposed algorithm identifies, in an online manner, stationary segments on the observed delay trace, in which $\hat{\zeta}_k$ represent shows positive drifts. However when non-stationarity is detected, a strong negative drift is perceived on $\hat{\zeta}_k$ (see Section 8.4.1).

Notice that the proposed segmentation algorithm clearly specifies that $\hat{\zeta}_k$ needs to comply with the conditions mentioned on see Section 8.4.1, when triggering the trace segmentation. Negative drifts on $\hat{\zeta}_k$ that do not comply with these specifications will not dictate trace segmentation, as it can be seen on Figure 8.6.b.

Figure 8.6.c shows the ACF distribution of the entire packet delay sample trace, together with the ones generated by some of the segmented regions. Numbering of the segmented regions starts from the left hand side. From here, it can be seen that in general ACF distribution of the entire trace differs from the distributions of each individual segment. This again is proof that ACF distribution of a sample packet delay trace does not always represents the dynamics of the trace at any given time, in addition that spikes on the packet delay samples may generate a false sense of LRD on the observations. Sample autocorrelation of the selected regions shown on Figure 8.6.c can be easily associated with their corresponding packet delay dynamics shown on Figure

8.6.a. In a nutshell, the performance results shown on Figure 8.6 proves that the proposed algorithm does not act as change point detection or spike detection mechanism, since instantaneous changes of the packet delay do not trigger the segmentation. On the contrary, it segments the packet delay trace based on the non-stationarity of the observations. This can be appreciated on segmented region 8 of Figure 8.6.a, for instance, in which numerous spikes are observed but none of them trigger the algorithm. Although $\hat{\zeta}_k$ decays at the beginning of region 8, as result of the spikes, it can be seen that it rapidly recovers from this stage and shows a positive drift, which denotes that observations on this region are indeed stationary. Stationarity in this case is represented by repeated similar irregularities on the observations, which make samples on this region more auto-correlated than the ones on the other segmented regions, see Figure 8.6.c.

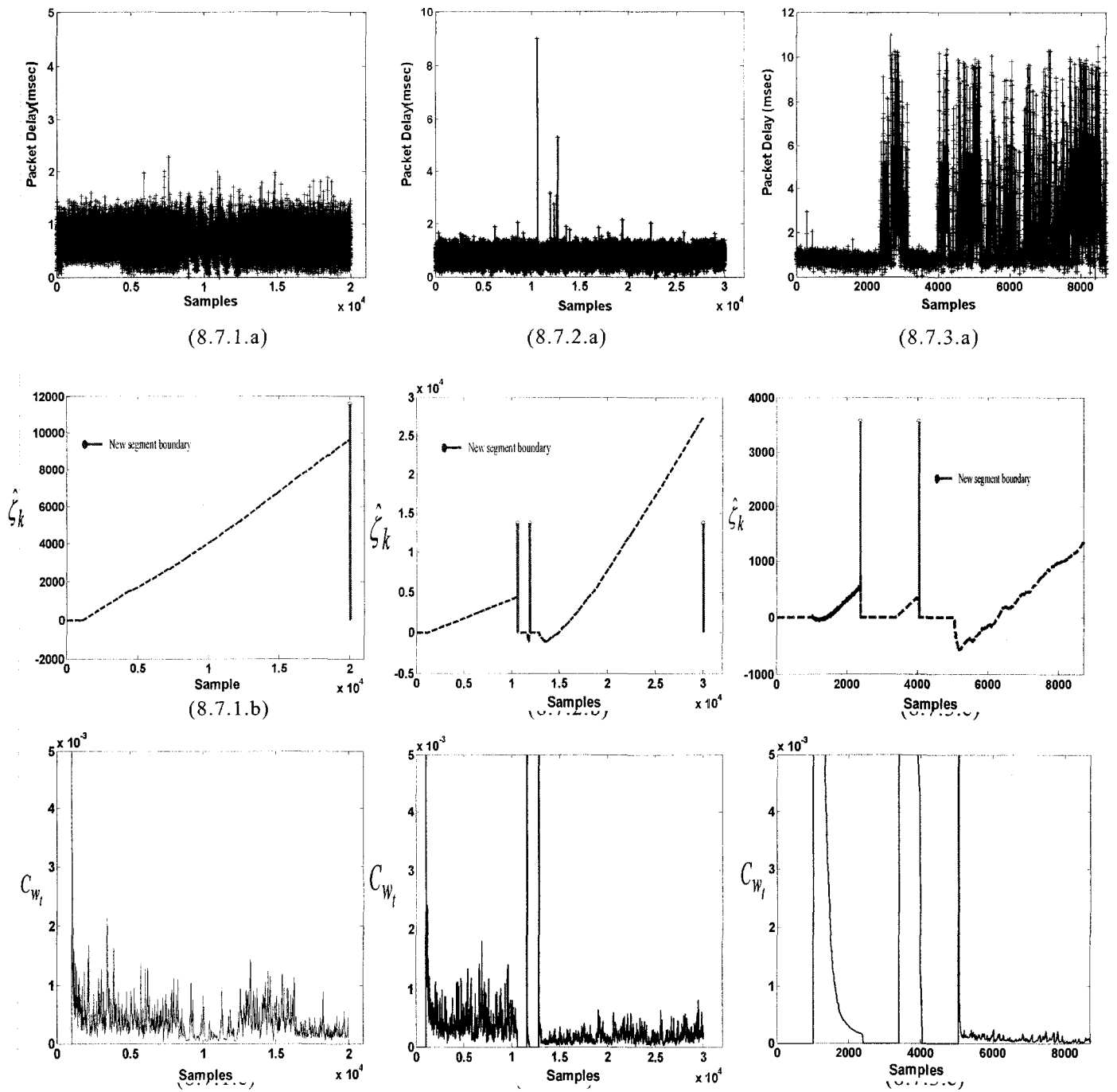


Figure 8.7. Segmentation analysis of packet delay trace for HUJI → UNAV experiment, (8.7.1.x) Run 4 using 2000pps, (8.7.2.x) Run 4 using 100pps, (8.7.3.x) Run 10 using 1000pps.

Figure 8.7 shows the performance analysis of the proposed modeling and segmentation algorithm under three packet delay traces from the HUJI→UNAV experiment, using three different sending rate conditions. Figures 8.7.1.a and 8.7.1.b show 20,000 packet delay samples, and their corresponding $\hat{\zeta}_k$, respectively, for Run 4 of 2000pps scenario. Due to the stationarity observed on the packet delay samples, Figure 8.7.1.b shows that not segmentation was needed. Figure 8.7.1.c shows the corresponding state noise covariance matrix, C_{w_t} , for the entire packet delay trace. In the context of Kalman Filter algorithm, C_{w_t} has been previously used to analyze changes on the system state [58], see equation (8.5). In addition C_{w_t} has been estimated recursively by means of $P_{\hat{A}_{t-1}}$, which it is associated to the stability of the system, see equation (8.10). C_{w_t} starts after the first n samples of each segment have been collected with the initial value of UC and varied in time according to the fluctuations of $P_{\hat{A}_{t-1}}$. From Figure 8.7.1.c it can be seen that statistical AR model used to capture the corresponding packet delay dynamics stays in a moderate narrow invariant stage, which denotes stationary on the observations, a fact that is also corroborated by the smooth positive drift observed on the corresponding $\hat{\zeta}_k$.

Figures 8.7.2.a, 8.7.2.b, and 8.7.2.c show the same analysis for 30,000 packet delay samples collected from Run 4 of the 100pps scenario. Three segments were created, which indeed were generated due to irregular spikes on the observations. Notice that although the proposed algorithm avoids, to the

extent possible, being triggered by instantaneous changes, spikes shown on Figures 8.7.2.a will create a false sense of LRD on the observations if not identified, due to their intensity and the stationarity of the remaining samples. Corresponding c_{w_i} shows again the segmented regions are stationary, however different degrees of stationarity can be perceived at each segmented region, a fact that is corroborated by $\hat{\xi}_k$ on Figure 8.7.2.b, in which each segment shows different positive slopes.

Finally in Figures 8.7.3.a, 8.7.3.b, and 8.7.3.c 9,000 packet delay samples collected from Run 10 of the 1000pps scenario are used to analyze the performance of the proposed mechanism. Packet delay samples in this case present higher variance than that in two previous scenarios; however this does not seem to affect the segmentation process. Three segments were created using the proposed mechanism, dynamics of packet delay samples at each segment vary significantly among each other. By analyzing the corresponding packet delay dynamics, $\hat{\xi}_k$, and c_{w_i} of each segment separately it can be concluded that segmentation in each region obeys different criteria. For instance, first region starts with low variance delay samples, followed by a train of high variance delay samples. Such irregularities are responsible of changing the process dynamics and thus trace segmentation is enforced. Second region starts when spikes disappear and low variance samples show up, again the region is segmented when a train of highly variance delay samples appears. In third region a peculiar phenomenon is observed, this region is started with highly varying

delay samples, however dynamics of the samples are observed to be consistent during a large number of samples and thus none segmentation is needed. Characteristics of $\hat{\zeta}_k$, and c_{w_i} denotes samples stationarity in each segment, even on the third one.

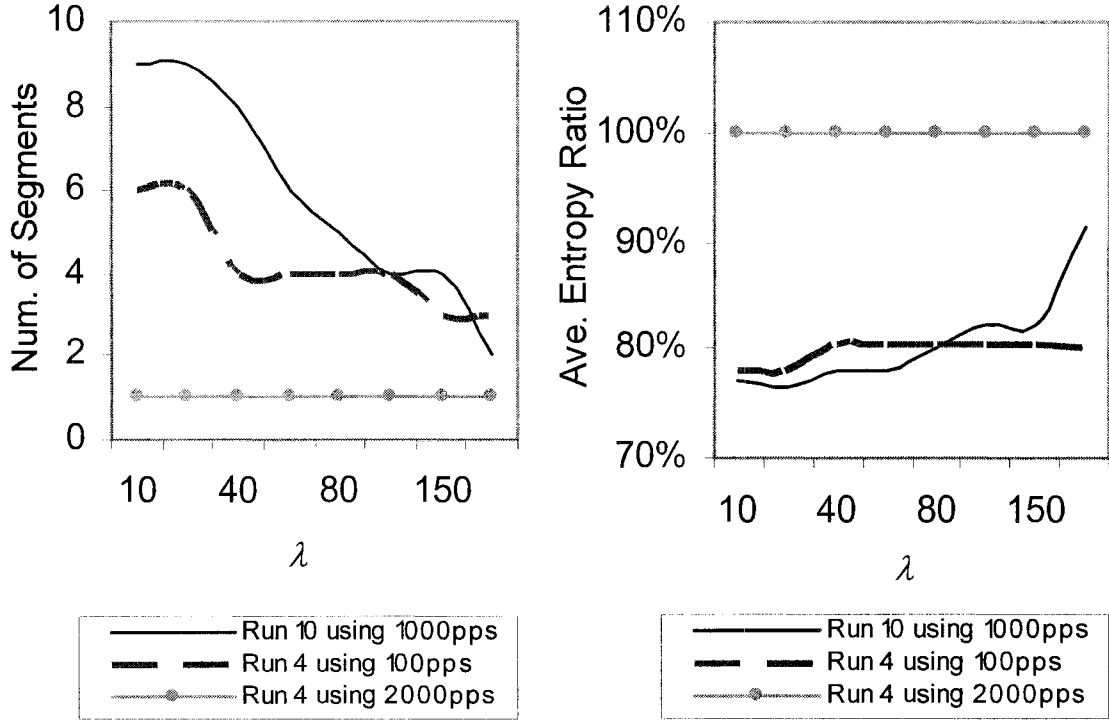


Figure 8.8. Performance of the proposed algorithm under different values of λ , (8.8.a) Number of segments vs. λ , (8.8.b) Average entropy vs. λ .

In Figure 8.8 performance of the proposed algorithm is tested for different values of λ using traces presented in Figure 8.7. Figure 8.8.a shows the number of segments to which the packet delay trace is fragmented for each value of λ , for each of the three packet delay traces. As can be seen, the number of segments

obtained tends to increase as λ decreases. This is due to the fact that λ controls the responsiveness of the segmentation algorithm. Thus smaller values of λ tend to make the algorithm more receptive to packet delay changes. This phenomenon depends also on the degree of stationarity and variability of the observed packet delay trace. For instance, only one segment is generated for any value of λ used on Run 4 using 2000pps scenario. However packet delay traces obtained on Run 10 using 1000pps scenario generates the most number of segments for small values of λ , compared to the other two traces.

Degree of disorder of packet delay samples within each segment is measured by means of entropy. Entropy is a concept used to define the randomness or disorder. The expected information content of a probability distribution, called entropy, is derived by weighing the information values by their respective probabilities [62]. Packet delay entropy is calculated on each segment. A set of entropy values is obtained for each packet delay trace. Average of this set of values is calculated and divided by the entropy of the entire packet delay trace. In this research, this metric is called average entropy ratio and it is presented in equation (8.16):

$$average_entropy_ratio = \frac{\sum_{i=1}^s \left(\frac{E^i}{k} \right)}{E_{total}} \quad (8.16)$$

where E^i is the entropy of segment i , E_{total} is the entropy of the entire packet delay trace, and s is the total number of segments in which the packet delay traces was fragmented using the proposed algorithm. As it can be seen from Figure 8.8.b, average entropy ratio tends to increase as λ increases. This can be explained due to the fact that small λ values tend to make the algorithm more aware to packet delay spikes and other uneven events on the packet delay series, and thus create small size segments. In this context, segments tend to group packet delay samples with similar dynamics, and thus entropy of them are more likely to be lower than entropy of segments generated using larger λ values. From Figure 8.8.b it can be seen that average entropy ratio of packet delay trace obtained from Run 10 using 1000pps scenario changes considerably for different values of λ , compared to the other two traces. This can be explained due to the variability of the packet delay samples observed for this trace, see Figure 8.7.3.a. Average entropy ratio of packet delay traces obtained from Run 4 using 100pps scenario show less variability for different values of λ . In fact, it is observed that average entropy ratio for this scenario reaches a steady state value for λ values larger than 40. This can be understood, since this packet delay trace only present few spikes. Finally, average entropy ratio of packet delay trace obtained from Run 4 using 2000pps scenario, shows no variability at all for any value of λ . This is due to the fact that only one segment is required for this trace, regardless of the λ value used on the algorithm.

Figure 8.9 shows the evolution in time of two of the four AR coefficients, $a_{1,k}$ and $a_{2,k}$, used for online modeling of packet delay trace shown on Figure 8.7.3.a., where k represents the sample number. As it can be seen, AR coefficients change drastically in time according to the packet delay dynamics. However they tend to vary within a narrow range in stationarity segments. Note that traditional time invariant AR models only capture the overall behavior of the packet delay trace, and thus fail of distinguishing such changes on the system dynamics, even when differentiation techniques are employed.

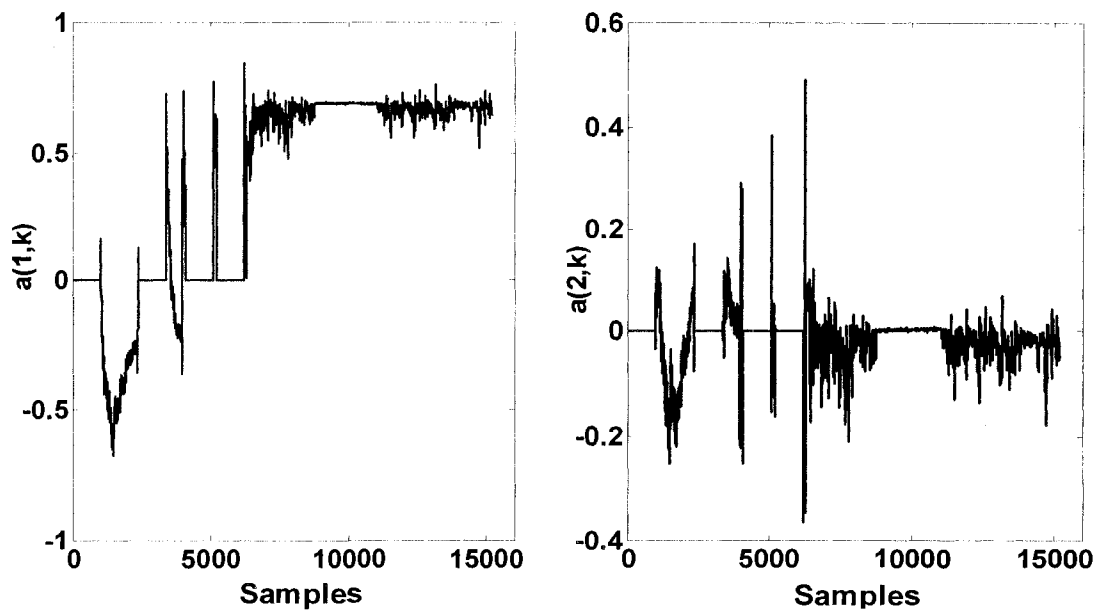


Figure 8.9. Evolution in time of two AR coefficients used for online modeling of packet delay trace shown on Figure 8.7.3.a.

In a nutshell, it can be seen that the proposed segmentation and modeling mechanism successfully separates and characterizes the dynamics of packet delay samples into a set of *AR* models based on the observed sample's stationarity. The proposed mechanism has demonstrated not being triggered by instantaneous changes on the trace dynamics, but to respond to the packet delay distribution's changes. Which is mainly driven by $\hat{\xi}_k$. It can also be seen that the embedded Kalman Filtering algorithm indeed captures effectively the evolution in time of the system. Moreover c_{w_i} reflects accurately the degree of non-stationarity on the segmented regions, which corroborates the underlying reason for the segmentation process. The proposed mechanism requires low computational and storage overhead, since a small set of parameters and recursive linear equations are needed to discover and model each segment. Algorithm settings are selected in such a way that segmentation is based on the non-stationarity of the packet delay samples. However, it can be expected that sensitivity of the algorithm and thus results may vary according to these settings.

8.5.1 MEMORY STORAGE SAVINGS VS. NON-STATIONARITY

In this subsection the memory storage savings capabilities of the proposed segmentation algorithm are tested against packet delay traces with different degrees of non-stationary. Traces presented on Figures 8.6 and 8.7 were used. To describe the degree of non-stationarity of a packet delay trace, the Index of Dispersion of Intervals (IDI) has been used. In general IDI measures the dependence between consecutive samples on a trace, and it is often used to

describe the burstiness of a signal [65]. IDI is defined as a sequence $\{c_k^2\}$, $k \geq 1$, where;

$$c_k^2 = \frac{k \text{Var}(S_k)}{[E(S_k)]^2} \quad (8.17)$$

and the random variable S_k is the sum of k consecutive samples on a trace.

If the trace represents a Poisson process, then $c_k^2 \equiv 1$ for every k . However if process has higher variance at some time scale, then c_k^2 will tend to increase as a function of k [35].

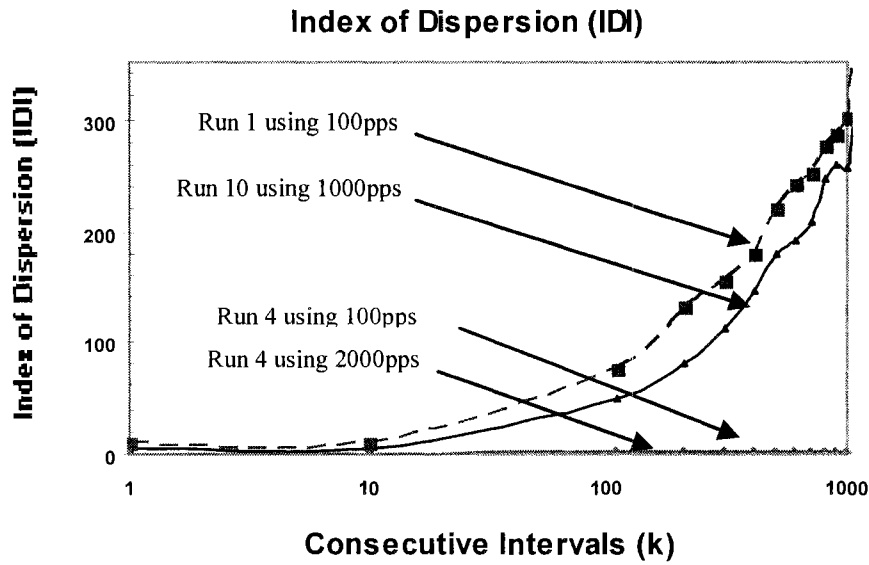


Figure 8.10. Index of Dispersion of Intervals (IDI) for blocks of k consecutive packet delay samples obtained from multiple traces.

Figure 8.10 shows the IDI for the packet delay trace presented on Figure 8.6 and the three traces presented on Figure 8.7. As it can be seen from Figure 8.10, IDI obtained from Run 4 using 2000 pps and Run 4 using 100 pps scenarios indicate these traces come from stationary Poisson process, which indeed it can be corroborated by Figures 8.7.1.a and 8.7.2.a. However, IDI obtained from Run 1 using 100 pps and Run 10 using 1000 pps scenarios indicates a noticeable degree of non-stationary on these traces, which also can be corroborated by Figures 8.6.a and 8.7.3.a. In general, it can be concluded that the first two traces of Figure 8.10, starting from the bottom, can be easily modeled as Poisson process. Thus packet delay distributions of the entire traces can be used when modeling these series. As a result, in the context of the proposed segmentation algorithm, these traces may need very few segments or just one when modeling them. Conversely, the other two traces will fit poorly into a Poisson distribution, and thus more segments are needed when modeling them in the context of the proposed segmentation algorithm. These remarks can be confirmed by the number of segments generated through the segmentation algorithm on these packet delay traces and shown on Figures 8.6.b, 8.7.1.b, 8.7.2.b, and 8.7.3.b.

Percentage of Memory Storage Savings using Segmentation Algorithm as a function of λ				
λ	Run 4 Using 2000pps	Run 4 Using 100pps	Run 10 Using 1000pps	Run 1 Using 100pps
10	99.96%	99.84%	99.10%	99.96%
20	99.96%	99.84%	99.10%	99.97%
40	99.96%	99.89%	99.20%	99.97%
60	99.96%	99.89%	99.40%	99.97%
80	99.96%	99.89%	99.50%	99.98%
100	99.96%	99.89%	99.60%	99.98%
150	99.96%	99.92%	99.60%	99.99%
200	99.96%	99.92%	99.80%	99.99%

Table 8.3. Percentage of memory storage savings using segmentation algorithm as a function of λ

Table 8.3 shows the percentage of memory storage savings using segmentation algorithm for the four mentioned packet delay traces, obtained through equation (8.15), as a function of λ . As it can be seen the proposed algorithm produces tremendous memory storage saving compared to a traditional approach of storing the entire trace for all the four scenarios presented. Memory storage saving in general not only depends on the degree of non-stationarity of the trace, but also on the number of samples of the trace and the sensibility of the algorithm, λ . This remark can be confirmed by Table 8.3 results.

8.6 REMARKS

A novel approach for online modeling end-to-end packet delay dynamics under non-stationary network conditions is presented. Proposed methodology models the network system characteristics based on the non-stationarity of the packet delay samples, while keeping computational and storage requirements

low. Such method is based on adaptive AR model, Kalman Filtering algorithm, and a modified version of the Divergence-Test.

Our findings show that the proposed methodology separates and model, in an online manner, packet delay traces based on significant changes on the system dynamics, and not based on isolated spikes on the trace. Performance of algorithm has been tested against different network and traffic conditions. Results indicate that segmented series obtained through this approach reflect stationarity within its samples, which indeed demonstrate the capability of the algorithm of modeling a non-stationary packet delay process as a sequence of stationary sub-processes. Number of fragmented segments, generated from packet delay traces using the proposed algorithm, and their duration clearly indicate a correlation to the algorithm settings and the network system dynamics. Responsiveness of the algorithm was also tested for different settings. Results indicate a tradeoff of accuracy by computational overhead and memory storage requirements according to the settings employed. False sense of LRD on packet delay was also studied in the context of the proposed algorithm, and the importance of distinguishing it when modeling packet delay processes is highlighted. In general, results shows that analyzing packet delay processes by modeling the segmented stationary traces yield to a better understanding of the network system dynamics.

CHAPTER 9. CONCLUSIONS

When tested under realistic scenarios, existing approaches for modeling network system and packet dynamics for network emulation are not scalable. This research proposes a measurement-based modeling methodology for the design of a network-in-a-box emulator that aims to overcome the limitations associated with computational overhead and complexity associated with traditional approaches to end-to-end network system modeling.

A framework for large scale IP network emulation, named OTRENET, has been formally introduced. OTRENET overcomes the overhead of packet-by-packet mapping and modeling, while keeping track on the consistency of the results, by means of a proposed Average Traffic Sampler by Time Frame Segmentation Algorithm. Design and operation performance of the proposed network emulation were described in detail. Performance analysis results shown indicate that the proposed OTRENET module fulfills our expectations of mimicking the overall behavior of a real network scenario.

Methodologies for modeling network system dynamics by means of packet delay and IPG characterization, with emphasis on cross traffic, sending rate, and packet size were discussed in this research. Findings presented leads to the conclusion that the behavior of end-to-end packet delay and IPG sequences can be captured effectively by ARMA and ARIMA models, under weakly-stationary network conditions and using CBR probe flows. Under these network conditions,

model goodness-of-fit results demonstrate modeling accuracy for both packet delay and IPG processes under low sending bit rate conditions. However, as sending bit rate increases as a fraction of the bandwidth, IPG becomes better alternative for network system modeling.

Network system modeling using end-to-end packet delay dynamics is extended to no-stationary network system conditions. A novel computational efficient methodology for online segmentation and modeling of packet delay series based on adaptive AR model, Kalman Filtering algorithm, and a modified version of the Divergence-Test was proposed. Our findings show that the proposed approach separates and models, in an online manner, packet delay traces based on significant changes on the system dynamics. Performance of algorithm has been tested against different network and traffic conditions. Results indicate that segmented series obtained through this algorithm reflect stationarity within its samples, which indeed demonstrate the capability of the algorithm of modeling a non-stationary packet delay process as a sequence of stationary sub-processes. Responsiveness of the algorithm was also tested for different settings, and under various network system conditions. Results indicate a tradeoff of accuracy by computational overhead and memory storage requirements according to the settings employed. False sense of LRD on packet delay was also studied in the context of the proposed algorithm, and the importance of distinguishing it when modeling packet delay processes is highlighted. In general, results shows that analyzing packet delay processes by

modeling the segmented stationary traces yield to a better understanding of the network system dynamics.

REFERENCES.

- [1] M. Aboy, J. McNames, O. Marquez, R. Hornero, T. Thong, and B. Goldstein, "Power spectral density estimation and tracking of nonstationary pressure signals based on Kalman filtering," *Annual International Conference of the IEEE Engineering in Medicine and Biology-Proceedings.*, pp. 156-159.04, San Francisco, California, September 2004.
- [2] J. Ahn, P. B. Danzig, Z. Liu, and L. Yan, "Evaluation of TCP Vegas: Emulation and Experiment," *Proc. of the ACM SIGCOMM*, pp 185-195, Cambridge, Massachusetts, Aug. 1995. ACM.
- [3] M. Allman, A. Caldwell, S Ostermann, "ONE: The Ohio Network Emulator," *TR-19972*, School of Electrical Engineering and Computer Science, Ohio University, August 1997.
- [4] R. Andre-Obrecht, "A new statistical approach for the automatic segmentation of continuous speech signals," *IEEE Trans. Speech and Audio Proc.*, vol. 36, no. 1, 1988.
- [5] M. Arnold, W. Miltner, H. Witte, R. Bauer, and C. Braun, "Adaptive AR Modeling of Non-stationary Time Series by Means of Kalman Filtering," *IEEE Trans. Biomed. Eng.*, vol. 45(5), 1998.
- [6] D.K. Arrowsmith, R.J.Mondragón, J.M. Pitts, and M. Woolf, "Internet packet congestion," *IEEE International Symposium on Circuits and Systems*, Bangkok, Thailand, May 2003.
- [7] S. Bajaj, L.Beslau, K Fall. "Virtual internetwork testbed: Status and research agenda," *Technical Report 98-678*, University of Southern California, July 1998.
- [8] T. Banka, A. Maroo, A. P. Jayasumana, V. Chandrasekar, N. Bharadwaj, and S. Chittibabu, "Radar Networking: Considerations for Data Transfer Protocols and Network Characteristics," *Proc. 21st International Conference on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*, Paper 19.1, San Diego, CA, January 2005.

- [9] M. Basseville and A. Benveniste. "Sequential detection of abrupt changes in spectral characteristics of digital signals." *IEEE Trans. on Information Theory*. vol. IT-29, pp. 709-723, Sept. 1983.
- [10] A. Begen, M. Begen and Y. Altunbasak, "Predictive modeling of video packet delay in IP networks", *Proc. IEEE Int. Conf. Image Processing (ICIP)*, Atlanta, GA, Oct. 2006.
- [11] J. Beran, "Statistics of long memory processes", *Monographs on Stats and Appl. Prob.* 61, Chapman & Hall, London.
- [12] J.-C Bolot, "End-to-end packet delay and loss behavior in the Internet", *Proc. ACM SIGCOMM '93*, pp. 289-298, Sept. 1993.
- [13] J.-C. Bolot, "Characterizing end-to-end packet delay and loss in the Internet," *Journal of High-Speed Networks*, vol. 2, pp. 305-323, Dec. 1993.
- [14] C. Bovy, H. Mertodimedjo, G. Hooghiemstra, H. Uijterwaal, and P. Mieghem, "Analysis of End-to-End Delay Measurements in Internet", *Proc. Passive and Active Measurements Conferences*, Colorado, March 2002.
- [15] G. E. P. Box, G. M. Jenkins, and, G. C. Reinsel, "Time Series Analysis, Forecasting and Control", 3rd ed. *Prentice Hall*, Englewood Cliffs, NJ, 1994.
- [16] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," *Proc. of ACM SIGCOMM '94*, pp. 24-35, Oct. 1994.
- [17] P. Brockwell and R. Davis, "Introduction to Time Series and Forecasting (2nd ed.)", *New York: Springer-Verlag*, 2002. ISBN 0-387-95351-1.
- [18] R. Brown, "Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem", *Communications of the ACM*, 31(10):1220-1227, October 1988.
- [19] M. Carson., and D. Santay, "NIST Net - A Linux based network emulator tool", *ACM SIGCOMM Computer Communications Review*, Vo1.33, Jan 2003, pages 111-126.

- [20] D. Cox and P. Lewis, "The Statistical Analysis of Series of Events," London, England: Methuen, 1966, pp71-72.
- [21] J. DeFreitas, M. Niranjan, and A. Gee, "Hierarchical Bayesian-Kalman models for regularisation and automatic relevance determination in sequential Learning", *Neural Computation*, 12, pp. 933-953. April 2000.
- [22] C. Demichelis, "Packet Delay Variation Comparison between ITU-T and IETF Draft Definitions," *in the IPPM mail archives*, November 2000.
- [23] C. Demichelis, P. Chimento, "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)", *RFC 3393*, November 2002.
- [24] G. Dharwarkar and O. Basir, "Enhancing Temporal Classification of AAR Parameters in EEG single-trial analysis for Brain-Computer Interfacing", *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005*. Sept. 2005 Page(s): 5358 – 5361.
- [25] J.-P. Eckmann, S. Kamphorst, and D. Ruelle. "Recurrence plots of dynamical systems". *Europhysics Letters*, 4:973-977, 1987.
- [26] K. Fall. "Network Emulation in the VINT/NS Simulator," *Proc. of 4th IEEE Symposium on Computers and Communications*, July 1999.
- [27] K. Fall, K. Varadhan, editors. NS notes and documentation. *The VINT project*, LBL, February 2000. <http://www.isi.edu/nsnam/ns/>
- [28] R. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM*, vol. 33, pp. 30-53, Oct. 1990.
- [29] K. Fukuda, H. Stanley, and L. Amaral. "Heuristic segmentation of a nonstationary time series". *Phys. Rev. E* 69, art. no. 021108, 1-12 (2004).
- [30] S. Haykin, *Adaptive Filter Theory*, 4th edition, Upper Saddle River, NJ: Prentice Hall, 2002, Ch 10.

- [31] L. Huang, and K. Sezaki, "An analysis of the characterization and prediction of network delay," *Proc. IEICE General Conference* 2000, SB-9-7, Japan, Mar 200 0.
- [32] E. Hernandez and Sumi Helal, "RAMON: Rapid Mobility Network Emulator". *Proc. of the 27th Annual IEEE Conference on Local Computer Networks (LCN)*, November 2002, Tampa, Florida.
- [33] A. Isaksson, A. Wennberg, and L. H. Zetterberg, "Computer analysis of EEG signals with parametric models," *Proc. IEEE*, vol. 69, pp. 451-461, Apr. 1981.
- [34] R. Jain, "A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks," *ACM Computer Communication Review*, vol. 19, pp. 56-71, Oct. 1989.
- [35] T. Karagiannis, M. Molle, and M. Faloutsos, "A nonstationary poisson view of internet traffic," *Proc. of IEEE INFOCOM*, pp. 1-1, March 2004.
- [36] J. Kohlmorgen and S. Lemm. "An On-line Method for Segmentation and Identification of Non-stationary Time Series". *Neural Networks for Signal Processing XI, IEEE*, NJ, 113-122, 2001.
- [37] L. A. Kulkarni and S. Q. Li, "Measurement-Based Traffic Modeling: Capturing Important Statistics," *Journal of Stochastic Model*, Vo. 14, No. 5, 1998.
- [38] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. "On the Self-Similar Nature of Ethernet Traffic," *In IEEE/ACM Transactions on Networking*, 1994.
- [39] M. Li, W. Jia, and W. Zhao, "A Method for Modeling Autocorrelation Functions of Asymptotically LRD Traffic and Its Verification," *Conf. Proc., ICCT2000, 16th IFIP World Computer Congress*, IEEE Press, 21-25, Aug., 2000, Beijing, China, pp. 62-65.
- [40] K. Lien, and J. Reeve, "A TCP/IP Network Emulator", *In IST2005 International Symposium on Telecommunications*, September 2005, Shiraz, Iran.

- [41] S.-Q. Li, S. Park, and D. Arifler, "SMAQ: A measurement-based tool for traffic modeling and queueing analysis Part I - design methodologies and software architecture," *IEEE Communications Magazine*, vol. 36, Aug. 1998.
- [42] S. Q. Li & C. L. Hwang, "Queue Response to Input Correlation Functions: Discrete Spectral Analysis," *IEEE/ACM Transactions on Networking*, vol. 1, No. 5, Oct. 1994, pp. 522-533.
- [43] L. Ljung, "System Identification - theory for the user". NJ: Prentice Hall, Englewood Cliffs, 1987.
- [44] M. Lopatka, C. Laplanche, O. Adam, J. Motsch, J. Zarzycki, "Non-Stationary Time-Series Segmentation Based on the Schur Prediction Error Analysis," *Statistical Signal Processing, IEEE/SP 13th Workshop*, July 17-20 2005 pp: 251 – 256.
- [45] N. Marwan. "Recurrence Plots and Cross Recurrence Plots," <http://www.recurrence-plot.tk/>
- [46] N. Marwan. Matlab CRP Toolbox 5.5. Available at <http://www.agnld.uni-potsdam.de/marwan>.
- [47] S. Makridakis, "A survey of time series", *Int. Stat. Rev.*, 44(1), pp:29-70, 1976.
- [48] J. Minho, K. Hyungdo, and K. Hyogon Kim, "An Adaptive Routing Method for VoIP Gateways Based on Packet Delay Information," *IEICE Transactions on Communications*, Feb. 2005.
- [49] S. B. Moon., J. Kurose, P. Skelly, and D. Towsley, "Correlation of packet delay and loss in the Internet," *Report UM-CS-1998-011*, *UM-CS-1998-011*, Mar. 1998.
- [50] D. Morató, E. Magaña, M. Izal, J. Aracil, F. Naranjo, F. Astiz, U. Alonso, I. Csabai, P. Hágá, G. Simon, J. Stéger, and G. Vattay. "The European traffic observatory measurement infrastructure (etomic): A testbed for universal active and passive measurements," In *Proc. of Tridentcom 2005*. Trento, Italy, February 23, 2005.
- [51] A. Morton, "Packet Delay Variation Applicability Statement", draft-morton-ippm-delay-var-as-00. Work in progress.

- [52] A. Nussi, A. Sridharan and N. Taft, "The problem of synthetically generating IP traffic matrices: Initial recommendations," *ACM SIGCOMM Computer Communication Review*, 35(3), 19-32 July 2005.
- [53] H. Ohsaki, M. Murata, and H. Miyahara, "Modeling end-to-end packet delay dynamics of the Internet using system identification". *In Proceedings of the International Teletraffic Congress 17*, pp 1027-1038, Dec. 2001.
- [54] L. Patomaki, J. Kaipio, and P. Karjalainen, "Tracking of nonstationary EEG with the roots of ARMA models", *IEEE Conf. EMBC-95*, 1995.
- [55] K. Pawlikowski, H. Jeong, J Lee, "On Credibility of Simulation Studies of Telecommunication Networks," *IEEE Communications Magazine*, January 2002, pp 132-139.
- [56] V. Paxson and S. Floyd, "Why We don't Know How to Simulate the Internet," *Proc. of the 1997 Winter Simulation Conference*, December 1997.
- [57] V. Paxson, "On calibrating measurements of packet transit times," *LBNL -41535*, <ftp://ftp.ee.lbl.gov/papers/vp-clockssigmatics98.ps.gz>, Mar. 1998.
- [58] W. Penny and S. Roberts, "Dynamic models for nonstationary signal segmentation," *Computers and Biomedical Research*, 32(6):483-502, 1999.
- [59] N. Piratla, A. P. Jayasumana and H. Smith, "Overcoming Effects of Correlation in Packet Delay Measurements using Inter-packet Gaps," *Proc. IEEE International Conference on Networks, Singapore*, November 2004, pp. 233-238.
- [60] N. M. Piratla, A. P. Jayasumana and T. Banka, "On Reorder Density and its Application to Characterization of Packet Reordering," *Proc. 30th IEEE Local Computer Networks (LCN) Conference*, Sydney, Australia, Nov. 2005.
- [61] A. Sang and S. Li, "A predictability analysis of network traffic," *Proc. of the 2000 IEEE Computer and Communications Societies*

Conference on Computer Communications (INFOCOM-00), pp 342–351, 2000.

- [62] C.E Shannon, "A Mathematical Theory of Communication", *Bell Sys. Tech. Journal*, vol. 27, 1948.
- [63] A. Schlogl, S. Roberts, G. Pfurtscheller, "A criterion for adaptive autoregressive models," *Proceedings of the 22nd EMBS International Conference*, pp. 1581-1582, 2000.
- [64] C. Spielvogel. "Design and Implementation of a Network Resource Service for QoS Aware Servers," *Master's thesis*, University Klagenfurt, Institute of Information Technology, November 2003.
- [65] K. Sriram and W. Whitt, "Characterizing Superposition Arrival Processes in Packet Multiplexers for Voice and Data," *IEEE Journal on Selected Areas in Communications SAC-4*, pp. 833-846, September 1986.
- [66] D. Vivanco and A. Jayasumana, "New QoS Approaches for Delay-sensitive Applications over DiffServ," *Proc. of ITCOM 2002 - SPIE Int. Symposium on The Convergence of Information and Technologies and Communications*, Aug. 2002.
- [67] D. Vivanco and A. Jayasumana, "Bandwidth Brokering and Dynamic Resource Allocation in DiffServ Domains for Heterogeneous Applications," *Proc. 27th IEEE Conference on Local Computer Networks*, Tampa, FL, Nov. 2002, pp. 372-381.
- [68] D. Vivanco and A. Jayasumana, "OTRENET: Overall Trend Replicating Network Emulator Tool," *Proc. International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'05)*, Philadelphia, PA, July 2005, pp.863-872.
- [69] D. Vivanco and A. P. Jayasumana, "A Measurement-Based Modeling Approach for Network-Induced Packet Delay," *32nd IEEE Int. Conference on Local Computer Networks*, Dublin, Ireland, Oct. 2007.
- [70] M. Yang, X.R. Li, H. Chen, and N.S.V. Rao, "Predicting Internet end-to-end delay: An overview", *Proc. of 36th IEEE Southeastern Symposium on Systems Theory*, pp. 210-214, Atlanta, Mar. 2004.

- [71]I. Yeom and A. L. Narasimha Reddy, "ENDE: An End-to-end Network Delay Emulator Tool for Multimedia Protocol Development," *Multimedia Tools and Applications*, No. 14, pp. 269-296, 2001.
- [72]L. Zheng, L. Zhang, and D. Xu, "Characteristics of network delay and delay jitter and its effects on voice over IP (VoIP)," *Proc. IEEE ICC*, Jun 2001, pp 122-126.
- [73]P. Zheng, and L. Ni, "EMPOWER: A Cluster Architecture Supporting Network Emulation," *IEEE Transactions on Parallel and Distributed Systems*, Volume 15, Issue 7, July 2004.
- [74]National Institute of Standards and Technology. Nistnet(<http://snad.ncsl.nist.gov/itg/nistnet>.)
- [75]<http://www.packetstorm.com>
- [76]http://www.stockcharts.com/education/IndicatorAnalysis/indic_movingAvg.html
- [77]Ixia Incorporated webpage, <http://www.ixiacom.com>
- [78]ITSM 2000 Professional Version 6.0, developed by Peter J. Brockwell and Richard (<http://www.stat.colostate.edu/~pjbrock/>).
- [79]The official website of Linux Kernel information. <http://www.kernelnotes.org>.
- [80]ETOMIC (Evergrow Traffic Observatory Measurement InfrastruCture), <http://www.etomic.org>.
- [81]History of ARPANET – <http://www.dei.isep.ipp.pt/~acc/docs/arpa>. Last Accessed December 14, 2007.
- [82]MATLAB Version 7.1 – Curve Fitting Toolbox v1.1.4.
- [83]<http://www.itl.nist.gov>
- [84]OPNET Modeler Documentation, OPNET Technologies Inc., Bethesda, MD, 2001.

APENDIX A. SCRIPT FOR TRAFFIC SAMPLER BY TIME FRAME

SEGMENTATION ALGORITHM

The source code of the traffic sampler by time frame segmentation algorithm, which was written in Matlab, is presented below. This algorithm was formally presented and described in detail in CHAPTER 5. A performance comparison analysis of this algorithm against similar algorithms was presented in CHAPTER 5. The code presented below separates packet delay metrics according to its variability.

A.1. MATLAB SCRIPT FOR TRAFFIC SAMPLER BY TIME FRAME SEGMENTATION

```
clc;
clear all;
close all;

load algo_prove\out_simul_per_second.txt;
A=out_simul_per_second;
sampled_delay=A(300:800,2);
sampled_drop=A(:,3);
sampled_time=A(:,1);
initial_thres_acc_delay=10;
initial_thres_acc_drop=10;
time_frame_number=1;
star_time_frame=sampled_time(1);
index_star_time_frame=1;
star_time_frame_p=0;
estimated_delay_actual_frame=0;
maximum_delay_deviation=0.4;
estimated_drop_actual_frame=0;
maximum_drop_deviation=0.4;
T=10;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-----Delay Analysis-----
for i=1:1:length(sampled_delay)-1;
    diff_ave_delay(i)= sampled_delay(i+1)- sampled_delay(i);
    acc_diff_ave_delay(i)=sum(diff_ave_delay(index_star_time_frame:i));
    signal_new_frame_1(i)=0;signal_new_frame_2(i)=0;

    if ((time_frame_number==1)& (i==1))
```

```

        thr_delay(time_frame_number)=initial_thres_acc_delay;
        thr_delay_time(i)=thr_delay(time_frame_number);
    else
        D=i-index_star_time_frame;
        thr_delay_time(i)=thr_delay(time_frame_number)*exp(-D/T);
    end

    if (abs(acc_diff_ave_delay_(i)) > thr_delay_time(i))
        T=50/(star_time_frame-star_time_frame_p);
        star_time_frame_p=star_time_frame;
        estimated_delay_(index_star_time_frame:i)=mean(sampled_delay(index_star_time_frame:i));
        estimated_delay_actual_frame=mean(sampled_delay(index_star_time_frame:i));
        time_frame_number=time_frame_number+1;
        star_time_frame=sampled_time(i+1);
        index_star_time_frame=i+1;
        thr_delay(time_frame_number)=1.3*thr_delay_time(i);
        signal_new_frame_1(i)=1;

        if ((abs(sampled_delay(i+1)-estimated_delay_actual_frame)) > maximum_delay_deviation)
            T=0.4;
            signal_new_frame_2(i)=1;
            thr_delay(time_frame_number)=initial_thres_acc_delay;
        end
    end

end

end

figure(1)
B=[1 0.5 0.25]/1.75;
A=[1];

estimated_gaussian_delay=filter(B,A,sampled_delay);
subplot(4,1,1);
plot(sampled_delay);hold on;plot(estimated_delay,'r');
plot(estimated_gaussian_delay,'g')
subplot(4,1,2);plot(abs(acc_diff_ave_delay_));hold on; plot(thr_delay_time,'g')
subplot(4,1,3);plot(signal_new_frame_1)
subplot(4,1,4);plot(signal_new_frame_2,'y')

mse(sampled_delay(1:length(estimated_delay_))-
(estimated_gaussian_delay(1:length(estimated_delay_))))

max(sampled_time)/sum(signal_new_frame_1)
max(sampled_time)/sum(signal_new_frame_2)

%%%%%%%%%%%%% EWMA ANALYSIS %%%%%%%%%%%%%%
%%%%%%%%%%%%%
x=sampled_delay;
z_previous=0;
lambda=0.05;
start_frame_time=1;
L=3;

```

```

for i=1:1:length(x)
    mean_x=mean(x(start_frame_time:i));
    x(i);
    sigma_x=sqrt(var(x(start_frame_time:i)));
    z(i)=lambda*x(i)+(1-lambda)*z_previous;
    z_previous=z(i);
    num=lambda*(1-(1-lambda)^(2*i));
    n=1; % sample size
    den=(2-lambda)*n;
    UCL(i)=mean_x+L*sigma_x*sqrt(num/den);
    LCL(i)=mean_x-L*sigma_x*sqrt(num/den);
    mean_x+L*sigma_x*sqrt(num/den);
    UCL_(i)=mean_x+L*sigma_x*sqrt(lambda/den);

    signal_EWMA(i)=0;
    if (UCL(i)<z(i)) | (LCL(i)<z(i))
        averaged_signal_EWMA(start_frame_time:i)=mean(x(start_frame_time:i));
        start_frame_time=i-1;
        signal_EWMA(i)=500;
    end
end

figure(1)
%subplot(4,1,1);
plot(averaged_signal_EWMA,'k')

(length(estimated_delay_))/(sum(signal_EWMA)/max(signal_EWMA))
(length(estimated_delay_))/(sum(signal_new_frame_1)+sum(signal_new_frame_2))
mse(sampled_delay(1:length(estimated_delay_))-estimated_delay_)
mse(sampled_delay-averaged_signal_EWMA)
legend('sampled_delay','Estimated delay using dynamic average traffic sampler by time
segmentation','averaged_signal_EWMA');

figure(2)
subplot(2,1,1);plot(z); hold on; plot(UCL,'g');hold on;plot(LCL,'y')
subplot(2,1,2);plot(signal,'g');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-----Drop Analysis-----
for i=1:1:length(sampled_drop)-1;
    diff_ave_drop(i)= sampled_drop(i+1)- sampled_drop(i);
    acc_diff_ave_drop(i)=sum(diff_ave_drop(index_star_time_frame:i));
    signal_new_frame_1(i)=0;signal_new_frame_2(i)=0;

    if ((time_frame_number==1)& (i==1))
        thr_drop(time_frame_number)=initial_thres_acc_drop;
        thr_drop_time(i)=thr_drop(time_frame_number);
    else
        D=i-index_star_time_frame;
        thr_drop_time(i)=thr_drop(time_frame_number)*exp(-D/T);
    end

    if (abs(acc_diff_ave_drop(i)) > thr_drop_time(i))
        T=50/(star_time_frame-star_time_frame_p);
        star_time_frame_p=star_time_frame;
        estimated_drop(index_star_time_frame:i)=mean(sampled_drop(index_star_time_frame:i));
        estimated_drop_actual_frame=mean(sampled_drop(index_star_time_frame:i));
        time_frame_number=time_frame_number+1
        star_time_frame=sampled_time(i+1);

```

```

index_star_time_frame=i+1;
thr_drop(time_frame_number)=1.3*thr_drop_time(i);
signal_new_frame_1(i)=1;

if ((abs(sampled_drop(i+1)-estimated_drop_actual_frame)) > maximum_drop_deviation)
    T=0.4;
    signal_new_frame_2(i)=1;
    thr_drop(time_frame_number)=initial_thres_acc_drop;
end

end

end

figure(2)
subplot(4,1,1);plot(sampled_drop);hold on;plot(estimated_drop_,'r')
subplot(4,1,2);plot(abs(acc_diff_ave_drop_));hold on; plot(thr_drop_time,'g')
subplot(4,1,3);plot(signal_new_frame_1)
subplot(4,1,4);plot(signal_new_frame_2,'y')

max(sampled_time)/sum(signal_new_frame_1)
max(sampled_time)/sum(signal_new_frame_2)

```

APENDIX B. SCRIPT FOR OTRENET EMULATOR

In this appendix the scripts used within the OTRENET network emulator module are presented. Appendix 2.3.2B.1 presents the TCL script used in NS network simulator to recreate a computer network topology and realistic cross-traffic situations. In Appendix B.2 the traffic sampler by time frame segmentation algorithm, presented on Appendix A.1, is used to characterize the simulated traffic metrics obtained through the network simulation presented on Appendix 2.3.2B.1. Code presented in appendix B.2 was written in awk.

The script used to link the three units of the OTRENET, which was explained in 4.3.4, is presented in appendix B.3. Code presented in appendix B.3 was written in Perl. In appendix B.4 the modified version of knistnet.c code, which was written in C, is presented. This code is used to capture in real time incoming traffic metrics using the time frame segmentation algorithm, which was formally presented and described in detail in CHAPTER 5, and input the traffic metrics into the network simulator.

B.1. TCL SCRIPT FOR NS NETWORK SIMULATION

```
#Create a simulator object
set ns [new Simulator]
while {[file exists
/home/daniel/Emul/total/final_module_2/dump_files/ns_input.tcl] == 0} {
    #puts "Simulator doesn't have input file..."
}
source /home/daniel/Emul/total/final_module_2/dump_files/ns_input.tcl
source simulator_scheduler_2.tcl
exec rm /home/daniel/Emul/total/final_module_2/dump_files/ns_input.tcl

#-----Switching Output traces-----
```

```

set f [open "| awk -f simulator_emulator_7.awk" w]
$ns trace-all $f

#-----
#-----Topology-----
#-----
#-----
#Create five nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
#-----

#-----
#Create a duplex link between the nodes
$ns duplex-link $n0 $n3 1000kb 1ms DropTail
$ns duplex-link $n1 $n3 1000kb 1ms DropTail
$ns duplex-link $n2 $n3 1000kb 1ms DropTail
#$ns duplex-link $n3 $n4 17kb 1ms SFQ
$ns duplex-link $n3 $n4 100kb 1ms SFQ
#-----

#----Orientation-----
$ns duplex-link-op $n0 $n3 orient right-down
$ns duplex-link-op $n2 $n3 orient right-up
$ns duplex-link-op $n1 $n3 orient right
$ns duplex-link-op $n3 $n4 orient right
#-----
#-----Initial values-----
set counter_ 0
set now_p -1
set real_end_time $end_time
#-----
#Create a UDP agent and attach to real stream
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
#-----
set packetsize_2 50
set interval_2 0.01
set packetsize_3 100
set interval_3 0.01
#-----
#Create three traffic sinks and attach them to the node n4
set sink0 [new Agent/LossMonitor]
set sink1 [new Agent/LossMonitor]
set sink2 [new Agent/LossMonitor]
$ns attach-agent $n4 $sink0
$ns attach-agent $n4 $sink1
$ns attach-agent $n4 $sink2

```

```

#-----
#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]

#Create a UDP agent and attach it to node n1
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1

# Create a CBR traffic source and attach it to udp1(0.384 Mbps)
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ $packetSize_2
$cbr1 set interval_ $interval_2
$cbr1 attach-agent $udp1
$udp1 set fid_ 2

#Create a UDP agent and attach it to node n2
set udp2 [new Agent/UDP]
$ns attach-agent $n2 $udp2

# Create a CBR traffic source and attach it to udp2 (0.768 Mbps)
set cbr2 [new Application/Traffic/CBR]
$cbr2 set packetSize_ $packetSize_3
$cbr2 set interval_ $interval_3
$cbr2 attach-agent $udp2
$udp2 set fid_ 2
#-----

#Connect the traffic source with the traffic sink
$ns connect $udp0 $sink0
$ns connect $udp1 $sink1
$ns connect $udp2 $sink2

#-----
#Schedule events for the CBR agent
$ns at 0 "parameters"
$ns at 0.0 "$cbr0 start"
#$ns at 0.1 "$cbr1 start"
#$ns at 0.2 "$cbr2 start"
#$ns at 5.0 "$cbr0 stop"
#$ns at 5.0 "$cbr1 stop"
#$ns at 5.0 "$cbr2 stop"
#-----
proc combine {} {
    global f ns cbr2 cbr1
    set ns [Simulator instance]
    set now [$ns now]
    $ns at $now "$cbr1 start"
    $ns at [expr $now+40] "$cbr1 stop"

```



```

    $ns at [expr $now+10] "$cbr2 start"
    $ns at [expr $now+30] "$cbr2 stop"
    $ns at [expr $now+50] "combine"
    puts "combine"
}

#-----
#Call the finish procedure after 5 seconds of simulation time
#$ns at 10 "finish"
#$ns at 50.1 "combine"

#Run the simulation
$ns run
#-----

```

B.2. AWK SCRIPT FOR MEASURING SIMULATED TRAFFIC METRICS AND ONLINE TRAFFIC SAMPLER BY TIME FRAME SEGMENTATION

```

BEGIN {
# Description: awk script for measuring delay, packet loss and jitter

# --- Configuration lines ---
noc = 10;
minflow = 1;
maxflow = 10;
# -----

highest_packet_id = -1;
highest_start_time_id=-1;
change_counter=1;
total_ave_delay_p=0;
total_ave_drop_p=0;
aver_applied_delay=1;
aver_applied_drop=1;
aver_applied_delay_th=1;
aver_applied_drop_th=1;

system_trigger_time=0;
#---Exponential initail paramters--
t=0;
j=1;
D=0;
T_delay=30;
T_drop=15;
intial_ampl_thr_delay=10;
intial_ampl_thr_drop=5;
ampl_thr_delay[j]=intial_ampl_thr_delay;

```

```

ampl_thr_drop[j]=initial_ampl_thr_drop;
thr_trigger=0;
signal_2=0;
#max_delay_deviation=0.4;
max_delay_deviation=0.4;
counter_deviation=0
#-----

for (i = minflow; i <= maxflow; i++) {

counter_passed_[i]=0;
counter_dropped_[i]=0;
total_pack_duration_p[i]=0;
total_packet_size_p[i]=0;
smallest_start_time_id[i]=0;
minimum_sampling_boundry_limit[i]=-1;
Throughput_p[1]=0;
Thr_p[1]=0;
acc_diff_thr_p[1]=0
acc_drop_p=0;
average_pack_duration_p[i]=0;
acc_diff_ave_pack_duration_p[i]=0;
percentage_pack_dropped_p[i]=0;
acc_diff_perc_pack_dropped_p[i]=0;
sampling_rate[i]=1;

}

}

#-----
{
# Get tokens from a string
action = $1;
time = $2;
temporal_source = $3;
temporal_destination = $4;
flow = $8;
source = int($9);
final_destination = int($10);
packet_id = $12;
packet_size= $6;

#####
#####

if (action == "h" || action == "r" || action == "d" || action == "+" || action == "-") {

if (action == "+")
{
start_time[flow,packet_id] = time;

if (((start_time[flow,packet_id] > start_time_p[flow,packet_id]) && (start_time_p[flow,packet_id]
!= 0 )) || (start_time_p[flow,packet_id] == 0 && packet_id == 0 && start_time[flow,packet_id] != 0 ) )
{
start_time[flow,packet_id]=start_time_p[flow,packet_id];
}
else {
start_time_p[flow,packet_id]=start_time[flow,packet_id];
}
}
}

```

```

    }

    if ( packet_id == 0 && action == "+" && (temporal_source == source) ) {
        smallest_start_time_id[flow] = time;
    }
}

b
    if ( minimum_sampling_boundry_limit[flow] < smallest_start_time_id[flow] ) {
        minimum_sampling_boundry_limit[flow] = smallest_start_time_id[flow];
        maximum_sampling_boundry_limit[flow] = (minimum_sampling_boundry_limit[flow]) +
(sampling_rate[flow]);
    }

    if (((action != "d") && (action == "r" && final_destination == temporal_destination)) || (action ==
"d")) {
        if ((action == "r" && final_destination == temporal_destination) && (action != "d")) {

            end_time[flow,packet_id] = time;
            counter_passed_[flow] = counter_passed_[flow]+1;

            pack_size[flow,packet_id] = packet_size;
            total_packet_size[flow]=total_packet_size_p[flow]+pack_size[flow,packet_id];
            total_packet_size_p[flow]=total_packet_size[flow];

            pack_duration[flow] = (end_time[flow,packet_id] - start_time[flow,packet_id])*1000;
            total_pack_duration[flow] = pack_duration[flow]+total_pack_duration_p[flow];
            total_pack_duration_p[flow] = total_pack_duration[flow];

            if (start_time[flow,packet_id] > highest_start_time_id) { #in case of reordering
                highest_start_time_id=start_time[flow,packet_id];
            }

            if ( start_time[flow,packet_id] > maximum_sampling_boundry_limit[flow] ) {
                average_packet_size[flow] = (total_packet_size[flow])/(counter_passed_[flow]);
                average_pack_duration[flow] = (total_pack_duration[flow])/(counter_passed_[flow]);
                percentage_pack_dropped[flow] =
100*(counter_dropped_[flow])/(counter_dropped_[flow]+counter_passed_[flow]);
                Throughput[flow]=
(0.008*average_packet_size[flow]*counter_passed_[flow])/(start_time[flow,packet_id]-
minimum_sampling_boundry_limit[flow]);
                minimum_sampling_boundry_limit[flow] = start_time[flow,packet_id];
                maximum_sampling_boundry_limit[flow] = (minimum_sampling_boundry_limit[flow]) +
(sampling_rate[flow]);
                counter_passed_[flow]=0;
                counter_dropped_[flow]=0;
                total_pack_duration_p[flow] =0;
                total_packet_size_p[flow]=0;

                if (flow==1) {
                    system_trigger=0;
                    diff_ave_pack_duration[1] = average_pack_duration_p[1]-average_pack_duration[1];
                    acc_diff_ave_pack_duration[1] = diff_ave_pack_duration[1] +
acc_diff_ave_pack_duration_p[1];
                    average_pack_duration_p[1] = average_pack_duration[1];
                    acc_diff_ave_pack_duration_p[1]= acc_diff_ave_pack_duration[1];
                    abs_acc_delay[1]= sqrt((acc_diff_ave_pack_duration[1])^2);
                    total_ave_delay=(total_ave_delay_p+average_pack_duration[1]);
                    total_ave_delay_p=total_ave_delay;

```

```

        diff_perc_pack_dropped[1] = percentage_pack_dropped_p[1] -
percentage_pack_dropped[1];
        acc_diff_perc_pack_dropped[1] = diff_perc_pack_dropped[1] +
acc_diff_perc_pack_dropped_p[1];
        percentage_pack_dropped_p[1] = percentage_pack_dropped[1];
        acc_diff_perc_pack_dropped_p[1] = acc_diff_perc_pack_dropped[1];
        abs_acc_drop[1] = sqrt((acc_diff_perc_pack_dropped[1])^2);
        total_ave_drop = (total_ave_drop_p + percentage_pack_dropped[1]);
        total_ave_drop_p = total_ave_drop;

        change_counter = change_counter + 1;
        #-----Exponential_Threshold_Algorithm-----
        if (j == 1 || thr_trigger == 1) {
            D = 0;
            trigger_time = start_time[flow, packet_id];
            thr_trigger = 0;
        } else {
            D = start_time[flow, packet_id] - trigger_time;
        }
        if (signal_2 == 1 && j != 2) {
            D = 0;
            T_delay = 0.1;
            T_drop = 0.5;
            signal_2 = 0;
            signal_peak = 1;
            ampl_thr_delay[j] = initial_ampl_thr_delay; #could be changed from 80 to thr_delay[t-1]
            ampl_thr_drop[j] = initial_ampl_thr_drop; #could be changed from 80 to thr_delay[t-1]
            printf("\n..... HERE..... \n");
        }
        thr_delay[t] = (ampl_thr_delay[j]) * exp(-D/T_delay);
        thr_drop[t] = (ampl_thr_drop[j]) * exp(-D/T_drop);
        #-----Thriggering_delay-----
        if (abs_acc_delay[1] > thr_delay[t]) {
            #T_delay=30; #should change according the the peaks periodicity
            system_trigger = 1;
            value_delay = ((average_pack_duration[1] -
aver_applied_delay_th)/aver_applied_delay_th);
            if (((average_pack_duration[1] - aver_applied_delay_th)/aver_applied_delay) > 0) {
                sign = 1;
            }
            else { sign = -1; }
            if (aver_applied_drop != 0) {
                value_drop = sqrt(((percentage_pack_dropped[1] -
aver_applied_drop)/aver_applied_drop)^2);
                printf("\n..... Signal_11..... :%f %f\n", value_delay, value_drop);

                if ((sqrt(((average_pack_duration[1] - aver_applied_delay)/aver_applied_delay_th)^2))
> 0.3) {
                    system_trigger = 1;
                    signal_2 = 1;
                    signal_22 = 1;
                    printf("\n..... Signal_12..... \n");
                }
            }

            #-----Thriggering_drop-----

            if ((abs_acc_drop[1] > thr_drop[t]) && (signal_2 != 1)) {
                ##T_drop=50; #should change according the the peaks periodicity

```

```

        #system_trigger=1;
        #printf("\n..... Signal_21..... \n");
        #if ((sqrt(((percentage_pack_dropped[1]-
aver_applied_drop)/aver_applied_drop)^2))>0.25){
        #signal_2=1;
        #printf("\n..... Signal_22..... \n");
        #}
    #}

#-----Display-----
    real_time_simulation = systime();

    print(1000*Throughput[1],real_time_simulation,"<Simulator_out_>",system_trigger_time,thr_drop[t],ab
s_acc_delay[1],abs_acc_drop[1],average_pack_duration[1],aver_applied_delay,percentage_pack_dr
opped[1],aver_applied_drop,1000*Throughput[2],maximum_sampling_boundry_limit[1]) >
    "simulated_trace.txt";

    #print(1000*Throughput[1],real_time_simulation,"<Simulator_out_>",thr_delay[t],thr_drop[t],abs_acc_
delay[1],abs_acc_drop[1],average_pack_duration[1],aver_applied_delay,percentage_pack_dropped[1
],aver_applied_drop,1000*Throughput[2]) > "simulated_trace.txt";

    close("simulated_trace.txt");
    printf("\n Thr_delay_time:%f | abs_acc_delay[1]:%f \n",thr_delay[t],abs_acc_delay[1]);
    printf("\n flow :%d |Throu(kbps):%f |ave_delay:%f |acc_diff_delay[1]:%f | ave_drop:%f
|acc_diff_drop[1]:%f| %f
|System_time_:%f\n",flow,Throughput[flow],average_pack_duration[1],acc_diff_ave_pack_duration[1],
percentage_pack_dropped[1],acc_diff_perc_pack_dropped[1],thr_delay[t],system_trigger_time);
    t=t+1;

#-----
#-----System Trigger-----
if (system_trigger == 1) {
    j=j+1;
    thr_trigger=1;
    ampl_thr_delay[j]=1.2*thr_delay[t-1];
    ampl_thr_drop[j]=1.2*thr_drop[t-1];
    if (signal_peak == 1)
{ampl_thr_delay[j]=initial_ampl_thr_delay;ampl_thr_drop[j]=initial_ampl_thr_drop;}
    aver_applied_delay=total_ave_delay/(change_counter-1);
    aver_applied_drop=total_ave_drop/(change_counter-1);
    aver_applied_delay_th=total_ave_delay/(change_counter-1);
    aver_applied_drop_th=total_ave_drop/(change_counter-1);
    total_ave_delay_p=0;
    total_ave_drop_p=0;
    acc_diff_ave_pack_duration_p[flow]=0;
    acc_diff_perc_pack_dropped_p[flow]=0;
    change_counter=1;
    system_trigger_time = systime();

    if (signal_2 == 0){counter_deviation=0;max_delay_deviation=0.4}

    if (signal_22 == 1){
        printf("\n..... System Drastical change| counter_deviation = %f
\n",counter_deviation);
        if (counter_deviation > 2 || counter_deviation == 0 || sqrt((value_delay)^2) > 1 ||
value_drop > 0.5){
            aver_applied_delay = ((2.5)^sign)*aver_applied_delay;
            aver_applied_drop = ((2.5)^sign)*aver_applied_drop;
            #max_delay_deviation=2.5*max_delay_deviation;

```

```

        printf("\n..... System Drasticall change| counter_deviation = %f
\n",counter_deviation);
        counter_deviation=0;
    }
    counter_deviation=counter_deviation+1;
    signal_22=0;
}

    printf("\n..... System_Signal....: %f..... \n",aver_applied_delay);
    #system("/root/download/nistnet/cli/./cnistnet -F");
    system("/root/download/nistnet/cli/./cnistnet -a 13.0.0.5 15.0.0.2 add new --delay
"aver_applied_delay" --drop "aver_applied_drop");
    #system("cnistnet -a 13.0.0.5 15.0.0.2 add new --delay 0 --drop 0");

}

#-----

}

}

}

} if (action == "d") {
    start_time[flow,packet_id] = time;
    end_time[flow,packet_id] = -1;
    counter_dropped[flow] = counter_dropped[flow]+1;
    #printf("Amount of packets dropped [1]:%d\n",counter_dropped[1]);
    #printf("Amount of packets dropped [2]:%d\n",counter_dropped[2]);
}

}

}

#####
#####

}

#-----

```

B.3. PERL SCRIPT THAT EXECUTES AND SYNCHRONIZES OTRENET UNITS

```
#!/usr/bin/perl

use threads;
use threads::shared;

my $collector_flag : shared = 0;
my $simulator_flag : shared = 0;
my $file_scheduler_flag : shared = 0;
my $file_counter_collector : shared = 0;
my $file_counter_simulator : shared = 1;
my $initial_flag : shared = 0;
my $input_size : shared = 0;
my $plotter_flag : shared = 0;

$input_size_p = -s "/var/log/messages";

my $collector = threads->new (\&traffic_collector, 1);
my $simulator = threads->new (\&network_simulator_emulator, 2);
my $file_scheduler = threads->new (\&scheduler, 3);
my $emulator_plotter = threads->new (\&plotter, 4);

#system("echo 0 > ns_availability");
system("/root/download/nistnet/cli/.cnistnet -u");
system("/root/download/nistnet/cli/.cnistnet -F");
#system("cnistnet -a 15.0.0.3 13.0.0.2 add new --delay 0 --drop 0");
system("/root/download/nistnet/cli/.cnistnet -a 13.0.0.5 15.0.0.2 add new --delay 0 --drop 0");
#system("/root/download/nistnet/cli/.cnistnet -S 13.0.0.5 15.0.0.2 >>file_1");
system("rm /home/daniel/Emul/total/final_module_2/dump_files/*.*");
system("rm /home/daniel/Emul/total/final_module_2/simul_continuous/dump_files/*.*");
system("rm -rf /home/daniel/Emul/total/final_module_2/simul_continuous/dump_files/*.*");
system("rm /home/daniel/Emul/total/final_module_2/simul_continuous/results/*.*");
system("rm -rf /home/daniel/Emul/total/final_module_2/simul_continuous/results/*.*");

$collector->join;
$simulator->join;
$file_scheduler->join;
$plotter->join;

sub traffic_collector {
    system("dmesg -c");

    while (1) {
        #while($file_counter_collector - $file_counter_simulator > 2) {sleep(1); }
        $input_size = -s "/var/log/messages";
        while ($input_size_p == $input_size){
            $input_size = -s "/var/log/messages";
        }
        print("$input_size $input_size_p \n");
        $input_size_p = $input_size;
        system("dmesg -c >> /home/daniel/Emul/total/final_module_2/dump_files/temp_file");

        if ((-z "/home/daniel/Emul/total/final_module_2/dump_files/temp_file")) {
            #print ("\n..temp_file exist and is empty...\n");
        }
    }
}
```

```

#print ("\n.....testing: Collector is on ..in between...\n");

if ((-e "/home/daniel/Emul/total/final_module_2/dump_files/temp_file") && (-s
"/home/daniel/Emul/total/final_module_2/dump_files/temp_file")) {
    print ("\n..temp_file exist and is not empty...\n");
    system("awk -f simulator_input_generator_2.awk
/home/daniel/Emul/total/final_module_2/dump_files/temp_file");
    system("rm /home/daniel/Emul/total/final_module_2/dump_files/temp_file");

    if ((-e "/home/daniel/Emul/total/final_module_2/dump_files/ns_input_temp") && (-s
"/home/daniel/Emul/total/final_module_2/dump_files/ns_input_temp")) {
        print ("\n...ns_input_temp exist and is not empty...\n");
        $file_counter_collector++;
        system("mv /home/daniel/Emul/total/final_module_2/dump_files/ns_input_temp
/home/daniel/Emul/total/final_module_2/dump_files/"$file_counter_collector".tcl ");
        system("mv
/home/daniel/Emul/total/final_module_2/simul_continuous/dump_files/ns_continuous_input_temp
/home/daniel/Emul/total/final_module_2/simul_continuous/dump_files/"$file_counter_collector".tcl ");
        print "\n Collector is done file $file_counter_collector \n\n";
    }
}

if ($file_counter_collector == 1) {
    $file_scheduler_flag = 1;
    cond_signal($file_scheduler_flag);
}

}

}

sub network_simulator_emulator {
    lock($simulator_flag);
    cond_wait($simulator_flag);
    if ($simulator_flag == 1) {
        print "Simulator is on...\n\n";
        $plotter_flag = 1;
        cond_signal($plotter_flag);
        system("ns simulator_3.tcl");
        #system("ns simulator_exp_back.tcl");
        #system("ns complex_simul_1.tcl");
        exit 0;
    }
}

sub plotter {
    lock($plotter_flag);
    cond_wait($plotter_flag);
    if ($plotter_flag == 1) {
        #system("/root/download/nistnet/cli/./cnistnet -S 15.0.0.22 13.0.0.22 ");
    }
}

sub scheduler {
    lock($file_scheduler_flag);
    cond_wait($file_scheduler_flag);

```



```

while (1) {
    if ($file_scheduler_flag == 1) {

        if (!(!-e "/home/daniel/Emul/total/final_module_2/dump_files/ns_input.tcl")
            || ($file_counter_simulator == 1)){
            print ("File_simulator:$file_counter_simulator - File_collector:$file_counter_collector\n");

            while ($file_counter_simulator > $file_counter_collector){ }
            while (!-e
"/home/daniel/Emul/total/final_module_2/dump_files/"$file_counter_simulator.".tcl"){ }

            system("mv
/home/daniel/Emul/total/final_module_2/dump_files/"$file_counter_simulator.".tcl
/home/daniel/Emul/total/final_module_2/dump_files/ns_input.tcl");
            print ".....Scheduler: File #$file_counter_simulator was switched..... \n\n";
            $file_counter_simulator++;

            if ($file_counter_simulator == 2) {
                $simulator_flag = 1;
                cond_signal($simulator_flag);
            }
        }
    }
}
}

```

B.4. MODIFIED VERSION OF KNISTNET.C

```

/* $Header$ */

/* knistnet.c - Linux implementation of "hitbox"-like functionality.
 * This code exists as a loadable kernel module. It gains access to the
 * entry points it needs through some patches to the existing Linux kernel.
 * (Unfortunately, there seemed to be no practical alternative to patching
 * to gain access to the basic packet handling routine entry points.)
 * The user-level interface provided is a device driver one, modeled on the
 * original SunOS-based hitbox.
 *
 * Mark Carson, NIST/UMPC
 * 1/1997
 */

#include "kincludes.h"

/* The following can only be included in one place! */
#define EXPORT_SYMTAB
#include <linux/module.h>
#include <linux/kernel.h>
#include "tabledist.h"

int nistnet_debug;

/* Breakpoints are only helpful when we're compiled into the kernel */
#ifndef MODULE

#ifdef DEBUG

```

```

#define BREAKPOINT(string) asm(" int $3")
#define DEBUG_SPINLOCKS 2
#else
#define BREAKPOINT(string) printk(string)/*@@minidebug@@*/
#endif

#else /* MODULE */

#ifdef DEBUG
#define DEBUG_SPINLOCKS 2
#endif

#define BREAKPOINT(string) printk(string)/*@@minidebug@@*/

#endif /* MODULE */

spinlock_t LinLockVar = SPIN_LOCK_UNLOCKED;

static int lock_ticker;
#define LinLock(string) \
do {++lock_ticker;\
    if (nistnet_debug > 4 && (!(lock_ticker&0x3f) || nistnet_debug > 5)) \
        printk("lock %s", string);\
    spin_lock_irqsave(&LinLockVar, pre_flags);} while (0)
#define LinUnlock(string) \
do {if (nistnet_debug > 4 && (!(lock_ticker&0x3f) || nistnet_debug > 5)) \
    printk(" unlock %s\n", string);\
    spin_unlock_irqrestore(&LinLockVar, pre_flags);} while (0)

#ifdef notdef
#define HASHSIZE 256
static struct lin_hitbox *hitable[HASHSIZE];
#endif

#ifdef DEBUG
static int hittable_count;
#endif

struct nistnet_globalstats ourstats;
#define STATS_START 0
#define STATS_PROCESS 1
#define STATS_UNPROCESS 2

void fixed_gettimeofday(struct timeval *tv);
void lin_hash_stats(int number);

void fast_fill(void);
void fast_empty(void);
struct fast_timer_list * fast_alloc(int how);
void fast_free(struct fast_timer_list *done);

int
addnistnet(NistnetTableEntryPtr entry)
{
    /* Check entry for sanity */
    if (entry->lteStats.hitreq.drd_min &&
        entry->lteStats.hitreq.drd_min >= entry->lteStats.hitreq.drd_max)
        return -EINVAL;
#ifdef CONFIG_ECN

```

```

    if (entry->lteStats.hitreq.drd_congestion) {
        if (entry->lteStats.hitreq.drd_min > entry->lteStats.hitreq.drd_congestion ||
            entry->lteStats.hitreq.drd_max < entry->lteStats.hitreq.drd_congestion)
            return -EINVAL;
    }
#else
    entry->lteStats.hitreq.drd_congestion = 0;
#endif
    /* Should ignore these fields, but it doesn't feel right... */
    entry->lteOldDrop = entry->lteDrop;
    entry->lteOldDup = entry->lteDup;
    entry->lteOldDelay = entry->lteDelay;
    entry->lteOldDelsigma = entry->lteDelsigma;

    /* Initialize packet timer */
    fixed_gettimeofday(&entry->lteStats.last_packet);
    entry->lteStats.next_packet = entry->lteStats.last_packet;
    /* Insert in table */
    if (!lt_add(entry))
        return -ENOMEM;
#ifdef DEBUG
    ++hittable_count;
#endif
    return 0;
}

int
addhitreq(struct lin_hitreq *hitreq)
{
    NistnetTableEntry nistnetreq;

    bzero(&nistnetreq, sizeof(nistnetreq));
    nistnetreq.lteSource = hitreq->src;
    nistnetreq.lteDest = hitreq->dest;
    nistnetreq.lteStats.hitreq = *hitreq;
    nistnetreq.lteDrop = hitreq->p_drop;
    nistnetreq.lteDup = hitreq->p_dup;
    nistnetreq.lteDelay = hitreq->delay;
    nistnetreq.lteDelsigma = hitreq->delsigma;
    return addnistnet(&nistnetreq);
}

int
rmnistnet(NistnetTableEntryPtr entry)
{
    /* Remove from table */
    switch (lt_rm(entry)) {
    case 1:
        break;
    case 0:
        return -ESRCH;
    case -1:
        return -EFAULT;
    }
#ifdef DEBUG
    --hittable_count;
#endif
    return 0;
}

int

```

```

rmhitreq(struct lin_hitreq *hitreq)
{
    NistnetTableEntry nistnetreq;

    bzero(&nistnetreq, sizeof(nistnetreq));
    nistnetreq.lteSource = hitreq->src;
    nistnetreq.lteDest = hitreq->dest;
    nistnetreq.lteStats.hitreq = *hitreq;
    return rmnistnet(&nistnetreq);
}

int
gethitstats(struct lin_hitstats *hitstats)
{
    NistnetTablePtr tableptr;
    int i;

    tableptr = lt_find_by_srcdest(hitstats->hitreq.src, hitstats->hitreq.dest);
    if (!tableptr)
        return -ESRCH;
    /* Put things where the old stuff expects it */
    tableptr->ltEntry.lteOldDrop = tableptr->ltEntry.lteDrop;
    tableptr->ltEntry.lteOldDup = tableptr->ltEntry.lteDup;
    *hitstats = tableptr->ltEntry.lteStats;
    /* Compute current_bandwidth */
    for (i=0; i < BAND_ARRAY; ++i)
        hitstats->current_bandwidth +=
            hitstats->bandwidth_array[i];
    if (hitstats->seats_used)
        hitstats->current_bandwidth /= hitstats->seats_used;
    return 0;
}

int
getnistnet(NistnetTableEntryPtr where)
{
    struct lin_hitstats *hitstats;
    NistnetTablePtr tableptr;
    int i;

    tableptr = lt_find_by_key(&where->lteKey, NULL);
    if (!tableptr)
        return -ESRCH;

    /* Copy things where the old stuff expects it */
    tableptr->ltEntry.lteOldDrop = tableptr->ltEntry.lteDrop;
    tableptr->ltEntry.lteOldDup = tableptr->ltEntry.lteDup;

    *where = tableptr->ltEntry;
    hitstats = &where->lteStats;
    where->lteOldDelay = where->lteDelay;
    where->lteOldDelsigma = where->lteDelsigma;
    /* Compute current_bandwidth */
    for (i=0; i < BAND_ARRAY; ++i)
        hitstats->current_bandwidth +=
            hitstats->bandwidth_array[i];
    if (hitstats->seats_used)
        hitstats->current_bandwidth /= hitstats->seats_used;
    return 0;
}

```

```

/* nice and high, but it is tunable: insmod hitmod.o major=your_selection */
static int major = HITMAJOR;

/*
 * The driver.
 */

/* read -- get what's in the table */
static
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
ssize_t
hw_read(struct file * file, char * buf, size_t count, loff_t *whoknows)
#else
int
hw_read(struct inode * node, struct file * file, char * buf, int count)
#endif
{
    extern int DumpPairs(char *buf, int count);
    extern int lt_read(char *buf, int count);

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
    switch (MINOR(file->f_dentry->d_inode->i_rdev))
#else
    switch (MINOR(node->i_rdev))
#endif
    {
        case HITMINOR:
            return DumpPairs(buf, count);
        case NISTNETMINOR:
            return lt_read(buf, count);
        default:
            return 0;
    }
}

static
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
ssize_t
hw_write(struct file * file, const char * buf, size_t count, loff_t *whoknows)
#else
int
hw_write(struct inode * node, struct file * file, const char * buf, int count)
#endif
{
    int ret;

    if (count >= tabledistsize()) {
        ret = tabledistfill(buf);
        if (ret) return ret;
        return tabledistsize();
    } else
        return -E2BIG;    /* ha, ha */
}

/*
 * mucky muck ioctl interface
 */
static int
hw_ioctl(struct inode * inode, struct file * file, unsigned int type, unsigned long arg)
{
    struct lin_hitreq hitreq;

```

```

NistnetTableEntry nistnetreq;
extern void kick_fast_rtc(void);
int rc = 0;

switch (type) {
case HITIOCTL_OFF:
case NISTNET_OFF:
    /* Shut down */
    ourstats.emulator_on=0;
    break;
case HITIOCTL_ON:
case NISTNET_ON:
    /* Turn on if not already on */
    ourstats.emulator_on=1;
    break;
case HITIOCTL_ADD:
    /* Get what they want */
    copy_from_user_ret(&hitreq, (struct lin_hitreq *)arg,
        sizeof(struct lin_hitreq), -EFAULT);
    /* Add it to the table */
    rc = addhitreq(&hitreq);
    break;
case NISTNET_ADD:
    /* Get what they want */
    copy_from_user_ret(&nistnetreq, (NistnetTableEntryPtr)arg,
        sizeof(NistnetTableEntry), -EFAULT);
    /* Add it to the table */
    rc = addnistnet(&nistnetreq);
    break;
case HITIOCTL_REMOVE:
    /* Get what they want */
    copy_from_user_ret(&hitreq, (struct lin_hitreq *)arg,
        sizeof(struct lin_hitreq), -EFAULT);
    /* Remove it from the table */
    rc = rmhitreq(&hitreq);
    break;
case NISTNET_REMOVE:
    /* Get what they want */
    copy_from_user_ret(&nistnetreq, (NistnetTableEntryPtr)arg,
        sizeof(NistnetTableEntry), -EFAULT);
    /* Remove it from the table */
    rc = rmnistnet(&nistnetreq);
    break;
case HITIOCTL_STATS:
    {
        struct lin_hitstats hitstats;

        /* Get what they want */
        copy_from_user_ret(&hitstats, (struct lin_hitstats *)arg,
            sizeof(struct lin_hitstats), -EFAULT);
        rc = gethitstats(&hitstats);
        /* Copy out individual stats */
        if (!rc) {
            copy_to_user_ret((struct lin_hitstats *)arg,
                &hitstats,
                sizeof(struct lin_hitstats), -EFAULT);
        }
        break;
    }
case NISTNET_STATS:
    /* Get what they want */

```

```

        copy_from_user_ret(&nistnetreq, (NistnetTableEntryPtr)arg,
                           sizeof(NistnetTableEntry), -EFAULT);
        rc = getnistnet(&nistnetreq);
        /* Copy out individual stats */
        if (!rc) {
            copy_to_user_ret((NistnetTableEntryPtr)arg,
                             &nistnetreq,
                             sizeof(NistnetTableEntry), -EFAULT);
        }
        break;
    case HITIOCTL_MODE:
        /* ?? */
        break;
    case HITIOCTL_TIMER:
        /* ?? */
        break;
    case HITIOCTL_MTU:
        /* ?? */
        break;
    case HITIOCTL_DEBUG:
    case NISTNET_DEBUG:
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
        rc = get_user(nistnet_debug, (long *)arg);
        if (rc < 0) break;
#else
        nistnet_debug = get_user((long *)arg);
#endif
#ifdef LT_DEBUG
    lt_set_debug_level(nistnet_debug);
#endif
    break;
    case HITIOCTL_GLOBALSTATS:
        copy_to_user_ret((struct lin_globalstats *)arg,
                         &ourstats.l,
                         sizeof(struct lin_globalstats), -EFAULT);
        break;
    case HITIOCTL_NGLOBALSTATS:
    case NISTNET_GLOBALSTATS:
        copy_to_user_ret((struct nistnet_globalstats *)arg,
                         &ourstats,
                         sizeof(struct nistnet_globalstats), -EFAULT);
        break;
    case NISTNET_KICK:
        kick_fast_rtc();
        break;
    case NISTNET_FLUSH:
        flush_fast_timer_list();
        break;
    }
    return rc;
}

/*
 * Our special open code.
 * MOD_INC_USE_COUNT make sure that the driver memory is not freed
 * while the device is in use.
 */
static int
hw_open( struct inode* ino, struct file* filep)
{

```

```

    MOD_INC_USE_COUNT;
    return 0;
}

/*
 * Now decrement the use count.
 */
static
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
int
#else
void
#endif
hw_close( struct inode* ino, struct file* filep)
{
    MOD_DEC_USE_COUNT;
    #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
        return 0;
    #endif
}

static struct file_operations hw_fops = {
    #if LINUX_VERSION_CODE > KERNEL_VERSION(2,3,25)
        owner: THIS_MODULE, /* struct module *owner */
        read:  hw_read,      /* read - get emulator table */
        write: hw_write,     /* write - fill distribution table */
        ioctl: hw_ioctl, /* ioctl - most of the controls */
        open:  hw_open,      /* open */
        release: hw_close,   /* release */

        #else /* 2.0 and 2.2 versions */

            NULL,          /* lseek - n/a */
            hw_read,       /* read - get emulator table */
            hw_write,      /* write - fill distribution table */
            NULL,          /* readdir - n/a */
            NULL,          /* poll/select - n/a */
            hw_ioctl, /* ioctl - most of the controls */
            NULL,          /* mmap - n/a, for now at least */
            hw_open,       /* open */
            #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
                NULL,      /* flush */
            #endif
            hw_close,      /* release */
            NULL,          /* fsync */
            NULL,          /* fasync */
            NULL,          /* check_media_change */
            NULL,          /* revalidate */
            #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
                ,
                NULL      /* lock */
            #endif

        #endif /* 2.0 and 2.2 versions */

};

/* Various statistics. We record some running totals in circular arrays,
 * to give an idea of how things are going.
 */

```



```

#define band_seat(time) ((time.tv_sec)%BAND_ARRAY)

/* How well is the hash table doing? */
void
lin_hash_stats(int number)
{
    static int hash_slot;

    /* average 50/50 with the last value we got */
    ourstats.l.hash_tries[hash_slot] = (number + ourstats.l.hash_tries[hash_slot]) >> 1;
    hash_slot = (hash_slot+1)%BAND_ARRAY;
}

/* How long is our processing time? */
void
global_stats(int process)
{
    static struct timeval start;
    struct timeval end;
    static int process_slot, unprocess_slot;
    long int usec_time;

    switch(process) {
    case STATS_START:
        fixed_gettimeofday(&start);
        return;
    case STATS_PROCESS:
        fixed_gettimeofday(&end);
        usec_time = timeval_diff(&end, &start);
        /* Check for bogus time values */
        if (nistnet_debug && usec_time < 0) {
            printk("nistnet: pretty fast processing, %ld usecs!\n", usec_time);
            return;
        }
        /* Average in with previous value 50/50 */
        ourstats.l.process_overhead[process_slot] =
            (usec_time+ourstats.l.process_overhead[process_slot])>> 1;
        process_slot = (process_slot+1)%BAND_ARRAY;
        break;
    case STATS_UNPROCESS:
        fixed_gettimeofday(&end);
        usec_time = timeval_diff(&end, &start);
        /* Check for bogus time values */
        if (nistnet_debug && usec_time < 0) {
            printk("nistnet: pretty fast unprocessing, %ld usecs!\n", usec_time);
            return;
        }
        /* Average in with previous value 50/50 */
        ourstats.l.unprocess_overhead[unprocess_slot] =
            (usec_time+ourstats.l.unprocess_overhead[unprocess_slot])>> 1;
        unprocess_slot = (unprocess_slot+1)%BAND_ARRAY;
        break;
    }
}

/* What's the traffic like on this entry? */
void
packet_stats(struct sk_buff *skb, struct lin_hitstats *hitme)
{
    struct timeval our_time;

```

```

long packettime;
int last_seat, our_seat, seat;

if (!skb->len)
    return;
fixed_gettimeofday(&our_time);
last_seat = band_seat(hitme->last_packet);
our_seat = band_seat(our_time);
packettime = timeval_diff(&our_time, &hitme->last_packet);
/* Check for bogus time values */
if (nistnet_debug && packettime < 0) {
    printk("nistnet: packet arrived in %ld usecs!\n", packettime);
    goto after_bandwidth;
}
hitme->last_packet = our_time;
/* compute bandwidth */
if (packettime > BAND_ARRAY*MILLION) {
    /* too long since last packet; zero out everything */
    memset(hitme->bandwidth_array, 0,
           BAND_ARRAY*sizeof(unsigned long));
    hitme->seats_used = 1;
} else if (last_seat != our_seat) { /* zero out skipped intervals */
    for (seat = (last_seat+1)%BAND_ARRAY; seat != our_seat;
         seat = (seat+1)%BAND_ARRAY) {
        hitme->bandwidth_array[seat] = 0;
    }
    /* plus get this one! */
    hitme->bandwidth_array[our_seat] = 0;
    if (hitme->seats_used < BAND_ARRAY)
        ++hitme->seats_used;
}
hitme->bandwidth_array[our_seat] += skb->len;

after_bandwidth:
hitme->current_size = skb->len;
hitme->bytes_sent += skb->len;
return;
}

#ifdef DEBUG
void
check_skb(struct sk_buff *skb, char *where)
{
    if (skb->data < skb->head) {
        printk("bug:check_skb:under:%s\n", where);
    }
    if (skb->tail > skb->end) {
        printk("bug:check_skb:over:%s\n", where);
    }
}
#endif /* DEBUG */

/* Receive packet interception */
static struct packet_type *ippt;

static struct packet_type ourpt;

/* We use an arbitrary spot in the skb control buffer to mark packets
 * which we've already processed.
 */
#define NISTNET_CB_MAGIC          66

```

```

#define NISTNET_CB_MAGIC_SPOT 33
#define we_saw_skb(skb) (skb->cb[NISTNET_CB_MAGIC_SPOT] == NISTNET_CB_MAGIC)
#define gaze_at_skb(skb) (skb->cb[NISTNET_CB_MAGIC_SPOT] = NISTNET_CB_MAGIC)

/* Resume processing of a delayed packet */
void
runpacket(struct fast_timer_list *info)
{
    struct nistnet_packetinfo *hpi = (struct nistnet_packetinfo *)info->data;
    unsigned long pre_flags;

    LinLock("runpacket1");
    packet_stats(hpi->skb, &hpi->nle->lteStats);
    LinUnlock("runpacket1");
    /* Non-local save/restore of flags doesn't work on some
     * architectures (notably Suns). We should be in an OK
     * situation without it, though...
     */
    /*restore_flags(hpi->flags);*/
#ifdef DEBUG
    check_skb(hpi->skb, "third");
#endif
    /* Mark this one as already having been queued */
    gaze_at_skb(hpi->skb);
    (void) netif_rx(hpi->skb);
    LinLock("runpacket2");
    if (!hpi->nle->lteStats.qlen) {
        BREAKPOINT("zero queue in runpacket");
    } else {
        --hpi->nle->lteStats.qlen;
    }
    fast_free(info);
    LinUnlock("runpacket2");
    MOD_DEC_USE_COUNT;
}

/* This is the (slightly modified) DRD algorithm for dropping */

/* probability factors * PROBFACOR */
static unsigned int drdtable[] = { /* constant "ramp up" */
    6554,
    9830,
    13107,
    16384,
    19661,
    22938,
    26214,
    29491,
    32768,
    36045,
    39322,
    42598,
    45875,
    49152,
    52429,
    55706,
    58982,
    62259,
    65535
};

```

```

#define DRDLIMIT (sizeof(drdtable)/sizeof(int))

/* packet_drop - compute probability of dropping packet. If we are using
 * DRD, use its table (adjusted for the queue length parameters we're
 * using), otherwise use a constant drop probability (which may be 0).
 * If both DRD and constant drop are specified, we use the DRD probability
 * in preference to the constant drop if the former is non-zero.
 *
 * Note that DRD drops are by definition uncorrelated. The whole point
 * in doing "preventive" drops is to avoid correlated loss and retransmit!!
 *
 * Return 1 if packet is to be dropped, 0 otherwise.
 */
int
packet_drop(NistnetTablePtr tableme, int *use_drd, int *use_ecn)
{
    struct lin_hitstats *hitme;
    int value;

    hitme = &tableme->ItEntry.lteStats;
    *use_drd=0;
    *use_ecn=0;
    if (hitme->hitreq.drd_max) { /* using DRD */
        unsigned int stretch = DRDLIMIT/(hitme->hitreq.drd_max-hitme->hitreq.drd_min);

        if (hitme->qlen < hitme->hitreq.drd_min) { /* below DRD limit */
            if (tableme->ItEntry.lteDrop) {
                value = (correlatedrandom(&tableme->ItEntry.lteDrop)&0xffff);
                return value < tableme->ItEntry.lteDrop;
            } else {
                return 0;
            }
        }

        value = (myrandom())&0xffff;
        *use_drd=1;
        if (hitme->qlen >= hitme->hitreq.drd_max)
            return value < drdtable[DRDLIMIT-1];
#ifdef CONFIG_ECN
        /* If using DRD, check whether the queue length is still
         * below the ECN limit. If so, the packet can be marked
         * with the ECN bit rather than dropped.
         */
        else if (hitme->qlen <= hitme->hitreq.drd_congestion)
            *use_ecn=1;
#endif /* CONFIG_ECH */
        return value < drdtable[stretch *
            (hitme->qlen-hitme->hitreq.drd_min)];
    } else {
        if (tableme->ItEntry.lteDrop) {
            value = (correlatedrandom(&tableme->ItEntry.lteDrop)&0xffff);
            return value < tableme->ItEntry.lteDrop;
        } else {
            return 0;
        }
    }
}

#ifdef CONFIG_ECN
/* ecn_skb - mark a packet for explicit congestion notification, if supported.
 */

```

```

int
ecn_skb(struct sk_buff *skb)
{
    struct iphdr *iph;

    /* Get the ip header */
    #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
        iph = skb->nh.iph;
    #else
        iph = skb->h.iph;
    #endif
    /* Check if ecn enabled */
    if (!(iph->tos & ECN_CAPABLE))
        return -1;
    /* Munge bit, if not already munged */
    if (!(iph->tos & ECN_NOTED)) {
        unsigned long checksum;

        iph->tos |= ECN_NOTED;
        /* Adjust checksum to account for munged bit */
        /* ip checksum is 1's complement in network byte order... */
        checksum = iph->check - htons(ECN_NOTED);
        checksum += checksum >> 16; /* catch carry */
        iph->check = checksum;
    }
    return 0;
}
#endif /* CONFIG_ECN */

/* Determine the amount of time to delay a packet. This is the maximum
 * of two quantities:
 * 1. Probabilistic packet delay time
 * 2. Bandwidth-limitation delay time
 *
 * Question: Should we take probabilistic delay into account in determining
 * bandwidth consumption? Answer: This complicates things a little too much.
 * Our model is that bandwidth throttling happens first at one virtual
 * choke point, then packets may get independently delayed at some later
 * point. This can result in packets getting bunched up, so the stated
 * bandwidth limitation is actually exceeded at some point.
 *
 * The problem with taking the delay into account is that the simplest
 * way of doing so would remove any possibility of reordering packets --
 * each packet could not be sent any sooner than its predecessor. It
 * thus seems more useful in terms of network effects to do things the
 * way they are here.
 */
int
packet_delay(struct sk_buff *skb, NistnetTablePtr tableme)
{
    int probdelay=0, bandwidthdelay=0, delay=0;
    struct lin_hitstats *hitme;
    struct timeval our_time={0,0};
    long packettime=0;

    hitme = &tableme->ItEntry.ItStats;

    /* Figure probabilistic delay */
    probdelay = correlatedtabledist(&tableme->ItEntry.ItDelay);

    /* Figure bandwidth-limitation delay */

```

```

if (hitme->hitreq.bandwidth) {

    fixed_gettimeofday(&our_time);
    /* We can't send until queued packets have been taken care of */
    bandwidthdelay = timeval_diff(&hitme->next_packet,
                                   &our_time);

    if (bandwidthdelay < 0) {
        bandwidthdelay = 0;
        hitme->next_packet = our_time;
    }
    /* Now determine how much time this packet will take in
     * usec, in order to schedule the following one.
     */
    /* skb->len can sometimes be too big (with other junk)(?) */
    packettime = (long)skb->len*(MILLION/hitme->hitreq.bandwidth) +
                 ((long)skb->len*(MILLION%hitme->hitreq.bandwidth)
                  + hitme->hitreq.bandwidth/2)/hitme->hitreq.bandwidth;
    /* Quick defensive hack: even at 1 byte/second, a packet
     * shouldn't take longer than MTU seconds!
     */
    if (packettime < 0 || packettime > 1500*MILLION) {
        if (nistnet_debug)
            printk("nistnet: wacky packettime of %ld, with length %ld and bandwidth
%d\n",
                                   (long)packettime, (long)skb->len, hitme->hitreq.bandwidth);
        packettime = 0;
    } else {
        timeval_add(&hitme->next_packet, packettime);
    }
}
#ifdef CONFIG_DELAYMIDDLE
    bandwidthdelay += packettime/2;
#elif defined(CONFIG_DELAYEND)
    bandwidthdelay += packettime;
#elif defined(CONFIG_DELAYSTART)
    #endif

}

delay = probdelay > bandwidthdelay ? probdelay : bandwidthdelay;

if (nistnet_debug > 4) { /* Print what we're doing every once in a while */
    static int ticker;

    if (!(ticker&0x3f)) {
        printk("nistnet: packet size %ld packettime %ld usec delay %ld usec\n",
               (long)skb->len, (long) packettime, (long)delay);
        if (bandwidthdelay)
            printk("nistnet: current time is %d.%06d, will send at %d.%06d\n",
                   (int)our_time.tv_sec, (int)our_time.tv_usec,
                   (int)hitme->next_packet.tv_sec, (int)hitme-
>next_packet.tv_usec);
        ++ticker;
    }

    return usec_to_minijiffy(delay);
}

int
packet_dup(NistnetTablePtr tableme)
{

```

```

int value;

if (!tableme->ItEntry.ItDup) return 0;
value = (correlatedrandom(&tableme->ItEntry.ItDup)&0xffff);
return value < tableme->ItEntry.ItDup;
}

int
default_munger(struct sk_buff *skb, struct net_device *dev, struct packet_type *ptype, struct
lin_hitbox *hitme)
{
    return 1;
}

int
DefaultNistnetMunger(struct sk_buff *skb, struct net_device *dev, struct packet_type *ptype,
NistnetTableEntry *hitme)
{
    return 1;
}

packet_munger other_munger = default_munger;
NistnetMunger OtherNistnetMunger = DefaultNistnetMunger;

void addmunge(packet_munger munger)
{
    other_munger = munger;
}

void AddNistnetMunger(NistnetMunger munger)
{
    /* New mungers take precedence */
    other_munger = default_munger;
    OtherNistnetMunger = munger;
}

void rmmunge(packet_munger munger)
{
    if (munger == other_munger)
        other_munger = default_munger;
}

void RmNistnetMunger(NistnetMunger munger)
{
    if (munger == OtherNistnetMunger)
        OtherNistnetMunger = DefaultNistnetMunger;
}

static struct timeval ingress_time_p={0,0};
static long int total_sampling_time_p=0;
static long int total_packet_size_input_emulator_p=0;
static sampling_counter=0;
static emulator_input_counter=1;
static long int average_packet_interval_gap_p=0;
static long int average_packet_size_input_emulator_p=0;
static long int diff_time_gap_p =0;
static long int acc_diff_time_gap_p =0;
static long int acc_diff_pack_size_p =0;

static void printk_double(double value, unsigned int places)

```

```

{
int whole;
int fraction;
int multiplier;
for(multiplier = 1; places; places--) multiplier *= 10;
whole = (int)(value);
fraction = (int)( (value - whole) * multiplier);
if(fraction < 0) fraction = -fraction;
printf("%d.%d ", whole, fraction);
}

#define munge_finish(string) {LinUnlock(string); if (skb2) (void) rcv_packet_munge(skb2, dev,
ptype);/* recursively process dup */ }

int
rcv_packet_munge(struct sk_buff *skb, struct net_device *dev, struct packet_type *ptype)
{
    unsigned long pre_flags;

    LinLock("rcv_packet1");
    if (ourstats.emulator_on && !we_saw_skb(skb)) {
        int use_drd, use_ecn, delaytime;
        NistnetTablePtr tableme;
        struct lin_hitstats *hitme;
        struct lin_hitbox dummy;
        struct fast_timer_list *screamer;
        struct nistnet_packetinfo *hpi;
        struct sk_buff *skb2=NULL;
        int ret=1,special_sampling_time;
        struct timeval ingress_time={0,0};
        struct timeval ingress_time_u={0,0};
        long int packet_interval_gap;
        long int total_sampling_time=0;
        long int average_packet_interval_gap,change;
        double total_sampling_time_seconds,ingress_time_seconds;
        long int total_packet_size_input_emulator,diff_pack_size,total_acc_diff_time_gap;
        long int
average_packet_size_input_emulator,diff_time_gap,acc_diff_time_gap,acc_diff_pack_size;
        long int final_acc_diff_time_gap;

        global_stats(STATS_START);
        tableme = lt_find_by_ipheader(skb);
        /*tableme = lt_find_by_srcdest(skb->h.iph->saddr, skb->h.iph->daddr);*/
        if (tableme) {
            hitme = &tableme->ltEntry.lteStats;
            if (other_munger != default_munger) {
                dummy.stats = *hitme;
                dummy.next = NULL;
                ret = (*other_munger)(skb, dev, ippt, &dummy);
            } else {
                ret = (*OtherNistnetMunger)(skb, dev, ippt, &tableme->ltEntry);
            }
        } else {
            hitme = NULL;
            if (other_munger != default_munger) {
                ret = (*other_munger)(skb, dev, ippt, NULL);
            } else {
                ret = (*OtherNistnetMunger)(skb, dev, ippt, NULL);
            }
        }
        if (ret <= 0) {

```



```

        global_stats(STATS_PROCESS); /* well, sort of */
        LinUnlock("global_stats1");
        return ret;
    }
    if (!tableme) { /* not intercepting */
        global_stats(STATS_UNPROCESS);
        LinUnlock("global_stats2");
        return ippt->func(skb, dev, ippt);
    }

    fixed_gettimeofday(&ingress_time);
    packet_interval_gap = timeval_diff(&ingress_time,&ingress_time_p);
    total_sampling_time = packet_interval_gap+total_sampling_time_p;
    total_sampling_time_seconds=(double)total_sampling_time/1000000.0;
    sampling_counter++;
    average_packet_interval_gap=total_sampling_time/sampling_counter;
    total_packet_size_input_emulator=(long)skb->len+total_packet_size_input_emulator_p;
    average_packet_size_input_emulator=total_packet_size_input_emulator/sampling_counter;

    diff_pack_size = average_packet_size_input_emulator_p-
average_packet_size_input_emulator;
    diff_time_gap = average_packet_interval_gap_p-average_packet_interval_gap;
    acc_diff_time_gap = diff_time_gap + acc_diff_time_gap_p;
    acc_diff_pack_size = diff_pack_size + acc_diff_pack_size_p ;
    memcpy(&acc_diff_time_gap_p,&acc_diff_time_gap,sizeof(long int));
    memcpy(&acc_diff_pack_size_p,&acc_diff_pack_size,sizeof(long int));

    if (emulator_input_counter !=1 && total_sampling_time_seconds > 1 && (abs(acc_diff_time_gap) >
1000 || abs(acc_diff_pack_size) > 6) ) {
        change =1;

        ingress_time_seconds=(double)(ingress_time.tv_sec) +
(double)(ingress_time.tv_usec)/1000000;
        printk("<Input_change> %ld %ld %ld\n",average_packet_size_input_emulator,average_packet_interval_gap,emulator_input_counter);
        printk_double(ingress_time_seconds,6);
        printk_double(total_sampling_time_seconds,6);
        printk("\n");
        total_sampling_time = 0;
        total_packet_size_input_emulator=0;
        sampling_counter =0;
        emulator_input_counter++;
        acc_diff_time_gap_p=0;
        acc_diff_pack_size_p=0;
    }

    if((total_sampling_time_seconds > 5)|| (sampling_counter >1000)){
        change =0;
        ingress_time_seconds=(double)(ingress_time.tv_sec) +
(double)(ingress_time.tv_usec)/1000000;
        printk("<Input_normal> %ld %ld %ld\n",average_packet_size_input_emulator,average_packet_interval_gap,emulator_input_counter);
        printk_double(ingress_time_seconds,6);
        printk_double(total_sampling_time_seconds,6);
        printk("\n");
        /*printk("<Emulator_input_previous> %ld %ld %ld\n",average_packet_size_input_emulator_p,average_packet_interval_gap_p,emulator_input_coun
ter,ingress_time);*/
        total_sampling_time = 0;

```

```

total_packet_size_input_emulator=0;
sampling_counter =0;
emulator_input_counter++;
}

memcpy(&ingress_time_p,&ingress_time,sizeof(struct timeval));
memcpy(&total_sampling_time_p,&total_sampling_time,sizeof(long int));
memcpy(&total_packet_size_input_emulator_p,&total_packet_size_input_emulator,sizeof(long int));
memcpy(&average_packet_size_input_emulator_p,&average_packet_size_input_emulator,sizeof(long int));
memcpy(&average_packet_interval_gap_p,&average_packet_interval_gap,sizeof(long int));
memcpy(&diff_time_gap_p,&diff_time_gap,sizeof(long int));

/* Assume we will queue until we find otherwise */
++hitme->qlen;

/* See if we're going to drop the packet */
if (packet_drop(tableme, &use_drd, &use_ecn)) {
#ifdef CONFIG_ECN
    /* ecn behavior: mark packet, don't drop */
    if (use_ecn && ecn_skb(skb) == 0) {
        ++hitme->drd_ecns;
    } else {
#endif /* CONFIG_ECN */
        our_kfree_skb(skb, FREE_WRITE);
        --hitme->qlen;
        ++hitme->n_drops;
        if (use_drd)
            ++hitme->drd_drops;
        else
            ++hitme->rand_drops;
        global_stats(STATS_PROCESS);
        LinUnlock("global_stats3");
        return 0;
#ifdef CONFIG_ECN
    }
#endif /* CONFIG_ECN */
}

/* See if we're going to duplicate the packet. Here,
 * we just do fixed probability.
 */

if (packet_dup(tableme)) { /* you get a new sister! */
    ++hitme->dups;
    skb2 = skb_copy(skb, GFP_ATOMIC);
}

/* Now see if we're going to delay the packet */
if (!(delaytime = packet_delay(skb, tableme))) { /* no delay */
    --hitme->qlen;
    packet_stats(skb, hitme);

    global_stats(STATS_PROCESS);
    munge_finish("no delay");
    return ippt->func(skb, dev, ippt);
}

screamer = fast_alloc(GFP_ATOMIC);

```

```

        /* If we can't allocate, punt! */
        if (!screamer) {
            --hitme->qlen;
            ++hitme->mem_drops;
            packet_stats(skb, hitme);
            global_stats(STATS_PROCESS);
            munge_finish("fast_alloc failed");
            return ippt->func(skb, dev, ippt);
        }
        hpi = (struct nistnet_packetinfo *)screamer->data;

        init_fast_timer(screamer);
#ifdef DEBUG
        check_skb(skb, "first");
#endif
        hpi->skb = skb_unshare(skb, GFP_ATOMIC);
#ifdef DEBUG
        check_skb(hpi->skb, "second");
#endif
        hpi->dev = dev;
        hpi->n timer = &tableme->ltEntry;
        /* We don't actually use this anymore, as non-local
         * save/restore of flags turns out not to work on
         * some architectures (notably Suns). But we'll
         * leave it in to indicate what we were thinking
         * about...
         */
        hpi->flags = pre_flags;

        /* Schedule something to happen in a little while */
        screamer->expires = delaytime;
        MOD_INC_USE_COUNT;
        add_fast_timer(screamer);
        global_stats(STATS_PROCESS);
        munge_finish("reg delay");
        return 0;
    } else {
        LinUnlock("ippt->func");
        return ippt->func(skb, dev, ippt);
    }
}

void
grab_ip_rcv(void)
{
    struct packet_type *us;

    ourpt.type = htons(ETH_P_IP);    /* IP packets (only) */
    ourpt.dev = NULL;               /* wild card, for any dev */
    ourpt.func = rcv_packet_munge; /* our handler */
    ourpt.data = NULL;              /* nothing we need to keep */
    ourpt.next = NULL;              /* filled out by dev_add_pack */
    /* Add our handler */
    dev_add_pack(&ourpt);
    /* Now we search for the old one. Yes, this is a dirty trick.
     * We are using our proffered handler as a Trojan horse to get
     * at the old one. Heh, heh, heh.
     */
    us = &ourpt;

```

```

    for (us = us->next; us; us = us->next) {
        if (us->type == ourpt.type) { /* Got 'em! */
            printk("grab_ip_rcv: Found ippt at %lx\n",
                (unsigned long int) us);
            ippt = us;
            dev_remove_pack(us);
            break;
        }
    }
}

void
release_ip_rcv(void)
{
    if (ippt) {
        dev_remove_pack(&ourpt);
        dev_add_pack(ippt);
        ippt = NULL;
    }
}

#ifdef MODULE

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,1,0) && LINUX_VERSION_CODE <
    KERNEL_VERSION(2,3,0)
extern int irq_desc_addr;
MODULE_PARM(irq_desc_addr, "i");
#endif

/* I don't know exactly when these various modules macros were defined;
 * the following is a rough cut...
 */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,1,0)
MODULE_AUTHOR("Mark Carson <carson@antd.nist.gov>");
MODULE_DESCRIPTION("NIST Net network emulator");
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,4,10)
/* See the README.License file for why this "license" is included */
MODULE_LICENSE("GPL and additional rights");
#endif
#endif

int
init_module( void)
#else
int
nistnet_init(void)
#endif
{
    #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
    #else
    void export_nistnet_symbols(void);
    #endif

    if (register_chrdev(major, "hw", &hw_fops)) {
        printk("nistnet: register_chrdev failed: goodbye world :-(\n");
        return -EIO;
    }
    else
        printk("nistnet: Hello, world\n");

    (void) install_fast_timer();

```

```

    fast_fill();
    lt_init();
    memset(&ourstats, 0, sizeof(ourstats));
    grab_ip_rcv();
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
#else
    export_nistnet_symbols();
#endif
    return 0;
}

#ifdef MODULE
void
cleanup_module(void)
{
    if (unregister_chrdev(major, "hw") != 0)
        printk("nistnet: cleanup_module failed\n");
    ourstats.emulator_on = 0;
    release_ip_rcv();
    lt_cleanup();
    fast_empty();
    if (uninstall_fast_timer() != 0) {
        printk("nistnet: uninstall_fast_timer failed\n");
        /* Well, we're in trouble now! */
    }
}
#endif

/* We allocate 1024 slots at startup, then allow for extra bunches
 * of 64 at a time to be allocated if needed. (We keep the extra
 * allocations small, since they are done at interrupt time, from
 * presumably precious locked-down kernel buffers.)
 *
 * Hence, the initial memory requirement is around 36K, while the
 * maximum allowed usage is on the order of 616K (17344 packets),
 * not counting the space used up by all those extra sk_buffs hanging
 * around.
 *
 * If you really are planning to delay enormous numbers of packets,
 * you'd be better off making FAST_RESERVE larger, more or less
 * equal to the maximum number of packets you anticipate delaying.
 */
#define FAST_RESERVE    1024
#define FAST_EMERGENCY    64
#define FAST_MAX 256

struct fast_timer_list *bigfastspace[FAST_MAX], *fast_stack;
struct nistnet_packetinfo *bighpispacspace[FAST_MAX];
int extra_count = 0;

void
fast_fill(void)
{
    struct fast_timer_list *newfast, *fastspace;
    struct nistnet_packetinfo *newhpi, *hpispacspace;
    int i, limit;

    if (!extra_count) { /* First time, allocate a few pages */
        limit = FAST_RESERVE;
        fastspace = (struct fast_timer_list *)
            vmalloc(sizeof(struct fast_timer_list)*limit);

```

```

        if (!fastspace)
            return;
        hspace = (struct nistnet_packetinfo *)
            vmalloc(sizeof(struct nistnet_packetinfo)*limit);
        if (!hspace) {
            vfree(fastspace);
            return;
        }
    } else if (extra_count < FAST_MAX) {          /* subsequent times, go for fairly small chunks */
        limit = FAST_EMERGENCY;
        fastspace = (struct fast_timer_list *)
            kmalloc(sizeof(struct fast_timer_list)*limit, GFP_ATOMIC);
        if (!fastspace)
            return;
        hspace = (struct nistnet_packetinfo *)
            kmalloc(sizeof(struct nistnet_packetinfo)*limit, GFP_ATOMIC);
        if (!hspace) {
            our_kfree_s(fastspace, sizeof(struct fast_timer_list)*limit);
            return;
        }
    } else {                                     /* somebody got too greedy */
        return;
    }
    bigfastspace[extra_count] = fastspace;
    bighspace[extra_count] = hspace;
    ++extra_count;

    for (i=0; i < limit; ++i) {
        newfast = fastspace+i;
        newhpi = hspace+i;
        newfast->data = (unsigned long) newhpi;
        newfast->function = runpacket;
        newfast->next = fast_stack;
        fast_stack = newfast;
    }
}

void
fast_empty(void)
{
    int i;

    vfree((void *)bigfastspace[0]);
    vfree((void *)bighspace[0]);
    for (i = 1; i < extra_count; ++i) {
        our_kfree_s(bigfastspace[i], sizeof(struct fast_timer_list)*FAST_EMERGENCY);
        our_kfree_s(bighspace[i], sizeof(struct nistnet_packetinfo)*FAST_EMERGENCY);
    }
    extra_count = 0;
    fast_stack = NULL;
    memset((void *)bigfastspace, 0, sizeof(bigfastspace));
    memset((void *)bighspace, 0, sizeof(bighspace));
}

struct fast_timer_list *
fast_alloc(int how)
{
    struct fast_timer_list *answer = fast_stack;

    if (answer) {
        fast_stack = answer->next;
    }
}

```

```

        return answer;
    }
    fast_fill();
    answer = fast_stack;
    if (answer)
        fast_stack = answer->next;
    return answer;
}

void
fast_free(struct fast_timer_list *done)
{
    done->next = fast_stack;
    fast_stack = done;
}

/* Export interfaces */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
EXPORT_SYMBOL_NOVERS(addmunge);
EXPORT_SYMBOL_NOVERS(rmmunge);
EXPORT_SYMBOL_NOVERS(AddNistnetMunger);
EXPORT_SYMBOL_NOVERS(RmNistnetMunger);
#else
static struct symbol_table nistnet_syms = {
#include <linux/symtab_begin.h>

    X(addmunge),
    X(rmmunge),
    X(AddNistnetMunger),
    X(RmNistnetMunger),

#include <linux/symtab_end.h>
};

void export_nistnet_symbols(void)
{
    register_symtab(&nistnet_syms);
}
#endif

```

APENDIX C. SCRIPT FOR ONLINE PACKET DELAY SEGMENTATION

ALGORITHM

In this appendix the script used for implementing the online methodology for modeling non-stationary end-to-end packet delay, which was formally introduced in CHAPTER 8, is presented. Script presented below was written on Matlab.

C.1. MATLAB SCRIPT FOR IMPLEMENTING PACKET DELAY SEGMENTATION

```
close all;
clc;
clear all;

change_threshold=150;
original_bias=0.25;%
p=8;
delay_data = load ('E:\Old DaTa\delay\net_1\data_delay\delay_50.del');
delay_data =delay_data-mean(delay_data);
x=(delay_data);

%----- Start of the program -----
%-----

%-----%
%           Initial values for algorithm           %
%-----%
segment_counter=0;
segment_index=1;
UC=0.05;
alpha=0.0001;
buffered_samples=1000;
start_sample_region=1;
end_sample_region=start_sample_region+buffered_samples-1;
segment_end_point=end_sample_region;
segment_start_point=end_sample_region+1;
time_segmentation=zeros((length(delay_data)),1);
combined_cumulative_cross_entropy_(1:end_sample_region)=zeros(end_sample_region,1);
original_factor_check_negative_slop=100;
%-----%
%           Kalman algorithm loop           %
%-----%

for n = 1:(length(delay_data)),
    if(n==(end_sample_region)) %Computing Model Theta_1 of the segment
        A=[];E=[];a_zero=[];AR_model_small_segment=[];
        var_forward_error_small_segment=[];Cov_e=[];e=[];
        Cov_w=[];P=[];
```



```

initial_sample=[];AR_model_small_segment=[];
initial_sample=x(start_sample_region:end_sample_region) ;
initial_sample=initial_sample-mean(initial_sample);
[A,E] = ARBURG(initial_sample,p);
A=-A(2:p+1); %AR coefficients
a_zero=A; % AR coefficients
a(n,:)=a_zero;
P{n} = eye(p);
AR_model_small_segment=a_zero;
var_forward_error_small_segment=E;
Cov_w(n)=0;
Cov_e(n) = E; % Noise variance of AR model
e(start_sample_region:end_sample_region)=normrnd(0,sqrt(E),end_sample_region-
start_sample_region+1,1);
bias=original_bias;
factor_check_negative_slop=original_factor_check_negative_slop;
ARRAY_AR_model_small_segments(segment_index,:)=AR_model_small_segment;
array_delay_small_segment=x(start_sample_region:end_sample_region);

if (0)
for h = p+20:1:end_sample_region-start_sample_region+1;
error_small_segment(h)= array_delay_small_segment(h)-
AR_model_small_segment*array_delay_small_segment(h-p:h-1);
error_half_small_segment(h)= array_delay_small_segment(h)-
AR_model_half_small_segment*array_delay_small_segment(h-p:h-1);

first_coeff =(2/var_forward_error_half_small_segment)*(error_small_segment(h))*(error_half_small_s
egment(h));
second_coeff _=-
1+(var_forward_error_small_segment/var_forward_error_half_small_segment)*(((error_small_segme
nt(h))^2)/var_forward_error_small_segment);
third_coeff _=(1-(var_forward_error_small_segment/var_forward_error_half_small_segment));
small_w(h)=(1/2)*(first_coeff _+second_coeff _+third_coeff _)+original_bias;
big_w(h)=sum(small_w);

end
plot(big_w);
pause(10);
end

elseif(n== (start_sample_region+((end_sample_region-start_sample_region+1)/2)))
end_half_small_segment= (start_sample_region+((end_sample_region-
start_sample_region+1)/2));
initial_sample=[];A=[];E=[];P=[];AR_model_half_small_segment=[];
initial_sample=delay_data(start_sample_region:end_half_small_segment) ;
initial_sample=initial_sample-mean(initial_sample);
[A,E] = ARBURG(initial_sample,p);
A=-A(2:p+1); %AR coefficients
AR_model_half_small_segment=A;
ARRAY_AR_model_half_small_segment(segment_index,:)=AR_model_half_small_segment;
var_forward_error_half_small_segment=E;

elseif(n >(end_sample_region))
Y{n} = x(n-1:-1:n-p);
e(n) = x(n) - a(n-1,:)*Y{n};
Cov_e(n)=var(e);

```

```

Cov_w(n)=(UC/p)*TRACE(P{n-1});
array_cov_w(n)=Cov_w(n);
Cov_w(n)=alpha*Cov_w(n-1)+(1-alpha)*Cov_w(n);
P_estimation= P{n-1}+Cov_w(n-1); % P_A(t/t-1)
K = (P_estimation)* Y{n} / ( Y{n}*P_estimation* Y{n} + Cov_e(n));
a(n,:) = a(n-1,:) + K*e(n);
P{n} = (P_estimation - K*(P_estimation*Y{n}));

%-----%
%% Segmenting the forwarding error prediction %%%
% Grap the var(e) and e(t) of the "updated" AR model and the var(e)
% and e(t) using a truncated for AR model. Then compare entropy of
% both outcome models
%-----%
segment_counter=segment_counter+1;
forward_error_small_segment(n)= x(n) -
AR_model_small_segment*Y{n};
%Small segment is the one analysed with the fixed AR model
forward_error_big_segment(n)=e(n);
%Big segment is the one analysed with the dynamic AR model
var_forward_error_big_segment=Cov_e(n);

first_coeff_=(2*((forward_error_small_segment(n)*forward_error_big_segment(n))/var_forward_error_
small_segment));
second_coeff_ = -
((1+var_forward_error_big_segment/var_forward_error_small_segment)*((forward_error_big_segmen
t(n))^2/(var_forward_error_big_segment))) ;
third_coeff_ = +(1-(var_forward_error_big_segment/var_forward_error_small_segment));

conditional_cross_entropy(segment_counter)=(1/2)*(first_coeff_+second_coeff_+third_coeff_)+bias;
cummulative_conditional_cross_entropy(segment_counter)=sum(conditional_cross_entropy);

[max_cummulative_conditional_cross_entropy,index_max_cummulative_conditional_cross_entropy]
= max(cummulative_conditional_cross_entropy);
delta=max_cummulative_conditional_cross_entropy-
cummulative_conditional_cross_entropy(segment_counter);
difference_peak_down=segment_counter-index_max_cummulative_conditional_cross_entropy;
%&( max_cummulative_conditional_cross_entropy > 0.8*change_threshold)

if(factor_check_negative_slop-segment_counter == 0)
slopes_ = cummulative_conditional_cross_entropy(segment_counter)-
cummulative_conditional_cross_entropy(segment_counter-original_factor_check_negative_slop+1);
slopes_=((slopes_)/original_factor_check_negative_slop);
if ((slopes_ < 0)&(cummulative_conditional_cross_entropy(segment_counter) < 0))
bias=0.25+bias;
end

factor_check_negative_slop=factor_check_negative_slop+original_factor_check_negative_slop;
end

%if (( (delta > change_threshold)&(segment_counter > 300)&(difference_peak_down < 200)))
(n=length(delay_data)))
%if (( (delta > change_threshold)&(segment_counter > 150)&(difference_peak_down < 150)))
(n=length(delay_data)))
if (( (delta > change_threshold)&(segment_counter > 200)&(difference_peak_down < 200)))
(n=length(delay_data)))
segment_end_point=segment_end_point+segment_counter;

combined_cummulative_cross_entropy_(start_sample_region:end_sample_region)=zeros(1,end_sam
ple_region-start_sample_region+1);

```

```
combined_cumulative_cross_entropy_(end_sample_region+1:end_sample_region+segment_count
er)=cumulative_conditional_cross_entropy;
```

```
segment_start_point=segment_end_point+1;
    %time_segmentation(length(combined_cumulative_cross_entropy_))=1;
    time_segmentation(n)=1;
    conditional_cross_entropy=[];
    cumulative_conditional_cross_entropy=[];
```

```
    time_segmentation_array(segment_index)=n
    if (segment_index ==1 )
        index_initial=1;
    else
        index_initial=time_segmentation_array(segment_index-1);
    end
```

```
    index_final=time_segmentation_array(segment_index);
    Entropy(segment_index)= entropy(x(index_initial:index_final))
```

```
    segment_counter=0;
    segment_index=segment_index+1;
    start_sample_region=n;
    end_sample_region=start_sample_region+buffered_samples-1;
```

```
end
```

```
end
```

```
end
```

```
%----- End of the program -----
```

```
time_segmentation_array
```

```
Entropy
```

```
total_entropy=entropy(x);
```

```
figure(4);
```

```
stem(time_segmentation_array,Entropy/(total_entropy));
```

```
coefficients_length=100;
```

```
figure(1);
```

```
plot(x-min(x),'b');
```

```
figure(2);
```

```
plot(combined_cumulative_cross_entropy_);
```

```
hold on;
```

```
stem(time_segmentation_array,(max(combined_cumulative_cross_entropy_))*1.2*ones(1,length(ti
me_segmentation_array)), 'r-');
```

```
counter_seg=1;
```

```
offset_=0;
```

```
for t = 1:1:length(delay_data);
```

```
    segmented_series(t-offset_)=x(t);
```

```
    if (t==time_segmentation_array(counter_seg))
```

```
        counter_seg=counter_seg+1;
```

```
        ACF_X_=akfrader(segmented_series,coefficients_length);
```

```
        offset_=t;
```

```
        plot(ACF_X_);hold on;
```

```
        segmented_series=[];ACF_X_=[];
```

```
    end
```

```

end
x=x-min(x);

start_point=1;
for t = 1:length(time_segmentation_array);
    end_point=time_segmentation_array(t);
    variance_array(t)=var(x(start_point:end_point));
    entropy_array(t)=entropy(x(start_point:end_point))
    start_point=end_point+1;

end
mean(variance_array)
mean(entropy_array)
length(variance_array)
figure(5);
plot(array_cov_w)

figure(6);
plot(a(:,1),'b');
figure(7);
plot(a(:,2),'b');

```