A Network Impairment Tool based on the IXDP425 Network Processor

by

Hai Pham

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Master of Science

Department of Computer Science
Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

A Network Impairment Tool Based on the IXDP425 Network Processor

BY

Hai Pham

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Contents

Chapter 1

Introduction

In the last few decades, we have witnessed the creation of an increasing number of network interconnection media and communication protocols. We have also observed a new trend towards implementing traditional software components as services on embedded devices. These two factors are putting increasing pressure on the verification process. The purpose of verification in the context of networking is to evaluate protocol compatibility, effectiveness, efficiency, and interoperability, as well as implementation correctness.

Two categories of tools are used during the verification step: network simulators and network emulators. Network simulators are used in the early stage of protocol development to select the best suitable protocol for a particular type of application. Simulators such as NS2 [5, 27] and OPNET [17] are very comprehensive and flexible. Not only can they simulate almost any existing protocols, but they can also be quickly used to develop simulation scenarios for new protocols. Despite all those advantages, simulators are not suitable when more realistic and subjective testing are required. In such situations, network emulators, with real code implemented on real devices, are used to meet the necessary requirements. Generally, emulators are less extensible and more expensive to build compared to simulators. On the other hand, an emulator can often simplify the verification process because it is fine-tuned and stream-lined out of the box. The users just need to connect the devices to the emulator; this is much more friendly than having to script all the needed simulation scenarios.

The objective of my thesis work is to develop a network impairment tool that can emulate common network faults (i.e. packet lost, delay, duplication and reordering) at wire speed. The tool will allow developers to verify and stress test their products and services under such faulty conditions. The use of the IXDP425 as an implementation platform will

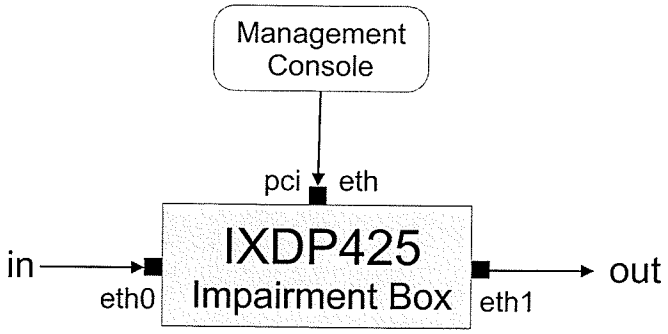

Figure 1.1: Network Configuration.

provide a high level of flexibility and efficiency at reduced cost. Figure 1.1 represents the typical system layout. The packets flow through one port of the impairment platform, and after being manipulated (impaired), they come out on the other port.

I outline several requirements that any sophisticated network impairment tool needs to support below:

High performance: Packet manipulation imposes overhead and the overhead needs to be carefully controlled so that it does not affect throughput and other impairment requirements. As an example, if a packet takes 50 μs to traverse through the impairment module, then any emulation of packet delay smaller than 50 μs will not be possible.

Flexibility: The platform needs to provide a complete set of well-known models for packet loss, duplicate, reordering and delay.

Transparency: External devices and/or applications should not have to be modified to use the tool.

Robustness: The impairment platform must be able to run smoothly for a long period of time, and to support dynamic configuration. Also, the sequence of emulation events, though randomly generated, must be reproducible.

Low cost: The total cost of the impairment platform should be within a few thousand dollars, so that it is accessible to both education institutions and small businesses.

My thesis work will specifically focus on the performance, flexibility, and cost aspects, which are not addressed sufficiently in the current solutions. To this end, I will use the `Intel IXDP425 Network Processor Evaluation Board` [15] (I will, from this point forward, simply refer to this system as IXDP425 BOARD) as the core component. The flexibility, robustness, and efficiency of the IXDP425 BOARD make it an ideal choice for use as an impairment tool.

The rest of this thesis is organized as follows. The rest of this chapter will describe in detail the motivation and the related work. Chapter 2 describes the system run-time environment. Chapter 3 provides a summary of the functionalities of the final impairment system. Chapter 4 goes into details of the design and implementation of the system; the reader can safely skip this chapter if desired. Chapter 5 elaborates on the random number generation implementation and validation. The next two chapters describe the testing and performance analysis. Finally, chapter 8 summarizes the results of the project and outlines the future work.

1.1 Problem Statement

Network faults such as packet loss, duplication ,reordering and delay are common within a network. These faults affect network applications at different levels, ranging from making the applications unusable to mildly degrading their performance. To resist network faults, communication protocols are built with layers of redundancy. The complexity of these protocols makes it very hard to verify the correctness of their implementations, especially in heterogeneous systems consisting of different operating systems, interconnection media, devices, and applications.

In this thesis, I will use a network processor to design an impairment tool that can emulate the above network faults flexibly and cost-effectively at wire speed. The tool will help to verify and stress test network products and systems. It can also be used to facilitate subjective evaluation of applications such as voice and/or video codecs. Finally the impairment tool can be used to emulate the behavior of a wide range of networks, thus allowing developers to realistically evaluate their products in a controlled environment.

My impairment tool will have improvements over existing tools in terms of flexibility, robustness, efficiency, and cost. The final impairment platform will support a multi-state model, the Markov chain [38], to emulate each type of impairments. Formally, the Markov chain is defined as a finite set S of states that are controlled by a transition probability

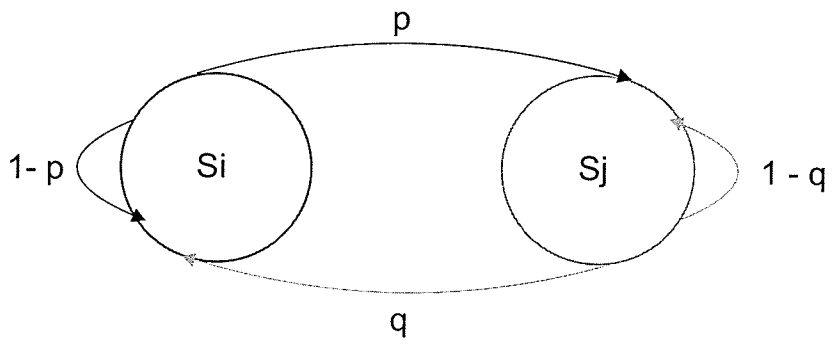

Figure 1.2: A 2-state Markov chain.

matrix P. The matrix P associates each pair of states with a value $0 \leq p_{ij} \leq 0$ to represent the probability of moving from state i to state j given that the system is currently in state i. Thus, the system will swing between states based on the value in the matrix P. Figure 1.2 illustrates this concept.

Furthermore, the platform will be able to run for a prolonged period of time without degrading the packet throughput. Once the other network devices are hooked up to the platform, the user can confidently control various emulation aspects remotely from a web-based console. Finally, with the hardware-accelerated network processing capacity provided in the IXDP425 BOARD as well as the extensive access library toolkit, the development time and the total cost are greatly reduced (many other network impairment platforms have to build their own hardware from the ground up).

1.2 Related Work

I will first introduce some recent work related to network measurement and modeling. I will then describe several existing network impairment tools in the industry and academic communities, along with their basic characteristics.

1.2.1 The Nature of Network Faults

Previously, I identified network faults as packet loss, delay, duplication and reordering. As one of the uses of network impairment tools is to emulate network faults after real scenarios, it is essential to be able to accurately model the variable delay, and the patterns of packet loss, duplication and reordering. In this section, I will briefly explain the causes of the network faults, and then introduce some previous work on network measurement and monitoring. These studies lay out various methods to infer suitable models of network faults.

The majority of packet losses are caused by buffer overflow in the devices [3, 10]; only a very small portion is due to bit errors on the transmission links. Additionally, there are four types of delays of interest: propagation, processing, transmission, and queuing delay. Propagation delay is dependent only on the distance and the physical medium properties. The last three delays occur at the switching devices, and of those, queuing delay is the most variable. Of the above faults, packet duplication is actually a side effect of packet loss and/or delay, created by the communication protocols. Packet reordering occurs due to many reasons such as differential service, packet re-transmission, and scheduling. For more information, please refer to [34, chapters 7, 31, and 49].

Much work has been done to characterize Internet packet behavior. Bolot [3] uses UDP probes to show that packets are lost randomly. Paxson [28] confirms this result using TCP probes. His work also indicates that packet reordering is very common and highly variable from path to path, and that packet delay variations occur primarily on time scales of 0.1–1.0 seconds. Bi *et al.* [2] observe that there is a correlation between packet delay and packet loss. The distribution of packet delay is usually unimodal when packet loss rate is small. However, this is not the case with increasing loss rate and the delay distribution is more dispersed. Further, Bellardo and Savage [1] have developed several methods to measure packet reordering rates. By varying the probe interval, their method can be used to measure the distribution of the reordering process over time.

Some recent work [24, 37, 38] demonstrates that a Markov model can be used to approximate network behavior. Wei *et al.* [37] model Internet packet delay as a continuous-time hidden Markov model [30]. The specific parameters for the model can be calculated efficiently after a certain period of network probing. packet For simpler networks, the distribution of packet delay and packet loss can be calculated from the classic single-server and finite-buffer queuing systems M/M/1/K [35, chapter 8] with parameters represented by the packet arrival rates, the processing and transmitting time, the phys-

ical medium characteristics, and the buffer sizes of network devices. Classical queuing theory can provide us the average waiting time of packets in the queue and the average queue size, among other properties. From there, the distributions of packet delay and packet loss can be derived. In many cases, the packet delay distribution can be as simple as uniform, or normal models; and the packet loss distribution as the Bernoulli or Gilbert [12] models. Depending on the packet arrival patterns, a multi-state/hierarchical Markov model [19, 37] based on time or packets may be used.

The behavior of a complex network is highly unpredictable and cannot be approximated by a simple model. Besides supplying trivial packet fault models for simple network configuration, a good impairment tool must be flexible enough to emulate the behavior of a complex real-world network.

1.2.2 Network Impairment Emulators

Seeing limitations in the final stage of product verification, several simulators have added support for emulation. The NS2 [5] simulator has recently added an emulation module to its core. The additional module permits real traffic to pass through the simulator, leveraging the comprehensive simulation scenario library on real packet streams. Of course, tools such as NS2 cannot scale up in terms of throughput when compared to pure network emulators because the simulation core generates a significant amount of processing overhead.

Other network impairment emulators modify the packet behavior along the communication stack. NIST NET [6] provides per-flow impairment at the network layer. DUMMYNET [32] and NETSHAPER [13] operate at the data link layer. The lower the layer, the more realistic the emulator is. This is because at higher layers, as in the case of NIST NET [6], the emulator will need to probe the packet headers; thus increasing the processing time. All these three emulators are similar in the sense that the impairment module is implemented as part of the operating system (BSD and Linux) kernel. With the exception of NIST NET, which uses an external clock, these systems are limited by the timer interrupt granularity (usually ten milliseconds). This means that without any tweaking, they cannot handle packet delays smaller than ten milliseconds. My proposed impairment tool will also use the Linux operating system. However, since the IXDP425 BOARD can signal packet-related events as interrupts, the tool will not suffer from the Linux operating system's timer granularity limitation.

Instead of specifying a static impairment model, a more realistic approach is to adapt

the loss and delay models to fit actual traffic patterns. Noble *et al.*'s strategy [26] is to first observe a network, interpret the observations, and finally replay the trace to inject the packet loss and packet delay. ENDE [39], on the other hand, calculates the packet delay in real-time. The core component of ENDE has three Ethernet interfaces: two of them connect to client and server machines, and the third connects to a real network whose behavior ENDE is trying to emulate. It periodically probes the network to retrieve real-time packet behavior, and then uses that information to define the emulated delay between the client and the server. Using this method, the client and the server reside on the same network but can still experience reasonably accurate Internet packet delays. The obvious drawback is that ENDE needs to be connected to a real network whose behavior is neither predictable nor reproducible.

Impairment tools are also very popular in the commercial world. They are typically powerful but also very expensive. The SPIRENT SX data link simulator [7] is probably the most popular impairment tool on the market. It supports many pluggable network interfaces such as DS1, DS3, E1, E3, and SONET. Naturally, for the above features to function, the SPIRENT SX allows users to plug in an external clock signal. It does not implement complex probability distributions for the delay and loss; instead, exact values (in microsecond unit) are used. To make up for this deficiency, SPIRENT SX allows users to define different parameters for up to 99 steps in a sequence. The minimum duration of each step is one second. With this feature, the user can observe a real network for a number of time slots, and then input the impairment parameters into these 99 steps. Nevertheless, the lack of exact probability distributions is a major limitation. In addition, packet loss, duplication, and reordering are not supported. Finally, the SPIRENT SX's management console is accessible either on the display panel of the box, or remotely through IEEE-488 or RS-232 interfaces. Even then, the remote control feature is still less accessible compared to a web-based management console.

In contrast to the SPIRENT SX, Maxwell's NETWORK IMPAIRMENT SYSTEM (NIS) [21] is highly flexible in terms of software implementation. Users can develop and deploy their own packet handling code through a set of published application programming interfaces. All possible types of packet faults are supported. The NIS can update the impairment parameters automatically in real-time based on the devices' response rates (similar to ENDE). A Java-based management console controls the NIS through the TCP/IP protocol. The Maxwell's NIS has dual CPUs running the Linux operating system, and supports remote access through `ssh` or `telnet` protocols. However, it has very limited

probability distributions for the impairment models.

There are also other network impairment tools. Most of them, however, are implemented on traditional personal computers. They are characterized by high packet processing overhead and very simple impairment models. While almost every tool claims that it can reach the maximum wire speed, what they do not mention is that their high overhead drives out certain impairment scenarios, where the time granularity is in terms of microseconds. Further, due to the limited buffer size on the network interface, the higher the overhead, the lower the packet throughput.

1.3 Network Processors

I will give a brief overview of the general architecture, applications, and some details on the components of the IXDP425 BOARD.

1.3.1 General Features

With the increase in bandwidth use and rapid development of new network protocols, comes the need for a general purpose, programmable device that is optimized for packet processing. These devices, called network processors, have exotic architectures with unconventional system architectures, enabling them to process packets at high speed. To explore these architectures, we first need to understand the nature of packet processing. Packet processing can be classified into data-plane and control-plane processing. The former performs operations like packet header checking and forwarding (tasks with a small number of instructions). Because data-plane operations must be completed in a short time to prevent packet drop, they are usually handled by multiple specialized processing units. On the other hand, control-plane processing deals with tasks that have long code segments and are more suitable for execution on a general purpose processor such as the StrongArm or the various Intel/AMD processors. Other factors such as the bus speed, the memory hierarchy, and the level of hardware-supported multithreading also influence packet processing performance.

On top of all the hardware components, a software development framework in the form of libraries or domain-specific languages allows the developer to program the network processor. Unfortunately, due to the unusual architectures and the high-throughput requirement, it is more difficult to design and implement applications for a network

processor than for a general purpose processor [20]. Developers often have to code in a mix of low and high level programming languages to maintain high performance.

For more detail on the architectures and the programming models of network processors, please refer to the recent survey by Peyravian and Calvignac [29].

1.3.2 The Intel IXDP425 Board

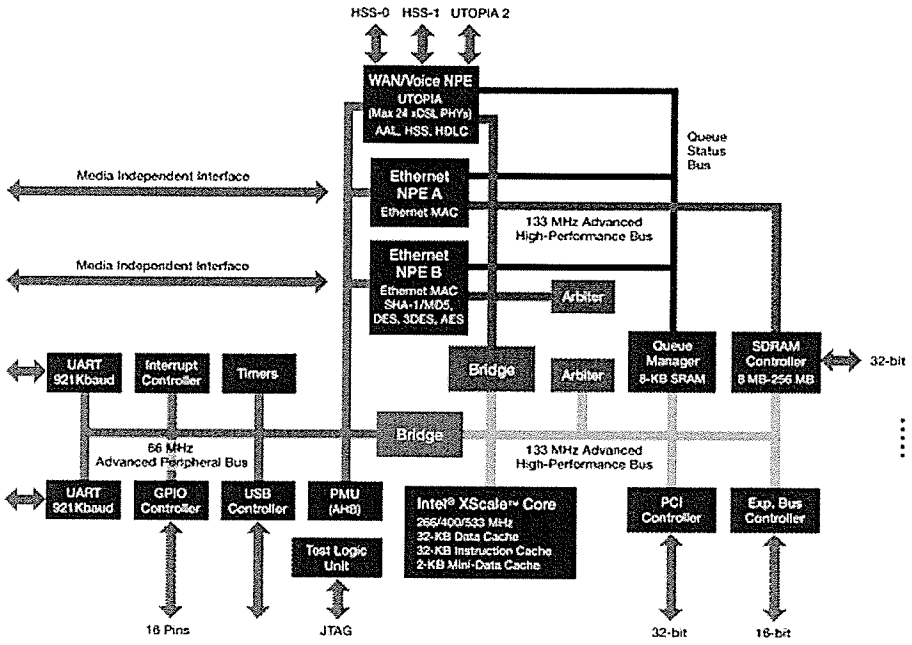

Figure 1.3: The IXDP425 Board Block Diagram [14].

The Intel IXDP425 BOARD (Figure 1.3) is targeted at the LAN/WAN market. It is a complete board with an XScale RISC processor core running up to 533 MHz, and with up to 256 MB of SDRAM. Two integrated 10/100 Ethernet NICs are connected to the XScale core, the SDRAM controller, and a queue manager through a 133-Mhz bus. The queue manager facilitates packet queuing in hardware for various components. For the Ethernet module, two queues holding up to 128 packets each are provided. The Ethernet module also provides a software queue to temporarily buffer packets in case the hardware queues are full. The software can use interrupts or polling to inquire about the hardware queue status. Accompanied with the XScale core are three Network Processor Engines (NPEs) capable of executing instructions in parallel with the main processor. The NPEs help to

offload common computing intensive operations like IP header inspection and packet filtering from the XScale core.

In terms of software, the IXDP425 BOARD supports two operating systems: VxWorks [31] and Linux. The interfacing library toolkit, named access library, consists of multiple layered modules, providing a rich set of accessible functions. Under Linux, this access library is implemented as a kernel module which somewhat reduces the level of freedom of the developer. The public application programming interface (API) is in the form of public C functions, sharing the kernel-mode memory space. This setting means that in order to use the API, the application has to be written as another Linux kernel module. Overall, however, the access library makes the IXDP425 BOARD much easier to program than many others.

Chapter 2

Environment

The run-time environment consists of three main components: the IXDP425 BOARD, a transmitting source and a receiving station. The three components are directly connected to each other using cross-over Ethernet cables. This selection saves the need for a hub in the middle. It is important to emphasize that the IXDP425 BOARD acts as a bridge, therefore the two ports involved (from here on I will refer to them as PORT1 and PORT2) do not require an IP address. The transmitting source and the receiving station reside in an isolated subnet.

2.1 Transmitting Source / Host System

The transmitting source is a regular PC running Mandrake 9.2 operating system. This machine is also the host system for the IXDP425 BOARD whereby it provides a root file system through the `nfs` carrier protocol. Two network cards are present on the machine, The first network card connects to PORT1 of the IXDP425 BOARD; this connection is used to transmit test data through the IXDP425 BOARD. The second network card is connected to the PCI network card on the IXDP425 BOARD and is used to transmit binary images of the code and other control information. It is used as a way to remotely control the board.

2.2 The IXDP425 Board

The IXDP425 BOARD has two 10 – 100-Mbit hardware-accelerated Ethernet ports. It also comes with another Ethernet card on one of the PCI slots. The IXDP425 BOARD is

loaded with Redhat's Redboot 1.92 boot loader and MontaVista 2.1 Linux kernel image (a modified version of Linux kernel version 2.4.17). This kernel version has proprietary driver for the hardware-accelerated Ethernet ports, as well as optimizations to make the kernel more real-time friendly. In addition, the kernel is compiled to load the root file system from the host system. Finally, this is a very bare-bone Linux distribution with only the absolutely necessary components. Because no standard libraries are provided, all programs have to be statically linked with the libraries.

The IXDP425 BOARD is accessible through a serial connection. On the host system, `minicom` is used to provide this connection.

Chapter 3

Impairment Capacities

All the impairment modes are equally implemented for both Ethernet ports on the IXDP425 BOARD, permitting separate impairments on the `forward path` as well as the `reverse path`. For the forward path, the packets are impaired within PORT1; for the reverse path, it is within PORT2. Likewise, statistical data are collected for each port individually.

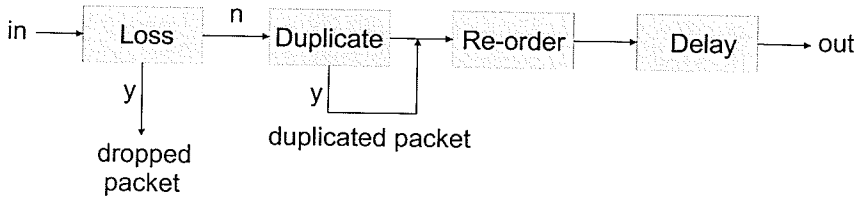

Figure 3.1: Packet traverse path within each Ethernet port.

Figure 3.1 displays the traverse path for each packet in each Ethernet port. Each rectangle box represents an impairment decision node. If the underlying model implies a YES answer, the impairment is applied. Except for the YES decision in the `loss` emulation where the packet is dropped immediately, packets will continue to traverse to other impairment decision nodes. In the `duplicate` box, an exact copy of the packet may be created. In that case, both the original packet and the duplicated one continue to travel to the `reorder` box.

Note that the impairment is done at the data-link layer. As such, there is no support for individual packet flow (a packet flow is defined as a traffic path from one IP address to another). The tool is best used in an isolated testing environment. Packet impairment

emulation at data-link layer has less processing overhead then at higher layers of the protocol stack.

Statistics are collected on a per-port basis. The information gathered are the number of packets received, transmitted, dropped, lost, reordered, and duplicated. The statistical module also provides snapshot information regarding the packet inter-arrival time, inter-departure time, the targeted packet delay, and the actual packet delay. In addition, the user can choose to collect a continuous sample of data which includes the emulated delay time, the packet processing overhead, the internal software queue size, and the inter-arrival and inter-departure time. This information is useful for model validation and performance analysis.

Two emulation modes are supported: model-based and trace-based. Each will be described in more detail below.

3.1 Model-based Emulation

The model-based emulation mode uses probability distributions to approximate the actual data patterns.

The emulation for packet delay consists of two components: a fixed delay plus a variable delay; both are in units of microseconds. The variable delay can be modeled by the UNIFORM and NORMAL distributions.

Packet loss and duplication are modeled by a BERNOULLI variate. The input is a value in the range $(0, 1)$ to indicate the proportion of packets to be emulated as lost or duplicated. More complex model such as the GILBERT loss model can be created with the help of the multi-state Markov chain.

Packet reordering has two components. The first one is the underlying BERNOULLI model similar to the one of packet loss and duplication to decide whether or not to reorder the packet. The second component is the `displacement distance`, which specifies how many positions the packet will be displaced with respect to its original position. The displacement distance component is modeled in a similar manner to packet delay, i.e. by a discretized UNIFORM and NORMAL distribution.

The key strength of the modeling system is the multi-state Markov chain. The Markov chain is characterized by the number of states, the duration and the impairment specifications in each state, and the transition probability matrix. The durations in the states are measured either in units of the number of arrived packets or the amount of elapsed time

since the beginning of the state. The transition probability matrix is used to calculate the next state.

Each mode of impairment can have its own Markov chain. Since we have four different impairment types, we could then have up to four different Markov chains in use concurrently. Nested Markov chain (within another Markov chain) is not supported however.

Let's illustrate the Markov chain concept by a concrete example. Consider a simple bursty loss scenario in which the packet stream has a bursty loss probability of 3%; and when that happens, 10 consecutive packets are to be dropped. Also, once a bursty loss occurs, the chance that we have another bursty loss immediately is 1%. We can model the above requirement using a two-state Markov chain. The first state depicts a usual packet stream; we set the length of this state to one packet. The second state depicts the bursty loss, and according to the requirement, we will set the length of the state to ten packets. In state one, no packet is dropped, thus the loss model's drop probability is 0%. On the other hand, in state two, we will have to drop all packets; so we will set the drop probability to 100%. Now we have to specify the transits from one state to another. At the end of each state (i.e. after the specified number of packets have arrived), the Markov chain model will decide the next state to move to based on the probability transition matrix. From the requirement above, we can formulate the probability transition matrix as:

$$P = \begin{pmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{pmatrix} = \begin{pmatrix} 0.97 & 0.03 \\ 0.01 & 0.99 \end{pmatrix}$$

We can emulate more complex models, such as varying the number of packets within each bursty loss, by adding more Markov states. In fact, any scenario that requires a nested Markov chain can be re-modeled into one Markov chain with additional states.

3.2 Trace-based Emulation

This mode assumes that the user has collected a set of real data. Each trace entry consists of the following fields:

port number, loss, duplication, reorder distance, delay time

The range of values are:

- Port number: 1 for port one and 2 for port two;
- Loss and duplication: 1 to imply a loss or duplication (different fields), 0 otherwise;
- Reorder distance: 0 to imply no reordering; non-zero positive value to represent the displacement distance (in term of packets);
- Delay time: 0 to imply no delay; non-zero positive value to represent the delay time in microseconds.

For instance, the entry $\boxed{1, 0, 0, 0, 30000}$ indicates that the associated packet arriving at PORT1 will be delayed for $30,000$ μsec.

The traces are applied in circular order starting with the first trace entry. Each entry is associated with one packet; when the last entry is reached, the sequence starts again from the first trace entry. This mode is most applicable in scenarios when the user can get hold of the traffic log. The user then needs to transform data from the log format to the trace format outlined above using one of the many available scripting languages. The field `reorder distance` is less useful in this mode as it is usually not possible to observe packet reordering from the traffic log.

Chapter 4

System Design and Implementation

Figure 4.1 displays three main components of the system:

1. The impairment core (implemented as a kernel module);
2. The web-based management console; and
3. The user-mode `server` program.

The major efforts are concentrated in the web-based management console (from now on I will refer to it as web-app) and the impairment kernel module. They will be described in detail below. The `server` program is just a simple application that tunnels messages from the web-app to the impairment kernel module, hence its detail will be skipped.

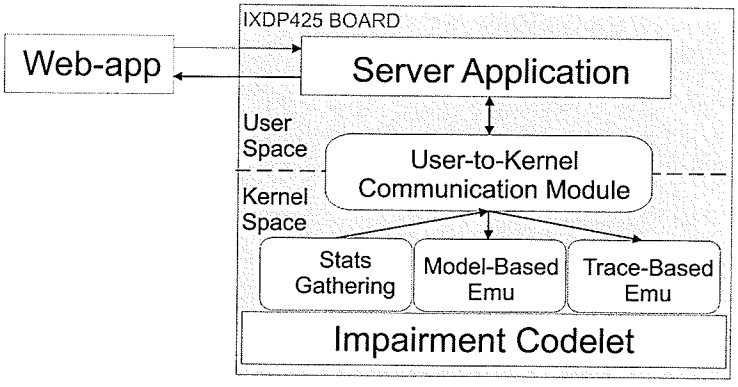

Figure 4.1: Software components.

4.1 The Impairment Kernel Module

Linux is a monolithic operating system where all the sub-systems are located within the kernel. The kernel code runs in a privileged mode (kernel-mode), having direct access to the system data and hardware. User applications run in the user-mode, and can only access system functions exported by the kernel. Kernel module is a way to extend the Linux operating system; hardware drivers and sub-systems are implemented as kernel modules.

The impairment codelet is built on top of the IXDP425 BOARD's access library, which in turn is written as a Linux kernel module. As a result, the impairment core also has to be developed as a kernel module (it is not possible to access kernel-mode's functions from the user-mode). Programming in kernel-mode is more complicated than in the user-mode because there is a number of limitations on what the programmer can do. The next few sections describe about some of the issues encountered while developing the impairment kernel module.

4.1.1 Data Structures

The impairment core has four software queues to implement various impairment modes (two for each port). One queue stores the delayed packets and the other stores the re-ordered packets. Packets in the reorder queue are arranged in increasing order based on the displacement distance, which is decremented each time a packet is transmitted. When the displacement distance reaches zero, the packet is moved to the delay queue.

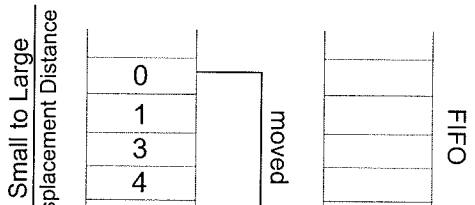

Figure 4.2: Interaction between the impairment core's software queues.

The IXDP425 BOARD already has hardware queues to buffer incoming and outgoing

packets. The `access library` provides additional software queues to take care of the case when the hardware queues are full. The software queues in the impairment core further increase the packet processing overhead. In particular, the semantic of the delay queues dictates that packets are to be transmitted in strictly FIFO (first-in, first-out) order. This means that a packet may be delayed longer than it is supposed to. Consider one particular moment where the delay queue of PORT1 has more than one packet; the first packet has a delay of 80-μsec, and the second of 40-μsec. Since the second packet has to wait for the first one, its delay is actually 80-μsec, twice longer than planned. The higher the number of items in the queue, the more severe the problem.

4.1.2 Issues with Linux Kernel Module Programming

As mentioned earlier, kernel modules are an integral part of the Linux operating system. They can be loaded and unloaded dynamically at run-time, making the kernel highly flexible. The general guideline for kernel module programming is to make the code as simple as possible. This is to reduce further risks that can de-stabilize the kernel. The consequence is that many common programming constructs and tools that the programmers have taken for granted in the user-mode are no longer available. Most notably lacking are the standard *C* library, I/O access, networking stack and floating point operations. In contrast to this, the IXDP425 BOARD's access library is a large kernel module. The impairment core is also another big kernel module. In fact, it is more of a regular application (albeit developed within the kernel-mode) than a hardware driver. As such, many kernel limitations need to be overcome. They are detailed below.

First, the kernel itself exports only a few number of functions that are accessible to all kernel modules. The list of these functions can be retrieved using command `cat /proc/ksyms`. Each kernel module can choose to export its functions as well. Therefore, the `ksyms` list can be different from system to system, depending on which modules are loaded at that time. Several common routines in the standard *C* library such as `string` handling have the equivalent versions in the kernel-mode. Other than that, if the programmer needs a routine in the standard *C* library, it should be re-implemented.

The lack of I/O and networking stack are addressed in this project by the user-mode `server` program. This program acts as a bridge between the web-app and the impairment kernel module. It listens to incoming commands on a TCP port and then tunnels the messages through the module's proc file system. The proc file system is a virtual file system that is designed to retrieve and to configure the kernel settings at run-time. As this

is the only viable way to communicate with the impairment kernel module, the proc file handler is implemented as a stateful communication sub-module. It handles emulation parameter settings and statistics-related messages, among others.

The next obstacle, no floating point supported in the kernel, is addressed by a soft emulation library. Hardware-supported float operations cannot be carried out within the kernel due to the inability to save floating point state during context switch. In any case, they are not relevant in this thesis because the IXDP425 BOARD does not have a floating point arithmetic logical unit. However, float operations are unavoidable during the random number generation process (see chapter 5), thus the use of the soft emulation library.

The final obstacle with kernel module programming is with the debugging process. Unlike a user-mode program, when a fatal error occurs, the entire system crashes. Because of the shared memory space, the impairment module is mangled with other modules. The most practical way to trace an error is to investigate the list of exported functions and their addresses in the kernel before it crashes and then cross reference to other binary symbols of the kernel module in question. Needless to say, this is a time consuming process.

4.1.3 Interrupt and Tasklet Handlers

Interrupt mechanism in Linux kernel is divided into two parts: the top-half (interrupt handler) and the bottom-half (tasklet handler). The top-half executes a short code and returns quickly. Its usual purposes are to acknowledge an interrupt and to transfer any data into a buffer for processing later by the bottom-half handler. In other words, any work that requires many CPU cycles is carried out in the bottom-half handler, which is scheduled at a safer time. The key difference is that during the execution of an interrupt handler, all interrupts are disabled; but they are enabled during the execution of a tasklet handler. Thus the use of a tasklet handler enables the processor to service more interrupts per unit time.

A tasklet handler is used by the impairment core to process incoming packets. Unfortunately, in order to address race condition and data corruption, the design requires interrupts to be disabled within the tasklet handler, defeating the original purpose of the bottom-half handler. This particular handler takes a long time to execute. With the interrupts disabled, certain side effects are observed and will be discussed at the end of this section.

Even though interrupt handlers and tasklet handlers are not scheduled, race condition and data corruption can still occur, even on a uni-processor machine. Consider the scenario illustrated in table 4.1.

Time Sequence	packet_received_handler	tasklet_handler
1		See the queue NOT empty
2	See the queue NOT empty `(interrupt occurs)`	
3	Process the queue	
4		Send out all packets in the queue

Table 4.1: An execution scenario that could cause data access violation.

The processor just finishes checking the state of the queue from the tasklet handler and sees that the queue is non-empty, then an interrupt occurs to process an incoming packet. The interrupt can manipulate the shared queue in a way such that when the tasklet handler is executed again, the queue is empty; thus causing the kernel to crash. Clearly, to avoid access violation, any interrupt and tasklet handlers that access shared data structures need to disable interrupts.

A long running interrupt handler or a tasklet handler (with interrupts disabled) can cause many side effects on the system. Consider the simplest case with the clock interrupt. The system relies on the clock interrupt to keep the time. During the time all interrupts are disabled, clock interrupts are not processed, causing the system clock to be out-of-sync. This symptom is observed with the impairment core; the internal clock counter would lag behind in terms of milliseconds after each cycle of 65 seconds.

4.1.4 Packet Scheduling

Timing granularity is a critical issue in the impairment core because packet delay emulation can require delays as low as a few microseconds. The Linux operating system was not originally designed as a real-time system; its clock granularity can only go down to ten milliseconds. Many network impairment implementations rely on the Linux kernel timer to schedule packets. Those projects address the Linux's clock granularity problem by either using an external timing device or by modifying the kernel to respond to the clock interrupt more frequently, thus reducing the timer granularity. The latter approach can potentially thrash the system because if the response frequency is too high, the kernel does not have any time slots left to carry out real work.

My work employs a different approach. To schedule packets with time granularity of microseconds, a tasklet handler is used. The frequency with which the tasklet handler gets executed is directly proportional to the frequency of the interrupts. The nice thing is that this does not rely only on the clock interrupt; in fact, any other kind of interrupt would do. In this case, every time a packet is received, an interrupt is generated. Thus the higher the incoming packet rate, the more frequent the tasklet handler is executed, and the more accurate the packet's release time is.

The packet's arrival and release times are stamped with the IXDP425 BOARD's internal clock, which is updated at every bus cycle. Several issues are observed with the internal clock counter. The clock counter is of type `unsigned int`, thus its range is from 0 to $2^{32} - 1$. With the bus running at 66-Mhz, the counter is reset (after reaching the maximum value) at roughly every 65 seconds ($(2^{32} - 1)/(66 * 1000 * 1000)$). So, packet delays cannot be longer than a minute, which is nevertheless acceptable for most practical uses. Special care must be taken when the clock value is close to the maximum value, because the addition of an emulated packet delay to the packet's arrival time can overflow the value.

4.2 The Web-based Management Console

The management console is a web-based application (web-app) written in the Java programming language. The web-based management console allows users to remotely manage the impairment system. The Java programming language, while generally suitable for web development, turns out to be a shortcoming in the final stage of the development of the impairment tool. The software required to run the web-app includes the Java runtime engine and the Java-based application server. To sever the dependency on the host machine, these two components must be embedded in the flash memory of the IXDP425 BOARD. This process is much more simple if the web-app had been written in a scripting language such as `perl`.

The communication protocol between the web-app and the user-mode `server` program is in ASCII. Thus, re-writing the web-app in another programming language is trivial.

Below are several screenshots of the current Java web-app. The first two pictures show the input screen for the model-based emulation mode. On the left, we have the menu for each individual impairment types. On the central right is the main screen for the currently selected impairment. Whenever the user changes the probability model, the screen will update with the new fields. At the bottom of the screen are six small fields representing the seeds of the multiple recursive random number generator (MRG). Setting these fields will force the system to use the MRG. Finally, the Synchronize with Server button updates the interface with the impairment parameters currently set within the impairment box.

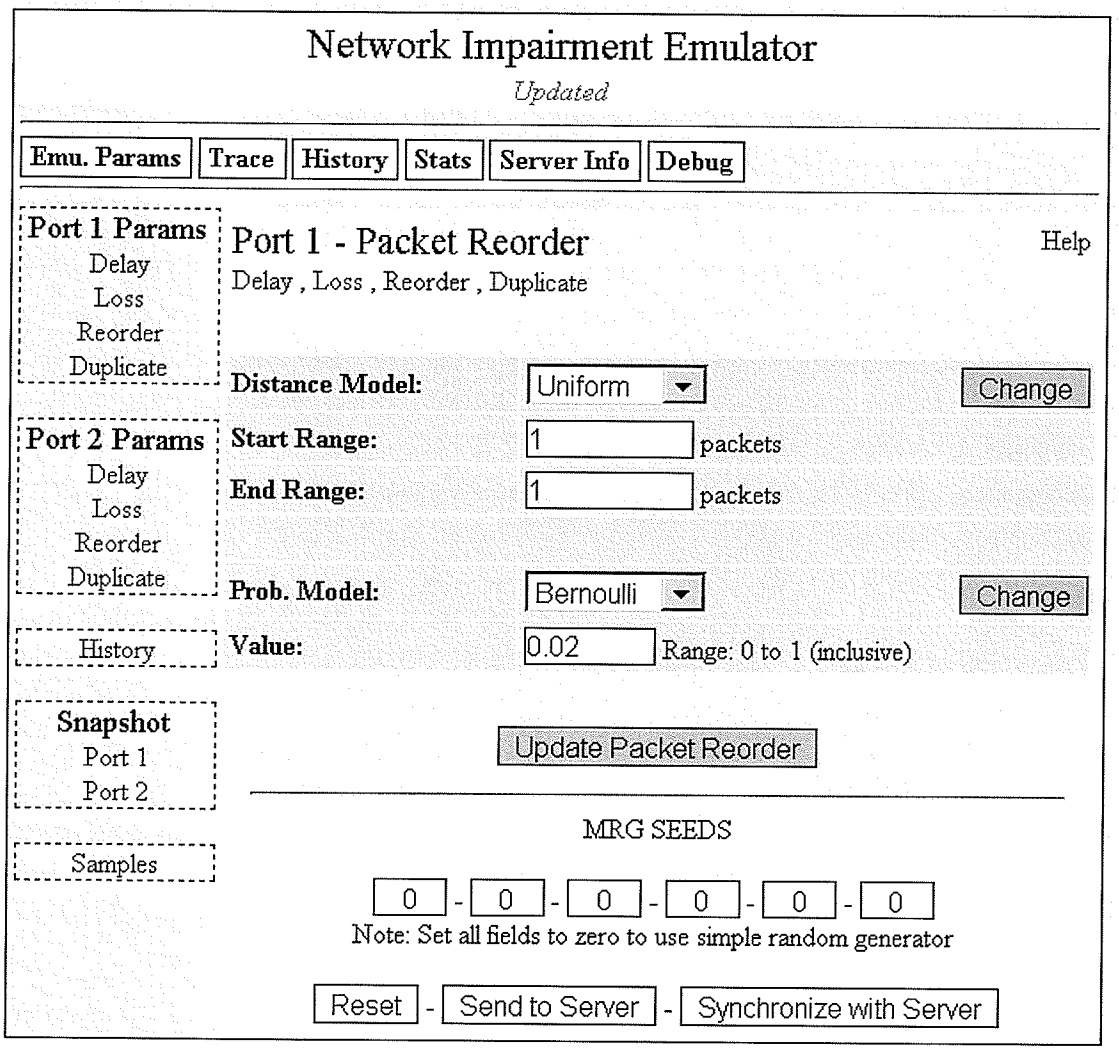

Figure 4.3: Packet reordering parameter setting for model-based emulation mode.

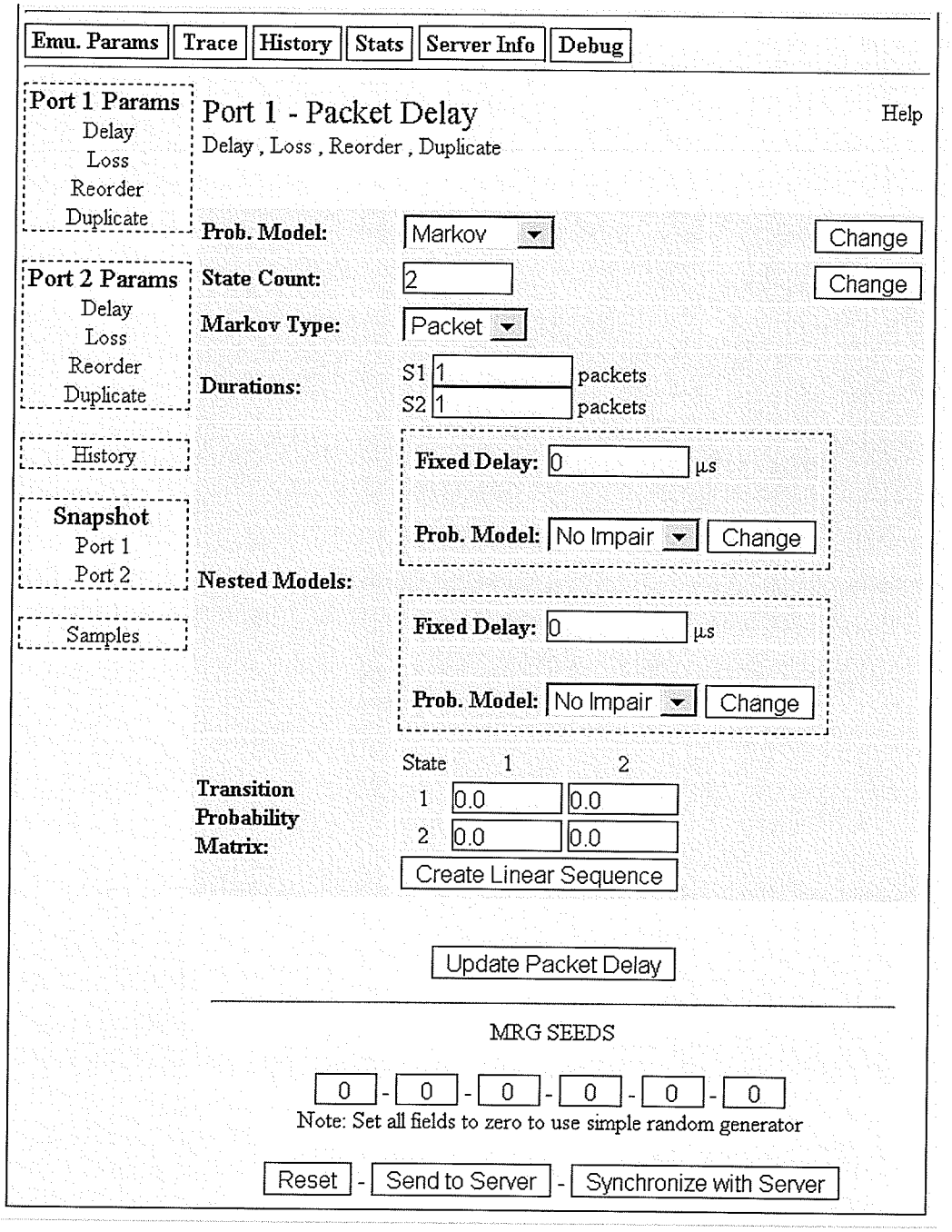

Figure 4.4: Packet delay parameter setting for model-based emulation mode.

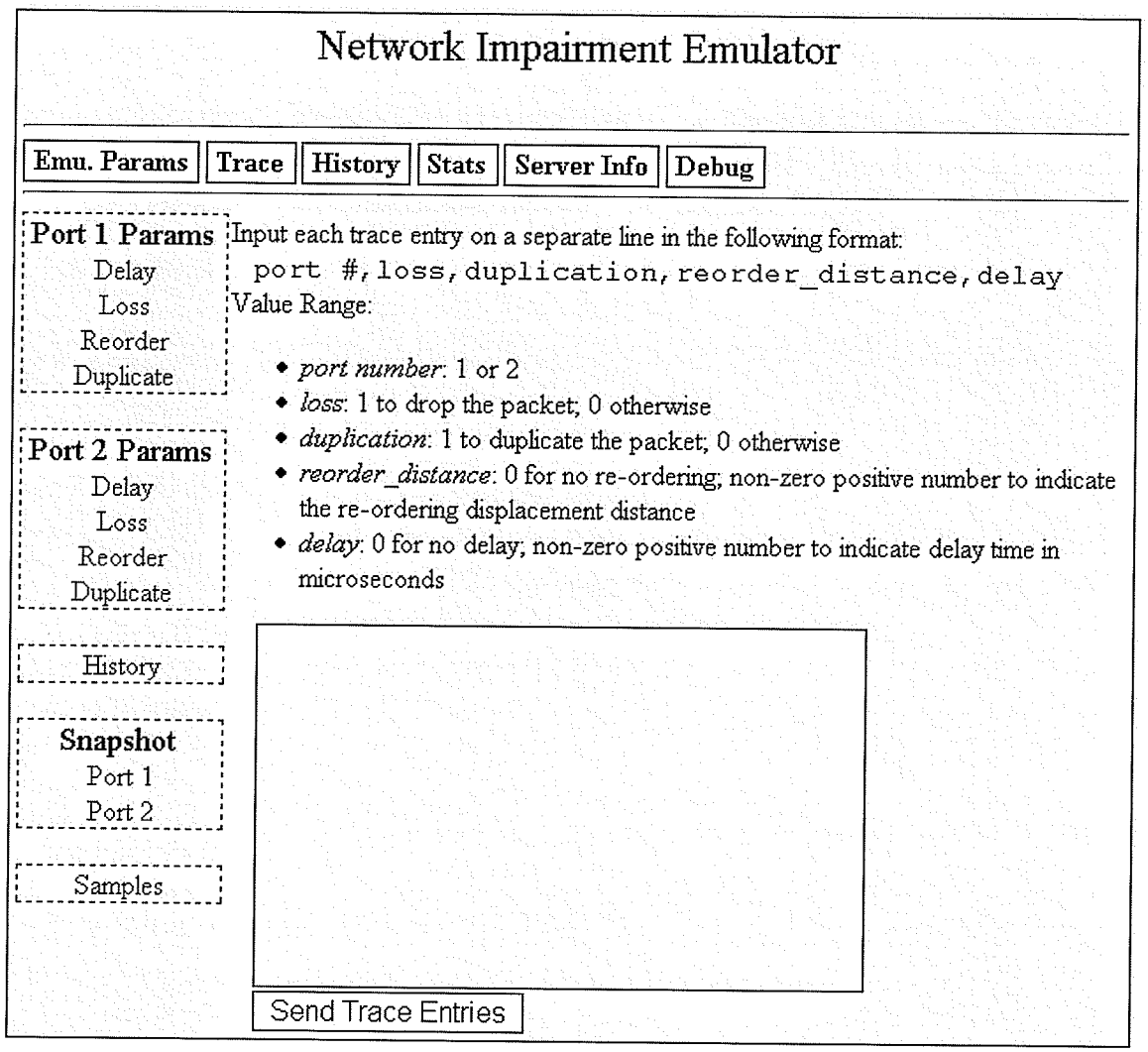

Figure 4.5: Trace entries input.

Chapter 5

Random Number Generation and Validation

Random number is an essential instrument in the model-based emulation mode. Within the scope of this thesis, the random numbers do not need to be truly and highly random; in fact, they are only required to be good enough. No new algorithm is proposed. Instead, several existing algorithms are re-implemented to fit the system's runtime environment. The emphasis is thus on the correctness of those implementations.

With that insight, this chapter describes various aspects involved in the process of generating and validating random numbers. The first section explains the rational behind the selection of the random number generator (RNG)'s algorithm and implementation. Specifically, I will briefly introduce the two types of RNG. The second section outlines the methods used to verify the correctness of the RNG implementations, i.e. to ensure that the generated random numbers really conform to the initial given distribution. For this purpose, two techniques, the exploratory data analysis from NIST [25] and classical quantitative, are used.

5.1 Generation

Random number generation is a complex task. The process requires two steps:

1. Generate a uniform random number over the interval $(0, 1)$; and
2. Apply transformations to the above number to imitate an arbitrary distribution.

The difficult work lies in the first step. Although true random numbers can be generated from physical events such as atomic decay or thermal noise, these methods, however, have several deficiencies such as the high cost, and the inability to reproduce the sequence. The more practical approach is to use algorithms to produce pseudo random numbers.

In this thesis, the objectives are to generate reasonably random numbers in the shortest time possible, and to make the random number stream reproducible. The speed requirement is obvious, because any unnecessary delay, even in terms of microseconds, can cause a packet to fail to meet the required parameters. Reproducible random number streams mean that given the same initial seeds, the generator will produce exactly the same sequence of random numbers. Reproducibility is necessary to repeat a sequence of run, which is useful during the develop-debug-refactor-test cycle.

Since the impairment core is written as a kernel module, there is no access to the usual user-mode C functions `rand` and `srand`. However, one of the kernel module exports an alternative version, `net_random`. `Net_random` is a weak feedback random generator, based only on an initial entropy and the system jiffies (a unit of time in microseconds). It is not possible to reproduce a sequence of random numbers using this function. On the other hand, `net_random` is very fast; the function requires only one multiplication, one addition, one XOR operation, and finally one division to transform an integer to a uniform random number in the range $(0, 1)$. From here on, the term *net_random* will be used to refer to both the C function as well as the algorithm implemented by the function.

To support reproducible random number stream, an alternative version based on L'Ecuyer's algorithm [22] is implemented. The algorithm is a combined multiple recursive random (MRG) number generator with period length of about 2^{205}, i.e. it takes that many numbers until the random number sequence repeats.

An MRG of order k is based on a kth-order linear recurrence of the form:

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k} + b) \bmod m;$$
$$u_n = x_n / m$$

where m and k are positive integers, while b and a_i are integers in the range $\{0, \ldots, m-1\}$.

A **combined** MRG is then simply a combination of more than one MRGs. The detail of the algorithm is, however, out of the scope of this thesis.

This particular algorithm uses 31-bit fixed number operations. Fischer *et al.* [11] proposed a faster algorithm based on floating point operations with longer bit length (resulting in longer period length); however, with the soft emulation library, the new algorithm is actually slower than the fixed-number version.

The implementation based on L'Ecuyer's algorithm is better than the build-in *net_random* function, not only because of its reproducibility but also in terms of randomness. It is, however, more expensive in terms of computing. To gauge their performance, each function is timed to produce one million uniform random numbers. The result: *net_random* takes a total of 1.28-sec and the other 2.04-sec, more than 1.5 times slower. To generate a single random number, the associated time required are 1.28-μsec and 2.04-μsec, respectively.

Both functions are supported in this thesis, with *net_random* is the default algorithm. The user can select the desired algorithm from the graphical user interface. In the case of L'Ecuyer's algorithm, the user also needs to specify a set of six integers for the initial seeds. Whenever the random number generator is changed, the set of pooled uniform and normal random numbers are re-generated (for performance purposes, a million random numbers of each types are pooled and cycled through).

5.2 Validation

The verification step validates whether the RNG algorithms have been implemented correctly. We wish to show that the generated uniform numbers are adequate to uniformity and the normal numbers are adequate to normality. To that end, both the exploratory data analysis (EDA) [25] as well as the classical quantitative test techniques are used. As the name implied, EDA techniques employs graphical plotting heavily to explore the patterns of data, whereas classical quantitative goodness-of-fit test techniques use numerical analysis to test a hypothesis. These two methods are complementary with each other. For a hypothesis to hold, the results of both methods have to agree.

5.2.1 Exploratory Data Analysis

EDA techniques assume that the measured data behaves according to the following four properties:

1. random drawings;

2. from a fixed distribution;

3. with the distribution having fixed location; and

4. with the distribution having fixed variation.

Given those assumptions, EDA then provides the following four simple graphical techniques to validate the assumptions.

1. run sequence plot;

2. lag plot;

3. histogram; and

4. normal probability plot.

The `run sequence plot` draws the data in the sequence of occurrences. The X-axis contains the index of the data and the Y-axis the value of the data at the given index. The run sequence plot can easily pin-point any shift in the location and the variation of the data, as well as any outliers. The level of variation is indicated by the highest and lowest data point relative to the vertical axis. If the variation is fixed then the highest data points should form a relatively straight line, and the same thing for the lowest data points; i.e. we should not see irregular patterns such as a long series of low data points followed by another long series of high data points.

The `lag plot` demonstrates the evenness property expected of random data. A lag is a fixed displacement between the observed data. Given a set of data Y, a lag plot of x consists of coordinates (Y_{i-x}, Y_i) for i from x to the last index of the data. If the data are random, then the points on the lag plot will not form any recognizable shape. Lag plot is usually used with the lag value of 1.

The `histogram` groups data into a given number of buckets. The horizontal axis contains the buckets and the vertical axis the frequencies. The histogram diagram shows the center, the spread, the skewness as well as the outliers of the data.

Finally, the `normal probability plot` indicates whether or not the data is approximately normally distributed, in which case, the plot will resemble a straight line from the left to the right. The vertical axis of the plot contains the ordered response values, and the horizontal axis contains the normal order statistic percentiles.

5.2.2 Quantitative Techniques

The Kolmogorov-Smirnov [25, section 1.3.5.16] and the Jarque-Bera [18] tests are used to test for goodness-to-fit to the uniform and normal distributions, respectively. Both tests are in the general category of `hypothesis tests`, which's format includes the following components:

- *Null hypothesis* (H_0): the original statement to be tested
- H_1: the alternative hypothesis
- *Significant level* (α): the degree of certainty regarding the conclusion. For example, a value of $\alpha = 0.05$ means that the hypothesis is rejected wrongly 5% of the time
- *Critical region*: The region containing the values of the test statistics that directly lead to the rejection of the hypothesis.

In this case, the null hypothesis is that the data has the uniform or normal distribution. The detail of the two tests can be explored further in the associated references. I will just briefly describe them below.

With the Kolmogorov-Smirnov test, besides the data under test, the user has to supply the cumulative distribution function (CDF) for each data point. The Kolmogorov-Smirnov test is based on the maximum distance between the theoretical CDF curve and the empirical CDF curve. On the other hand, the Jarque-Bera test determines if the data follows a normal distribution based on the sample skewness and kurtosis. This test does not require the mean and standard deviation to be specified.

Matlab is used to perform these two tests. The built-in functions, `jbtest` and `kstest` [16], return 0 if the hypothesis cannot be rejected or 1 otherwise, at the significant level of 5%.

5.3 Uniform Distribution

Figure 5.1 shows the four plots for a sample of $1,000$ uniform random numbers having values in the range $(30, 100)$, generated by the *net_random* algorithm. The run sequence indicates that there is no shift in the location and variation of the data. The lag plot shows that the data are uniformly distributed within the rectangle. The histogram indicates that the frequencies are quite flat across the range of data. Finally, the normal probability

affirms that the data cannot be approximated by a normal distribution. Thus, all EDA techniques indicate that the data are in fact uniform.

Similarly, figure 5.2 displays $1,000$ numbers generated using L'Ecuyer's algorithm. The various plots also indicate that the set of data is truly uniform.

In both cases, the readers can notice that the histograms are a bit uneven at the bottom left of the screen. The reason is quite simple: we do not have enough data. Another test shall be required to confirm that the histograms really represent data from a uniform distribution. The chi-square test for standard deviation [25, §1.3.5.8] is used for that purpose. I make use of an existing matlab chi-square test implementation [33], and run it with the interval parameter of 70 for the first RNG and 50 for the second RNG at the significant level of 5%. Both results indicate that the hypothesis that the data come from a uniform distribution does hold.

Matlab's Kolmogorov-Smirnov test returns 0, which also means that the data really follows a uniform distribution.

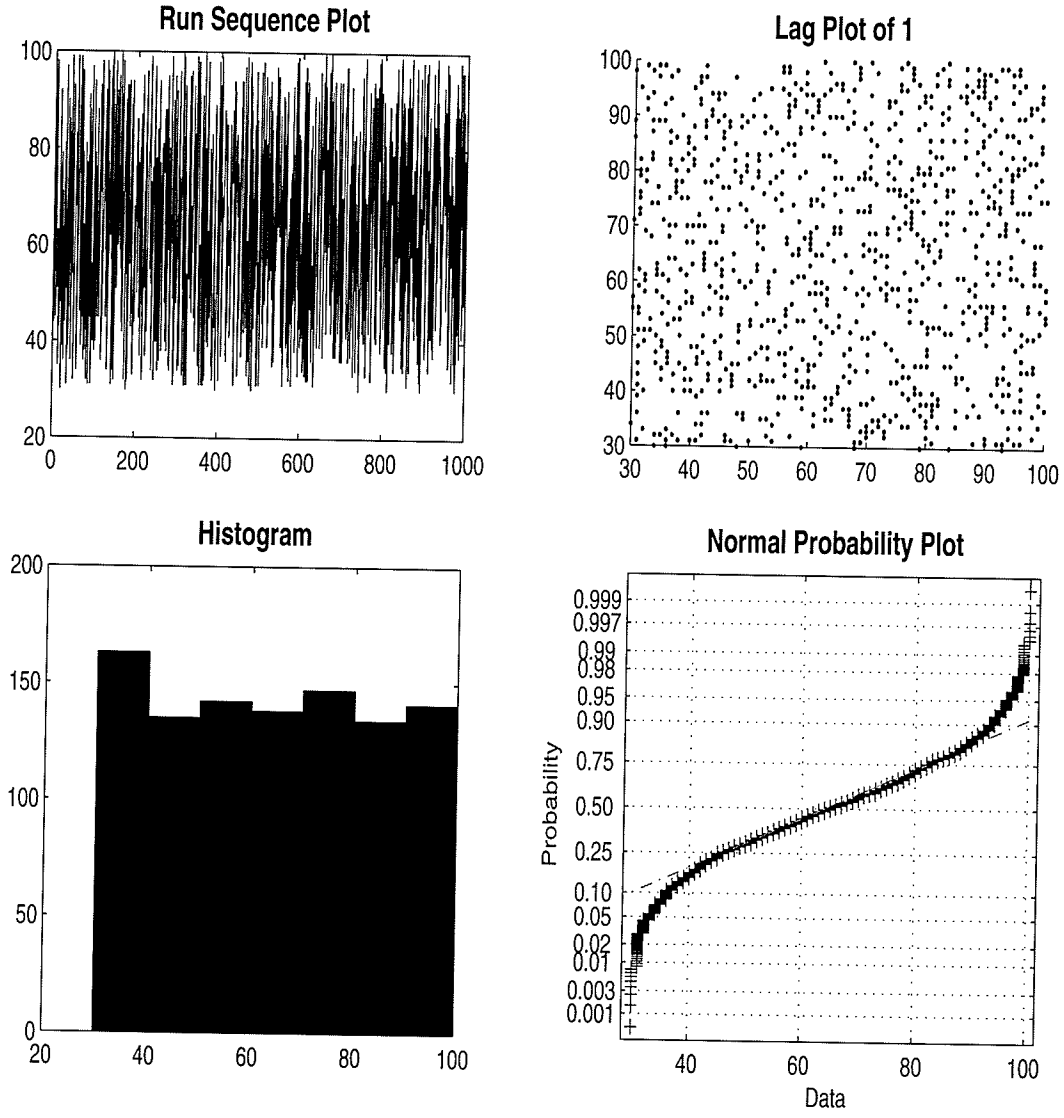

Figure 5.1: EDA plots over a sample of $1,000$ UNIFORM random numbers in the range $(30, 100)$ generated using *net_random* function.

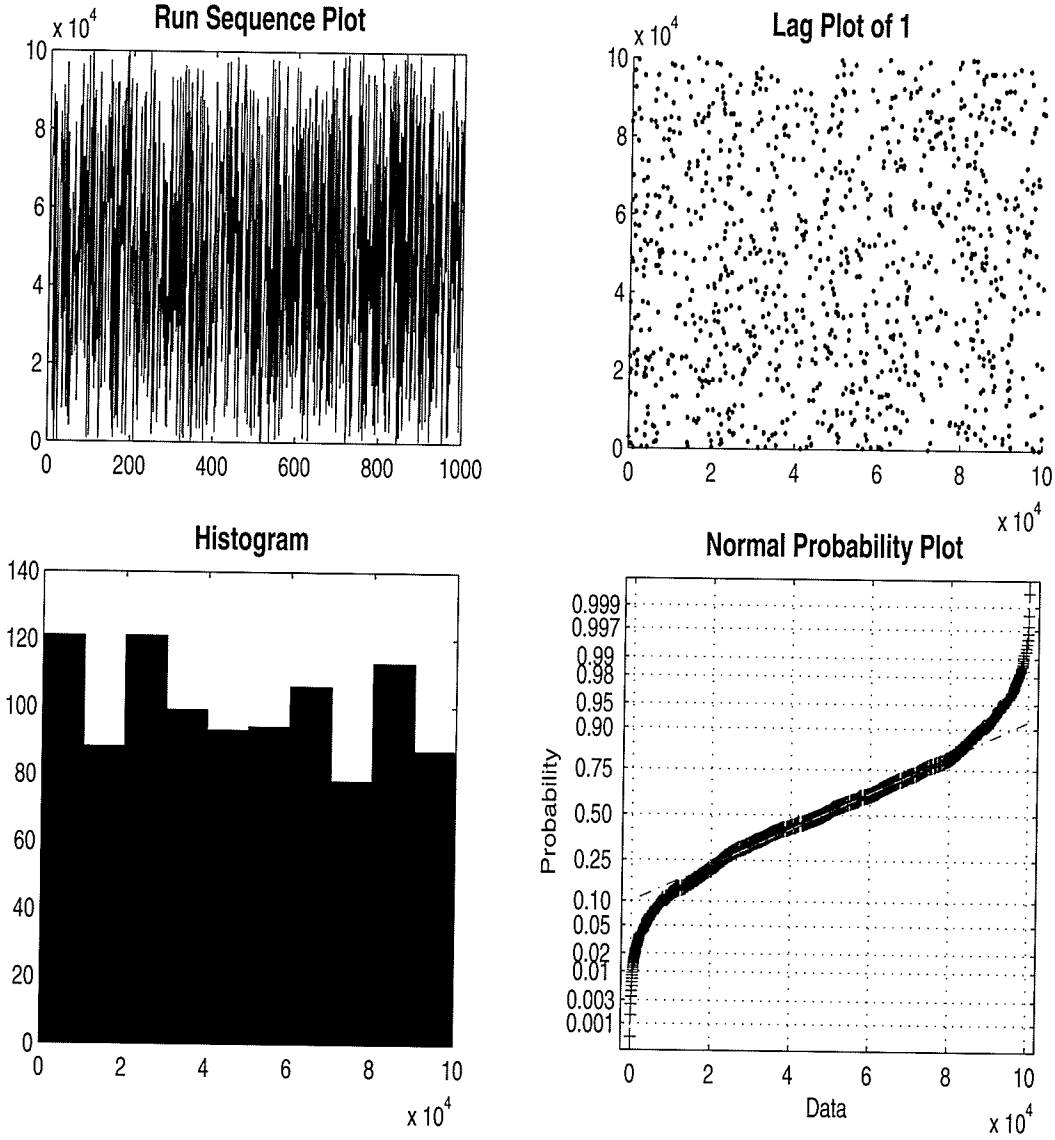

Figure 5.2: EDA plots over a sample of $1,000$ UNIFORM random numbers in the range $(0, 100, 000)$ generated using L'Ecuyer's algorithm.

5.4 Normal Distribution

Unlike other distributions where the `invert` [9, section 3.2] method is usually used to create a random number, there is no exact equation for computing normal random numbers in this method. Instead, Leva's [23] quadratic-bound fast normal random generator, based on ratio of uniform deviates method, is used in this thesis. This algorithm is one of the fastest normal random generators, consuming on average 2.74 uniform numbers to generate one normal number. When choosing the algorithms, the following factors are considered:

- the speed of the algorithm; and
- the level of complexity.

Since normal random variates are not correlated with each other, they can be mass pre-generated at start-up time. Subsequent normal random number requests will be retrieved from the pre-generated pool. Thus, the speed constraint listed above is not the most critical requirement. In fact, one has to be careful between the trade-off of complexity and speed. Several algorithms, such as Brent's [4] or Wallace's [36], are noticeably faster but are more complicated to implement. Particularly, kernel-mode execution environment also favors simple algorithms. Within the kernel-mode, the programmer does not have access to the `math` library; as a result, any `math` related functions such as `log` or `sqrt` have to be re-implemented.

The pseudo code of Leva's algorithm [23] is shown below:

1. Generate numbers u and v in the interval $(0, 1)$ using a uniform random number generator. Set $v = 1.7156(v - 0.5)$. This scales v to be uniform in $(-r, r)$, where $r = \sqrt{2/e} = 0.8578$.
2. Set $x = u - s$, $y = |v| - t$, $Q = x^2 + y(ay - bx)$. The point $(s, t) = (0.449871, -0.386595)$ is the center of the quadratic form. The remaining parameters are $a = 0.19600$ and $b = 0.25472$.
3. If $Q < 0.27597$ go to step (6).
4. If $Q > 0.27846$ go to step (1).
5. If $v^2 > -4u^2 ln(u)$ go to step (1).
6. Return v/u as the pseudorandom number.

Figure 5.3 displays four plots used by the EDA method to validate the correctness of the algorithm. In the run sequence, there are a few data points reaching above and below the average vertical plane, but over all the data location and variation are relatively stable. The lag plot does not resemble any particular shape; the data are correctly distributed around the mean. The histogram is of a bell shape but rather distorted in the middle. Again, the reason is because of the small number of intervals as well as the small amount of data. However, unlike the previous case of uniform data, the chi-square test for the standard deviation does not conclude as the range of the data is too small. Finally the normal probability plot is a relatively straight line (the line is not continuous since the data are discretized). Hence, the data sample is of a normal distribution.

It should be noted that here we are actually testing the algorithm to transform a uniform random number into a normal random number. The tests assume that the uniform random number generator functions correctly. As a result, only one set of data is required, instead of each for the *net_random* and L'Ecuyer's algorithms.

On the quantitative technique side, with the same set of data, the Jarque-Bera tests returns $H = 0$, which indicates that the hypothesis that the sample has a normal distribution cannot be rejected at significant level of 5%.

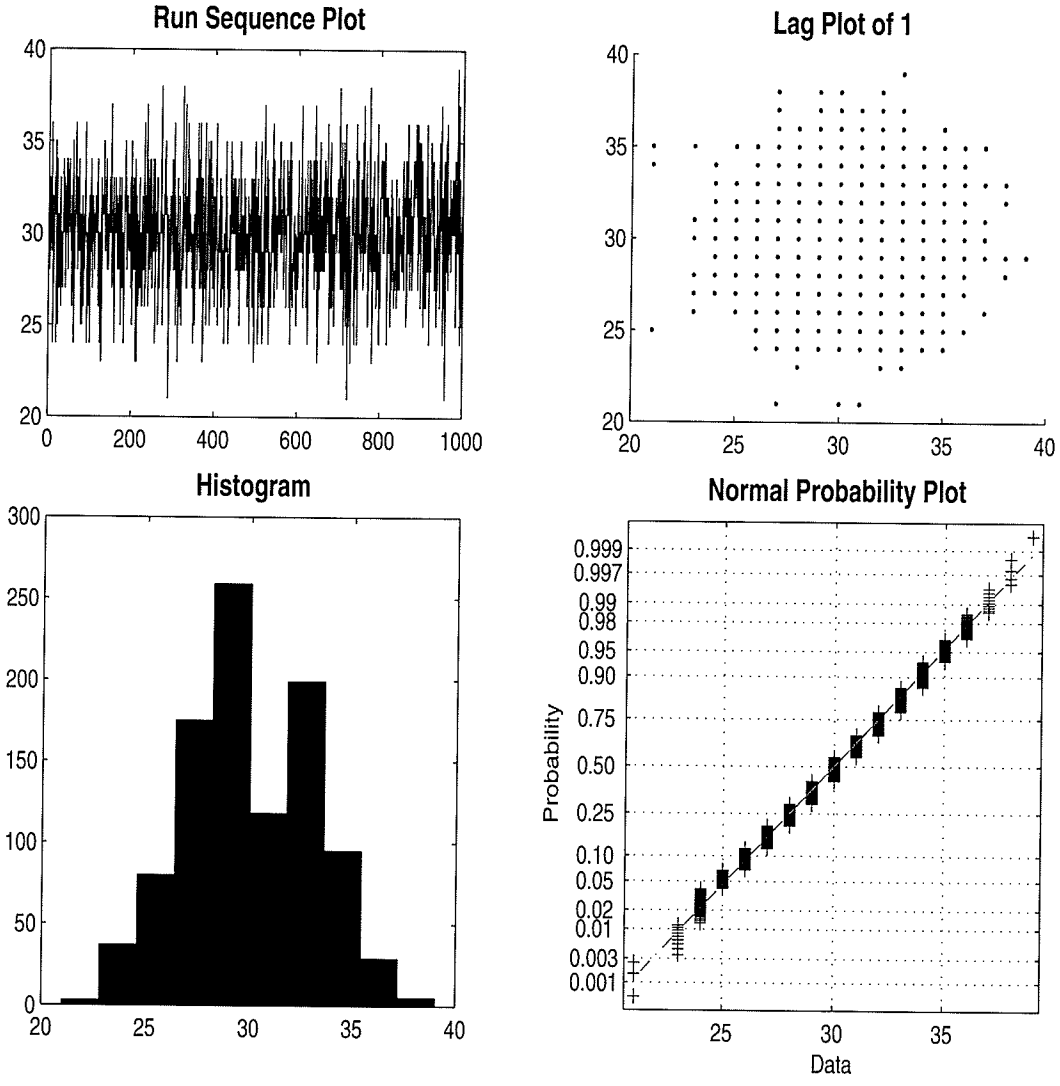

Figure 5.3: EDA plots over a sample of 1,000 NORMAL random numbers of mean 30 and standard deviation 3.

Chapter 6

Multi-state Discrete Markov Chain

The objective of this chapter is to provide some preliminary evaluation on the correctness of the Markov chain implementation. For detail description of the Markov chain, please refer back to section 3.1. I will introduce two test cases, one for the markov chain with the durations in the units of number of packets arrived, and the other one for durations in the units of time elapsed. For each test case, I will first use analysis to arrive at the theoretical latency values, and then use actual runs to compute the empirical values. The Markov chain implementation is correct if the theoretical numerical values are closely matched with the empirical values.

To analytically calculate the theoretical latency, one has to calculate the state probabilities at `stationary`. The state probabilities are at stationary when the transition matrix does not have any effect on the current state probabilities.

Let i be the number of states, $\vec{v} = \begin{pmatrix} v_1 & v_2 & \dots & v_i \end{pmatrix}$ be the state probabilities at a particular time, and P be the transition matrix.

$$P = \begin{pmatrix} p_{11} & p_{12} & \dots & p_{1i} \\ \dots & \dots & \dots & \dots \\ p_{i1} & p_{i2} & \dots & p_{ii} \end{pmatrix}$$

Then the discrete Markov chain is in stationary if:

$$\begin{cases} \vec{v} = \vec{v} \times P \\ \sum_{k=1}^{i} v_k = 1 \end{cases} \tag{6.1}$$

Solving the above set of linear equations will give us the stationary state probabilities, and from there, the theoretical latency.

6.1 Test 1: Packet-based Discrete Markov Chain for Packet Delay

	State 1	State 2
Duration	200 packets	100 packets
Var. Delay	Uniform(20, 20)	Uniform(50, 50)
Trans. Prob.	(0.4, 0.6)	(0.8, 0.2)

Table 6.1: Packet delay parameters for PORT1.

The packet delay parameters are displayed in table 6.1. This scenario is a bit more complicated because the duration of the first state is twice as long as that of the second state, which alters the stationary state probabilities. We can convert the parameters into standard form (i.e. all states have the same duration) by introducing another state having duration of 100 packets. The state transition matrix is then:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0.4 & 0.6 \\ 0.8 & 0 & 0.2 \end{pmatrix}$$

Using the set of equations (6.1), the stationary state probabilities are calculated to be $\vec{v} = (0.25, 0.43, 0.32)$. Since the first two states have the same delay impairment parameters, the theoretical packet delay is then $((0.25 + 0.43) \times 20\mu sec) + (0.32 \times 50\mu sec) = \boxed{29.6\mu sec}$.

The PING command is used to verify the implementation of the Markov chain. The approach is to first ping flush the network when the system is in NO impairment mode to collect the base round time trip (RTT). Then the system is ping flushed again with the packets being impaired; the output RTT is deducted from the base RTT to get the emulated delay. Finally, the inferred emulated delay is compared against the theoretical calculations.

```
BASE RTT:
---------
[root@hpham hpham]# ping -f -c 100000 10.3.1.5
PING 10.3.1.5 (10.3.1.5) 56(84) bytes of data.

--- 10.3.1.5 ping statistics ---
100000 packets transmitted, 100000 received, 0% packet loss, time 14136ms
rtt min/avg/max/mdev = 0.099/0.104/1.558/0.011 ms, ipg/ewma 0.141/0.104 ms

IMPAIRED RTT:
-------------
[root@hpham hpham]# ping -f -c 100000 10.3.1.5
PING 10.3.1.5 (10.3.1.5) 56(84) bytes of data.

--- 10.3.1.5 ping statistics ---
100000 packets transmitted, 100000 received, 0% packet loss, time 17529ms
rtt min/avg/max/mdev = 0.117/0.134/1.605/0.016 ms, ipg/ewma 0.175/0.126 ms
```

The base RTT is about 104-μsec and the impaired RTT is around 134-μsec; thus the average delay is inferred to be $134-104 = \boxed{30-\mu sec}$, which is very close to the theoretical one, 29.6-μsec.

6.2 Test 2: Time-based Discrete Markov Chain for Packet Loss

	State 1	State 2
Duration	5 seconds	5 seconds
Duplicate Probability	0.1%	100%
Trans. Prob.	$(0.8, 0.2)$	$(0.9, 0.1)$

Table 6.2: Packet duplication parameters for PORT1.

We now evaluate the implementation of the time-based discrete Markov chain for the duplication impairment mode. The parameters are show in Table 6.2. Using equation (6.1), the stationary state probabilities are calculated as $\vec{v} = \begin{pmatrix} 0.82 & 0.18 \end{pmatrix}$. The overall duplication probability is then $(0.82 \times 0.001) + (0.18 \times 1) = 0.18$.

Below is the output after ping flush the network for five minutes.

```
[root@hpham hpham]# ping -f -w 300 10.3.1.5
PING 10.3.1.5 (10.3.1.5) 56(84) bytes of data.

--- 10.3.1.5 ping statistics ---
1938179 packets transmitted, 1938179 received, +350642 duplicates,
0% packet loss, time 299991ms
rtt min/avg/max/mdev = 0.100/0.113/5.664/0.016 ms, pipe 2,
ipg/ewma 0.154/0.120 ms
```

The actual percentage of duplicated packets is $350,642/1,938,179 = \boxed{18\%}$ which agrees with the theoretical calculation.

Chapter 7

Performance Analysis

The goal of the performance analysis phase is to show that the system, consisting of the entire IXDP425 BOARD as a black box, performs correctly and with adequate response/throughput. The focus will be on the manipulation of incoming network traffic whereby the system is expected to be able to impair packets at wire speed. In particular, the analysis is based on the following two metrics: the packet passthrough rate and the packet impairment processing overhead.

I will evaluate the system under various impairment scenarios. In addition to the parameters specific to each impairment mode, each scenario is examined under the same set of workload parameters, which are the packet size and the packet rate. The three packet sizes chosen are: $64B$, $759B$, and $1,518B$. This selection covers the minimum and maximum packet sizes allowed by the Ethernet standard. Then for each of those sizes, three packet rates are selected: at 10%, 50%, and 100% of the maximum bandwidth. Note that the maximum wire speed is 100Mbit/s and that the packet rate is directly influenced by the packet size. Hence, for each test scenario, there are a total of nine runs.

The IXIA400 [8] traffic generator is used to inject packets into the system. It is capable of generating traffic at the maximum wire rate (for 100-Mbps, at packet size of 64-KB, the transmission rate is $148,809$ pkts/s). Due to various system overheads, a typical PC cannot generate this kind of traffic.

The packet processing overhead is calculated using two different methods: by measurements internally within the IXDP425 BOARD, and by measurements externally on the IXIA400 traffic generator. The measurements start at small packet size and low transmission rate and then at gradually increased packet size and transmission rate.

For internal measurement, the packet's arrival time is stamped when the kernel mod-

ule receives a notification interrupt from the hardware queues, i.e. the packet has been queued in the hardware queues. The packet's release time is stamped just before it is transferred from the kernel module into the hardware queues for transmission. Hence, this method measures the time spent within the impairment core only; waiting times in the hardware queues are not considered. Semantically, to act as a bridge, the driver has to transmit the received packet from one port to the other port for re-transmission. In the baseline case, where no impairment is imposed, the total time to perform such operation is considered to be zero. When factoring in the impairments, the packet processing overhead can be calculated by subtracting from the packet release time the sum of the arrival time and the emulated delay (if any). The result is the actual packet processing overhead with respect to the baseline case. For each test, a sample of $10,000$ continuous packets are collected after the initial 5-second interval. The min, max, mean, `trimmed mean` and standard deviation of the processing overhead are calculated from the collected sample. The trimmed mean of the sample is computed by first discarding 5% of the lowest and highest values and finally calculating the mean from the middle 80% of the data. Of the two means, the trimmed mean is more resistant to extreme values. Hence, it will be used whenever there is a comparison between two sets of data.

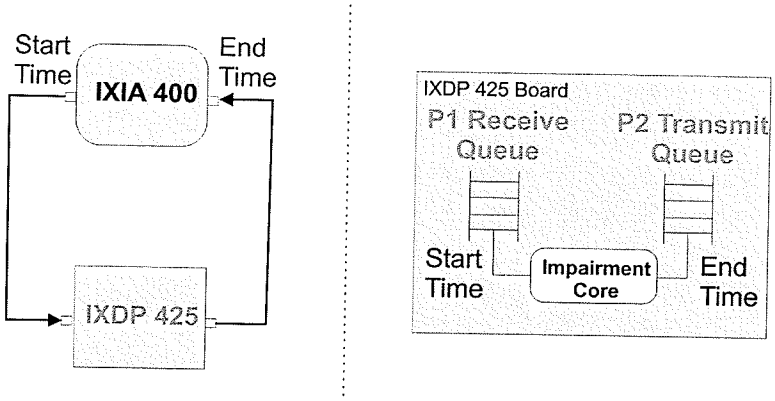

Figure 7.1: External latency measurement *(left)* and internal overhead measurement *(right)*.

External measurement of the packet processing overhead relies on the latency measurement capacity of the IXIA400 traffic generator. Packets are transmitted one-way from the input port to the output port. The average latency is measured over a sample of at least $50,000$ packets, and is carried out in the `store and forward` mode, i.e.

from the time the last bit of the input frame reaches the input port to the time the first bit of the frame reaches the output port. In the baseline case, no impairment is imposed, and the overhead is considered to be zero. The overheads for other impairment settings are calculated by subtracting the associated mean latencies by the baseline mean latency. Thus, unlike the internal measurement method, not only the IXDP425 BOARD overhead but also the transmitter and the receiver's overheads are considered in the measurement. Nevertheless, the packet processing overhead calculated by either method should be proportional to one another.

For all internal overhead measurements, the packet is created of frame type Ethernet-II and of protocol TCP/IP. However, in order to measure latency on the IXIA400 traffic generator, the frame's data link layer protocol must be set to 'None', i.e. no protocol specific handling is performed. It is observed that for this setting, the internal processing overhead is almost double that of Ethernet-II frames (an average of 6.09 μsec vs. 3.73 μsec at packet rate of $15,000$ pkts/sec, packet size of 64B and no impairment imposed). The possible cause is that the IXDP425 BOARD does not recognize the frame header and has to do some extra look-ups, or perhaps for Ethernet-II frame, it can perform some optimizations. Even though more consistent results can be obtained by specifying the same frame header for both the internal and external overhead measurement, Ethernet-II is still chosen for internal overhead measurements as it is the most common frame type on the Internet.

Table 7.1 displays the time required to execute the code block of each `active` impairment mode. An impairment mode is active if its impairment model is set to anything but the null mode. The values are measured internally within the IXDP425 BOARD with null model parameters:

- Uniform delay with start range of 0 μsec and end range of 0 μsec;
- Normal delay with mean of 0 μsec and standard deviation of 0 μsec;
- Loss and duplication impairments with occurrence probability of 0%; and
- Reordering with occurrence probability of 100% and displacement distance of 0.

These code blocks are executed by the central processor XScale at clock 533 MHz. Although both the standard uniform and normal variates are pre-generated, the required

	Delay		Loss	Reorder	Duplication
	Uniform	Normal			
Time	2.61	1.02	1.68	3.59	1.68

Table 7.1: Amount of time (μsec) required to execute each impairment mode.

time for the uniform delay is a larger because its transformation algorithm has one additional float operation than that of the normal delay. For the loss, reordering and duplication path, a new standard uniform random number is generated (not pooled). The reordering path also invokes the uniform/normal random number generator function to retrieve a value for the displacement distance. By adding up the numbers together, it would seem that the impairment core requires too much execution time. This is true when no delay impairment is set. However, as the reader will see, even a 10 μsec delay will hide away most of the execution time.

The sections below further investigate the packet overhead under different impairment settings. I will initially consider the overhead for one-way traffic, and then move on to bi-directional traffic at the end of this chapter. It is expected that, the overhead will be a bit higher for round-trip traffic as the XSCALE processor has to handle twice the work load.

7.1 No Impairment

The purpose of this test scenario is to first perform a check to ensure that the system does route packets from one port to the other. Secondly, the test measures the latency imposed on the passing-through packets. The system is expected to be able to accept the packet stream at the maximum transmission rate without dropping packets. Also, the latency should be consistently proportional to the packet rate.

Table 7.2 summarizes the packet processing overhead (in unit of microseconds) when there is no impairment, i.e. traffic is in the `pass-through` mode. Note that in column seven, C.I. stands for the confidence interval. As expected, the data indicate that the packet processing overhead is proportional to the transmitted packet rates. The sequence diagrams in figure 7.2 show that the overhead variation is small most of the time. This overhead is caused by the internal packet queuing in the impairment module. The higher the transmitted rates, the larger the number of items in the queues, and thus the higher

the overhead. Certainly, if a `fast-path` is implemented to skip the software queues, then the overhead will be somewhat reduced.

Pkt Size (Byte)	Rate pkt/s	BW %	Ext. Mean **Latency**	**Int. Processing Overhead**				
				Mean	Std	95% C.I. of μ	Min	Max
64	14,881	10%	14.2	4.59	0.86	4.58 - 4.61	3.00	12.00
64	74,405	50%	19.0	13.65	4.74	13.55 - 13.74	8.00	45.00
64	148,809	100%	133.6	168.51	30.00	167.92 - 169.10	75.00	271.00
759	1,605	10%	17.1	4.69	0.85	4.67 - 4.71	3.00	15.00
759	8,023	50%	17.2	4.60	0.82	4.58 - 4.62	3.00	10.00
759	16,046	100%	19.1	4.59	0.84	4.58 - 4.61	3.00	11.00
1,518	813	10%	17.1	4.62	0.86	4.60 - 4.64	3.00	9.00
1,518	4,064	50%	16.9	4.59	0.80	4.58 - 4.61	3.00	11.00
1,518	8,127	100%	19.1	4.59	0.82	4.57 - 4.60	3.00	24.00

Table 7.2: TCP/IP packet processing overhead (in μsec) when no impairment imposed. Estimates from $n = 10,000$ packets for internal overhead; $n = 50,000$ for external latency.

7.2 With Delay

Let d be the targeted emulation delay of a packet, and let θ be the allowable processing overhead margin, such that $d \geq 0$ and $\theta \geq 0$. Then the targeted emulation delay objective is said to be achievable if and only if the amount of time the packet is held in the impairment system is within the range $(d, d + \theta)$. As an example, let $d = 10\mu sec$ and $\theta = 0.5\mu sec$. The emulation parameters are achievable if the holding time of all packets (within the impairment system) is in the range $(10\mu sec, 10.5\mu sec)$.

I will first compare my system's result from a particular test run with NIST.net; and then perform additional tests with constant delay parameter and non-constant delay parameter.

7.2.1 Comparison with NIST.net

One of the network impairment tools I have mentioned earlier in section 1.2 is NIST.net [6]. In that paper, the authors publish the packet processing overhead values from a test run of 1.1Mbit/sec packet stream of 64-byte packets ($1,645$ pkt/s). I use the same workload parameters to perform the test on my system. The result is displayed in table 7.3.

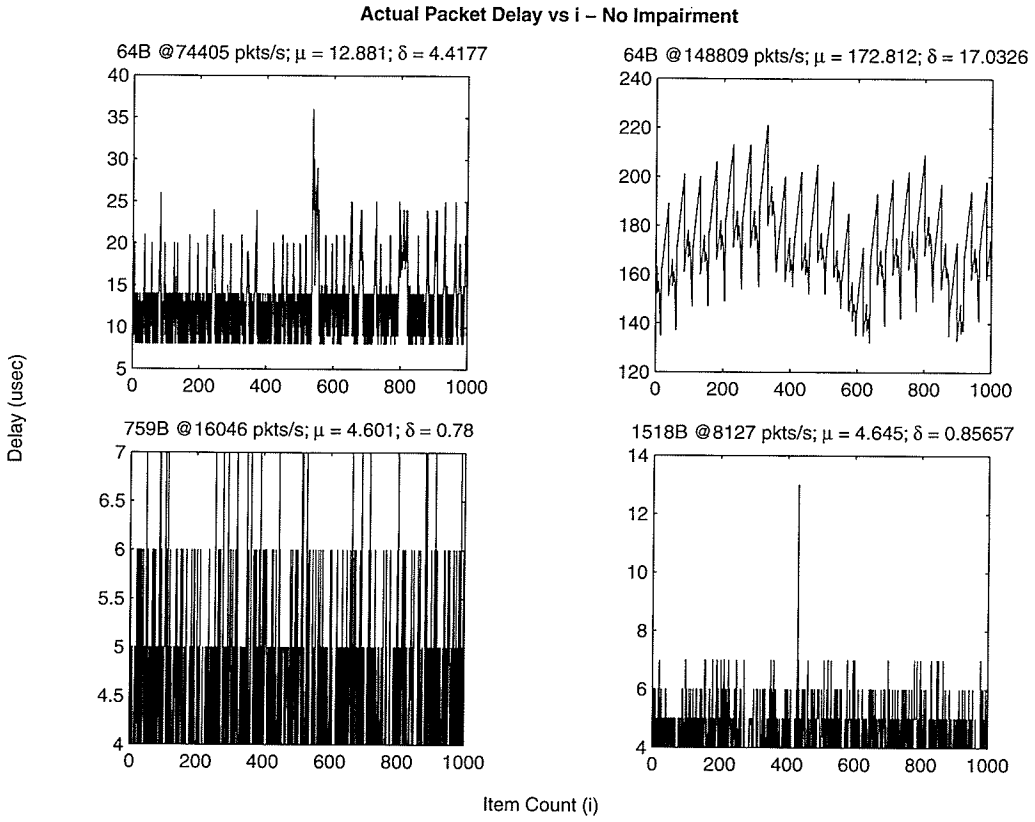

Figure 7.2: Actual packet delay time when no impairment is imposed; samples of $1,000$ continuous packets.

The packet processing overhead at each delay setting is calculated by subtracting the associated mean latency by the sum of the imposed delay (first column) and the baseline mean delay (first row). Note that the baseline case is when the system routes traffic without imposing any delay ($0 - msec$).

Recall that external latencies are measured at the IXIA400 traffic generator. For the delay setting of 100 and $1,000$ msec, the packet rate is reduced to $1,200$ and 100 pkt/s respectively. This is because at that rate, all the available buffers are queued in the software queues. To receive a packet, the hardware queue must get hold of a free buffer. If it cannot do that, the packet is dropped; in this case the buffers are replenished back to the hardware queue at a lower rate than the packet arrival rate. It shall be noted that the processing packet rate can be increased somewhat by extending the number of buffers; however, due to limited resources, eventually the system will reach a point where it cannot keep up with the incoming packet streams. This is natural if one looks at the system

Delay *msec*	Mean Latency *μsec*	Overhead *μsec*	NIST.net Mean Latency *msec*	NIST.net Overhead *μsec*
0	14.3	0.0	17.9	0.0
1	1,014.0	0.0	1,064.4	46.5
10	10,014.0	0.0	10,097.8	79.9
*100	100,020.0	5.7	100,063.4	45.5
*1,000	1,000,036.0	21.7	1,000,081.5	63.6

Table 7.3: Comparison of external latencies for various constant delay settings at $1,645$ pkt/s and packet size 64B vs NIST.net's (* denotes that some packets are dropped.)

while taking into account the number of packets currently on the wire. As the delay time increases but the transmitting rate stays the same, the number of packets on the wire will keep increasing and the sum of the packet sizes gradually approaches the wire's maximum bandwidth.

The reader can notice in table 7.3 that the latencies under the delay settings of 1-msec and 10-msec are slightly less than that of the baseline case (the first row of table 7.3). The explanation lies in a common code block that any packet going through the impairment core has to traverse through (no matter whether the system is in the pass-through mode or whether there are some impairments set). Let t denote the amount of time to execute that common code block. Now, in the baseline case, when there is no impairment imposed on the packet stream, t is then part of the packet latency. However, if there is a delay set, the arrival time and release time of a packet are stamped before and after the execution of the common code block, respectively. Thus not all of t makes up the packet latency since part of t (or all of it in the case the delay is greater than t) is within the delay. This diminishing overhead symptom is more clear under the internal measurement method where the overhead is measured from the arrived time to the release time of the packet.

7.2.2 Constant Delay

The delay impairment parameter is constant if its value never changes during the time the test is carried out. The opposite to this is the non-constant delay modeled using the packet-based or time-based Markov chain, which will result in different delay parameters on the same packet stream. For this test, two delay values are chosen: 10-μsec and 100-msec. The former is used to gauge the system's performance near the extreme end, while the later is within a latency range usually observed in many connection mediums. In both

cases, the packet processing overhead is expected to be rather uniform.

Figure 7.3 displays the internal and external packet processing overhead for 64, 759, and 1,518-Byte packet streams when they are impaired with a 10-μsec delay; data is shown in table 7.4. For each packet size, the tests are ran at 10%, 50% and 100% of the associated maximum packet rate. The achieved packet rate is plotted against the y axis on the right side. Except for the case of 64-Byte packets at 100% bandwidth (148,809 pkt/s), the overhead is very low. Figure 7.4 shows the sequence plot of the targeted delay and the actual delay for one thousand 759-Byte packets at packet rate of 16,046 pkt/s. The standard deviation is low, 0.55 μsec. Also, no packet is released pre-maturely. At 10-μsec delay, the impairment system can sustain transmission rate at up to 125,000 pkt/s (tested by going from maximum packet rate and then reducing it until there is no packet dropped).

Both figures thus indicate that very small packet delay is achievable at packet rate of less than 16,046 pkt/s.

Pkt Size (Byte)	Rate pkt/s	BW %	Ext. Mean **Overhead**	**Int. Processing Overhead** (μsec)				
				μ	Std	95% C.I. of μ	Min	Max
64	14,881	10%	2.1	1.16	0.68	1.15 - 1.17	0.00	7.00
64	74,405	50%	19,9	25.90	12.29	25.65 - 26.14	7.00	90.00
64	148,809	100%	831.9	611.89	106.93	609.78 - 613.99	399.00	769.00
759	1,605	10%	1.2	1.16	0.67	1.15 - 1.17	0.00	5.00
759	8,023	50%	1.3	4.60	0.82	4.58 - 4.62	3.00	10.00
759	16,046	100%	1.6	1.21	0.79	1.19 - 1.22	0.00	11.00
1,518	813	10%	1.0	1.17	0.69	1.15 - 1.18	0.00	6.00
1,518	4,064	50%	1.7	1.16	0.64	1.14 - 1.17	0.00	6.00
1,518	8,127	100%	0.4	1.19	0.77	1.18 - 1.21	0.00	10.00

Table 7.4: TCP/IP packet processing overhead (in μsec) under 10-μsec constant delay.

When testing under the 100-msec emulated delay, the processing overhead values are almost similar in all cases. The reason is because, under such delay model, the impairment box can only accept incoming traffic at the rate of about 1,300 packet per seconds. The cause of this problem is explained in detail in § 7.2. In the tests, the IXIA400 traffic generator keeps bombarding the system at the specified transmission rate. For any real world system, the transport layer protocol (such as TCP) would have reduced the transmission rate when packet loss is detected.

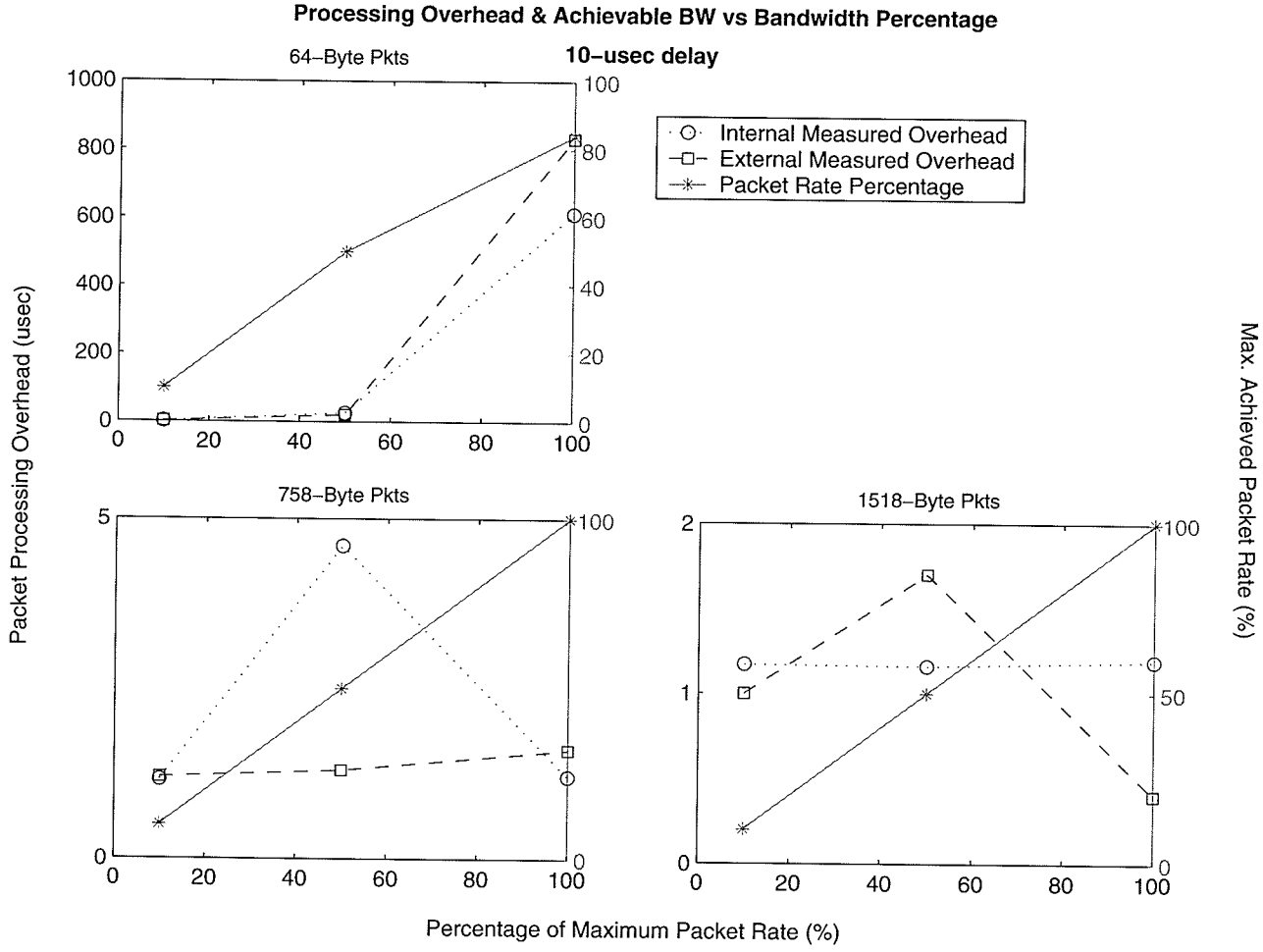

Figure 7.3: Internal and external measured packet processing overhead and achieved bandwidth for various packet size when running at 10 μsec delay.

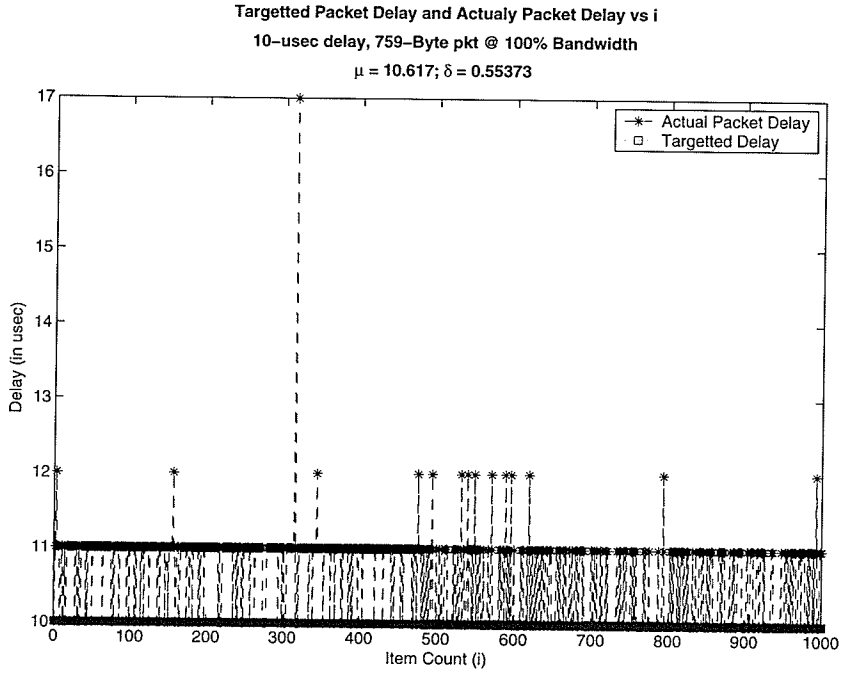

Figure 7.4: Sequence plot of one thousand continuous 759-Byte packets at 16,046 pkt/s and 10 μsec delay.

Pkt Size (Byte)	Rate pkt/s	BW %	Ext. Mean **Overhead**	**Int. Processing Overhead** (μsec)				
				μ	Std	95% C.I. of μ	Min	Max
64	14,881	10%	9.29	1.40	1.47	1.37 - 1.43	0.00	22.00
64	74,405	50%	10.30	2.73	2.88	2.67 - 2.78	0.00	20.00
64	148,809	100%	10.60	3.13	3.26	3.07 - 3.20	0.00	40.00
759	1,605	10%	2.78	1.09	0.93	1.07 - 1.10	0.00	4.00
759	8,023	50%	2.96	1.08	0.63	1.07 - 1.10	0.00	8.00
759	16,046	100%	4.28	1.09	0.65	1.08 - 1.10	0.00	9.00
1,518	813	10%	3.12	1.08	0.66	1.07 - 1.09	0.00	6.00
1,518	4,064	50%	2.60	1.10	1.13	1.08 - 1.12	0.00	39.00
1,518	8,127	100%	1.67	1.08	0.62	1.07 - 1.09	0.00	6.00

Table 7.5: TCP/IP packet processing overhead (in μsec) under 100-**msec** constant delay.

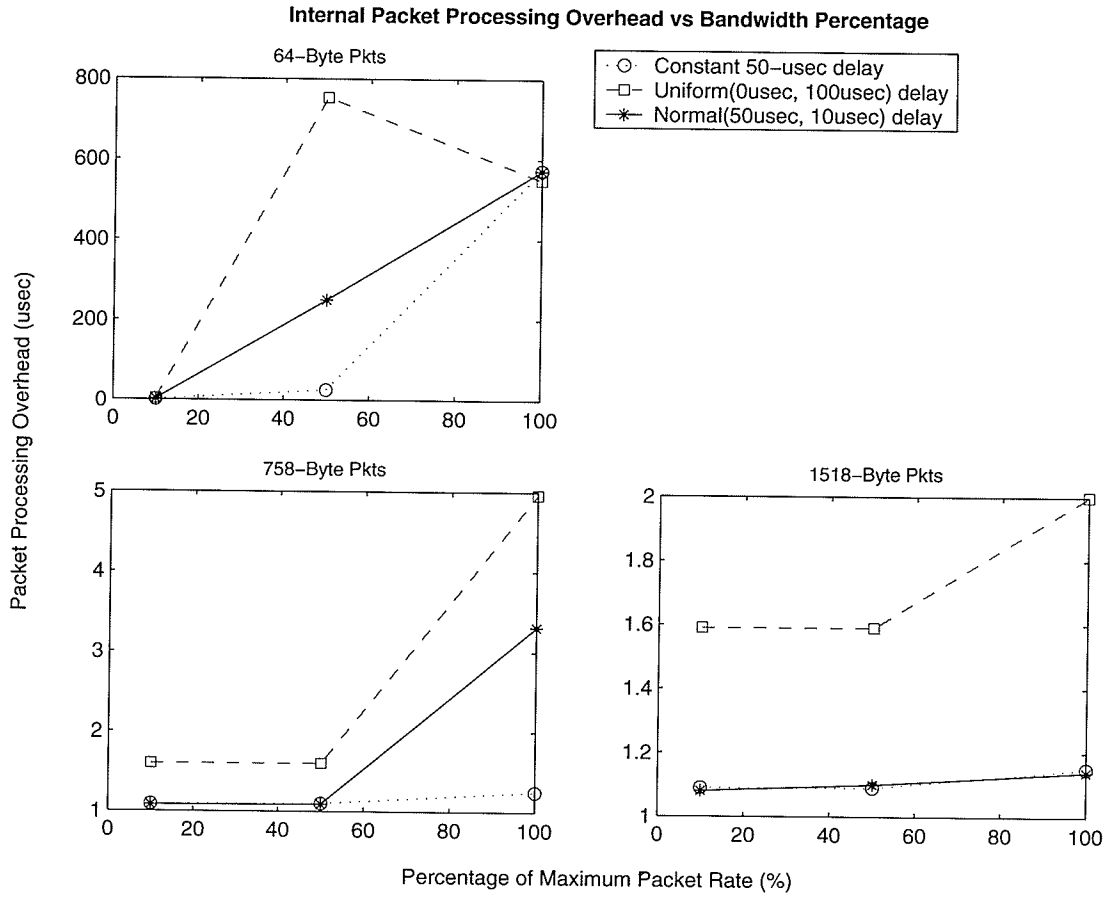

Figure 7.5: Mean packet processing overheads for constant 50-μsec, UNIFORM(0-μsec, 100-μsec), and NORMAL(50-μsec, 10-μsec) delay at various packet sizes.

7.2.3 Non-constant Variable-delay

In this section, I measure the side effect of non-constant delay. For comparison purpose, the mean packet overheads of the UNIFORM packet delay of range (0 μsec, 100 μsec) and of the NORMAL delay of mean 50 μsec and standard deviation 10 μsec are plotted along with the mean processing overhead of constant packet delay of 50 μsec in figure 7.5 (also refer to table 7.6). Note that for both the uniform and normal delays, the mean targeted delay is 50 μsec. Each mean is computed from a sample of $10,000$ continuous packets using internal measure. Figure 7.5 shows that in most test cases, the mean overhead is higher than the values computed in the constant 50-μsec delay model. When compared against the constant 50-μsec delay model, the means in the NORMAL model are less variable than those of the UNIFORM model. This is due to the nature of the NORMAL distrib-

P Size (Byte)	BW %	Constant 50 μsec			Uniform $0-100$ μsec			Normal $50-10$ μsec		
		μ	Std	C.I. of μ	μ	Std	C.I. of μ	μ	Std	C.I. of μ
64	10	1.07	0.69	1.05 - 1.08	3.80	5.60	3.69 - 3.91	2.05	2.98	1.99 - 2.11
64	50	25.26	15.83	24.9 - 25.6	753.58	474.41	744.2 - 762.9	249.01	377.17	241.6 - 256.4
64	100	572.46	106.70	570.4 - 574.6	548.67	114.96	546.4 - 550.9	572.33	105.93	570.2 - 574.4
759	10	1.09	0.65	1.08 - 1.11	1.60	1.90	1.56 - 1.64	1.08	0.66	1.07 - 1.10
759	50	1.10	0.61	1.09 - 1.11	1.60	1.88	1.56 - 1.64	1.09	0.67	1.07 - 1.10
759	100	1.25	1.38	1.22 - 1.27	4.96	6.95	4.83 - 5.10	3.31	4.21	3.23 - 3.39
1,518	10	1.09	0.60	1.08 - 1.10	1.59	1.88	1.55 - 1.63	1.08	0.63	1.07 - 1.09
1,518	50	1.09	0.66	1.08 - 1.10	1.59	1.87	1.55 - 1.63	1.10	1.22	1.07 - 1.12
1,518	100	1.15	0.80	1.13 - 1.16	2.00	2.40	1.96 - 2.05	1.14	0.84	1.12 - 1.15

Table 7.6: Internal packet processing overhead (in μsec) for various delay settings.

ution whereby the majority of data is clustered close to the mean value.

The use of non-constant packet delay creates an unavoidable amount of packet processing overhead. To see this problem clearly, consider a constant packet delay model. Packets arrive in chronological order; because of the constant delay, their release times are in increasing order with the first packet in the queue having the smallest release time. Thus, assuming that the first packet in the queue is popped out on time, then subsequent packets are also removed from the queue at the correct time slot. This is not the case for non-constant delay however. Even though the second packet in the queue arrived after the first one, its generated delay value could be less than that of the first packet. Therefore, the queue's FIFO (first in first out) policy forces the second packet to be transmitted later than it should be. The direct consequence is that packets are released in burst; their inter-departure time has one large value followed by a number of small values. This pattern is well presented in figure 7.6.

This undesired delay overhead only exists when the average queue size is greater than one, which implies that the transmitted packet rate must be very high and/or the targeted delay value is big. For example, in figure 7.7, under the same set of data we used earlier, we can see that the empirical CDF almost overlaps with the theoretical CDF. However, when the packet rate doubles (64-B packet at $14,881$ pkts/s in figure 7.8), there is a noticeable bump between the empirical and theoretical CDF.

Figure 7.9 displays the overhead gap between the targeted delay and the actual delay for 64-Byte packets at transmission rate $74,405$ pkts/sec. Each vertical line connects the targeted delay to the actual delay of a particular packet. It is easy to observe the relationship between the overhead of the current packet and the overhead and targeted delay time of the previous packet. Let's consider a sample interpretation from the left plot of figure 7.9. The second vertical bar has a targeted delay of about 95-μsec. The next packet

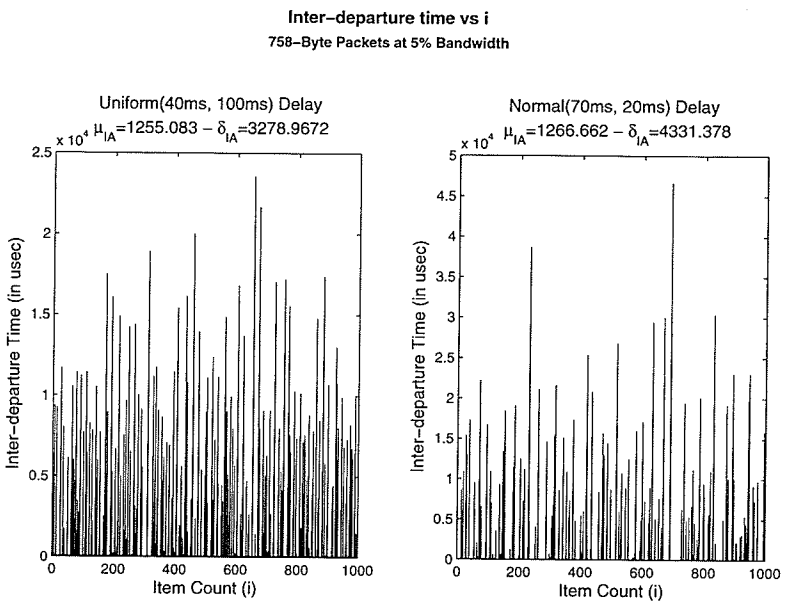

Figure 7.6: Inter-departure time for 759-byte packets under UNIFORM(40-msec, 100-msec) delay *(left)* and NORMAL(70-msec, 20-msec) delay *(right)*.

has a targeted delay of 25-μsec, much lower than the previous one. As a result, you can see that the third vertical line is very long, implying that the delay for that packet is much higher than desired. This strong correlation between the packets significantly distorts the initial theoretical model. In this case, the large delay difference between two packets is the cause.

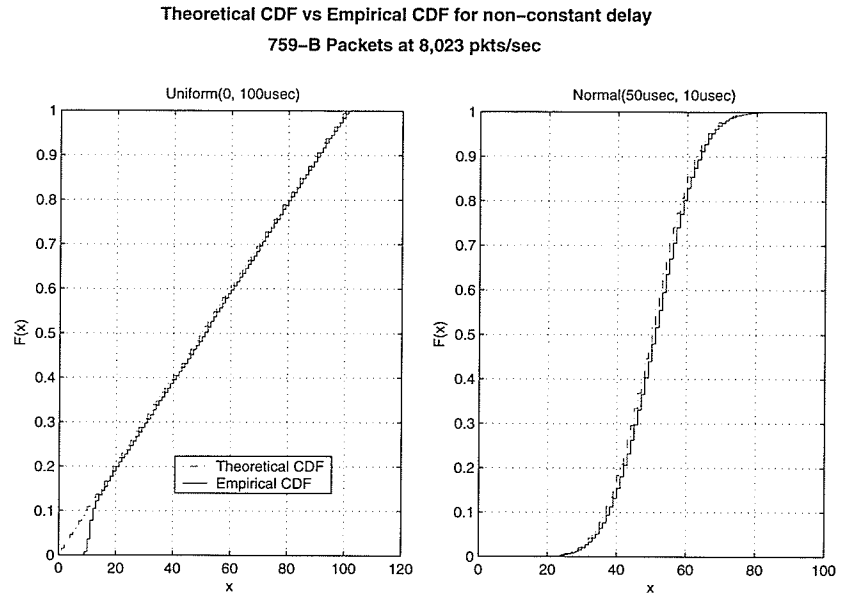

Figure 7.7: Theoretical vs Empirical CDF for 759-B packets at 8,023 pkts/s under delay settings of UNIFORM(0-μsec, 100-μsec) *(left)* and NORMAL(50-μsec, 10-μsec) delay *(right)*.

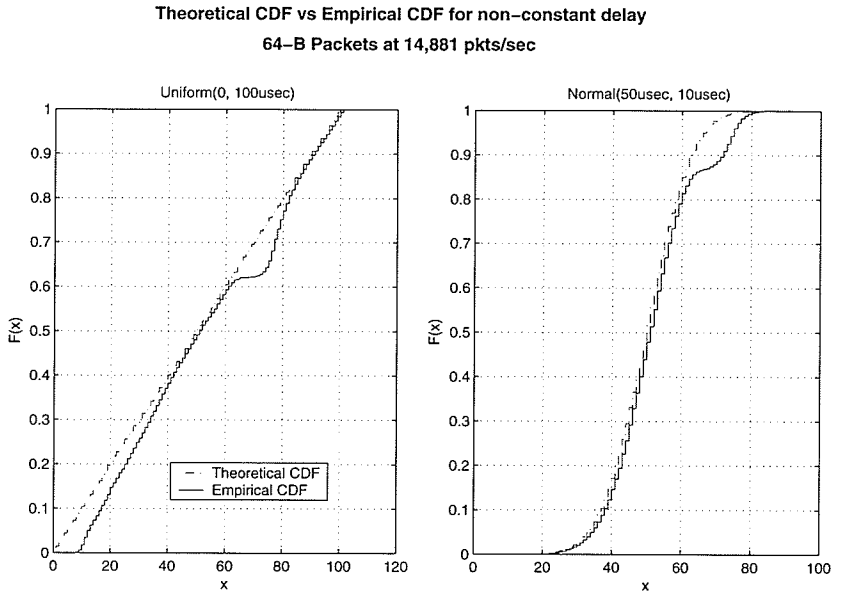

Figure 7.8: Theoretical vs Empirical CDF for 64-B packets at 14,881 pkts/s under delay settings of UNIFORM(0-μsec, 100-μsec) *(left)* and NORMAL(50-μsec, 10-μsec) delay *(right)*.

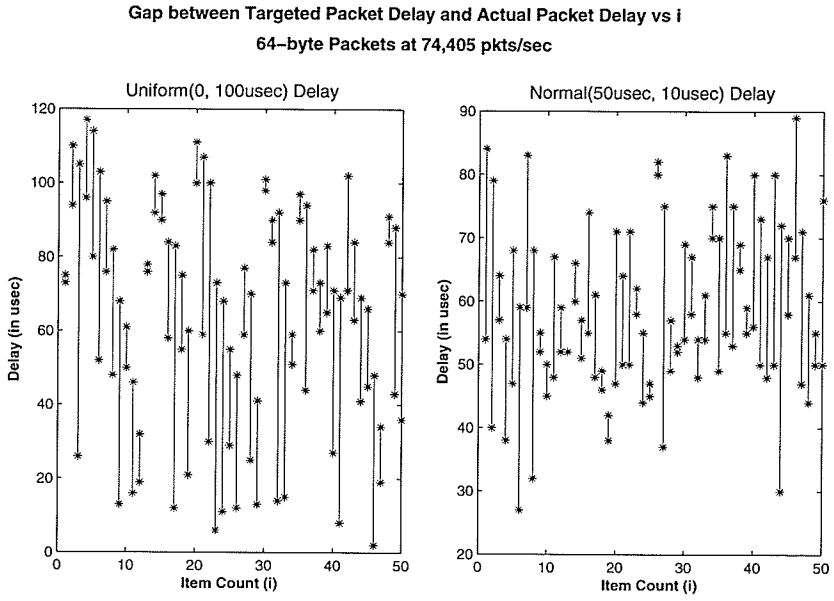

Figure 7.9: The overhead gap between targeted delay and actual delay for UNIFORM(0-μsec, 100-μsec) and NORMAL(50-μsec, 10-μsec) delay models; samples of 50 packets.

7.3 With Duplication

Figure 7.10 displays the mean and standard deviation processing overhead as well as the effective output bandwidth at different duplication probabilities. The initial packet rate is 30% of the maximum bandwidth; as the reader can see, due to the duplication, the effective output bandwidth increases linearly along with the duplication probability. The mean overhead plot indicates that the amount of time required to duplicate a packet is roughly 25 μsec (from the last data point at 100% duplication probability). Keep in mind that the mean overhead is calculated from both duplicated and regular packets. That is why the mean overhead plot increases gradually along with the duplication probabilities rather than a flat line at the value of 25 μsec. The duplication time is illustrated more clearly in figure 7.11.

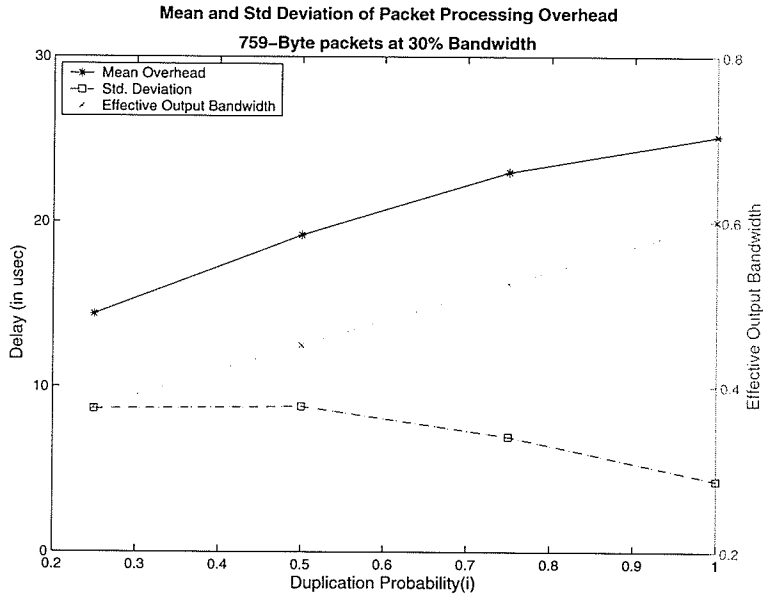

Figure 7.10: Packet processing overhead at different duplication probabilities.

Let i be the incoming packet rate and p be the duplication probability. To avoid packets being dropped, the output packet rate $j = i \times (1 + p)$ must be smaller than the maximum packet rate.

Presently, there is a shortcoming within the Access Library and the impairment implementation such that buffers for duplicated packets are not re-claimed if the packets are dropped. This memory leak eventually crashes the IXDP425 BOARD. The only work

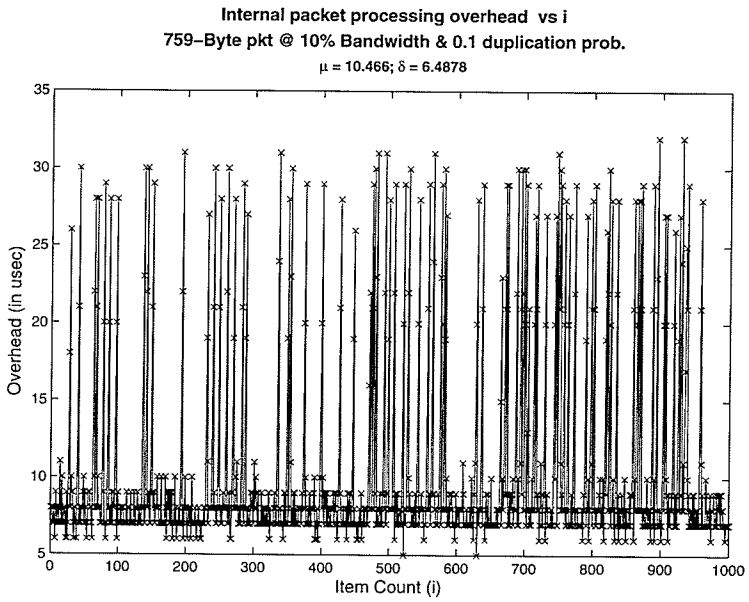

Figure 7.11: Packet processing overhead for 759-Byte packets at 30% maximum bandwidth and at duplication probabilities of 10%.

around is to make sure that the output packet rate is lower than the maximum packet rate.

7.4 With Reordering

The reorder impairment is tested with 759-Byte packets at packet rate of 16,046 pkt/s and reorder probability of 20%. Figure 7.12 shows the packet processing overhead for each packet displacement distance. The packet's delay time increases proportionally to its displacement distance. However, the data is highly variable at each reordered packets with some values spike up above 300 μsec. This is an ongoing problem that is currently being investigated.

7.5 The Effect of Reordering on Packet Delay

In this section, I will discuss the relationship between the reordering model and the delay model. Recall the packet flow in figure 3.1, the reorder module is in front of the delay module. Thus, the reorder module can influence the work of the delay module, BUT not

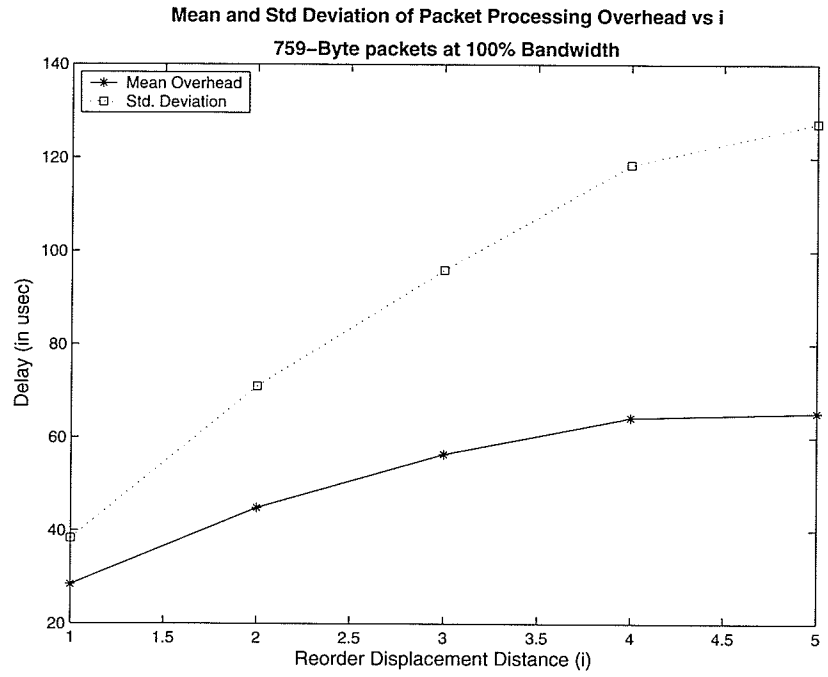

Figure 7.12: Packet processing overhead at 20% reorder probability and displacement distance from 1 to 5.

the other way around.

Let's start off with a quick review of the implementation. There are two software queues involved here; one for the reordering and the other for the delay module. The system reorders a packet by holding it within the reordering queue until a specific number of packets behind it has been processed. The reorder queue must necessarily be a priority queue ordered by the displacement distance of the packets.

On the other hand, the delay queue must necessarily be a FIFO (first in first out queue), so as to preserve the order of packets. One may be tempted to sort the packets in descending order based on the targeted delay to further reduce the undesired overhead. This will work perfectly, save for the fact that we are altering the incoming order of the packets which is strictly disallowed (unless this impairment is specifically requested).

The duration a packet is held in the reorder queue depends on the packet rate. The faster the packet rate, the less time the packet is held in the queue. Thus, if a packet is associated with both reordering and delay modeling, the actual time the packet will be delayed are influenced by four factors:

1. The targeted delay time,

2. the packet rate.

3. the displacement distance, and

4. the targeted delay for the packets behind the current one.

I will clarify the last one. Say a packet named A is to be reordered by a displacement distance of three packets; i.e. it will be held in the reorder queue until three packets after it (B, C, D) have exited the reorder module. Once packet A exits the module, it will be placed after the above packets in the delay queue, assuming that they haven't been sent out. The content of the delay queue could potentially be like this: B - C - D - A. There, you can see why the actual delay of A also depend on the delay of the three packets before it.

When dealing with non-constant packet delay, we have to worry not only about the reordering effect (if set), but also about the non-deterministic nature of the delay. Added to this is the overhead defiency in the reordering module I have mentioned in a previous section. The end effect is that with these two combined, we have some very wildly unexpected output, compared to the input model.

Figure 7.13 shows the theoretical and empirical CDF for the scenario of delay only (on the left side) and for the same delay setting along with the reordering probability of 0.2% and reorder displacement distance of 2. We can see that on the right plot, the tail of the CDF is extremely bent. That is because it is associated with reordered packets where the overhead is high. Keep in mind that in that figure, the delay is set at low values (tens of micro seconds). With large delay (in terms of milliseconds), the cost of reordering is largely hidden. As shown in figure 7.14, there is not much difference between the two plots.

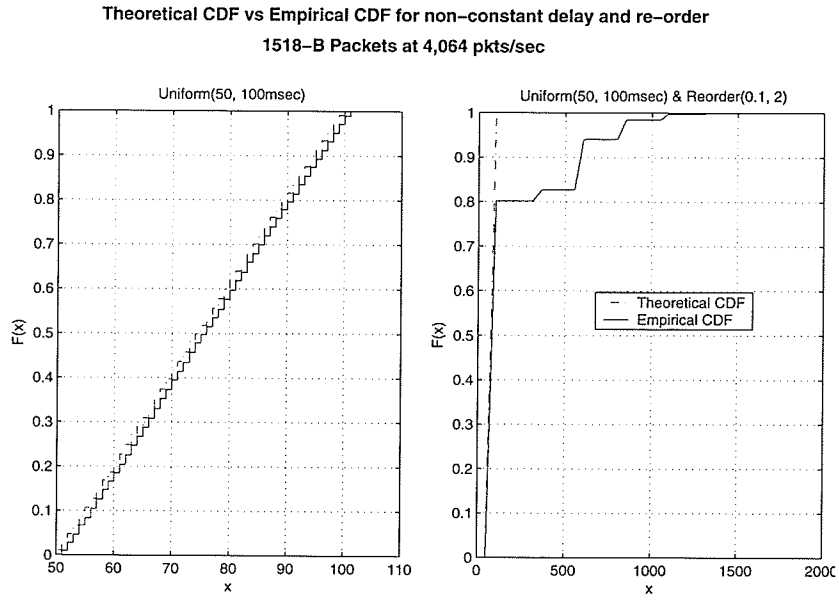

Figure 7.13: Theoretical vs Empirical CDF for 1518-B packets at 4,064 pkts/s under delay UNIFORM(50-μsec, 100-μsec) *(left)* and addition reorder settings of probability 20% and displacement distance 2 packets *(right)*.

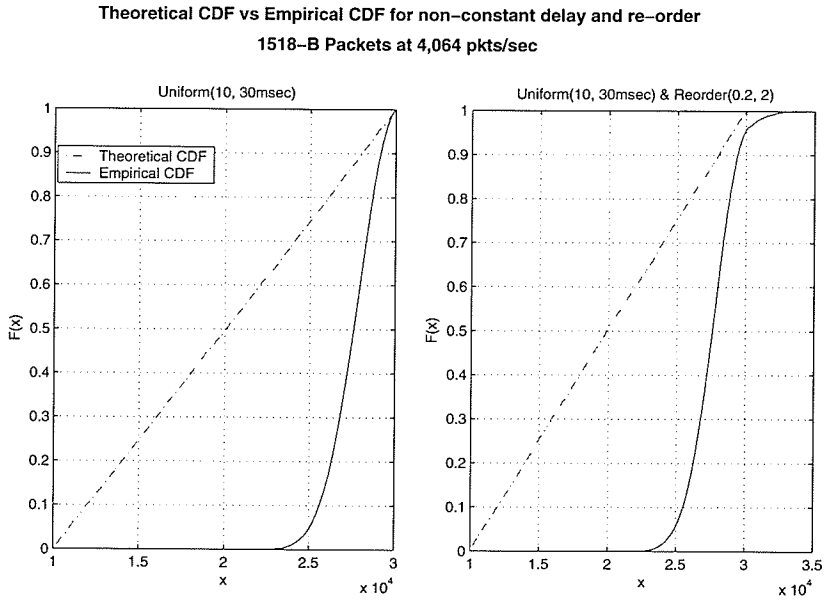

Figure 7.14: Theoretical vs Empirical CDF for 1518-B packets at 4,064 pkts/s under delay UNIFORM(10-msec, 30-msec) *(left)* and addition reorder settings of probability 20% and displacement distance 2 packets *(right)*.

7.6 Bi-directional Traffic

The impairment box's packet processing rate is greatly reduced in the presence of bi-directional traffic. Table 7.7 shows the maximum packet rate the impairment box can accept in the pass-thru (no impairment) mode. Note that the packet rate is for `each` direction.

	64B	759B	1,518B
Max. Theoretical Rate	148,810	16,046	8,127
Max. Achievable Rate	80,000	16,046	8,127

Table 7.7: Maximum achievable packet rate for bi-directional traffic in pass-thru mode.

To fully maximize bandwidth for bi-directional traffic, the test equipments must be set with `auto-negotiate` option on and 100-Mbps full duplex enabled. The IXDP425 BOARD turns on auto-negotiate by default; upon receiving no message, it will automatically go back to half-duplex mode, which reduces the bandwidth by half.

Figure 7.15 compares the packet processing overhead between uni-directional and bi-directional traffic at $8,127$ pkts/s and no impairment. Note that $8,127$ pkts/s happens to be the maximum packet rate for $1,518$-byte packets. Under bi-directional traffic, the software queue occasionally has more than one packets queued concurrently. That explains the sudden spike in the graph (please note that if the test has been carried out at all possible discretized packet size values, you will not see this spike; instead the overhead will gradually increases along with the increase in packet size).

7.7 Trace-based Emulation

This emulation mode uses the same engine as the model-based emulation mode. The only difference is the way the various impairment values are retrieved. Figure 7.16 gives an overview of the packet processing overhead for various types of impairment under this mode. Each value point is the trimmed mean of a $10,000$-item sample. The trace entries are purposedly formed so as to finally arrive at the impairment parameters shown in the graph legends.

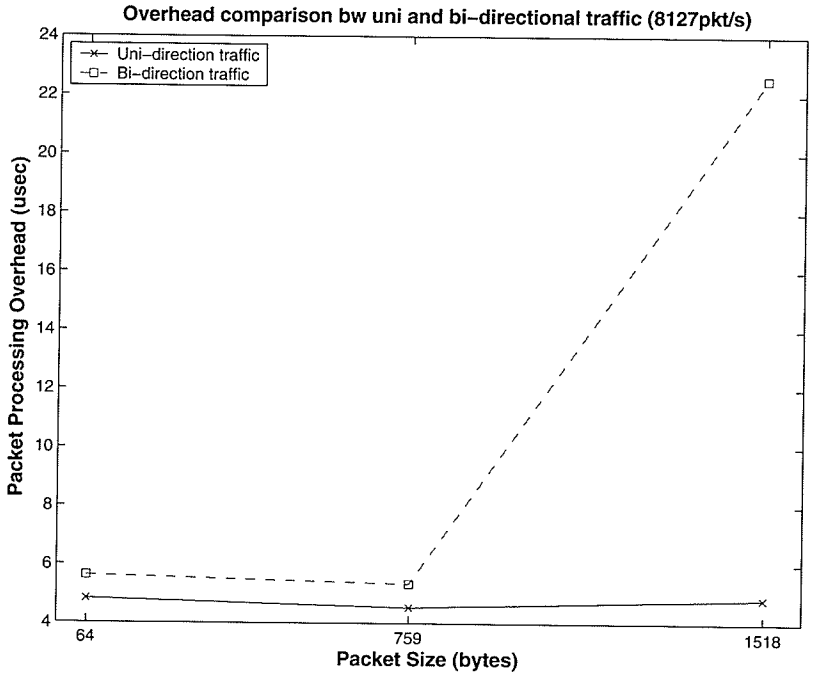

Figure 7.15: Packet processing overhead comparison between uni-directional and bi-directional traffic at packet rate of $8,127$ pkts/s and no impairment.

7.8 Overview Discussion

In this chapter, a series of performance tests have been carried out at various packet sizes and different incoming packet rates. The packet processing measurements are done both internally within the IXDP425 BOARD and externally at the end points. I first started out with the trivial case of simply bridging traffic between the two ports (i.e. no impairment), and then gradually moved on to each impairment mode.

The test result indicated that with no impairment imposed, the system can bridge `uni-directional` traffic at any packet size and packet rate. However, with `bi-directional` traffic, in the extreme case (packet size of 64B), the system can only handle 53.7% of the maximum packet rate ($148,810/s$). The major limitation is the packet rates (which is directly influenced by the packet size). At high packet rate, the internal packet queues are filled up very quickly, causing delay and consequently dropping packets.

With the existence of constant delay, the packet processing overhead is rather low, and within expectation. However, with non-constant delay, the processing overhead is non-deterministic. The duplication also performs well as planned. The reordering module,

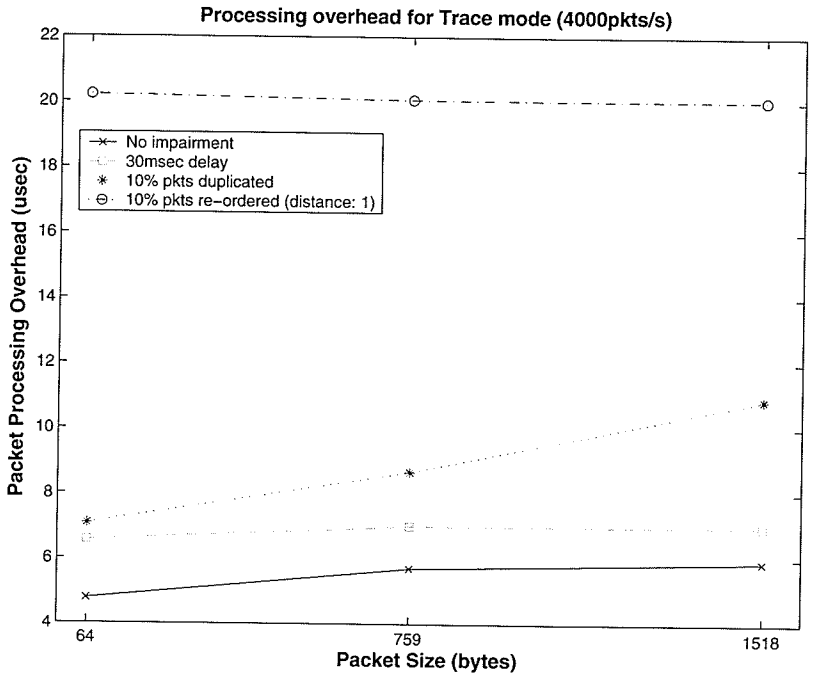

Figure 7.16: Packet processing overhead for various impairment when operated under TRACE mode at $4,000$ pkts/s.

on the other hand, is having some performance problem. The packets are reordered correctly according to the schedule, but in most cases, they are delayed much more than they should be (the overhead in many cases reaches 300-μsec).

Regarding the non-constant delay impairment, there is no guarantee that the distribution of the actual packet delay time will conform to the initial input model (e.g.: the input model is UNIFORM(10-msec, 50-msec), but the CDF is no where resembling a UNIFORM CDF. One may argue that the system could somehow foresee the non-deterministic delay effect, and shape the packet delays accordingly, so as to arrive at the outgoing distribution closely matches up with the input. This is a great amount of work that may not be feasible; I have decided not to follow up on it. Instead, the user should be aware of this effect when inputting the model.

Chapter 8

Conclusion

In this project, I have designed and developed a network impairment tool. The tool can inject packet loss, duplication, reordering, and delay at two different emulation modes: model-based and trace-based.

The final impairment system has proved to be quite flexible. Besides the simple models such as UNIFORM and NORMAL, the packet-based and time-based Markov chains are powerful tools to model complex system behaviors. In addition, the trace-based emulation mode gives the user another approach to examine the network devices under test. The various tests carried out in chapter 7 has indicated that the packet processing overhead is very small; negligible in most impairment scenarios. Except for the case of bi-directional traffic at packet size smaller than 120-Byte, the impairment system is capable of processing packets at wire speed. The system is also very reliable; no deficiency is observed after running continuously for several days. Thus, it can be concluded that most of the targeted requirements have been satisfied.

The work remained to be done is to resolve several problems with packet duplication and reordering that were mentioned previously in chapter 7. Another major task is to embed all the software components within the flash memory of the IXDP425 BOARD. Once done, the impairment system will be self-contained within the IXDP425 BOARD, eliminating its dependency on an external host system.

Bibliography

[1] J. Bellardo and S. Savage. Measuring packet reordering. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurement workshop*, pages 97–105. ACM Press, November 2002.

[2] J. Bi, Q. Wu, and Z. Li. Packet delay and packet loss in the Internet. In *Proc. of the 7th Int. Symp. on Computers and Communications (ISCC'02)*, pages 3–8. IEEE Computer Society, July 2002.

[3] J. C. Bolot. End-to-end packet delay and loss behavior in the Internet. In *Conference proc. on Communications architectures, protocols and applications*, pages 289–298. ACM Press, October 1993.

[4] R. P. Brent. Algorithm 488: A gaussian pseudo-random number generator. *Commun. ACM*, 17(12):704–706, 1974.

[5] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[6] M. Carson and D. Santay. NIST Net: A Linux-based network emulation tool. *SIGCOMM Computer Communication Review*, 33(3):111–126, July 2003.

[7] Spirent Communications. Spirent SX data link simulators. http://www.spirentcom.com/documents/439.pdf.

[8] IXIA Corp. Ixia hardware test platforms - 1600t/400t/100. http://www.ixiacom.com/datasheets/pdfs/ch_1600t_400t_100.pdf.

[9] L. Devroye. *Non-uniform random variate generation*. Springer-Verlag New York Inc., 1st edition, 1986.

[10] P. Dube, O. Ait-Hellal, and E. Altman. On loss probabilities in presence of redundant packets with random drop. *Perform. Eval.*, 53(3–4):147–167, 2003.

[11] G. W. Fischer, Z. Carmon, D. Ariely, G. Zauberman, and P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[12] E. N. Gilbert. Capacity of a burst noise channel. *Bell Syst. Tech. Journal*, 39:1253–1265, September 1960.

[13] D. Herrscher and K. Rothermel. A dynamic scenario emulation tool. In *Proc. of the 11th Int. Conf. on Computers Communications and Networks (ICCCN'02)*, pages 262–267, October 2002.

[14] Intel. Inc. Intel ixp425 network processor. http://www.intel.com/design/network/products/npfamily/ixp425.htm.

[15] Intel Inc. Intel(r) IXDP / IXCDP1100 development platform. http://www.intel.com/design/network/products/npfamily/ixdp425.htm.

[16] MathWorks Inc. Matlab documentation, distribution testing: Function reference. http://www.mathworks.com/access/helpdesk/help/toolbox/stats/refere12.html.

[17] OPNET Technologies Inc. OPNET modeler – accelerating networking R&D. http://www.opnet.com/products/modeler.

[18] C. M. Jarque and A. K. Bera. Efficient tests for normality, homoscedasticity and serial independence of regression residuals. *Economics Letters*, 6(3):255–259, 1980.

[19] S. A. Khayam and H. Radha. Markov-based modeling of wireless local area networks. In *Proc. of the 6th Int. workshop on Modeling analysis and simulation of wireless and mobile systems (MSWiM'2003)*, pages 100–107. ACM Press, September 2003.

[20] C. Kulkarni, M. Gries, C. Sauer, and K. Keutzer. Programming challenges in network processor deployment. In *Proc. of the Int. Conf. on Compilers, architectures and synthesis for embedded systems*, pages 178–187. ACM Press, October 2003.

[21] InterWorking Labs. Maxwell, the network impairment system. http://www.iwl.com/Products/maxwell/index.html.

[22] P. L'Ecuyer. Combined multiple recursive generators. *Operations Research*, 44(5):816–822, 1996.

[23] J. L. Leva. A fast normal random number generator. *ACM Trans. Math. Softw.*, 18(4):449–453, 1992.

[24] M. Mitzenmacher and R. Rajaraman. Towards more complete models of TCP latency and throughput. *Journal of Supercomputing*, 20(2):137–160, September 2001.

[25] NIST/SEMATECH. e-handbook of statistical methods. http://www.itl.nist.gov/div898/handbook.

[26] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proc. of the ACM SIGCOMM '97 Conf. on Applications, technologies, architectures, and protocols for computer communication*, pages 51–61. ACM Press, October 1997.

[27] NS. The network simulator - ns-2. http://www.isi.edu/nsnam/ns.

[28] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transaction on Networking*, 7(3):277–292, June 1999.

[29] M. Peyravian and J. Calvignac. Fundamental architectural considerations for network processors. *Computer Networks*, 41(5):587–600, April 2003.

[30] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, February 1989.

[31] Wind River. Wind river - device technologies - operating systems - vxworks 5.x. http://www.windriver.com/products/device_technologies/os/vxworks5.

[32] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, January 1997.

[33] L. Salomone. Chi-square test for continuous distributions. http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=4779&objectType=file.

[34] Cisco Systems. *Internetworking Technologies Handbook*. Cisco Press, August 2003.

[35] K. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley and Sons, 2nd edition, 2001.

[36] C. S. Wallace. Fast pseudorandom generators for normal and exponential variates. *ACM Trans. Math. Softw.*, 22(1):119–127, 1996.

[37] W. Wei, B. Wang, and D. Towsley. Continuous-time hidden markov models for network performance evaluation. *Performance Evaluation*, 49(1–4):129–146, September 2002.

[38] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *Proc. of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM'99*, pages 345–352. IEEE, March 1999.

[39] I. Yeom and A.L. N. Reddy. ENDE: An end-to-end network delay emulator tool for multimedia protocol development. *Multimedia Tools and Applications*, 14(3):269–296, August 2001.