A FORMAL APPROACH TO PROTOCOL OFFLOAD FOR WEB SERVERS

By

Juan Manuel Solá-Sloan

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTING INFORMATION SCIENCE AND ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

May, 2009

Approved by:

Nayda G. Santiago-Santiago, Ph.D Member, Graduate Committee	Date
Manuel Rodríguez-Martínez, Ph.D Member, Graduate Committee	Date
Domingo Rodríguez-Rodríguez, Ph.D Member, Graduate Committee	Date
Isidoro Couvertier-Reyes, Ph.D President, Graduate Committee	Date
Dorial Castellanos-Rodríguez, Ph.D Representative of Graduate Studies	Date
Nestor Rodríguez-Rivera, Ph.D Chairperson of the Department	Date

ABSTRACT

The use of networks as computational tools has been fundamental in Computer Information Science and Engineering research. As computers have evolved, the demands for faster data transmission have incremented. Protocol processing and application processing compete for a share of CPU time when the underlying network evolves.

Offload Engines have been proposed as ways to alleviate the CPU of the host of the cost of processing the protocols used in an end-to-end communication. These engines are proposed as part of the network interface hardware or as a component near or even inside the CPU. However, an increase in performance is not guaranteed after the installation of this type of engine. The operating system of the host is capable of processing the protocols faster than the engine in *some* particular occasions. A probabilistic model that estimates the performance metrics of utilization and delay, before and after extracting the protocol processing from the CPU of the host is presented in this document. This novel approach focuses on solving the issues presented by abstracting the inherent characteristics of protocol processing. A test bed was setup to validate the analytical model. The test bed was designed using an emulation-based approach mixed with a real implementation to reproduce the behavior of a TCP Offload Engine installed as part of a server that runs Apache 2.2. The file system used for populating the Web server has been divided into five different classes based on its sizes. Each file class has been requested using benchmarks to stress the server in different ways.

The analysis of the results obtained shows that the analytical model estimates the utilization and delay of the system. The analysis shows that, for some cases,

the inclusion of an offload engine improves overall system performance. The inclusion of the model into the operating system can be used to determine when and which objects are going to be handled by the offload engine and which are going to be handled by the operating system. Also, by using our model, we can provision the operating system with the capability of balancing the load between the offload engine and the host.

RESUMEN

El uso de redes como instrumentos de investigación ha sido fundamental en Ciencias e Ingeniera de la Información y la Computación. La demanda por una transmisión de datos más rápida ha incrementado gracias a la evolución en la computación. El procesamiento de los protocolos de comunicación así como el procesamiento de programas de aplicación compiten por una fracción de tiempo del procesador principal cuando la red que transmite los datos evoluciona de una velocidad menor a una mayor.

Los motores de descarga[1] han sido propuestos como una alternativa para aliviar al procesador principal del computador anfitrión del costo de procesar los protocolos de comunicación. Estos motores pueden ser incluidos como un equipo adicional que se añade a la tarjeta de red o como un componente cerca, o dentro, del procesador principal. Sin embargo, un aumento en el desempeño del computador no está garantizado después de la instalación de un motor de descarga. El sistema operativo del computador anfitrión es capaz de procesar los protocolos de comunicación más rápido que el motor de descargue en algunas ocasiones particulares. Un modelo probabilístico que estima utilización y demora, antes de y después de extraer el procesamiento de los protocolos de comunicación del procesador principal, se presenta en este documento. Este enfoque novedoso se centra en resolver los asuntos expuestos representando en el modelo analítico las características inherentes del procesamiento de los protocolos de comunicación. El

[1] "Offload Engines"

modelo analítico fue validado utilizando un experimento. Nuestra base experimental fue diseñada utilizando un enfoque basado en la emulación y complementado con una implementación real que reproduce la conducta de un motor de descarga de TCP/IP instalado como parte de un servidor que corre Apache 2.2. El sistema de archivos que se utilizó para poblar el servidor de Web ha sido dividido en cinco clases que están basadas en el tamaño del archivo. Archivos de cada clase fueron solicitados del servidor de Web utilizando los programas de *estándar de comparación* ("benchmarks") de diferentes maneras.

El análisis de los resultados expone que el modelo analítico estima la utilización y la demora del sistema. El análisis muestra, que para algunos casos, la inclusión de un motor de descarga mejora el desempeño general del sistema. La inclusión del modelo como parte del sistema operativo puede ser utilizada para determinar cuando y cuales objetos serán manejados por el motor de descarga y que será manejado por el sistema operativo. También, utilizando nuestro modelo, se le puede proveer al sistema operativo con la capacidad de equilibrar la carga entre el motor de descarga y el procesador principal del computador anfitrión.

Dedication

This dissertation is dedicated to the ones with character, courage, conviction, determination, and strength.... For those that are true to their ideals and convictions and not the ones imposed by others. For all of you my brothers and sisters, I raise my sword into the wind.

ACKNOWLEDGMENTS

First of all, I want to thank the NSF and the PRECISE program at CECORD, coordinated by Dr. Domingo Rodríguez and Dr. Nestor Rodríguez for the financial support during the first years of this reasearch. Also, I want to thank the AGEP program of the University of Puerto Rico and its coordinator Dr. Manuel Gómez of the Rio Piedras campus for their support. I specially want to thank Dr. Isidoro Couvertier for informally *drafting* me as his Ph.D. student on July 2001. Also I want to thank my graduate committee: Dr. Manuel Rodríguez, Dr. Nayda Santiago, and Dr. Domingo Rodríguez. Also I want to thank Dr. Yi Qian for being part of my committee during my first years at the Ph.D. program. I would like to extend special thanks to my immediate family and my relatives Elda and Hiram Gay for their support during my first year at the CISE Ph.D. program.

I would like to thank the following people for helping me during this research. First of all, Dr. José Vega Vilca of the Institute of Statistics of the University of Puerto Rico at Rio Piedras for the invaluable time he spent reviewing and commenting on the mathematical model. He should be the fifth member of the committee. Also I would like to thank Dr. Hector J. Huyke and Dr. William Frey for their advise and comments on the ethics chapter. Also I like to thank Dr. Ramón Vásquez Espinosa and Dr. Edgar Acuña for guiding me.

I would like to thank Luis Ortiz Ortiz and José Vega Vilca for being my studying partners during my preparation for the qualifying examination.

Also I liked to thank the following professors for being an inspiration to me during my early years: Ricardo Coronado, Jose J. Díaz-Caballero, Filiberto Arniellas and Ismael García. Also, I want to thank my friends from the Humanities department and from Pluriverso: Dr. Luis Pabón, William Rios, Dr. Javier Villalonga and Francisco García.

Also I would like to thank Urayoán Camacho and Rafael Linero for their help during tough times.

Also I would like to thank the following people for their emotional support: Jaime Alfonso Alcover, Octavio Rodríguez, José Vélez and the Morales family, and attorneys: Nelson Rodríguez, Jesús Rivera Delgado and Arturo Negrón Jr.

The following were the people that help in maintaining my sanity during tough times. First of all, my other half, Jeyka Laborde Pérez for her unconditional love. Special thanks to Dr. Francisco O'Neil Sulsona for his therapies and recommendations. I would like to thank the underground heavy metal community for encouraging me into staying on the program and not quitting.

THANKS TO ALL OF YOU...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASP	: Active Server Page
CR	: Commander and Reader Process
CDF	: Cumulative Distribution Function
CGI	: Common Gateway Interface
CMP	: Chip-Level Multi-Processor
DMA	: Direct Memory Access
EMO	: Extensible Message Oriented Model
FP	: Forker Process
Gbps	: Gigabit Per Second also (G/s)
HTML	: Hypertext Markup Language
HTTP	: Hypertext Transfer Protocol
IP	: Internet Protocol
iNIC	: Intelligent Network Interface Card
K-S	: Kolmogorov-Smirnov Analysis
OE	: Offload Engine
OS	: Operating System
P-K	: Pollaczek-Khinchin formula
PCT	: Process Control Table
PID	: Process Identifier
PPBP	: Poisson Pareto Burst Process
SP	: Slave Process

Mbps	: Millions of Bits Per Second
NIC	: Network Interface Card
NNTP	: Network News Transfer Protocol
RDMA	: Remote Direct Memory Access
SMP	: Symmetric Multi Processor
SMTP	: Simple Mail Transfer Protocol
SURGE	: Scalable URL Reference Generator
TCP	: Transport Control Protocol
TOE	: TCP/IP Offload Engine
ULP	: Upper Layer Protocol
URL	: Uniform Resource Locator
WS	: Web Server
WWW	: World Wide Web

LIST OF SYMBOLS

ρ : utilization.

μ : service rate of a server.

X_a : random variable that represents the service time for processing the application at the CPU.

X_p : random variable that represents the service time for processing the communication protocols at the CPU.

X_q : random variable that represents the service time for processing the communication protocols at the offload engine.

X_c : random variable that represents the service time for processing the control signals at the offload engine.

X_o : random variable that represents the service time for processing the control signals at the CPU.

μ_p : the service rate of processing the protocols at the CPU.

μ_a : the service rate of processing the application at the CPU.

μ_q : the service rate of processing the protocols at the offload engine.

μ_c : the service rate of processing the controls signals at the offload engine.

μ_o : the service rate of processing the controls signals at the CPU.

T_{cpu} : the expected total time a request spent on the CPU.

W_{cpu} : the expected time waiting to be serviced that a request spends on the CPU.

T_{oe} : the expected total time a request spends on the offload engine.

W_{oe} : the expected time waiting to be serviced that a request spends on the offload engine.

T : the expected total time in the system.

U_{oe} : utilization of the offload engine.

U_{cpu} : utilization of the CPU.

U_{noe} : utilization of a host without the support of an offload engine.

N_q : expected number of requests waiting to be service at the offload engine.

N_{oe} : expected total requests waiting inside the offload engine.

CHAPTER 1
INTRODUCTION

The computer has evolved from been a solely independent computational device to be a significant component that provides computational services within a network. Traditionally, network protocols are implemented in software as part of the operating system of the host. Increases in the quantity and resolution of the data, and the need for faster data transmission, encourage their implementation in specialized hardware. Hardware implementations have a tendency of achieving a higher performance increase than software ones, but with a significant increase in the implementation effort due to their complexity and cost. Meanwhile, advances in technology have made software implementation attractive. Novel methods for understanding the specific features of protocol processing are needed for deciding if a hardware or software implementation suffices.

On the advent of multi-core multi-threaded processors and new technologies on outboard processors, a new debate related to where protocol processing must be performed arises. An offload engine (OE) is a specialized entity outside the main CPU that processes in whole or in part the communication protocols that traditionally are processed by the operating system of the host. The intention of including an offload engine is to reduce the load incurred when processing communication. Some protocol offload engines have been proposed as part of the

Network Interface Card (NIC) [1, 2]. Others have proposed placing the OE near the CPU or even inside the CPU [3–6]. The most common offload engine is used to process the TCP and IP protocols. This engine has been called the TCP Offload Engine (TOE) (see Figure 1–1) [7]. The main issue addressed by processing the protocols efficiently is to alleviate the problem that arises when the speed of the underlying network transfers data in a way that imposes a heavy load on the CPU of the server. The same CPU also processes the network application that uses the data transferred. Consequently, the CPU becomes the bottleneck of the system.

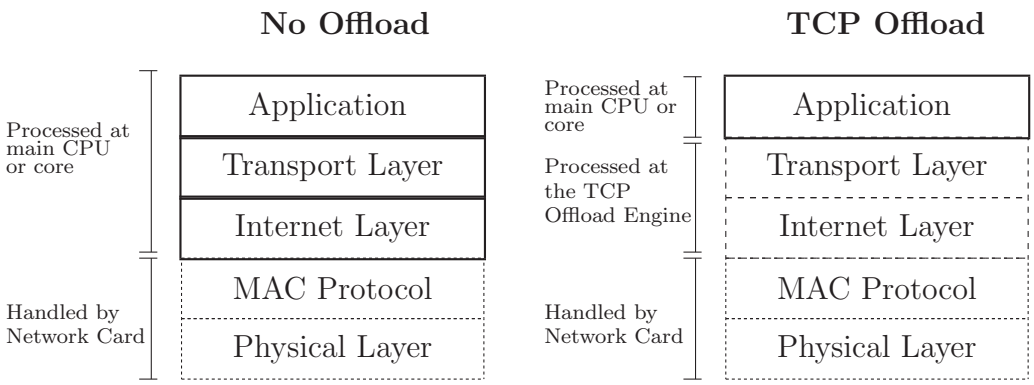


Figure 1–1: Internet Protocol Suite with and without a TCP Offload Engine

1.1 Motivation

There is no guarantee that the inclusion of an OE within the system results in performance increases. The interactions between the CPU and the protocol offload engine are independent of the underlying network technology. A poorly designed interface between the host and the OE is likely to ruin most of the performance benefit brought by protocol offload [9]. There are some situations in which the OE could be lagging behind the server and degrading the system [10].

The system is degraded if converting a file into an object to be transfered takes more time after the inclusion of the offload engine. Also the system is degraded when the actual overhead used to communicate with the offload entity is equal or takes longer than performing the actual transmission at the OS. This results in a system that is unable to provide its clients with better or similar response times than the traditional-host[1] . However, the inclusion of an OE *could be* beneficial in some cases [11].

Most of the protocol offload research found during our survey of literature is experimental. The experiments focus in processing the protocols in an OE as part of the Network Interface Card (NIC), into another core of the same CPU, or into a dedicated General Purpose Processor (GPP). However, during our exhaustive survey of literature, we were unable to find a Queueing Theory based model for estimating the change in performance for protocol offload engines. A limited number of articles deal with protocol offload using analytical modeling [10, 12]. The models found are deterministic and oriented to a single aspect of protocol offload [12].

Performance modeling is an important part of the research area of Web servers. Without a correct model, it is difficult to give an accurate prediction of performance metrics. Queueing Theory has been used for performance modeling of Web servers and also for Web traffic generation[2] . However, there is no consensus whether which distribution is the best for modeling the length of the

[1] a host without protocol offload

[2] The M/G/1 queue is used for both topics

files that are going to be converted into traffic streams [13–15]. Classical modeling gives a very poor approximation [16, 17]. A probabilistic model was design using the tools provided by the M/G/1 queue after examining the available literature. This model is centered on protocol offload engines.

1.2 Problem Definition

This dissertation presents a probabilistic model that is capable of estimating the performance increase in utilization and delay of a system when protocol processing is extracted from the OS and performed in another entity. The model uses Queuing Theory for approximating these performance measures. The probabilistic model was devised under the context of a Web Server that handles HTTP requests. Finally the model was validated using a case study oriented to TCP offload.

The uniqueness of our research abstracts analytically some of the inherent characteristics of protocol processing for estimating the performance metrics to determine *a priori* in what situations protocol offloading is beneficial. This is performed by modeling the system as a network of queues as presented in Figure 1–2. Using the steady state properties of the M/M/1 and M/G/1 queues we obtained a new formulation of the problem for the utilization and delay of the system.

In our case study, the model is validated using a mixture between a real implementation at the Web Server (WS) and an emulator of an OE in an external PC (Figure 1–3). The experimental setup includes a study focused on the behavior of the host CPU when confronted to different object sizes using an approach similar to the one presented in [18]. This research aims to study the pros and cons of protocol offloading and its related environment.

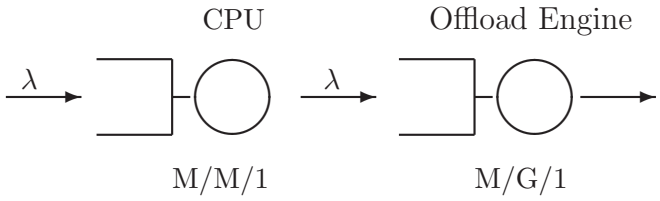


Figure 1–2: Abstraction of a CPU Supported by an Offload Engine

1.3 Contributions and Significance

The first main contribution was producing **a probabilistic model that estimates the performance increase or decrease in utilization and delay when protocol processing is shifted from the operating system to a protocol offload engine**. Our analytical model considers the stochastic nature of the end-to-end transmission. Consequently our model includes the process for converting the requested object R into packets of protocol P and how it was distributed. The analytical model is simple and extensible.

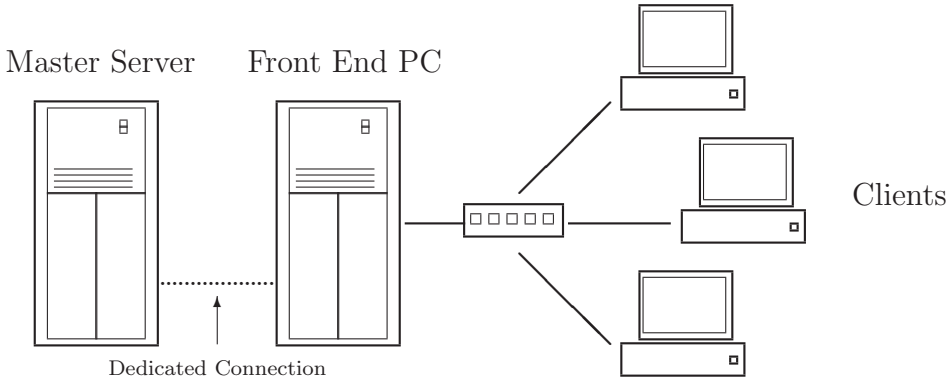


Figure 1–3: Experimental Setup

The second contribution was the development of **a TCP Offload Engine emulator as a tool used to validate the model. The emulator provides an insight of the real problem beyond the limitations of the analytical model**. The levels of detail of our emulation-based approach are finer than those provided by the oversimplification of the probabilistic model. The emulation-based approach provides our research with an experimental platform.

Our approach of **classifying files based on their sizes present an innovation for studying offloading from a totally different perspective**. This is our third main contribution. Dependencies on the size of the requested object stresses the host in different ways [18, 19]. This approach has been followed for further improvements on the Web server application but has not been oriented to protocol processing improvements. We were unable to find investigations on the standpoint of protocol offload that segregated the requested objects and analyzed its impact on performance. Most of the research conducted in TCP offload abstract the file system as a whole [20–22] or even uses workloads that benefits their test bed [4, 21, 23]. Studying only the general case, or the extreme case, could conceal the real benefits of protocol offload over a wide variety of different situations [24]. As mentioned before, it is known that the server behaves differently when handling different file sizes.

The fourth main contribution was **investigating the performance of the Web server before and after protocol processing extraction not only when the system saturates but also when the system is in equilibrium**. Related research [6, 11, 20] used benchmarks to saturate the system and obtained

measures for analyzing what happens under these extreme conditions. Our approach not only studies the behavior when the system is saturated but also studies the behavior of the system when it is not.

The fifth main contribution **was confronting the analytical model with a real implementation of the common Web server used today**. Apache 2.2 was used as part of the test bed. Infimal changes have been done to Apache 2.2 core application when interfaced with the TCP offload engine emulator. Therefore, our research implies that a network application does not need to be modified substantially if an offload engine is included as part of the system.

1.4 Marginal Contributions

The research conducted provided a mathematical model that was validated with a test bed composed of an emulator and a real-implementation. Since the analytical model uses the notions of Queuing Theory our research scope is not bound to a specific hardware or coded into any specific on-board embedded processor. Therefore, it is not limited by a specific hardware constraint that can ruin or bias our findings.

This research uses full TCP and IP offload. The test bed used in this research allows no constraints for achieving full protocol offload.

1.5 Dissertation Overview

This document is divided into six chapters. Chapter 2 provides a review of published work, both theoretical and experimental, that are fundamental for sustaining the relevance of the problem, analyze the accomplishments of other research and to study the areas of knowledge that contribute to our findings. Chapter 3 formally states the problem, presents the methods used to conduct the research, and describes the test conducted on our experimental setup. Chapter

4 presents the results obtained from the validation of the analytical model. Also this chapter presents some additional results obtained by stressing the test bed beyond our original intentions. Chapter 5 reviews the main findings of the investigation highlighting its contributions. Also this chapter presents the limitations encountered during the whole research process. Moreover, Chapter 5 presents final remarks and future directions for further improvements on this research area. The ethical considerations taken into preparing this research are presented in Chapter 6. Finally, a series of appendices present supporting material related to the scope of this dissertation.

CHAPTER 2
SURVEY OF LITERATURE

A fundamental part of our research consisted in reviewing the literature on the nature of performance modeling, protocol processing, and network application behavior. The uniqueness of this research contributes in finding how these areas can complement each other for further understanding of the problem. This chapter reviews both the analytical and the experimental studies in the field emphasizing the issues and techniques that are closely related to the contributions presented on this document.

2.1 Analytical Modeling

This section summarizes, discusses, and presents the issues that are related when modeling the system analytically. Experimental studies have shown that Web traffic can have different characteristics than traditional voice traffic [16, 17, 25]. These characteristics are directly related to the objects from within the traffic originates.

2.1.1 Limitations of Classic Modeling

Some features of classical models used for telephonic and telegraphic research are not suitable for modeling the traffic over a wide area network. Classically the Poisson distribution has been used to describe job arrivals in a server. Hanning, Samorodnitsky, Marron, and Smith state that modeling packets as jobs arriving

into a server following a Poisson distribution is an assumption that could be misleading [16]. In [17], Paxson and Floyd presented that modeling Web area traffic as Poisson is not appropriate for some Upper Layer Protocols (ULP). Perhaps, the problem can be approached from a different point of view by analyzing the behavior of the sessions.

A session initiation is a natural phenomenon that is consonant with the arrival rate of a job in telephonic and telegraphic research. In Web traffic, a human behind a computer initiates a session. Therefore, **session** arrivals for FTP and HTTP can be modeled as a Poisson process. On the contrary, sessions with other ULPs as the Network News Transfer Protocol (NNTP) and the Simple Mail Transfer Protocol (SMTP) have inter-arrival times that are far from *exponential* [17].

The actual data transfer is not well described with a Poisson process. The inter-arrival times between packets within a session does not follow an *exponential* distribution [16, 17]. Also, session length modeling using the approach of the traditional voice traffic is not suitable for modeling the Web session lengths [25]. Within a Web session, the data is carried by a series of *bursts* [26]. A burst is a collection of related packets from the same connection that constitute an active period of data transmission. In [16] it is demonstrated that modeling bursts using the classical approach leads to deceptive results. The article presents a graphical representation of the real traffic and the traffic generated using a simulation. It is concluded then that an *exponential* distribution is a poor approximation to the service distribution [16].

2.1.2 Novel Approaches of Analytical Modeling

Novel research has explored the use of the Poisson distribution to model the arrival of Web sessions. In [27], Miorandi, Kherani, and Altman present a queuing

model for HTTP traffic over wireless LANs. Time in between Web sessions was modeled using an *exponential* distribution. In [28], Kherani and Kumar modeled the request for file transfers as a Poisson process. In [29], Guo, Crovella, and Matta, studied Web traffic under *chaotic* conditions. Arrivals were modeled using a Poisson process. In [30], Cao, Andersson, Nyberg, and Kihl presented a Web server performance analysis using Queuing Theory. Sessions followed a Poisson distribution. In [16], Hanning, Samorodnitsky, Marron, and Smith suggested modeling the Web traffic initiations using an homogeneous Poisson process. In [25], Tudjarov, Temkov, Janevski, and Firfov presented an Empirical modeling of Internet traffic at middle-level *burstiness* that uses a Poisson process to model Web session arrivals. Most of the investigations found in our survey of the literature modeled the arrival of sessions as a Poisson process. Therefore, it is common in the scientific community to use the Poisson distribution to model this type of process.

Queuing Theory emerges as a candidate for modeling Web Servers whenever the arrival process of a session is asserted as Poisson. Markov models are performance analysis tools that capture the inherent uniqueness of Web session initiations. In [30], Cao and others presented an M/G/1/K queue for Web server performance modeling. Another approach using tandem queues was used by Van der Mei, Hariharan, and Resser in [31]. The model was used to predict Web server performance metrics and was validated through measurements and simulations. In [32], Cherkasova and Phaal, presented a similar Markovian model with Poisson arrivals and deterministic session times. The novelty of our approach relies on characterizing protocol offload in Web servers using M/G/1 queues. This model is simple and renders a smaller parameter space, thus it is easier to estimate. A

simple model (e.g. M/M/1/K or M/D/1/K) can predict some Web server performance metrics, but making the assertion that the service time distribution is *exponential* or deterministic is risky [30]. In the survey of literature conducted we were unable to find models that used these type of queues applied to protocol offload engines.

Queuing Theory emerges as a candidate for modeling Web Servers whenever the arrival process of a session is asserted as Poisson. Markov models are performance analysis tools that capture the inherent uniqueness of Web session initiations. In [30], Cao and others presented an M/G/1/K queue for Web server performance modeling. Another approach using tandem queues was used by Van der Mei, Hariharan, and Resser in [31]. The model was used to predict Web server performance metrics and was validated through measurements and simulations. In [32], Cherkasova and Phaal, presented a similar Markovian model with Poisson arrivals and deterministic session times. The novelty of our approach relies on characterizing protocol offload in Web servers using M/G/1 queues. In the survey of literature conducted we were unable to find models that used Queuing Theory for protocol offload. This model is simple and renders a smaller parameter space, thus it is easier to estimate. A simple model (e.g. M/M/1/K or M/D/1/K) can predict some Web server performance metrics, but making the assertion that the service time distribution is *exponential* or deterministic is risky [30].

Web sessions are initiated whenever a task is required from the server. In our context, a session consists of a client requesting an object that is going to be transferred from the server's location to its destination. This web object is converted to a series of packets that constitute a data stream, also called a burst. We can approximate the behavior and length of these bursts by studying traffic

over the Internet. Characterizing the length and behavior of the traffic streams was a key issue that was not overlooked during the research process.

2.1.3 Innovations on Web Traffic Modeling

It is known that Internet traffic is shown to be self-similar and *bursty* by nature [25, 33]. Aggregation of many of these bursts produce self-similar traffic [34]. Traffic that is self-similar posses fractal behavior when observed in various time scales [17, 26]. Recent work by Tudjarov leads to stronger self similarity for UDP traffic than TCP traffic [25]. Their work does not consider the effect that packet loss imposes in TCP congestion control mechanism. On the contrary, a transmission using TCP under a congested link is sensitive to packet loss. Even without any variability in terms of packet lengths or network delays, TCP itself can sometimes exhibit a behavior that produces traffic series that show properties similar to those of self-similar traffic. In [29] Guo and others present that a single TCP connection over a *lossy* link can produce a time-series that can be misidentified as self-similar. This phenomenon is called pseudo self-similarity [29].

Understanding the characteristics of the underlying processes that creates the fractal behavior of Web traffic is a key issue that gears the research toward understanding the processes that describe the lengths of traffic bursts. One way of generating Web traffic has been achieved using the Poisson distribution and the Pareto distribution in what has been called a Poisson Pareto Burst Process (PPBP) [35]. The PPBP has gained its appeal since it is formed in a way consistent with the typical behavior of Internet traffic. The PPPB has been used to model real traffic streams in recent work [36–38]. The Pareto distribution is a heavy tailed distribution. Heavy-tailed distributions assign relatively high probabilities to regions far from the mean. These distributions has an asymptotic shape as

a power-law with exponent α less than 2 [13]. A random variable that follows a heavy-tailed distribution can give rise to extremely large values with non-negligible probability:

$$P[X > x] \sim x^{-\alpha} \qquad 0 < \alpha \leq 2 \tag{2.1}$$

where $a(x) \sim b(x)$ means $lim_{x\to\infty} a(x)/b(x) = c$ for some constant c. These distributions have heavier tails than other common models, as the *exponential* and Weibull distribution [39]. A similar approach for generating traffic is using an $M/G/\infty$ queue to model Web traffic. This was suggested in a seminal article by Paxson and Floyd [17]. On this method, the arrival process follows a Poisson distribution and the service rate follows a power law distribution. Figure 2–1 presents this type of queue.

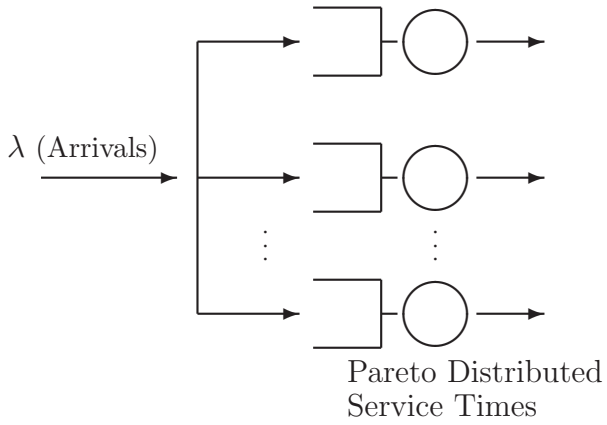


Figure 2–1: Poisson Pareto Process M/G/∞ Queue

2.1.4 Innovations in Characterizing File Sizes and Burst Lengths

It has been shown on [26] that the distribution of file sizes is directly related to the length of the bursts seen in Web traffic. These lengths are described using

either one or mixed probability distributions. While no standard distribution is exactly right, the heavy tail Pareto and the light tail *lognormal* distributions appear sensible in the tails [16]. The goodness of fit over Internet traffic of the *lognormal* distribution raises interesting questions about the widely accepted premise that only heavy tailed/power law distributions lead to long range dependence [16].

Currently, there is no definite consensus on which distribution follows the file and burst sizes [13, 14, 16, 39, 40]. During the past decade, file sizes and bursts lengths were modeled by heavy-tailed/power law distributions[17, 26, 40, 41]. In [14], Downey suggested that the *lognormal* distribution may be more appropriate than classic heavy tailed distributions such as the Pareto. Some new theoretical work revealed that these distributions are not inconsistent as was previously thought. This question of whether a distribution follows a Pareto or *lognormal* distribution has been studied in other fields such as Finance, Biology, Chemistry, Ecology, Astronomy, and Information Theory [39]. Power law and *lognormal* distributions are intrinsically connected. Using Pareto for approximate the burst lengths results in a poor fit for small data values. The *lognormal* yields a substantially better fit in the body of the distribution at the price of poorer fit on the upper tail [16]. The problem arises for fitting the bulk of the sample (small files), instead the emphasis is on the tail where the fit is *reasonable* acceptable. Large files, above a certain size, might be rare currently, and hence, both *lognormal* and power law distributions might capture the rare events adequately [39]. Some data analysis suggested that both the heavy tail Pareto and the light tail *lognormal* give reasonable fits in the upper tail by choosing the right parameters, although neither the Pareto nor the *lognormal* distribution is a perfect fit. Also, second moments, finiteness of variance, provide a poor way of understanding the

type of distributional properties that are important to Internet traffic and file size distribution [16].

A decade ago, Crovella and Bardford proposed the creation of a workload generator based on the characteristics inherent to Web traffic [15]. They applied a number of observations of Web Server (WS) usage to create a realistic Web workload generation tool that acts a set of real users accessing a server. The result was called the Scalable URL Reference Generator (SURGE). This tool generates references matching an *empirical* measurements of file size distribution, request size distributions among other characteristics of the workload. The distributions used during the development of SURGE agreed with the empirical measurements obtained from the files stored on the server. Crovella and Bardford criticized their work on [26] and they modeled the file system for SURGE with a distribution that is accurate not only for the tail but also for its body. A similar approach based on a *lognormal-Pareto* distribution for modeling web file sizes has been studied recently in [43]. Crovella and Bardford used censoring techniques to determine where to split between the *lognormal* distribution for the body and distribution for the tail.

2.1.5 Linear and other models

Deterministic models have been proposed for estimating the impact of offloading all or some parts of the protocol processing. In [12], Maccabe and Gilfeather, present the Extensible Message Oriented (EMO) model for protocol offload. They defined a conceptual model that captures the benefits of protocol offload in the context of high performance computing systems. The EMO is a linear extensible model that emphasizes communication in terms of messages rather than bursts

length. Their model allows protocol developers to consider the tradeoffs associated with offloading protocol processing. This includes the reduction in message latency along with benefits associated with the reduction in overhead. Also the EMO model allows the developer to observe the behavior of the model to predict throughput improvements [12]. The EMO is extensible since it can be used to model the behavior beyond transport protocol layers. Shiviam and Chase, [10] map this model into the LAWS model presented. The LAWS model was created to begin and quantify the debate over offloading the TCP and IP protocols. Also, the model attempts to characterize the benefits of transport layer offload. LAWS model reduces then to the classical LogP model. However, the LogP model was not designed for protocol offload. The probabilistic model presented on this document aims to achieve a higher level of abstraction and complexity than EMO and LAWS.

2.1.6 Summary

Performance modeling is an important part of the research area of Web servers. Without a correct model, it is difficult to give an accurate prediction of performance metrics. A Poisson process cannot describe the individual packet arrivals. However, a Poisson process can describe the initiation of sessions. M/G/1 queues has been used for performance modeling of Web servers and also for Web traffic generation (i.e. PPBP). Moreover, this model was the ideal candidate for developing the offload framework presented in this document. However, it is not clear which distribution is the best for modeling the length of the files that are going to be converted into traffic streams. The *exponential* distribution is a very poor approximation. The Weibull distribution gives much worse fit than either the Pareto or *lognormal* distributions [16]. As presented by Mitzenmacher in [39],

"just because one finds a compelling mechanism to explain a power law does not mean that there are no other, perhaps simpler, explanations". Given the close relationship between the two distributions, it is not clear that a definitive answer is possible. Mitzenmacher states that "for a more pragmatic view, it might be reasonable to use whichever distribution makes it easier to obtain results" [39].

2.2 Experimental Research

Most of the scientific research related to protocol offload found through the survey of literature is experimental. The experimental research steers in three different directions. The first is focused on processing the protocols outside the CPU into a special purpose processor. Most of the experiments focus in processing the protocols in an OE as part of the Network Interface Card (NIC). The second direction is geared on processing the protocols within the CPU but into another core or into a dedicated General Purpose Processor (GPP). The third approach is focused into performance of the protocol processing itself. The application of this technology is centered on two main fields: High Performance Computing (HPC) and Web server performance. The models are also tested either in an outboard processor near the NIC, another processor in a Symmetric Multi-Processor (SMP), or into another node.

2.2.1 State of the Art

In order to address the increasing bandwidth demands of modern networked computer systems, there has been significant interest in offloading TCP/IP processing from the operating system of the host [4]. TCP offloading can potentially reduce the number of host processor cycles spent on networking tasks, reduce the amount of local I/O interconnect traffic, and improve overall network throughput [4, 5, 46].

Today, network hardware manufacturers are commercializing Intelligent Network Interface Cards (iNIC)s that are capable of executing TCP/IP on a special purpose processor. These types of iNICs are referred as TCP Offload Engines (TOE). The term was first defined in a white paper published in 2002 by the 10 Gigabit Ethernet Alliance (10GEA)[1] . The purpose of their research is producing technology necessary to handle data flow at link speed of 10 Gbps and beyond[2] [5]. Network protocol processing can saturate servers when running high-speed TCP/IP based applications. High overhead for packet processing can leave few CPU cycles available for actual application processing. In [6], Brecht, Janakiraman, Lynn, Saletore, and Turner stated that while future TOE designs will likely show improvements, the processing elements in TOE are likely to always be behind the performance curve of mainstream processors. Furthermore, if processor performance is incremented by the current trend of multi-threaded multi-core CPUs, then fast processors of future TOEs might not be adequate for a single connection[5].

2.2.2 Offload Engine Technology Issues

Although offloading selected functions of a protocol has proved to be successful, so far there is limited use of them. This type of technology relies on firmware and specialized hardware implementations, which are more difficult to upgrade and customize than software based implementations. TOE devices have not yet

[1] Hewlett Packard, Intel Corp., Alacritech, Adaptec, and Qlogic Corp.

[2] 40 Gbps

demonstrated significant performance benefits [9, 10, 47] possibly because the current lack of NIC hardware resources that limits scalability [6]. Full offload is not new, however, it has never succeeded for complex general-purpose protocols such as TCP and IP, as vendors claim [5]. The resource limitations of a peripheral device limit the maximum processing capability and memory capacity of a TOE device [4].

In clusters of hundreds of thousands of nodes, resource management of communication will become more critical. One of the aspects of this scalability bottleneck is the amount of memory necessary to maintain communication. This problem is severe when offloading protocol processing onto different architectures is considered. One way to make TCP competitive with respect to latency and overhead for large clusters is to offload some protocol processing. However, protocol offload engines for TCP/IP are very expensive and limited [48].

Recent NIC designs provide some degree of offloading ability to execute *some* packet processing operations that previously had to be executed on the CPU. In particular, modern mainstream NICs can perform TCP/IP checksums and segmentation, and provide programmable timers used for interrupt moderation (example Intel PRO/1000). However, OS device drivers and protocol software must be designed or modified to take advantage of these capabilities and most of the time, they are not suited for that purpose [6].

Problems in current TCP offload NICs can be summarized in various aspects. The first one are bottlenecks at the TOE and/or the network interface that disrupt performance. The second issue identified is related to difficulties in designing efficient software interfaces between the operating system and the OE. The third problem is that modifying the existing network stack implementations is difficult

and sometimes unfeasible due to the inflexibility of the hardware. Consequently, accessing the internals of a TOE is cumbersome and sometimes impossible.

As multi-core processors provide mechanisms for fast inter-core communication, it becomes more attractive to dedicate some cores to protocol processing. This approach is complementary to proposals for placing the network interface hardware close to or even integrated onto the processor chip [3] and techniques for intelligently transferring data directly between network hardware and CPU caches [3, 49].

2.2.3 TCP and IP Processing Issues

There are five main issues found during the survey of literature that affect the performance of a host that processes TCP/IP [6, 23]. The first one is the low availability of processor time for application processing. This is a result concerning the second aspect which is hardware and software interrupt handling. In some operating systems, handling interruptions is a priority. Switching from interruption processing to protocol or application processing leads into the next problem: multiple context switches between the OS mode and user mode affect performance. The fourth issue is the overhead required for protocol processing. Finally, abusive use of memory allocation and memory copies results in poor memory management and degrades overall performance. The following paragraphs present seminal investigations on TCP/IP performance and innovative research on the issues of processing TCP and IP.

Most operating systems provide modules for processing the protocols but the way that these modules are handled by the operating system does not provide a suitable environment for the protocols. In the seminal paper by Clark, Romkey, and Salwen, they analyzed the cost of processing TCP and IP packets in the

context of the OS [50]. They showed that the first overhead is the operating system. This was also found by Kant in [11] and Ray and Pasquale in [51]. The functions for handling communication turn out to be very expensive. The other mayor overhead is protocol processing of small packets. Another source of overhead are the data-touching operations of large packets. The data-touching operations are all the operations that need to read all the data within a packet. These are the calculation and verification of the checksum, data movement (data copy), and memory allocation for storing each packet. Processing of large packets is dominated by the data-touching overheads while the smaller ones are dominated by protocol processing. It is important to consider the effects of non-data touching operations on performance, since most packets observed in real networks are small [16, 52]. The non-data touching overheads consume a majority of the total software processing time.

The TCP/IP protocol processing requires a significant amount of host system resources and competes with the application processing time. This problem becomes more critical under high loads. Also, when a packet finally reaches the socket, it may be dropped because of insufficient resources. However, packet drops occurs only after a considerable amount of system resources have already been spent on the protocol processing of the packet. A packet arrival results in an asynchronous interrupt, which preempts the current execution irrespective of whether the server has sufficient resources to process the newly arrived packet in its entirety [53].

Processing the TCP protocol outside the operating system have been suggested as early as 1988. In [9, 50, 51], the authors suggest that some of the operations needed to handle the packets could be outboard onto a special controller.

Efficient entities that can process TCP and IP outside the operating system in a more suitable environment are an attractive idea that could outperform the OS. This is true only if the entity plus the time that the OS uses to communicate with it is less than the time used for processing the modules that handle TCP and IP within the OS. However, in the survey of literature conducted we were unable to find articles that have been published in ways to abstract these events.

An optimized memory management mechanism is an important issue that has to be addressed for achieving a flawless transition from traditional protocol processing to protocol offloading. If memory is not manage adequately, TCP offload results in a "dumb idea" as Mogul has expressed explicitly in [9]. These suggestions come in accordance to the ones presented by Nahum, Tsipora, and Kandlur in [55]. In their article, they proposed a new function for transmitting files called *send_file()*. This function eliminates the copies from user space to kernel space whenever a file is transferred. Figure 2–2 presents how this function works. Traditionally, every time a file is to be transferred, the user reads the file and then writes its contents via a socket. The data travels from file to kernel space, and then it is copied from kernel to user space. This data is copied back again from user to kernel space, into a socket buffer and finally the data is copied from the socket buffer to the buffer of the network interface card. Lastly, the NIC transfers the data through the medium. Memory copies are reduced when *send_file()* is used. The data is not copied into user space as presented in Figure 2–2. However, the data is copied from kernel to socket buffers and sometimes is mapped directly from kernel buffers to be transferred then by the NIC.

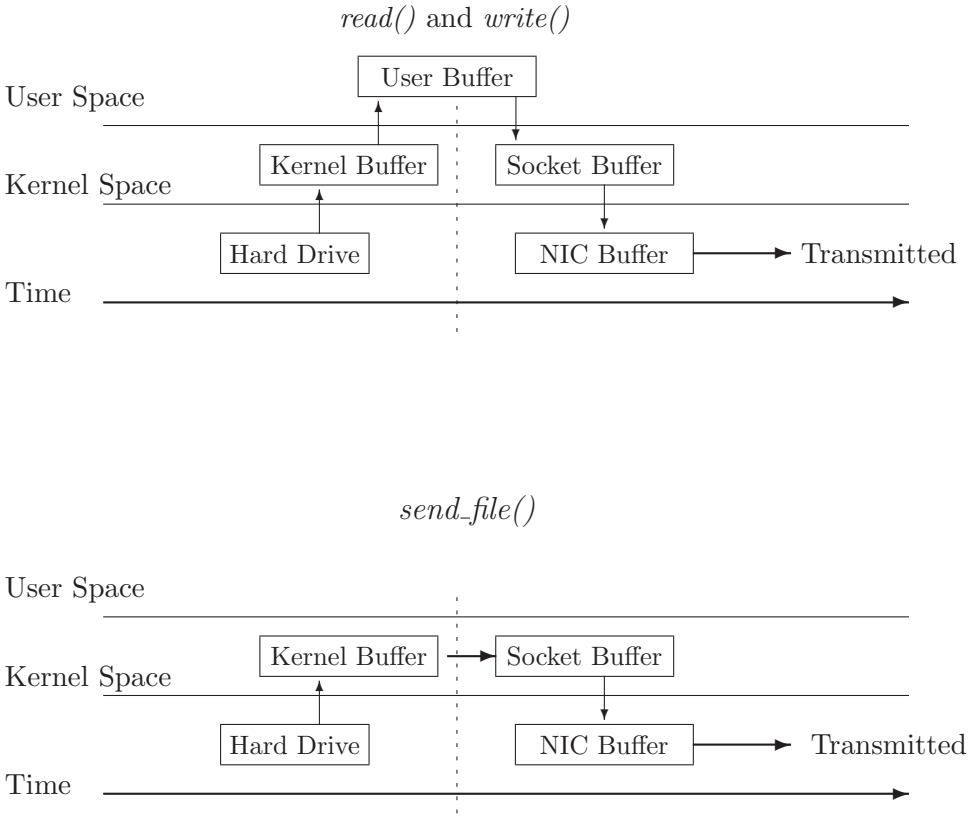


Figure 2–2: Function send_file() explained

If *send_file()* is tied to an integrated I/O system, which does not copy data, the function, provides a substantially better performance. This results in incrementing throughput by 51%. A similar notion was used in [6].

2.2.4 Novel Approaches for Processing TCP/IP on the NIC

Not all the research related to TCP/IP offloading attempt to outboard the processing into an entity outside main CPU. There exist investigations that are focused on optimizing TCP processing itself for reducing the resources that TCP consumes. The goal in [48] is to lessen the overhead of inactive connections by

deactivating the heavy-weight socket and replacing it with a place holder that allows the connection to be reactivated when needed. They use mini sockets[3] and open requests to efficiently handling the connections by deactivating and activating them. They were able to drastically reduce memory usage for open TCP connections, thus increasing the scalability of TCP, especially for offloaded TCP.

Other research focus on a mixture of hardware and software techniques that allowed proper processing of the protocols. In [1], Gilfeather and Maccabe present an alternative offload mechanism called Splintering TCP. The principle of splintering is not OS bypass rather the use of the OS efficiently. The splintering concept assumes that the application issues a *pre-posted* socket read before data has arrived. The OS captures this event by assigning a descriptor to the application buffer. Whenever, data arrives, the iNIC searches for the list of descriptors and copies the data directly into user space using Direct Memory Access (DMA), then makes the header available to the OS for further processing. It can bee seen that memory management is handled by the OS rather than by the iNIC. The drawback of this technique is that relies on interrupt coalescing for load balancing the OS and the iNIC. In [1] Maccabe and others mention that in standard environments, interrupt coalescing introduces a great deal of jitter in communication. This was also found by Prasad, Jain, and Dovrolis in [53]. Consequently, packets that arrive shortly after the iNIC has generated an interruption have longer latencies.

[3] also known as minisocks

2.2.5 Leading Approaches For Protocol Offload on Symmetric Multi Processors and Multi-cores

The higher degree of parallelism enabled by multi-core multi-thread processors is a good fit for concurrently processing multiple network streams [6]. This is true since the memory controller interfaces and network interfaces are integrated much closer to processor cores reducing memory and I/O overheads and bottlenecks. This is the direction proposed by Intel Corp. and Hewlett Packard Corp. in [6] and also has been proposed in [22, 56]. The authors have suggested that network packet processing efficiency can be improved by dedicating a subset of the server cores for network processing and by using asynchronous I/O for communication between the network processing and the application [6] (see Figure 2–3). Connection handoff has been proposed to allow the OS to selectively offload a subset of the established connections to the NIC [57, 58]. At any time, the OS can easily opt to reduce the number of connections on the NIC or not to use offload at all [4]. First, the NIC must ensure that TCP processing on the NIC does not degrade the performance of connections that are being handled by the host. Second, the OS must not overload the NIC since that would create a bottleneck in the system. The NIC can avoid overload conditions by dynamically adapting the number of connections to the current load indicated by the length of the receive packet.

Studying network protocol by emulation is the approach followed by the research conducted at Sun Microsystem by Westrelin, Fugier, Nordmark, Kunze, and Lemoine in [5]. On this work, a TOE was emulated by partitioning an SMP machine to evaluate different protocol offloading scenarios as presented in Figure 2–4. A full processor board run a stand-alone event piece of software emulating an OE. Their emulated TOE yields 600 to 900% improvement while still relying on

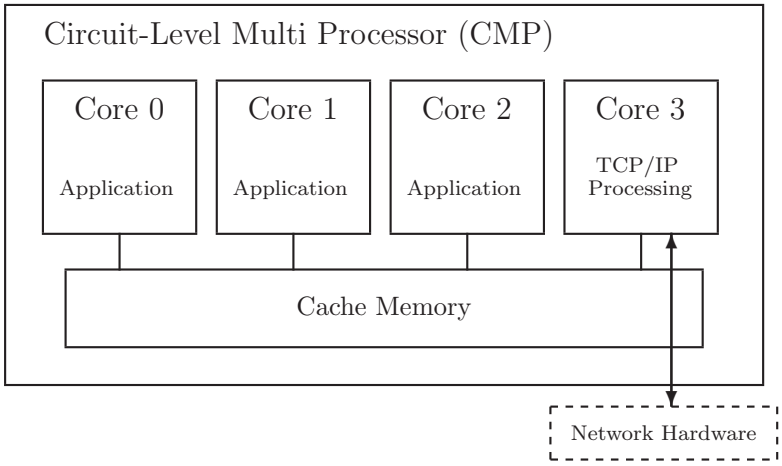


Figure 2–3: Processing the Protocols in Other Cores

memory copies at the operating system [5]. However, they only confronted their test bed with transactions that were favorable for their TOE approach [5]. They encountered that a poorly designed interface between the host and the OE is likely to ruin most of the performance benefit brought by protocol offload. However, this issue was not abstracted.

2.2.6 Quantitative Analysis Of Web Server Behavior

Studying the behavior of network applications, especially Web servers, raises interesting issues related to protocol offload. The time allotted to network processing from the execution time of a Web server was quantified by Banerjee in [23] when the server was overloaded with client requests[4] . They measured the time spent in every function during the execution path during send and receive, as well as the time spent processing interrupts. They concluded that the time spent

[4] The Web server application was Apache HTTP server

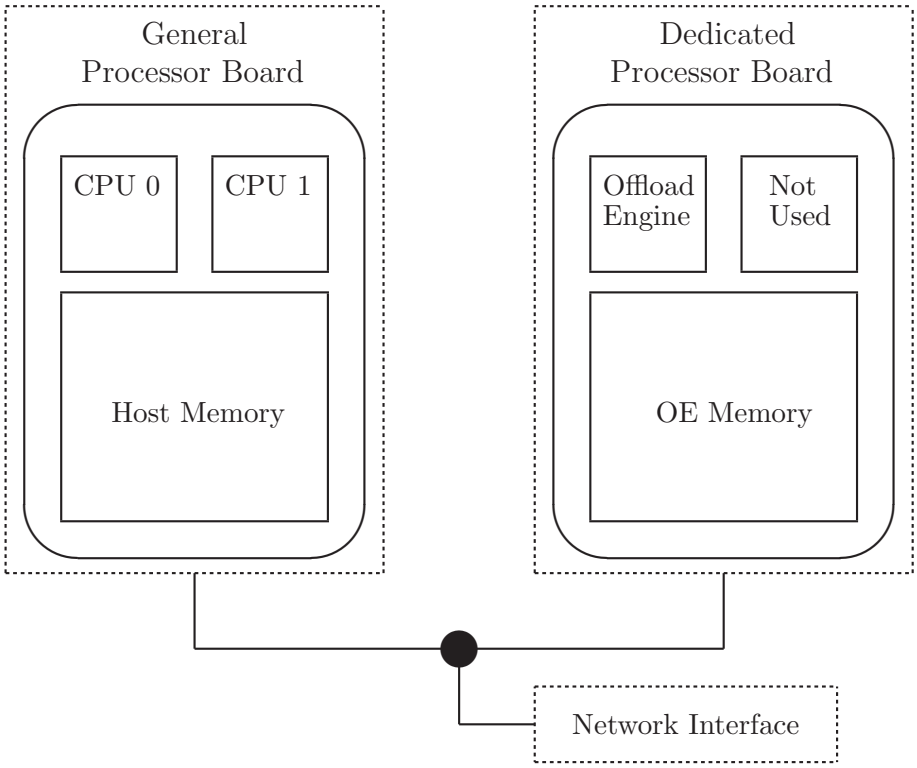


Figure 2–4: SMP Processor Boards and Emulated Offload Engine

in TCP/IP processing is significantly higher than the time spent on actual application processing. This is true for their original configuration. However, further studies have concluded that the Web server is stressed differently when different file sizes are requested. This is taken into consideration by our framework.

In [18], Hu, Nanda, and Yang presented the way the Web server interacts when confronted with different types of workloads. Webstone [59], was used for studying the behavior of the Web server with different file sizes. Webstone is a benchmark originally developed by Silicon Graphix and currently supported by Mindcraft Corporation. This benchmark was configured in such a way that it uses the files generated by SPECWEB [60] and not by the ones bundled with Webstone.

SPECWEB was developed by Standard Evaluation Performance Corporation and is the first standardized benchmark for measuring performance of Web Servers. The reason for using SPECWEB generated file system is that Webstone 2.5 is not bundled with a file set that could stress the Web server sufficiently as they intended to. In their work, files were classified based on their size. Table 2–1 presents these classes. Class 0 was used for files ranging from 0 to 1 Kilobyte. Class 1 was used for files with sizes ranging from more than 1 Kilobyte to 10 Kilobytes, and so on.

Classes	Range
0	0 - 1KB
1	1KB - 10KB
2	10KB - 100KB
3	100KB - 1 MB

Table 2–1: File Classes

The authors found that, on average, Apache spends about 20 to 25% of the total CPU time on user code, 35-50% on kernel system calls, and 25-40% on interrupt handling. Also, they found the TCP/IP stack and the network interrupt handling were the *major* performance bottlenecks for systems with reasonable RAM sizes [18]. Hu, Nanda, and Yang focused their work in the way they could improve the implementation of the Web server and did not concentrate in the way it processes the protocols. They proposed seven techniques to improve the performance of Apache. The most important suggestion was to implement a way for directly sending data from file system cache to the TCP/IP network [18]. This is similar to the *send_file()* function that was proposed by Nahum, Tsipora, and Kandlur in [55] and discussed previously.

In [61], He and Yang studied Web server performance beyond the use of static Web pages. They evaluated the Web server based on realistic workloads representing E-Commerce applications exemplified by a large amount of Common Gateway Interface (CGI), Active Server Pages (ASP), Servelet calls, and static Web pages. They found that most Web Servers can obtain higher throughput and less utilization, under a mixed workload, that under the static page workload. Static page requests are a little heavier to handle since they demand high disk I/O activities and memory copies. Therefore, studying the way static pages stress the server does not underestimate the impact on performance of a Web server, rather, in *some* cases, overestimate it. Our research considers the use of static Web Pages.

2.2.7 The Problem of Choosing the Right Workload

Choosing what is the representative workload for a specific network service had become an important issue with the evolution of the Internet and the introduction of new technologies that demand higher throughput. In [62], Floyd exposed that the Internet is an immense moving target. The way the Web is currently used is not the same way it will be used in the future. The data set collected within a trace represents only a snapshot at one point in the history of the evolution of the Web [45]. Analysis of the evolution of the Web and the way objects are requested raises issues about the way tests should be performed. Through 1995 to early 2000 SPECWEB and SURGE assigned access probabilities of around 80% for files less than 10KB. This means that most of the files requested by these benchmarks reside on Class 0 and 1 of Table 2–1 [15, 60]. Web sites had evolved to provide different type of applications that were not permissive a decade ago as commodity hardware provided users with higher throughput [63]. Animations used as part of content on some Web pages had an average file size that reside on Class 3 of

Table 2–1. These animations became common during early 2000. SPECWEB changed the access probabilities for each file class and introduced three new profiles in 2005 [60]. Different access probabilities were assigned to each class for each profile. It is evident that the demand for files that reside on Class 4 had increase with the advent of social networks and sites that provide, video, games and "heavy" content that was not permissive ten years ago [64, 65]. Therefore, the researcher has to consider these inherent characteristics of the evolution of the Web when performing tests and choosing workloads. Conclusions have to be analyzed carefully. In [18] the authors divided the file set in classes similar to the ones used by SPECWEB. They stressed the Web server by requesting each file class individually. This approach is also followed by the tests performed and presented in this document. Our aim is to surpass the historical constraints and provide new scenario that could be followed by other researchers in the future.

2.2.8 Summary

On the advent of multi-core multi-threaded processors and new technologies on outboard processors a new debate related two where protocol processing must be perform arises. Interactions between the host and the OE are independent of the underlying network technology. A poorly designed interface between the host and the OE is likely to ruin most of the performance benefit brought by protocol offload. The uniqueness of the research presented in this dissertation contributes into abstracting this characteristic for a proper protocol offload implementation. Our approach captures the overheads by abstracting them into the analytical model. In our case study, the model is validated versus a mixture between a real implementation at the Web server and the emulation of an OE in an external PC. The experimental setup includes a study focused on the behavior of the host CPU

when confronted to different files classes using an approach similar to [18]. The novelty of our research is based on a different perspective about the pros and cons of protocol offload.

CHAPTER 3
METHODOLOGY

The research methodology was conducted in two stages. The first stage was the development of an analytical model. The second stage was a case study to validate the model. This chapter is composed of three main sections. The first section discusses the methodology alternatives considered during this research. Also this section compares and contrasted our approach versus other methods of conducting research. Section 3.2 presents the analytical model used to estimate the performance metrics for protocol offload. Finally, the last section, presents the experimental setup used to validate the model.

3.1 Research Methodology

There exist different levels of abstraction and complexity when modeling real world scenarios. The traditional method of conducting research is usually made up of the following approaches: (1) analytical modeling, (2) simulation, or (3) implementation [66]. Another method that has recently surfaced is emulation-based approach [67–69]. Usually, a combination of these approaches is employed for increasing the robustness of the research. Combining methods is a way of dealing with the advantages and disadvantages each approach has (see Figure 3–1). This is true since each method has a different level of abstraction, complexity, development time, and cost. The methodology used during this research combines

analytical modeling and an emulation-based approach. The following paragraphs present the methodology followed to confirm our contributions and conduct our research in more detail.

The first stage of this research was developing an analytical model for abstracting our system. An analytical model is a representation of the essential aspects of an existing object, an improvement of the object, or an object that will be constructed [66]. This model can be either deterministic, or probabilistic. A probabilistic model is characterized by the fact that its future behavior is not predictable in a deterministic fashion [70]. A queuing model uses Probability theory and Markov chains to approximate a real system that uses queues and servers [66]. Our approach uses queuing theory to estimate the performance metrics before and after protocol processing extraction. The benefits of using queuing models are that there exist a number of useful steady state performance metrics that can be determined. The advantage of an analytical model is that is a generalization of the system. However, this generalization is also its disadvantage. (see Figure 3–1).

The second stage of our approach includes a high level of complexity using emulation-based mixed with a real implementation. Emulation-based analysis can serve as the bridging phase between the simulation and the real implementation. This analysis focuses on the reproduction of the external behavior of the device and/or system been implemented. Emulation does serve as an acceptable replacement for the system been modeled. This is in contrast to some other forms of computer simulation, which are a reproduction of the abstract mathematical model of the system been simulated. In simulations, peculiarities inherent to the problem could be concealed due to the abstractions used to create them [69]. Emulation enables conducting performance analysis of the system in the context of

real world applications. This approach has been used also recently in [67, 68] as an alternative to full physical test bed implementation. The complexity of emulation-based analysis is higher than the simulation as Figure 3–1 presents.

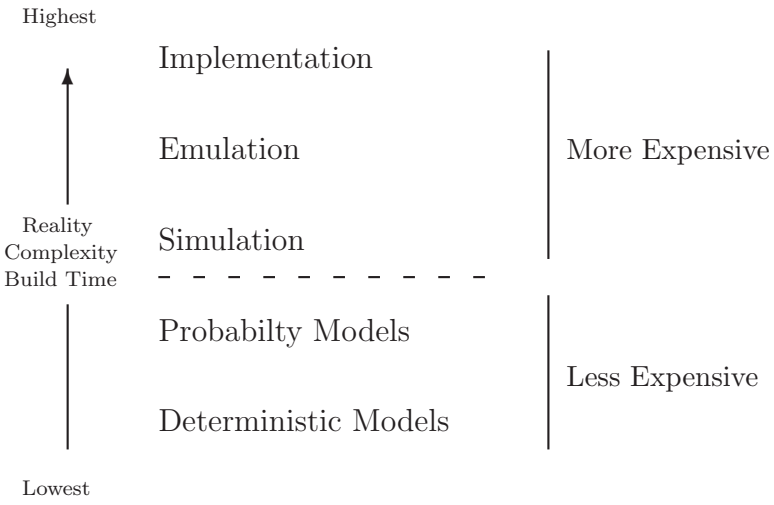


Figure 3–1: Levels of Complexity and Abstraction

The only true reliable evaluation of the system is the implementation [66]. The implementation of a system is the actual realization of that system as it was described. The main disadvantage is that not always the developer can afford to build the real system. Implementations are expensive and may impact the environment in which they have been applied. Since the implementation is complex is also time consuming and costly. The level of complexity is the highest as Figure 3–1 presents.

The research conducted uses a combination of the different levels of abstraction. First, a probabilistic model was devised. This model serves as a generalization of reality and captures the performance measures that are needed for

approximating, based on the assumptions, the outcomes of the real implementation. Moreover, a real implementation mixed with an emulator is used to validate the results obtained from the probabilistic model.

3.2 Probabilistic Model for Protocol Offload

This section presents the analytical model used for estimating the performance metrics of offload engines. The analytical model abstracts the most relevant features of implementing network processors. First, the performance metrics and the variables used to calculate them are presented. Subsequently, the CPU of the host is presented as a single server problem and modeled as in [30–32]. Then, the system is modeled as a network of queues consisting of one queue that abstracts the CPU of the system and another that models the offload engine. Finally, the model is extended to obtain the expected number of request within the system.

3.2.1 Performance Metrics

There are certain variables that need to be stated in order to understand the performance metrics of the system. Therefore, an introduction to these variables is needed for further understanding of the model. First, let X_a be a random variable that describes the time it takes to validate and to process the request. This is the time it takes to the application on top of the communication protocols to process the request. This process validates the request and initiates a session. Let X_p be a random variable that describes the time it takes for creating the packets that constitutes a burst that carries the requested object on the CPU of the host. Let X_q be a random variable that describes the time it takes for creating the packets that constitute a burst that carries the requested object on the offload engine. Then, let $E[X_a]$, $E[X_p]$, and $E[X_q]$ be the expected service time for X_a, X_p, and

X_q respectively. Throughout this document and for the sake of simplicity, the notations $E[X_a]$ and $E[X_p]$ are sometimes used as $1/\mu_a$ and $1/\mu_p$ interchangeably.

The most important performance metrics in this research is the utilization of the CPU of the host that is acting as a Web server and the delay of the whole system. The utilization is the percent of time the system is busy over an interval of length t. Examining only the utilization does not guarantee a performance increase even when the utilization of the CPU has been reduced. If the CPU idles could be for one of two reasons: Either the offload engine is handling the workload efficiently, or the offload engine has become the bottleneck of the whole system. Therefore, the delay on the system is a performance measure that must not be overlooked. An efficient offload engine reduces CPU utilization and maintains similar or lower response time to the clients in service. If the delay on the system is greater when the offload engine is added, then its inclusion **degrades** the whole system.

3.2.2 Abstraction of a System Without Offload Engine

A system that handles HTTP requests and does not posses a protocol offload engine is modeled with a single queue. The CPU of this server is represented with an M/G/1 queue as presented in Figure 3–2. The arrival of requests is modeled with a Poisson distribution as in [25, 27–32] and suggested by [17]. The service time for a request is a combination of the time spent processing the request and the time the protocols used to convert the requested object[1] into packets of protocol P. An exponential process describes only the interpretation and validation of the request.

[1] A file

The random variable that represents the service time of converting the requested object into a stream of data[2] does **not** follow an exponential distribution. The time it takes to create these packets depends on the length of the requested file. A strong relationship between the length of the file and the bursts length exists as presented in [14, 29]. This is true since the requested object has to be fitted into n packets of protocol P in order to be transferred back to the sender. Therefore, this time follows a *lognormal* or a Pareto distribution.

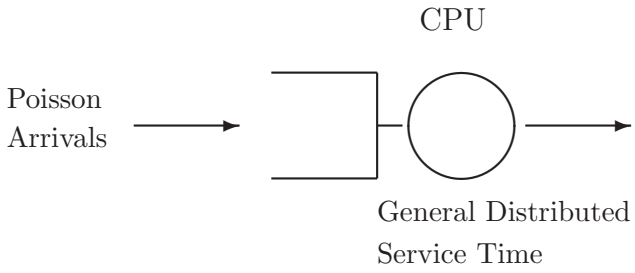


Figure 3–2: M/G/1 Queue, An Abstraction of a Non-Supported Host

The utilization for a host that has a CPU without the support of an offload engine is calculated with the following equation:

$$U_{cpu} = \lambda \left(\frac{1}{\mu_a} + \frac{1}{\mu_p} \right). \tag{3.1}$$

The distributions that describe the expected service time are needed to calculate the following performance measure. It is assumed that the mean service time for handling the communication between the offload engine and the CPU (X_o and

[2] a data stream an bursts are used interchangeably during this document

X_c) follows an exponential distribution. However, X_p and X_q are described either by a *lognormal* or by a *Pareto* distribution.

The first and second moment of the random variables are needed to obtain the delay of the CPU using the Pollaczek-Khinchin (P-K) formulas. The P-K formula require the first and second moments of the distributions that describe the service time (see Appendix B). The first moment is defined as the mean of the distribution and this is $E[X_a + X_p] = E[X_a] + E[X_p]$. However, the second moment of this random variable is calculated differently. The following paragraphs describe the approach used to obtain the second moment.

A host *without* an offload engine has a first moment $E[X_a + X_p]$. The second moment is then: $E[(X_a + X_p)^2] = E[(X_a^2 + X_p^2 + 2X_aX_p)]$. As a result:

$$E[(X_a + X_p)^2] = E[X_a^2] + E[X_p^2] + 2E[X_aX_p] \tag{3.2}$$

Since X_a and X_p are independent random variables, then:

$$E[(X_a + X_p)^2] = E[X_a^2] + E[X_p^2] + 2E[X_a]E[X_p] \tag{3.3}$$

Now, if X_p is described by a *lognormal* distribution. Substituting, results in:

$$E[(X_a + X_p)^2] = \frac{2}{\mu_a^2} + e^{2(\nu+\sigma^2)} + \frac{2e^{\nu+\sigma^2/2}}{\mu_a} \tag{3.4}$$

Then, simplifying:

$$E[(X_a + X_p)^2] = \frac{2}{\mu_a}\left(\frac{1}{\mu_a} + e^{\nu+\sigma^2/2}\right) + e^{2(\nu+\sigma^2)} \tag{3.5}$$

The utilization is needed to use the P-K formula for the M/G/1 queue (see Appendix B). Let U_{noe} represent the utilization of a system with no offload engine

and $U_{noe} = \lambda(1/\mu_a + e^{\nu+\sigma^2/2})$. Let W_{noe} be the average waiting time (delay) a requests lingers *before* entering service. Substituting in the P-K formula:

$$W_{noe} = \lambda \left[\frac{2\mu_a^{-1}(\mu_a^{-1} + e^{\nu+\sigma^2/2}) + e^{2(\nu+\sigma^2)}}{2(1-U_{noe})} \right] \tag{3.6}$$

Moreover, the expected waiting time is simplified as follows:

$$W_{noe} = \frac{2\mu_a^{-1}U_{noe} + \lambda(e^{2(\nu+\sigma^2)})}{2(1-U_{noe})} \tag{3.7}$$

Then:

$$W_{noe} = \frac{2U_{noe}}{2\mu_a(1-U_{noe})} + \frac{\lambda e^{2(\nu+\sigma^2)}}{2(1-U_{noe})} \tag{3.8}$$

Simplifying:

$$W_{noe} = \frac{1}{(1-U_{noe})} \left(\frac{U_{noe}}{\mu_a} + \frac{\lambda E[X_p^2]}{2} \right) \tag{3.9}$$

Notice that a general formula has been found that can be used either if X_p follows a *lognormal* or follows a Pareto distribution. Let T_{noe} be the expected time the customer lingers inside the CPU with no offload engine support. Then:

$$T_{noe} = \frac{1}{(1-U_{noe})} \left(\frac{U_{noe}}{\mu_a} + \frac{\lambda E[X_p^2]}{2} \right) + E[X_a] + E[X_p] \tag{3.10}$$

3.2.3 Abstraction of a System With The Support of an Offload Engine

An offloaded CPU has an entity whose main function is converting the requested object into packets of protocol P to be transferred through the network. Therefore, two queues interact for modeling our host. Figure 3–3 present the network of queues that represent our CPU with an attached offload engine.

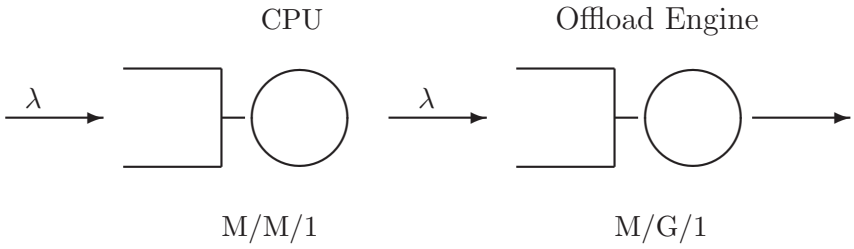


Figure 3–3: Network of Queues for Modeling a Supported Host

The service provided by the first queue is interpreting, validating and processing the request. This queue is abstracted as a Markovian model of an M/M/1 queue that represents the CPU. The task of the second queue is transferring the requested object by creating a series of packets to be transmitted in a data stream. The last queue represents the offload engine and is an M/G/1 queue.

The utilization of the system in a network of queues is calculated as the sum of the utilization of both service centers. However, we are only interested in the calculation of the utilization of the CPU. The utilization of the CPU is $U'_{cpu} = \lambda/\mu_a$. After protocol processing extraction, let X_q be the random variable of the service time for processing the protocols at the offload engine. Then $E[X_p]$ is not necessarily equal to $E[X_q]$. The expected time the offload engine uses to prepare packets of the object requested could be similar, lower, or larger depending on the capabilities of the offload engine. Now, the utilization of the offload engine is $U_{oe} = \lambda E[X_q] = \lambda/\mu_q$. The utilization for creating the packets has been shifted to the offload engine. Consequently, the utilization of the *system* in a host that has the support of an offload engine is:

$$U = U'_{cpu} + U_{oe} = \lambda \left(\frac{1}{\mu_a} + \frac{1}{\mu_q} \right). \tag{3.11}$$

Shifting the creation of the packets to the offload engine can introduce additional overhead. This aspect cannot be overlooked unless this overhead is negligible. Let $E[X_o]$ be the expected service time of processing the overhead added to the CPU that represents the actions taken for passing control messages from the CPU to the offload engine. Consequently, the utilization of the CPU of a host with the support of an offload engine is:

$$U'_{cpu} = \lambda \left(\frac{1}{\mu_a} + \frac{1}{\mu_o} \right). \tag{3.12}$$

Another overhead has been also added to the offload engine. Let X_c be the random variable that represents the time it takes on the offload engine to process the control information for communicating with the CPU of the host. Then, $E[X_c]$ is the expected service time for the overhead added to the offload engine for processing the control data sent by the CPU[3] . Then, the utilization of the offload engine is:

$$U_{oe} = \lambda(E[X_c] + E[X_q]). \tag{3.13}$$

The utilization of the CPU after protocol processing extraction must be lower than before. If $E[X_o]$ is greater than $E[X_p]$ when processing file F, then, the

[3] if $E[X_c]$ is substantially low, the expectation of X_c could be negligible

inclusion of the offload engine **degrades** the whole system. These aspects of offloading a protocol are *elusive* as it was stated by Shivam and Chase in [10].

The expected waiting time of both queues are needed to obtain the delay of the system. Appendix B presents the performance measure for calculating the delay in an M/M/1 queue. Let T_{cpu} be the time spent inside the queue that represents the CPU[4] . Let $U_{cpu} = \lambda(E[X_a] + E[X_o])$. Then, the delay of the first queue from the notions of the M/M/1 queue:

$$T_{cpu} = \left(\frac{1}{\lambda}\right) \frac{U_{cpu}}{1 - U_{cpu}} \tag{3.14}$$

Equation 3.14 represents the expected time the request spends in the first queue. The expected waiting time a request waits before been serviced by the CPU is obtained by subtracting the mean service time form the previous equation. Let W_{cpu} be this expected waiting time then:

$$W_{cpu} = T_{cpu} - E[X_o] - E[X_a]. \tag{3.15}$$

Let $Y = (E[X_o] + E[X_a])$ for a moment. Then: $W_{cpu} = T_{cpu} - Y$. Consequently:

$$W_{cpu} = \left(\frac{1}{\lambda}\right) \frac{\lambda Y}{1 - \lambda Y} - Y, \tag{3.16}$$

Then:

[4] This is also called the delay inside the CPU

$$W_{cpu} = \frac{Y}{1 - \lambda Y} - Y, \tag{3.17}$$

Simplifying:

$$W_{cpu} = Y \left(\frac{1}{1 - \lambda Y} - 1 \right), \tag{3.18}$$

Substituting:

$$W_{cpu} = (E[X_o] + E[X_a]) \left(\frac{1}{1 - U_{cpu}} - 1 \right), \tag{3.19}$$

Obtaining the delay on the second queue is quite differently. The second moment of $X_c + X_q$ is needed since the offload engine is modeled by an M/G/1 queue. Previously was stated that $E[X_o]$ is exponentially distributed. Let X_c be similar to X_o. Then, the time it takes to serve the requested object at the offload engine, is $E[(X_c + X_q)] = E[X_c] + E[X_q]$. Then the first moment is $E[(X_c + X_q)] = 1/\mu_c + e^{(\nu+\sigma^2/2)}$ when X_q follows a *lognormal* distribution. The second moment is similar to the one presented in 3.2. Then:

$$E[(X_c + X_q)^2] = \frac{2}{\mu_c} \left(\frac{1}{\mu_c} + e^{\nu+\sigma^2/2} \right) + e^{2(\nu+\sigma^2)} \tag{3.20}$$

The time a customer waits before entering service at the offload engine is given by the P-K formula. The second moment is plugged into the formula to obtain the following:

$$W_{oe} = \lambda \left[\frac{2\mu_c^{-1}(\mu_c^{-1} + e^{\nu+\sigma^2/2}) + e^{2(\nu+\sigma^2)}}{2(1 - U_{oe})} \right] \tag{3.21}$$

This equation is similar to the equation 3.7, therefore, substituting:

$$W_{oe} = \frac{1}{(1 - U_{oe})} \left(\frac{U_{oe}}{\mu_c} + \frac{\lambda E[X_q^2]}{2} \right) \tag{3.22}$$

It is important to state here that the utilization of the offload engine is calculated using $U_{oe} = \lambda(\mu_c^{-1} + e^{\nu+\sigma^2/2})$. The expected total time in the offload engine (T_{oe}) is calculated by adding the mean service time in the offload engine ($E[X_e]$), therefore:

$$T_{oe} = W_{oe} + E[X_c] + E[X_q] \tag{3.23}$$

Then:

$$T_{oe} = \frac{1}{(1 - U_{oe})} \left(\frac{U_{oe}}{\mu_c} + \frac{\lambda E[X_q^2]}{2} \right) + E[X_c] + E[X_q] \tag{3.24}$$

Let T be the delay of the system which includes the CPU and the OE. Since the system is a network of queues, the delay of the system is obtained by identifying the maximum delay of all the queues composing the network. Therefore:

$$T = \max(T_{oe}, T_{cpu}) \tag{3.25}$$

In our previous equation, the random variable that describes the time it takes to create a burst follows a *lognormal* distribution. The Pareto distribution is another candidate for describing the service time for creating a data stream of protocol P from file F. When X_q follows a Pareto distribution then:

$$E[X_c + X_q] = E[X_c] + \frac{\alpha k}{\alpha - 1} \tag{3.26}$$

The second moment of $E[X_c + X_q]$ is obtained by substituting in equation 3.2:

$$E[(X_c + X_q)^2] = \frac{2}{\mu_c}\left(\frac{1}{\mu_c} + \frac{\alpha k}{\alpha - 1}\right) + \frac{\alpha k^2}{\alpha - 2} \tag{3.27}$$

Now plugging the second moment on the P-K formula, the time a customer waits before acquiring service is:

$$W_{oe} = \lambda\left(\frac{2\mu_c^{-1}[\mu_c^{-1} + \alpha k(\alpha - 1)^{-1}] + \alpha k^2(\alpha - 2)^{-1}}{2(1 - U_{oe})}\right) \tag{3.28}$$

Then W_{oe} reduces to Equation 3.22, this is:

$$W_{oe} = \frac{1}{(1 - U_{oe})}\left(\frac{U_{oe}}{\mu_c} + \frac{\lambda E[X_q^2]}{2}\right)$$

It was mentioned previously that the expected total time in the offload engine is given by adding the mean service time in the offload engine, therefore:

$$T_{oe} = W_{oe} + E[X_e] \tag{3.29}$$

Then:

$$T_{oe} = \frac{1}{(1 - U_{oe})}\left(\frac{U_{oe}}{\mu_c} + \frac{\lambda E[X_q^2]}{2}\right) + \frac{\alpha k}{\alpha - 1} + E[X_c] \tag{3.30}$$

As stated previously, the delay in the whole system is then $T = \max(T_{oe}, T_{cpu})$.

3.2.4 Expected Number of Requests

The model can be used for determining the expected number of customers waiting to be serviced. Let N_q be the expected number of requests waiting to be serviced in the queue. Then using Little's Law:

$$N_q = \lambda(W_{oe}) = \frac{\lambda}{(1 - U_{oe})} \left(\frac{U_{oe}}{\mu_c} + \frac{\lambda E[X_q^2]}{2} \right) \tag{3.31}$$

The percent of time that the offload engine uses for processing the overhead is $U_c = \lambda/\mu_c$. Therefore, the previous equation can be simplified to:

$$N_q = \frac{1}{(1 - U_{oe})} \left(U_c U_{oe} + \frac{\lambda^2 E[X_q^2]}{2} \right) \tag{3.32}$$

Let N_{oe} be the expected total number of request in the offload engine (waiting and in service). Then using Little's Law:

$$N_{oe} = \lambda(T_{oe}) = \frac{1}{(1 - U_{oe})} \left(U_c U_{oe} + \frac{\lambda^2 E[X_q^2]}{2} \right) + U_c + U_q \tag{3.33}$$

Notice that in N_{oe} the utilization for handling the data to be transferred and converted it into packets is given by: $U_q = \lambda(E[X_q])$.

3.3 Experimental Setup

In order to validate our model a test bed was setup. The test bed is composed of two PCs that act as a single host. One of these machines acts as the Web Server and the other as the TOE device (see Figure 3–4). The PC that acts as the TOE device, is called the front-end PC. These two PCs are connected together via a dedicated line. The Web server is connected to the Internet via the front-end PC. The front-end PC emulates the TCP Offload Engine using an emulator. This emulator was called the TCP Offload Engine Emulator (*TOE-Em*) and is discussed in detail in Section 3.3.1. Two more nodes were used to generate the workload and the requests sent to the test bed. This is discussed in Section 3.3.7. The software and hardware used in the test bed is discussed in the following sections.

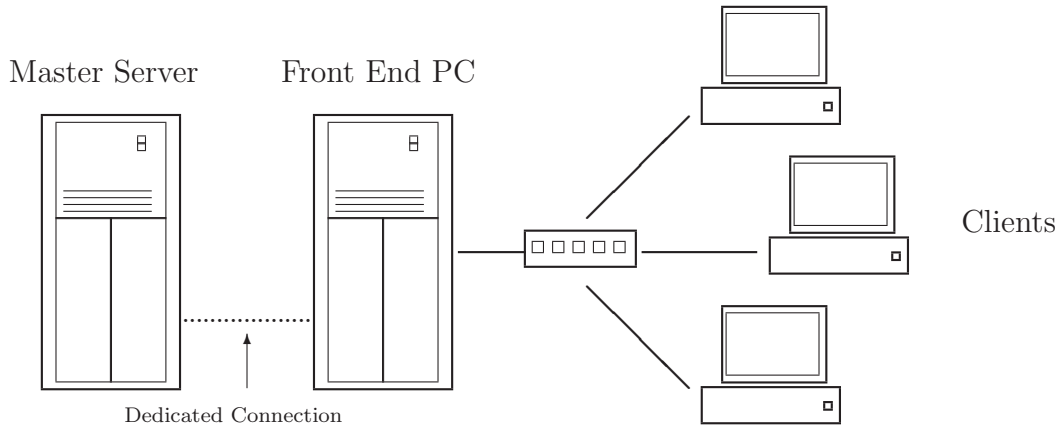


Figure 3–4: Experimental Setup Used to Test the System

3.3.1 The TCP Offload Engine Emulator

The TCP Offload Engine Emulator (*TOE-Em*) is a **multi-process** program written in **ANSI C** that has all the functionality of a TCP Offload Engine. The *TOE-Em* resides on the front-end PC and provides the socket interface layer to the Web server. This program is the gate in which packets come in and go out of the whole system. The following paragraphs describe the *TOE-Em* in detail.

The *TOE-Em* is a program that spawns processes and maps them to similar processes running at the Web Server by using dedicated processes that interact with each other. The *TOE-Em* is composed of three types of processes: The Commander and Reader Process (CR), the Forker Process (FP), and the Slave Processes (SP). All of them interact by UNIX System V Inter-Process Communication (IPC), share memory, semaphores and signals [71–73]. The Commander and Reader is the most important process of the *TOE-Em*. It is the one that reads all the commands sent by the Web server. Also, the CR is the one who signals any of its slave processes for processing the command. The slave process

is the process that executes the command assigned by the commander and reader. The FP spawns new slave processes and allocates their resources. The following paragraphs discuss the main functionality of each one of the processes presented.

The Commander and Reader Process is the most important process of the *TOE-Em*. Upon start up, the Commander and Reader Process enters into a critical loop that only end when an error is found or when the OS removes it. If there is a command waiting to be read, the CR reads and stores the command. If not, the process halts. This is a case that has been presented in the flowchart of Figure 3–5. If the read is successful then three conditions are tested. The first condition is to examine the command and if it is a command that is not destined to a particular SP, then, the Commander and Reader Process looks for an available slave that could handle it. Second, if the command is destined to a specific SP, then, the CR checks if the SP is available to handle the command. If not, the CR waits until the SP finishes. The SP receives this signal from the CR, whenever the SP is idling or blocked (see Figure 3–5). Finally, if the command is intended to the FP, then the CR checks if the FP is available and not in used by another command. Then, the CR signals the FP. If any of these conditions is **not** met the command is unknown and an error occurs. Notice that if an error occurred the *TOE-Em* stops.

The function of the slave process is executing the commands that have been chosen for it by the CR. Immediately, after a slave is spawned, the slave process enters into a critical loop. Figure 3–6 presents the flowchart that describes the functionality of the SP. Each and every SP, including the FP, checks if there is a notification from the CR indicating that a command is awaiting to be processed. If there is a command destined to it, the SP identifies the command and validates

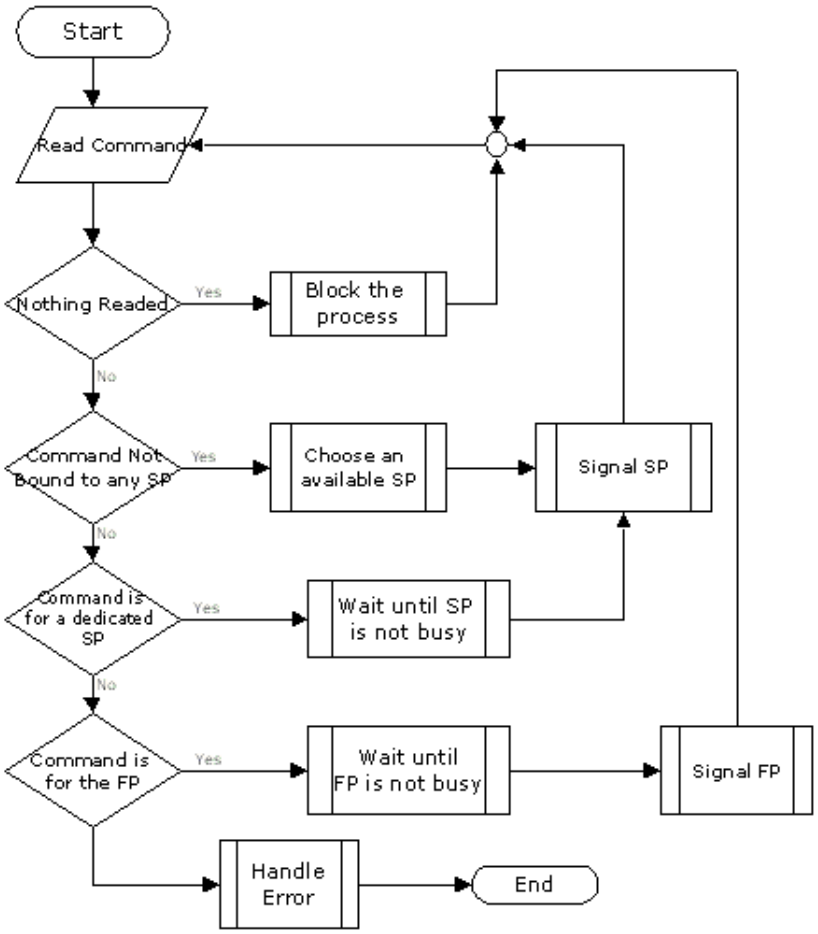


Figure 3–5: Commander Reader Flowchart

it. If the command is invalid, the process ends. If the command is valid, the SP issues the command. After issuing the command the SP releases its resources and loops back to the beginning (see Figure 3–6). The SP blocks if there is nothing to do.

The Forker Process is another SP. The only purpose of the Forker Process is to spawn more slaves and allocate resources for them. Also, the FP could remove any slaves and free the resources used by it. The FP is created when the first *fork*

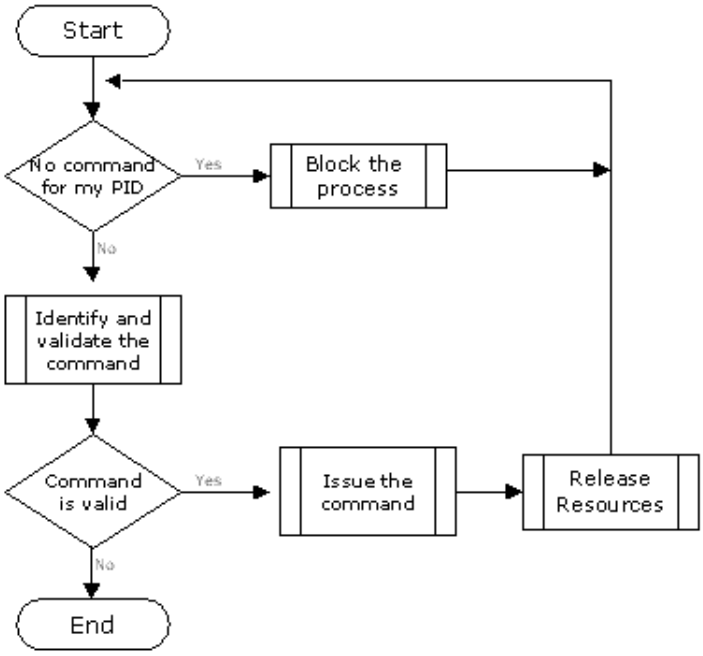


Figure 3–6: Slave Process Flowchart

command is received by the *TOE-Em*. In [74] the source code of the *TOE-Em* is available. Please refer to [74] for detail information.

3.3.2 TOE-Em Memory Management

During the design and implementation phase of the *TOE-Em* memory management was a key issue that was not overlooked. The design had as a top priority avoiding heavy memory allocations and copies. Stacks were used to reduce memory management. Figure 3–7 presents the memory map of the *TOE-Em*. The *TOE-Em* initializes a share memory area of a size that is around 370 KB upon startup. This size is the result of multiplying the maximum number of processes that the *TOE-Em* can handle (250) by the size of the Ethernet Frame (1514 octets). This is done since the maximum frame size, in our implementation is of 1514 octets and every command that the Web server generates travels within a

frame. Consequently, the 370 KB area can be partitioned in 250 slots of 1514 bytes each. This makes addressing an easy task that can be carried by offsetting from the address of the first byte of our memory area. This simple calculation is faster than searching over the entire memory area for a slot. The stack is used to store all the addresses that are free. Therefore, every time a command arrives from the Web server the *TOE-Em* pops an address from this stack, and uses this address to store the incoming command. After identifying the target SP, the CR, assigns the slot to an SP and then signals it. This slot is not pushed into the stack again until the SP releases its resources (recall Figure 3–6). When this happens, the SP pushes the address of the slot it was using onto the stack. Therefore, this slot becomes a free slot. Then the CR, without the need to allocate more memory, could reuse this buffer whenever it pops a free slot from the stack. Notice that race conditions could surface by using this design, however, semaphores were used to avoid this type of errors.

Memory management was a key issue to optimize our implementation. Any other implementation of a similar engine has to consider the way the memory is handled. One thing that has to be avoided at all costs is copying and dynamic memory allocation and de-allocation on user space.

3.3.3 Process Mapping

Every process in the *TOE-Em* needs to be mapped into a process in the Web server. This is used since the Web Server application is a multi-process program. Therefore, the task of the Commander and Reader Process is mapping every command to an appropriate Slave Process that could handle the command efficiently. The CR process does not designate a command to an SP at random. It uses a criterion for that purpose.

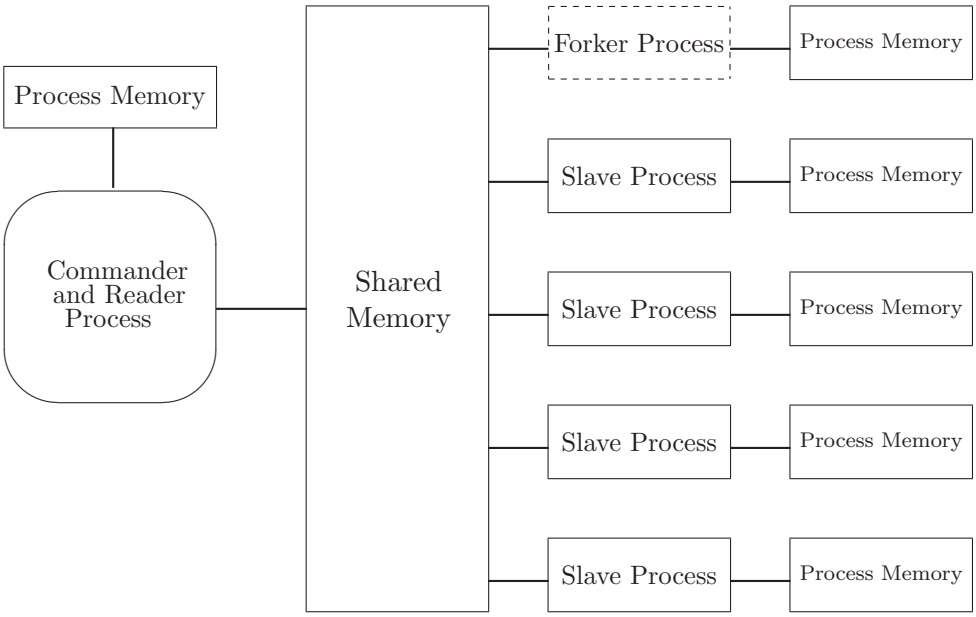


Figure 3–7: TCP Offload Engine Emulator Memory Map

A socket descriptor is an identifier returned by the OS used to reference an instance that is associated to a socket. The descriptors created by each SP are **only available** for that particular SP. This happens since the kernel at the front-end PC assigns the socket descriptor to a specific process identifier (PID). Also, all the state variables used to handle the TCP module are in a memory area assigned by the OS kernel for that specific PID. However, there was an issue. Two different PID could assign the same socket descriptor number, but reside in two different contexts. Since two processes can have the same socket descriptor a way of identifying to which process the socket descriptor number belongs was devised. This socket descriptor is an integer value and the Web server is expecting that from the *TOE-Em*. Therefore, a new descriptor was created for the Web server that is a combination of the PID and the socket descriptor. This was done by a

function that maps the socket descriptor within the right context. This function was called the build key function and is presented in Figure 3–8. The function prepares the new descriptor by multiplying the actual socket descriptor number by 10^6 and then adds the PID to it. This works since the maximum size for a PID in LINUX is six digits. This new socket descriptor is given to the Web server. The benefit of using this type of socket descriptor is that the appropriate PID that has this socket is known by the *TOE-Em* and also by the Web server. There is also a function that converts this socket descriptor to the PID and socket descriptor number pair. This function is called the break key function and is illustrated in Figure 3–8.

```
#define PID_KEY_ 1000000

int build_key(u_int mypid, int skt) {
    return (mypid + skt * PID_KEY_);
}

void break_key(int k, u_int *npid, int *rskt) {
     (*rskt) = (int) k / PID_KEY_;
     (*npid) = (u_int) k % PID_KEY_;
}
```

Figure 3–8: Source Code to Build and Break the Key

3.3.4 Process Coordination

In order for the CR and SP to coexists and work together they need to communicate with each other. This is done via a table that resides in shared memory. Both the SP and the CR consult this table. This table is called the process control table (PCT). The data structure for the PCT is depicted in Figure 3–9. This data structure is used as the data type of an array. The CR uses the PCT for consulting if the SP is available or busy. Also, the CR uses the PCT for

```
struct mini_control {
     u_int pid;        /* Process Pid */
     int data_entry;   /* Slot on share memory area */
     int rtable_entry; /* Entry on socket table */
     char state;       /* Process state */
     int sons;        /*  If I am an accept parent how many sons I have */
     int available;   /* is available for reuse */
     int blocked;     /* is blocked? */
};
```

Figure 3–9: Process Control Table Data Structure

indicating the address of the next command that the SP is going to execute. The SP consults the table to know if there is a next command to execute. This is done by examining the *data_entry* field of the structure. If this field is empty there is nothing to do and the process blocks. If there is a slot to handle, the SP begins processing right away. The process remains blocked until a signal is received from the CR.

One advantage that this approach has is that the SP always points to its appropriate control slot. Upon creation, a slot to this table is assigned to the PID of the SP. The address to this slot references the record in the PCT for the PID of the SP. The SP uses this address to consult the PCT.

3.3.5 Raw Sockets and Protocols

Raw Sockets were used to bypass the IP and TCP layers of the Web Server. UNIX raw sockets let the developer bypass any protocol modules used by the OS and handle any frame directly into the data link layer. Sometimes, this type of socket is referred as a packet socket. The use of raw sockets provides the developer with the capability of creating a frame directly from user space. In order to send commands to and from the TOE a special control protocol was used.

The protocol was called the raw protocol and is presented in Figure 3–10. This emulates the communication between the CPU and the TOE. The raw protocol header is composed of four main fields. The first one is the command field, which is an identifier that labels the command to be executed at the destination, either the Web server or the TOE. The second field is the optional parameters length. This is zero when no parameters are sent. The third field is the data length and is used on some commands that send data directly via the payload of the frame. The fourth field is the destination PID. This is used to indicate which PID is going to handle the command. Not all the commands use this field. The last field is optional and is used to send additional data. The length of this field varies and is indicated by the second field.

The raw protocol header consumes at most 16 octets from each frame. This maximizes the payload size of the frame by reducing overhead. However, the header allows for additional parameters to be sent. These are referenced at the end of the protocol header. The length used by the additional parameters depends directly on the command used. The commands that consume the largest parameter space are the *getaddrinfo()* and *getnameinfo()*. However, these commands are only issued once per connection.

Every POSIX socket command is *mapped* into the *TOE-Em*. Therefore, every time a socket function is invoked at the server, a frame is created containing the raw protocol header. The *TOE-Em* reads this frame, validates the raw protocol header and executes the command. If the *TOE-Em* needs to send results back to the Web server another frame containing a raw protocol header is created.

There are three type of commands implemented on the whole test bed. The first commands are the ones used to provide the Web server with the proper socket

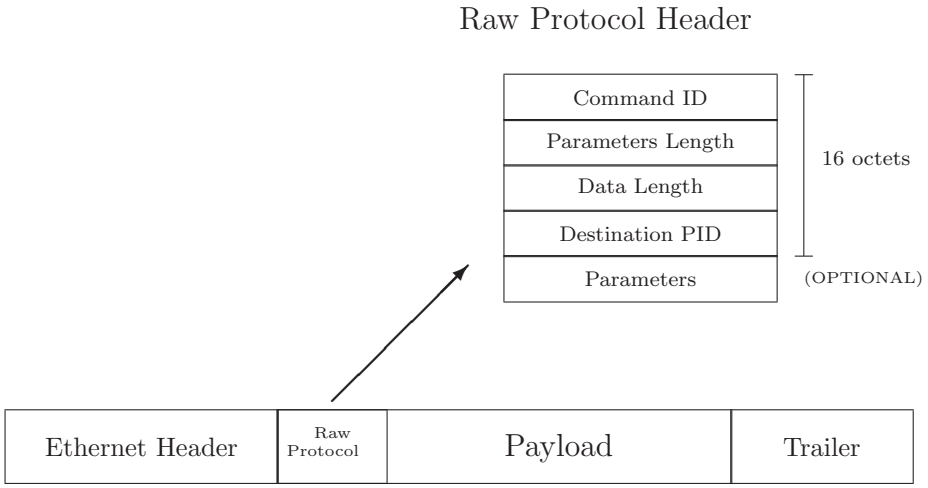


Figure 3–10: Raw Protocol

interface layer. These commands are presented in Table 3–1. The second type are fourteen additional commands used to return the results obtained at the *TOE-Em*. These commands have been supported in the Web server application. The method used for handling these commands is presented in Section 3.3.6. The third type of commands are specific control commands for fine tunning the *TOE-Em*. These commands are presented in Table 3–2.

The *TOE-Em* provides a way of emulating the way whole files are sent through a stream socket using *sendfile()*. It is important to notice that whenever files are going to be sent via *sendfile()* the file resides on both PCs. The file is open and processed normally as the Web server application does it. However, to emulate the *sendfile()* system call, all the files required for our test are replicated on the front-end PC. This emulates a TOE with the capabilities of using a zero copy mechanism [9, 55].

Command	Comment
RAW_CREATE	Initialize a connection
RAW_BIND	POSIX bind()
RAW_LISTEN	Listen to a TCP port
RAW_CONNECT	Establish a connection via TCP
RAW_ACCEPT	Passively listen to a TCP port and blocks
RAW_OPTIONS	Used to set options on a TCP socket
RAW_GETSOCKNAME	Use to store the bound name of the specific socket
RAW_GETPEERNAME	POSIX getpeername()
RAW_GETADDRINFO	Resolves DNS names into OS format
RAW_GETNAMEINFO	Translates a socket address to a node name
RAW_SEND	Used to send using a TCP socket
RAW_SENDFILE	Used to send a file *send_file()*
RAW_RECEIVE	Receive data from a TCP socket
RAW_SHUTDOWN	Close one side of full-duplex TCP connection
RAW_CLOSE	Close a connection

Table 3–1: Main commands supported by the TOE-Em

Our approach, emulates this exchange of control messages between the offload engine and the web server. The raw protocol creates an abstraction of the commands involved in the communication between the TOE and the Web server. In a real implementation, this happens inside the host. We assumed that when the TOE is part of the host, the communication between the TOE and the CPU, is faster than in our current test bed. However, the contribution of this research

Command	Comment
RAW_BLOCK_OPT	Used to turn on or off the non-blocking sockets
RAW_TIMEOUT	Sets timeout on a non-blocking socket
RAW_REGISTER_PID	Matches a PID from a PID at the Web server
RAW_KILL_PID	De-allocates resources in the *TOE-Em*
RAW_FORK	Creates a new process on the *TOE-Em*
RAW_EPIPE	Indicates if one half of the connection has been closed

Table 3–2: TOE-Em Specific Control Commands

is using the experimental setup to validate the abstractions of the mathematical model. Also, our approach is apt for analyzing the performance of the system.

3.3.6 TCP Offload Engine Support for Web Servers

The Web Sever application selected was Apache version 2.2 [75]. Apache is a free Unix-based Web server developed by a group of volunteers/contributors based on NCSA httpd 1.3 [76]. Apache is the most popular Web server running today. This Web server accounts for approximately 65% of all Web domains on the Internet [77]. Another important aspect in choosing Apache is that it is open source. Its source code is available and distributed freely. This is an advantage for us since the code can be changed for suiting our needs. The version used was Apache 2.2, which incorporates some of the improvements proposed in [18, 55].

A *hacked*[5] version of Apache 2.2 was used to interface it with our *TOE-Em*. This version includes a modified socket interface that uses raw sockets to communicate to the *TOE-Em*. All the socket interface of Apache 2.2 was changed in order to adapt it to our needs. Specifically, four of these sources were modified:

1. **sockets.c** : mostly used to handle connection establishment
2. **sockaddr.c** : used to handle address resolution and other calls
3. **sockopt.c** : used to handle socket options
4. **sendrecv.c** : used for handling the calls of sending and receiving

The Apache 2.2 **core application** was mostly maintained intact. Altering the Web server application substantially would have resulted in diverging from our scope. However, the socket interface was changed substantially. The *hacked*

[5] A way of altering a computer program beyond its original design goals.

```
...
mpm_state = AP_MPMQ_RUNNING;
bucket_alloc = apr_bucket_alloc_create(pchild);
while (!die_now) {
    conn_rec *current_conn;
    void *csd;

  rv = raw_register_child(ap_my_pid);
  if (rv < 0) {
     fprintf(stderr,"CHILD COULD NOT BE REGISTERED");
     clean_child_exit(APEXIT_CHILDFATAL);
  }

    apr_pool_clear(ptrans);
    if ((ap_max_requests_per_child > 0 ...
```

Figure 3–11: Modified prefork.c on Apache 2.2

version of Apache 2.2 stays as a multi-process program as the original non-modified version is. No other substantial modification has been made to Apache 2.2 **core**. Just a slight modification was made to Apache 2.2 core and is described as follows. Apache starts and forks into several child process by default. Each child process listen to TCP port 80 and waits for a connection. In order to alert the *TOE-Em* of this event a new function was called from the source code, **prefork.c** (see Figure 3–11). This enables the modified Apache to issue a notification that a new child process has been created. Then, the *TOE-Em* could spawn a new SP that is mapped to this newly created child. This is the only instruction added to the Apache core.

Apache 2.2 has been modified also to run as root. Apache 2.2 could not run as root by default for security reasons. However, the modified Apache 2.2 has to run as root since the only valid user for opening a raw socket is the root. The hacked version bypasses the section where it validates what user is running Apache 2.2.

3.3.7 System Architecture

This section presents the system architecture used in the test bed. This is the hardware used to run Apache 2.2, Apache 2.2 with the modified socket interface layered, and the benchmarks used. The software used to create the workload was based on two benchmarks: SURGE and Webstone. The benchmarks and the objects used to test the system are discussed on the following paragraphs.

The hosts have the following configuration: The front-end PC has an Athlon XP 2600+ 1.9 Ghz with a cache size of 512 KB and 756MB of RAM, the PC acting as our Web server is a Genuine Intel Pentium 1.8 Ghz with a cache size of 128KB with 512MB of RAM. Both PCs have the same operating system, i.e. Linux Fedora Core (FC) 6 with kernel version 2.6-18. The Web server runs Apache 2.2.

Several nodes were used to act as users requesting pages from the Web server. The main machine was a Genuine Intel Pentium 1.6 Ghz with a cache size 512MB of RAM. The OS for this PC was Linux Mandriva with kernel version 2.6-18. Another PC was used for some of our test and this was an AMD K6-2 600Mhz with Linux kernel 2.4.8. The benchmarks used were run from these nodes and will be discussed in the following sections.

3.3.8 Benchmarks: Webstone and SURGE

This research combines notions from different benchmarks to stress our test bed in different ways. Our target is to study how an offloaded host behaves for different types of requested objects. Webstone 2.5 was used for testing our system [59]. Originally, Webstone was developed by Silicon Graphix. Currently, Mindcraft supports the benchmark. Webstone was chosen for the following reasons:

- Webstone is open source

- Previous works are available for validating our results
- Webstone can be modified in a number of different ways

Since Webstone 2.5 is bundled with a limited file system, SURGE was used for generating the set of files for our test bed. The benchmark software stresses the system until it reaches the saturation point. However, SURGE could not be configured easily for our purposes.

The file set generated by SURGE has been classified and divided into four different classes similar to what was done in [18]. Table 3–3 presents these classes. Files with sizes ranging from 0 to 1 KB has been classified as Class 0. Class 1 contains files with length greater than 1KB and less or equal than 10KB. The files with sizes greater than 10KB and less or equal than 100KB constitute Class 2. Class 3 files are those files ranging from 100KB to 1MB. Finally, Class 4 files have sizes greater than 1MB.

Classes	Range
0	0 - 1KB
1	1KB - 10KB
2	10KB - 100KB
3	100KB - 1 MB
4	Over 1MB

Table 3–3: Classification of files

Tests were run for all the classes three times for 10 minutes each. Our test bed was confronted with a default Apache 2.2 configuration for a single processor system with no protocol offload capabilities. The /proc file system of Linux was used to acquire our results with a hacked version of the Unix program for monitoring the host called *top*.

3.4 Summary

This chapter presented the methodology used to conduct the research and confirm our contributions. The first phase of the research was developing an analytical model for estimating the utilization and delay before and after the extraction of protocol processing from the system. The second phase uses an emulation-based approach mixed with a real-implementation of the most common Web server used today, Apache 2.2. The implementation of the emulator and the real Web server is used to validate our model. The TCP Offload Engine Emulator (*TOE-Em*) is a **multi-process** program that have all the functionality of a TCP Offload Engine. This emulator provides the socket interface layer to the real Web server application. The core of the Web server application was almost left intact, however, the socket interface was changed substantially. This research combines notions from different benchmarks to stress the experimental setup in different ways. Our target was studying the performance benefits of an offloaded host handling different types of requested objects. Since confronting our experimental setup with a general case conceals the real benefits of protocol offload, the requested objects were classified in five different classes based on their size. Tests were run for every class stressing the server until it reaches the saturation point. Results were obtained from each configuration and presented in the following chapter.

CHAPTER 4
ANALYSIS OF RESULTS

This chapter presents the analysis of the results when comparing the results obtained from the analytical model with the ones measured from the emulator. Section 4.1 presents the validation of the probabilistic model discussed previously. After validating the probabilistic model, a quantitative analysis was performed from the data gathered from our test bed. This is presented in Section 4.2 where the performance metrics of goodput, utilization, and response time were analyzed further. Also, the number of connections that the system was able to handle when it was saturated is presented in this chapter.

4.1 Probabilistic Model Validation

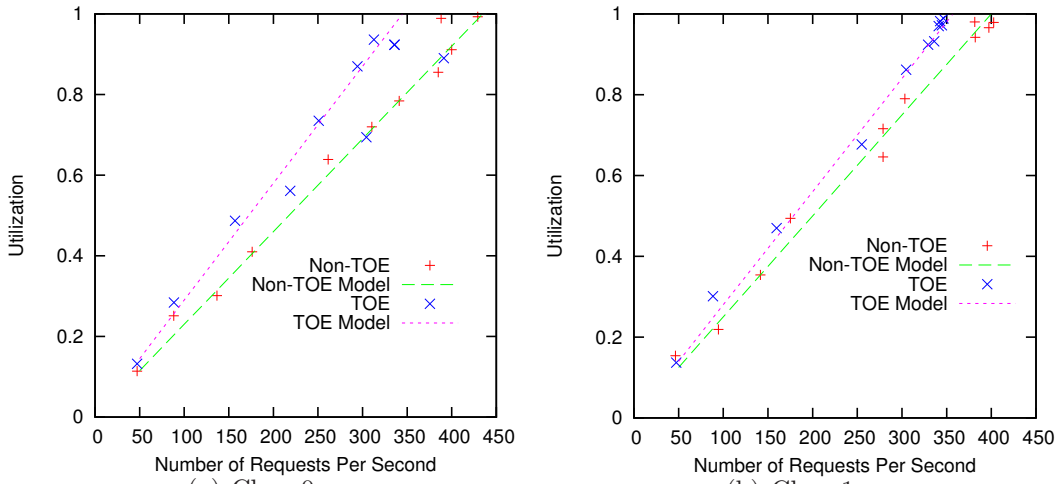
Webstone 2.5 was slightly modified in order to compare the estimated utilization obtained by the probabilistic model with the utilization obtained by our emulated Offload Engine. Webstone was modified for producing arrivals following a *Poisson* distribution. This means that Webstone produced sessions with inter-arrival times that followed an *exponential* distribution. The following sections presents the results obtained for these arrivals.

4.1.1 Utilization

The utilization of the host was measured before and after protocol processing extraction for all of our file classes. Figure 4–1 presents the utilization obtained by

the probabilistic model versus the measurements obtained from the experimental setup. The graph presents the utilization versus the number of requests per second. The values obtained by the analytical model are presented as lines. Points in the graph present the measurements obtained from the emulator. The non-supported host has been label as the Non-TOE and the supported host as the TOE. The analytical model is labeled as Non-TOE Model or TOE Model. These labels are used in some of the plots of this section.

The analytical model estimated that protocol offload via our TOE-Em would not be beneficial for files with sizes ranging from 0 to 10KB. These are files classified in Classes 0 and 1.



(a) Class 0 (b) Class 1

Figure 4–1: Utilization for Classes 0 and 1

This was true in our context, since the application plus the overhead for communicating with the TOE-Em is in fact greater than the application plus the actual transmission before offloading the CPU. This means that $E[X_o] > E[X_p]$ in

the probabilistic model. In order to achieve better performance for these classes, the time to process the overhead has to be reduced.

The expected time to process the overhead ($E[X_o]$) increases as a result of processing the protocols on another host. Every time Apache 2.2 uses the socket interface, a command is invoked via an Ethernet frame to the TOE-Em at the front-end PC. The process at the Web server must wait until the result of this command returns. $E[X_o]$ should be smaller in a real implementation since the offload engine resides inside the same host and these overheads bring by the inherent characteristics of our test bed could be reduced substantially.

Figure 4–2 shows the utilization versus the requests per second for both, the probabilistic model and the measurements obtained from the TOE-Em. The analytical model approximates the measurements. Offloading TCP/IP is in fact beneficial for files with sizes greater than 10KB in our experimental setup. Figure 4–2 presents the results for classes 2, 3 and 4.

As the files grow in size, more arrivals could be handled and the CPU of the host consumes less time. Table 4–1 presents the saturation point of the system for classes 2, 3 and 4. Notice that the saturation point of the system is approximately 350, 300, and 40 requests per second for classes 2, 3 and 4, respectively, in the TOE supported system. However, the saturation point for the non-supported host was approximately 300, 100, and 12 requests per second. This is an increase of 15% for Class 2. The saturation point was three times higher for class 3 and 3.33 times higher for class 4.

The CPU of the host never reaches the saturation point for files with sizes greater than 1024 KB. This means that the bottleneck of the system is not the CPU of the Web Server anymore. The bottleneck now resides on the front-end

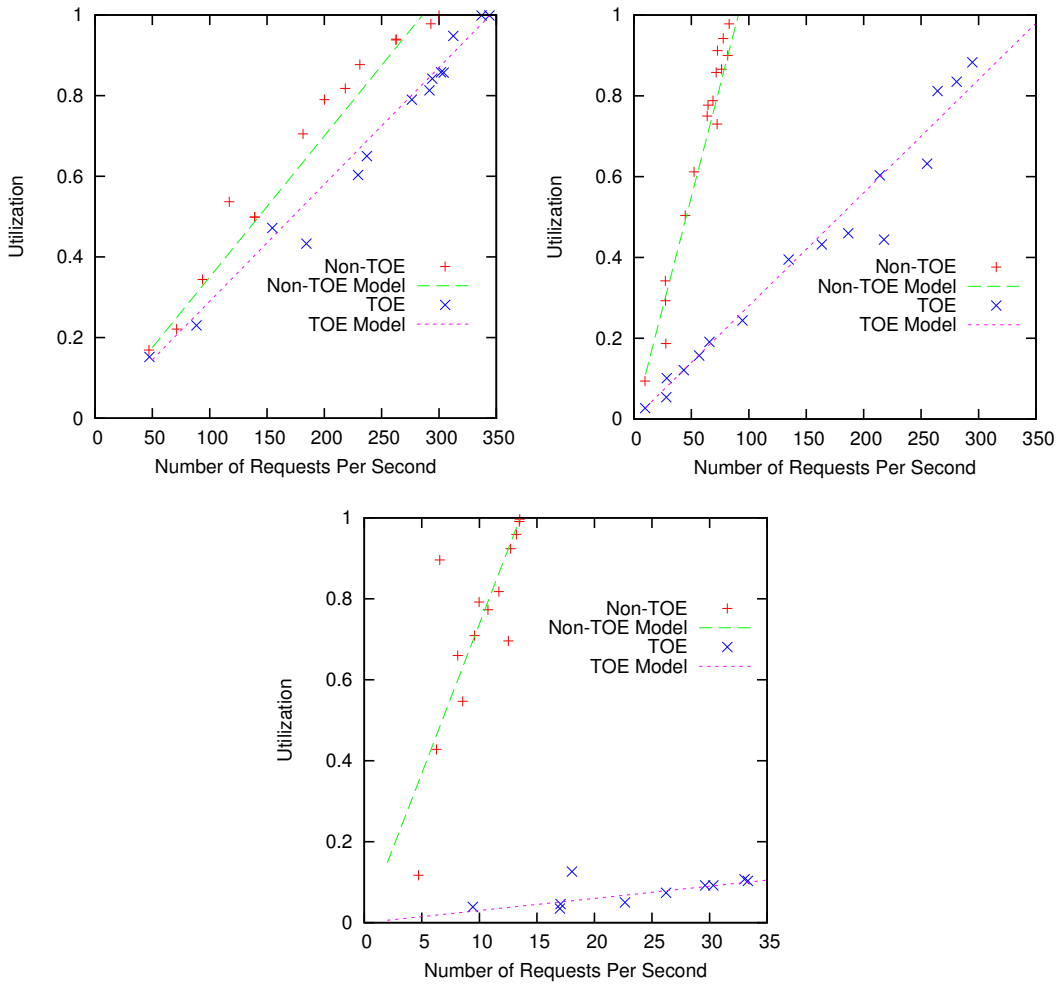


Figure 4–2: Utilization for classes 2(top left), 3 (top right) and 4 (bottom)

PC where the TCP offload engine is emulated. This is discussed further in Section 4.2.

4.1.2 Delay of The System

An efficient offload engine reduces the utilization of the CPU and maintains or reduces the delay after protocol processing extraction. This section analyzes the results obtained from the analytical model and compares them to the results obtained from the test bed.

File Class	Non TOE	TOE
2	300	350
3	100	300
4	12	40

Table 4–1: Saturation Point for Classes 2, 3 and 4

Figure 4–3 and Figure 4–4 present the delay estimated by the model and the delay measured from the Web Server for the Non-TOE and TOE configurations for class 0 and 1. The results obtained from the probabilistic model are plotted with solid lines while the results obtained by the test bed are plotted with points. Notice that for every class the model approximates the values obtained from the model with the ones obtained from the test bed. This is validated further using the Kolmogorov-Smirnov (K-S) analysis presented in Section 4.1.3. As predicted by the probabilistic model, the inclusion of an Offload Engine in our system is not beneficial for these file classes. This is evident for files with sizes lower than 10 KB where the Non-TOE outperformed the TOE configuration.

The mean response time obtained from the Non-TOE and TOE configurations for files with sizes greater than 10KB is presented in the following figures: Figure 4–5 presents the results for class 2. Figure 4–6 presents the results for class 3. Then, the results obtained for class 4 are presented in Figure 4–7. The TOE alternative outperformed the Non-TOE by reducing the average delay while increasing the number of requests per second that the whole system can handle. The probabilistic model predicted this behavior.

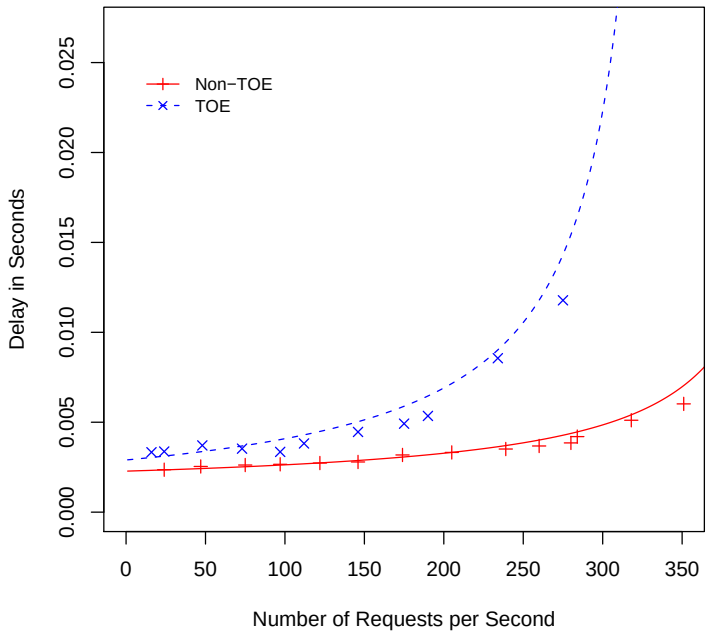


Figure 4–3: Estimated and Measured Average Delay for Class 0

4.1.3 Kolmogorov-Smirnov Analysis

A Kolmogorov-Smirnov (K-S) test was done for validating the performance measurements obtained. The K-S test is a form of minimum distance estimation that determines if two sets of data differ substantially [78]. The K-S test is non-parametric and distribution independent (see Appendix C). The statistical software environment used to perform this test was R version 2.7.1 [79]. The R Statistical Environment is a project developed at Bell Laboratories and mostly used in scientific computing [79]. This tool was used to generate the values presented in Table 4–2 which are the values obtained from the K-S test. We obtained a p-value greater than 0.20 for all of our data sets.

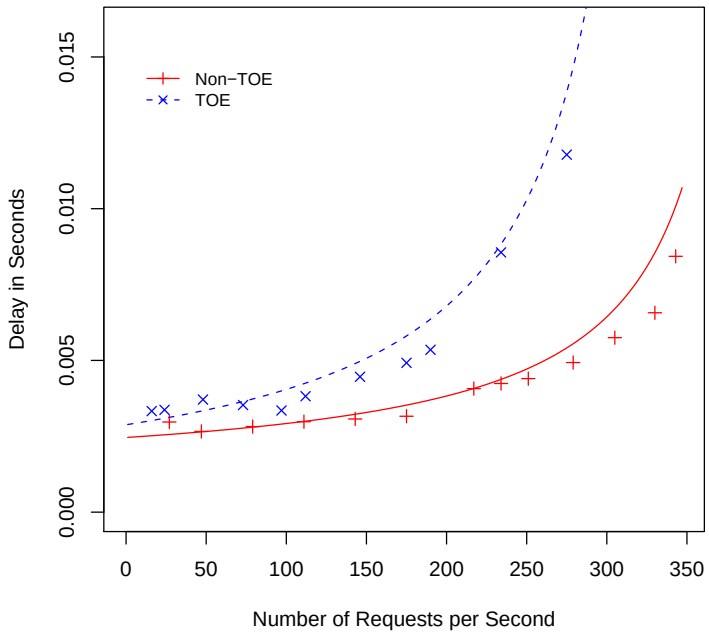


Figure 4–4: Estimated and Measured Average Delay for Class 1

4.2 Quantitative Analysis of The Experimental Setup

In this section, benchmarks were used to study our design beyond that the abstraction of the analytical modeling can provide us. In our context, a benchmark is the program used to find the upper bound in requests per second, connections, and goodput that the whole system can handle. The emulation-based approach provides the capabilities of testing the real system with real workloads to obtain accurate performance measurements beyond the capabilities of the analytical model. Therefore, this research had been bold into studying what occurs beyond the results of the analytical model. This is true since Queuing Theory is limited in estimating performance measurements in an unsteady state.

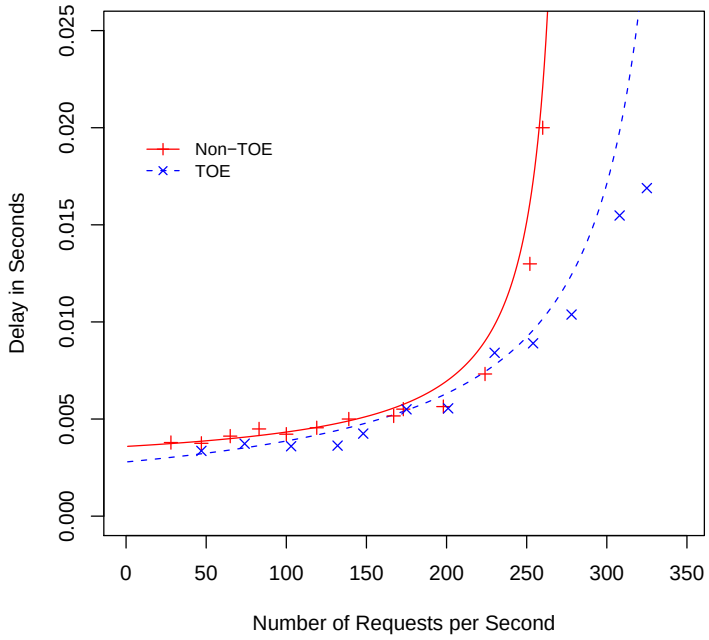


Figure 4–5: Estimated and Measured Average Delay for Class 2

Webstone 2.5 was run without any inter-arrival time between session initiations to study the impact on the saturation point. Other performance measurements were obtained and analyzed essentially from the TCP Offload Engine Emulator.

4.2.1 Utilization

Figure 4–8 presents the utilization for a default configuration of Apache without TOE-Em support. The utilization of the CPU is divided in seven types according to the percent of time spent handling:

- software interruptions
- hardware interruptions

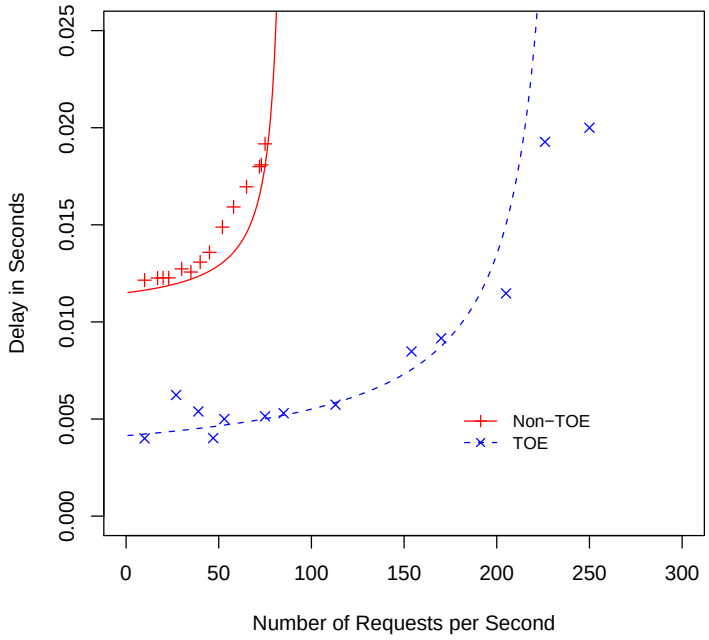


Figure 4–6: Estimated and Measured Average Delay for Class 3

- input and output system calls (I/O)
- kernel (operating system software)
- user (software application software)
- nice (non intrusive calls)
- idling (A CPU doing nothing)

In Figure 4–8 the server is stressed until saturation. However, for classes 2, 3 and 4, files greater than 10KB, the time the CPU spends on the kernel and handling interrupts increments substantially reducing the percent of time the system is handling the user space (application). In our work, the reduction in software interruptions reflect the reduction in processing TCP/IP protocols. It is

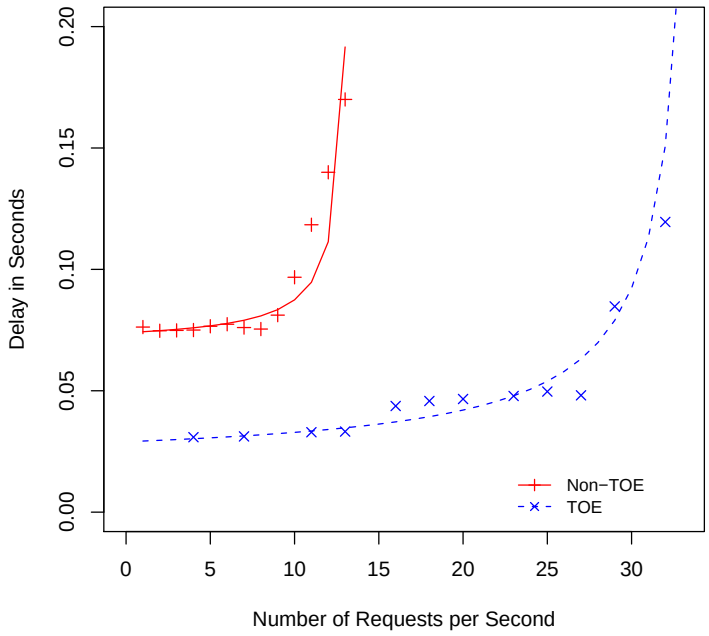


Figure 4–7: Estimated and Measured Average Delay for Class 4

known, by analyzing the source code of the Linux kernel that protocol processing generates software interruptions [80]. Notice that for every file class the percent of time used by the CPU for handling each type varies. This has been achieved using the capabilities provided by Linux Kernel 2.6. The behavior of the whole system is similar to what has been analyzed by Hu, Nanda and Yang in [18].

No idle time is visible for a server that runs the default Apache 2.2 without TCP offload. Notice also that the percent of time spent on the kernel and software interrupts is similar for classes 3 and 4.

Figure 4–9 presents the utilization of the host for the configuration that has the modified Apache with TOE support. The utilization for handling software

Configuration	D^+_-	P
Non-TOE Class 0	0.1429	0.9996
Non-TOE Class 1	0.1616	0.9996
Non-TOE Class 2	0.1667	0.9985
Non-TOE Class 3	0.2143	0.9205
Non-TOE Class 4	0.2500	0.8690
TOE Class 0	0.1818	0.9934
TOE Class 1	0.1818	0.9970
TOE Class 2	0.1667	0.9985
TOE Class 3	0.2308	0.8978
TOE Class 4	0.2727	0.8326

Table 4–2: Kolmogorov-Smirnov Test Results

interruptions has decreased to around 7.5% for all file classes. This reduction in software interruptions is a result of offloading TCP and IP from the server that handles the Web server application. Notice that the utilization for user space[1] remains the same (44% approximately) for all the classes. This utilization is superior for file classes 0, 1, and 2 of the default Apache configuration (33%, 31%, 24% respectively). The utilization consumed by hardware interruptions increments for every class on the Non-TOE (see Figure 4–8) and reaches 27% and 34% for classes 3 and 4. This is not reflected on the host supported by the TCP Offload Engine Emulator. The percent of time used for handling interruptions reflects a utilization close to 10.3% for every file class except for file class number 4. File class 4 showed a 2% of utilization in Figure 4–9. The percent of utilization in user and kernel space is barely the same for files of classes 0, 1, and 2. The utilization at

[1] application time

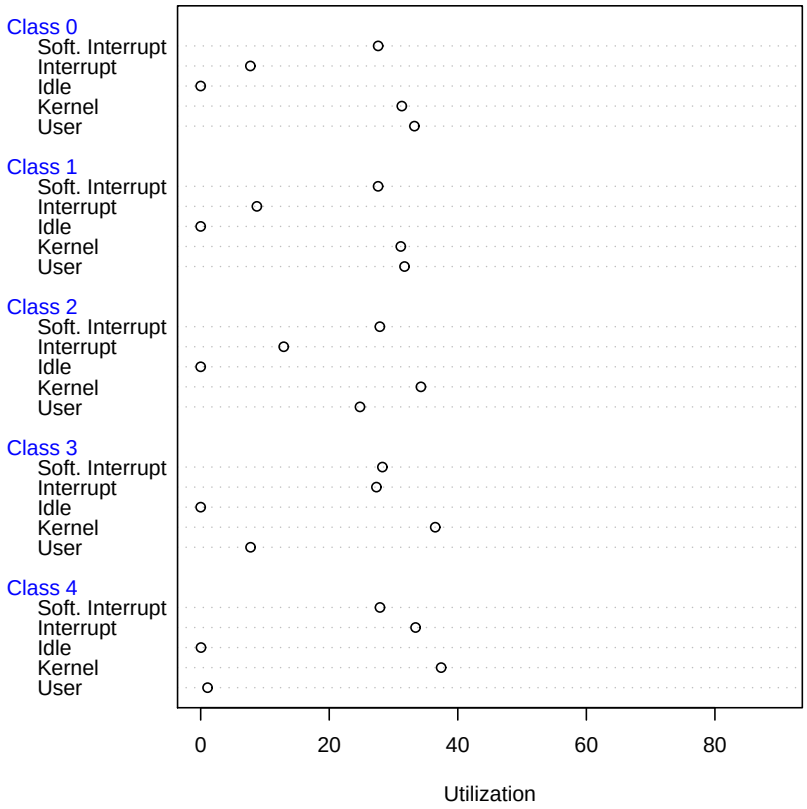


Figure 4–8: Default Apache 2.2 (protocols handled by the OS)

file classes 3, and 4 varies. This happens since the Web server, in some occasions, idles when handling file classes 3, and 4.

Figure 4–10 presents the utilization of the CPU of the machine that is running the TCP Offload Engine Emulator, (the front-end PC). Notice that for file classes 0, 1, and 2 the CPU of the front-end PC idles for 80%, 77% and 68% respectively. However for class 3, and 4 this idle time reduces considerably to around 2%. The bottleneck of the system has been shifted from the host handling the Web server to the front-end PC for file classes 3, and 4. The utilization for software interruptions

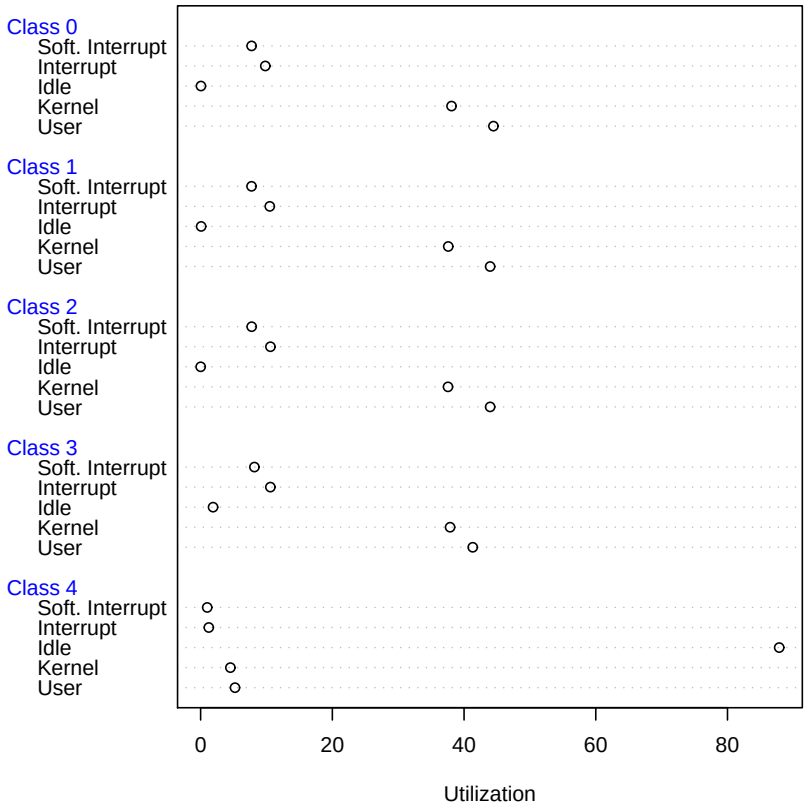


Figure 4–9: CPU Utilization of Apache 2.2 with the Support of the TOE-Em

increments as the length of the file is incremented. It does not increment so much for classes 0 and 1 (around 4% and 5% respectively). This issue can be explained by understanding the behavior of Apache 2.2 core application and the method it uses for transmitting small files. This is explained on the next paragraph. The utilization for software interruptions for file classes 2, 3, and 4 is 10.8%, 49.4% and 58% respectively. Also notice that the time spent in the kernel is greater for classes 2, 3, and 4 (13.5%, 40%, 35%) than of classes 0, and 1 (9% and 9.7%).

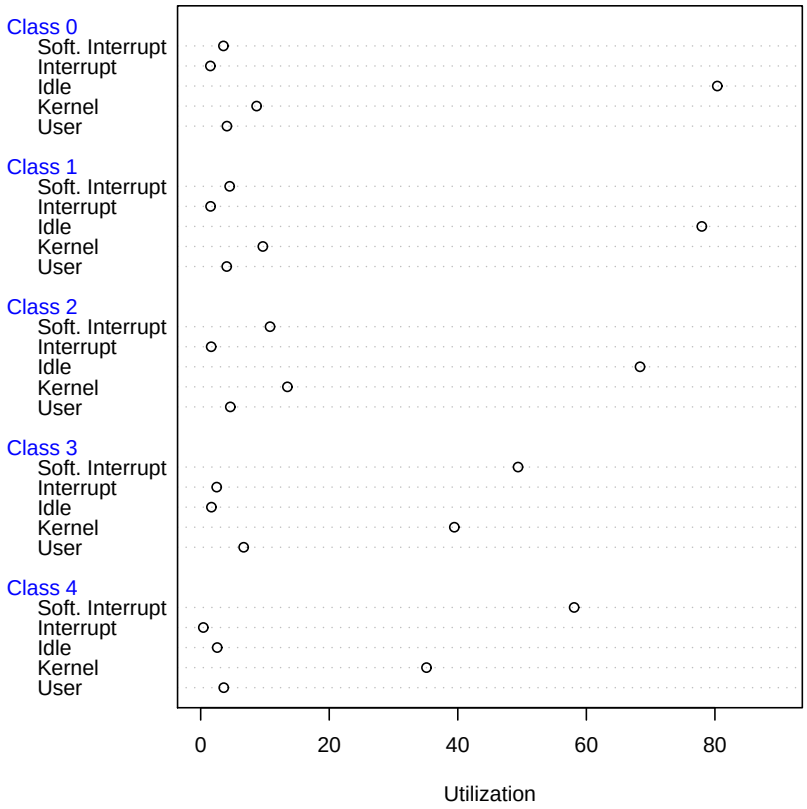


Figure 4–10: TOE-Em Front End PC Utilization

The host supported by the TOE-Em is stressed similarly for file classes 0 through 3. However, in class 4, the front-end PC became the bottleneck.

Apache 2.2 handles small files differently than large ones. Apache does not use *sendfile* for small files (8000 bytes or less). It uses *writev* instead. This makes the data travel from I/O to kernel, from kernel to user, and then, from user to kernel to be transmitted. Then, it reaches the TOE-Em and travels from kernel to user and then back from user to kernel. Large files are handled differently. Since a large file uses *sendfile*, a message is sent to the TOE-Em indicating which file

is to be transferred. The TOE-Em, then, transfers the file and notifies the Web server of its success or failure.

Notice also the increment in utilization by the kernel for classes 2, 3, and 4 in Figure 4–10. Data travels from I/O to kernel and form kernel to the device without a memory copy into user space when *sendfile* is used. Since these files are read into kernel buffers and then transferred the utilization in the kernel increases.

4.2.2 Connection Handling

Figure 4–11 presents the average number of connections per minute handled by both configurations when the system is saturated. Notice that for file class 0 the Web server without TOE support could handle more connections than the supported one. This is approximately 6% more connections for class 0. Also, for file class 1, the Web server without TOE support handled approximately 1% more connections than the TOE supported. The Web server with TOE-Em support out-performed the default one for file classes 2, 3, and 4. The TOE-Em supported Web server handled 21% more connections for class 2, around 62% more for class 3, and 66% for class 4.

It is worth noticing that the Web server was the bottleneck of the system when confronted to file with sizes less than 10Kb (class 0 and 1). The system is not properly balanced and the mean time spent for handling Apache 2.2 core dominates over the TCP/IP protocol processing. However, improvements in Apache can be achieved to solve this.

Table 4–3 presents the average time it takes to establish a connection when the system is saturated. Notice that for all the file classes, the connection establishment time is around 0.28 ms for the non-supported host. However, for the supported host, the mean time to establish a connection for classes 0, 1, and 2 is

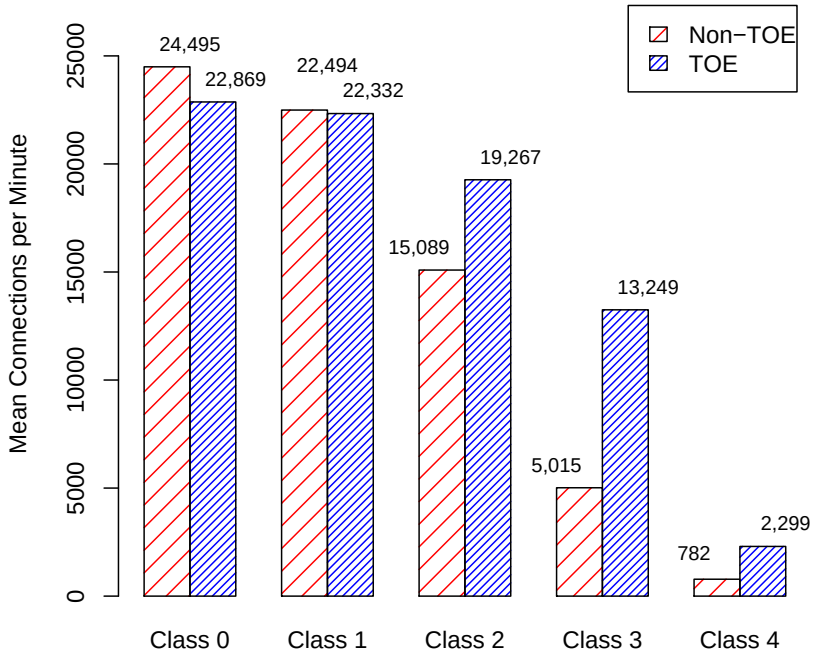


Figure 4–11: Mean Connections per Minute Histogram

approximately 0.09 ms. The mean time to establish a connection is around 0.12 ms and 0.8 ms for classes 3, and 4. Connection establishment when the system is supported by a TCP offload engine is faster because is handled at the front-end PC. As mentioned previously, the front-end PC is not the bottleneck of the system for file less than 100KB (class 0, 1, and 2). The CPU at the Web server is handling

other processes while the CPU of the host running the TOE-Em is handling the connection establishment[2] .

Configuration	default WS	Test bed
Class 0	0.27	0.09
Class 1	0.28	0.08
Class 2	0.28	0.09
Class 3	0.28	0.12
Class 4	0.29	0.78

Table 4–3: Connection Establishment in Milliseconds

4.2.3 Goodput

Throughput is a performance metric of the capacity of the medium. It is defined as the rate at which data can be sent through the network, specified in bits per second (b/s). The performance metric that is obtained by measuring the amount of data transfer per unit of time is called the goodput [52]. Table 4–4 presents the average goodput in megabytes for both configurations when the system is saturated (also see Figure 4–12). Notice that for file class 0 the non-supported Web server transfer 61.4 KB over the supported one. For class 1 both configurations behave similarly, just a slight performance increase is achieved by not using the offload alternative. The *TOE-Em* supported host out performed the default Web server for classes 2, 3, and 4 by 2.4 MB, 27 MB and 49 MB respectively. This is 23%, 58% and 67% of improvement. Figure 4–12 presents these values in an histogram.

[2] TCP three way handshake

Configuration	Apache 2.2	Test bed	Difference Testbed - WS
Class 0	317.4 KB/s	256 KB/s	(61.4 KB/s)
Class 1	1.36 MB/s	1.35 MB/s	(10.24 KB/s)
Class 2	7.8 MB/s	10.2 MB/s	2.36 MB/s
Class 3	19.6 MB/s	46.5 MB/s	26.87 MB/s
Class 4	24.8 MB/s	74.1 MB/s	49.28 MB/s

Table 4–4: Measured Goodput

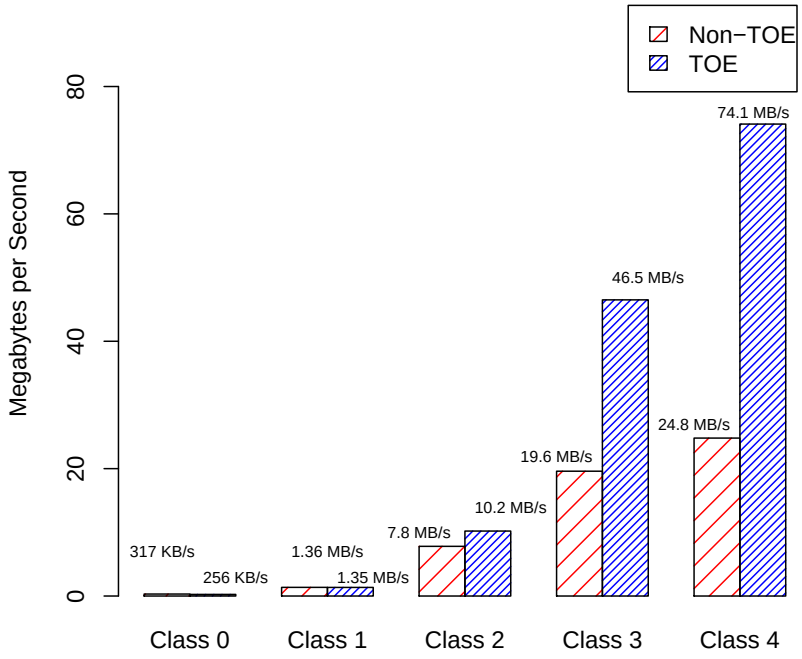


Figure 4–12: Goodput per Minute Histogram

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

This chapter presents the conclusions and their implications related to our contributions of the research presented. Also, this final chapter restates the research problem and reviews the methodology used for the research conducted. This chapter includes possible future directions that could propel the research even farther.

5.1 Brief Summary of the Dissertation

As explained in Chapter 1, on the dawn of multi-core multi-threaded processors and new technologies on outboard chips, a new debate related to where protocol processing should be performed arises. TCP offload engines have been proposed as an option for alleviating the CPU of the host of the cost of processing TCP and IP. These proposals tend to locate the OE inside the network interface hardware. Another, approach has been including the OE as a component near or even inside the CPU of the host. However, the interactions between the host and the OE are independent of the underlying network technology. The interface between the host and the offload engine could spoil most of the performance benefits brought by the inclusion of an OE. The performance of the host is degraded, in some cases, after including another entity to process the protocols. However, there are situations when the inclusion of an entity is beneficial. The uniqueness of our

research abstracts some of the inherent characteristics of protocol processing for identifying in what situations protocol offloading is beneficial. The research aims to study the pros and cons of protocol offloading and its related environment.

Chapter 3 presented the methods used to conduct the research and confirm our contributions. The first part of the research was developing an analytical model, based on probability theory, for estimating the utilization and delay before and after the extraction of protocol processing from the system. A network of queues is used to abstract the system and obtain the performance measures. The second stage uses an emulation-based approach mixed with a real-implementation of the most common Web server used today[1] . The emulator and a *hacked* version of the Web server is used to validate our mathematical model. The *TOE-Em*[2] is the program that emulates the functionality of a TCP offload engine. The tests conducted merged features from benchmarks to strain the experimental setup in diverse ways. Since it is known that the system behaves differently when handling different types of request, our target was studying the performance benefits of an offloaded host that handles these requests. Since confronting our experimental setup with a general case conceals the performance benefits of protocol offloading, the requested objects were classified based on their size. Tests were run for every class stressing the server in different ways until the system reaches the saturation point.

---

[1] Apache 2.2

[2] A TCP Offload Engine Emulator

In Chapter 4 the results obtained from our research were presented. The performance measurements obtained from the probabilistic model were compared and contrasted with the results obtained from the experimental setup. CPU utilization and delay in the system were estimated by the analytical model with p values greater than 0.75 under the Kolmogorov-Smirnov test. Therefore, we can conclude that the model estimated the performance measurements very well. Also, results obtained from the benchmarks were studied to analyze our design even further.

The following paragraphs present our contributions, conclusions and possible future directions related to the research presented in this document.

5.2 Main Contributions

The main contribution was producing **a probabilistic model that estimates the performance increase or decrease in utilization and delay when protocol processing is shifted from the operating system to a protocol offload engine**. As presented in Chapter 2, most of the research on protocol offload and engines is experimental. No work related to protocol offload was found during the survey of literature that handled the issue using Queuing Theory. Most of the analytical models are deterministic. Our analytical model considers the stochastic nature of the end-to-end transmission. Consequently, our model includes the process for converting the requested object R into packets of protocol P and how it was distributed. The analytical model is simple and extensible.

The second contribution was the development of **a TCP Offload Engine emulator used to validate the model but also provides an insight of the real problem beyond the limitations of the analytical model**. The levels of complexity and detail of our emulation-based approach are superior to

the ones provided by the generalities of the analytical model. We were unable to find research under our survey of literature that implement an offload engine emulator using our innovative approach. This approach pitches our research into the experimental ground. The *TOE-Em* could be modified for further inclusion into similar environments. The emulator is going to be available for the scientific community to study TCP offload engines even further.

Our approach of **classifying files based on their sizes present an innovation for studying offloading from a totally different perspective**. This is our third main contribution. Dependencies on the size of the requested object stress the host in different ways [18, 19, 81]. This approach has been followed for further improvements on the Web server application but has not been oriented to protocol processing improvements. We were unable to find investigations on the standpoint of protocol offload that segregated the requested objects and analyzed its impact on performance. Most of the research conducted in TCP offload abstracts the file system as a whole [20–22] or even uses workloads that benefit their test bed [4, 21, 23]. Studying only the general case, or the extreme case, could conceal the real benefits of protocol offload over a wide variety of different situations since it is known that the host behaves differently when handling different objects [18, 81].

The fourth main contribution was **studying the behavior of the Web server before and after protocol processing extraction not only when the system saturates, but also when the system is in equilibrium**. Experimental work [6, 11, 20] used benchmarks to saturate the system and obtained measurements for analyzing what happens under these extreme conditions. Our

approach not only studies the behavior when the system is saturated, but also studies the behavior of the system when it is not.

The fifth main contribution **was comparing the analytical model with a real implementation of a common Web server used today**. Apache 2.2 was used as part of the test bed. Minimal changes have been done to the Apache 2.2 core application when interfaced with the TCP offload engine emulator. Therefore, our research implies that a network application does not need to be modified substantially if an offload engine is included as part of the system.

5.3 Marginal Contributions

These section present marginal contributions that have been obtained from our research in transport layer protocol offload. Also, the following paragraphs present some additional conclusions and remarks.

The research conducted provided a mathematical model that was validated with a test bed composed of an emulator and a real-implementation. Since the analytical model uses the notions of Queuing Theory our research scope is not bound to a specific hardware or coded into any specific on-board embedded processor. Therefore, it is not limited by a specific hardware constraint that can ruin or biased our findings as others has faced [82].

This research was focused on full TCP offload. The topology used in this research allows no constraints for achieving full or partial protocol offload.

The analytical model is not bound to any specific characteristic of the protocols. This means that the model is not limited to the protocols TCP and IP.

5.4 Conclusions

Protocol offloading is not beneficial in scenarios in which the mean time to process the overhead of communicating with the offload engine at the CPU[3] is greater than the average time of converting requested object R into packets of protocol P in the same CPU before protocol processing extraction. The cost of processing the overhead could not exceed the time it takes to handle the protocols at the CPU rather than in the OE. In our context, an increase in utilization that resulted in degrading performance occurred for files with sizes equal or less than 10KB when the protocol processing was offloaded from the CPU of the Web Server. If the expected time to process the overhead is less than the expected time to process the protocols ($E[X_o] \leq E[X_p]$), then the CPU of the host has been offloaded successfully, however, we cannot state that this will result in a performance increase. The estimated delay in the system needs to be calculated to confirm an actual performance increase.

The bound on the mean time for processing the overhead can be found using this model. The bounds on overhead of the CPU and the OE can be found using the performance metrics. These, bounds are found by comparing the expected service time for processing the protocols and the overheads[4] . Even if the overhead is unknown, the bound on this variable can be found by examining the expected time of processing the protocols.

---

[3] Represented in the model with random variable X_o

[4] random variables: X_p, X_o and X_c

The model estimates the behavior of the delay of the system before and after protocol processing extraction. This is evident when the measurements obtained from the emulator are compared to the ones obtained by the analytical model. The results obtained from the K-S analysis performed on the performance measurements acquired from the experimental setup resulted in p values greater than 0.75.

Abstracting the system composed of a CPU with the support of an OE as an M/M/1 queue in tandem with an M/G/1 queue resulted in a suitable approximation of the real network scenario. Although others have claimed that a Web server can be modeled as an M/G/1 or an M/G/1/K [30–32], it seems than when protocol processing is removed from the main CPU, the CPU begins to behave as an M/M/1 queue. Therefore the research suggests that in an offload scenario the CPU can be modeled as an M/M/1 queue. Consequently, processing communication overhead as an exponential process fairly approximates the real scenario. However, this claim needs further analysis and is left as future work.

The performance measurements obtained from the probabilistic model can be used to properly balance the system by administering the resources of the host correctly. The third and fourth main contributions provide the bases for a handoff technique that uses our probabilistic model as part of the OS for deciding when to offload protocol processing and what objects to offload.

5.5 Limitations

There are some milestones that were encountered when conducting the investigation presented in this dissertation. These limitations are presented in the following paragraphs.

Obtaining measurements when the host utilization was near 100% on the TOE supported configuration was cumbersome sometimes. Ethernet frames containing the commands that travel to and from the Web server were dropped when the front-end PC was saturated. This happens because the front-end PC was unable to process the incoming frames while it was preparing the next burst to be transferred. Frames containing commands began to be dropped when the NIC's buffer overflowed. The raw protocol used by the *TOE-Em* did not provide a way of detecting this error. Consequently, the whole system hanged near the saturation point of the front-end PC. This means that the design of the raw protocol that is handled by the *TOE-Em* depends on a reliable connection and the underlying protocol does not provide it. This has to be solved if the *TOE-Em* is going to be used in a real scenario as a front-end.

Our *TOE-Em* could be optimized even further. The *TOE-Em* has a design flaw that needs to be improved on a second version of the emulator. This is a problem of the Commander and Reader Process. This process stops reading frames from the NIC whenever it finds that the Slave Process that must handle the command is busy. Therefore, the CR, blocks until the Slave Process is available to handle the command. All other incoming frames must wait for this slave process to be ready. Incoming frames are buffered and are not available until the next *recvfrom()*. Adding a queue for storing the commands received by the CR and destined to an unavailable SP seems to unravel this issue. This was analyzed under the design phase of the *TOE-Em*, however, the idea of implementing a queue conflicted with the memory management constraints imposed by us. Two approaches can be used; either handling the memory in a shared area, or using dedicated memory for every process and passing the incoming frames via pipes.

Consequently, this approach impacts the original design heavily. The designer of a real offload engine must maintain memory copies at minimum for achieving better performance.

The *TOE-Em* was confronted using the capabilities of Webstone 2.5. Webstone 2.5 is not capable of generating HTTP requests that use pipelines. Simple HTTP 1.0 requests were used in our test bed. The HTTP 1.1 specifications allow more than one object to be transferred via a single request. This means that if two objects are going to be requested from the WS using HTTP 1.0, two separate connections are established for transferring each object. HTTP 1.1 uses only one connection and maintains it open until one of the peers closes the connection. However, our model is capable of abstracting this event. The experimental setup was confronted with this type of requests for exploring the capabilities of the TCP offload engine emulator. These requests are transparent to the functionality of the *TOE-Em* because the emulator process the protocols that lie beneath the application layer (TCP/IP).

Another approach studied during the early stages of this research was implementing the *TOE-Em* and the Web server application inside an SMP. The implementation of the *TOE-Em* required exclusive access and full privileges to change the OS internals of the SMP resulting in heavy modifications of its functionalities. This approach was not pursued further due to the issues presented.

5.6 Future Work

The performance metrics presented by the analytical model are: utilization, delay, and average number of active connections: inside the system, in the server, and waiting to be serviced. However, there are other performance metrics that have not been presented. Throughput is a performance metrics that was not

considered by our analytical model but studied experimentally. A Poisson process does not necessarily describe this performance metric. Although, the arrival rate (λ) of requests approximates throughput if and only if the arrival rate is observed as the same time scales than the throughput, and the system is under equilibrium ($\rho < 1$). During the Kolmogorov-Smirnov analysis of the values obtained from the experimental setup we found evidence that suggests that the output of the system, in some occasions, follows a *lognormal* distribution. This was found as a byproduct of the tool used to perform the K-S test. However, this needs further analysis and is recommended as a future work.

In our test bed, a requested object is abstracted as a file. This file could be a static HTML page, an image, an object, an animation, a video feed, and any other static object. It has been presented on [61] that static objects stress the Web server more than some workloads that include CGI, ASP, and some Servlets. However, static pages do not stress the system more than pages that contain results generated by scripts that use database connections. Extending the analytical model to include this type of workload is left as a future work.

In order to achieve the maximum benefits from an offload engine the application has to be aware of its capabilities. If Apache 2.2 core is aware of the presence of the *TOE-Em*, then it could benefit of the capabilities provided by the offload engine. Moreover, the overhead incurred into communicating with the *TOE-Em* could be reduced substantially. The Web server could issue more than one command to the *TOE-Em* in order to reduce overhead and maximize parallelism. This could be used also for reducing the utilization obtained from transferring objects with lengths less than 10 KB with the current approach. Modifications to Apache 2.2 core to maximize the use of the *TOE-Em* are left as future work.

The analytical model can be used in a real environment to predict when to rely on the offload engine and when to use the operating system of the host to transfer data of requested object R in packets of protocol P. The input parameters for our model can be obtained by maintaining statistics at the OS. The OS could discriminate which objects the OE is going to handle and which is not. This is beneficial when balancing the load within a system. Moreover, the OS could detect if the OE is lagging behind the capabilities of the CPU.

This model was applied to an Apache 2.2 Web Server running in a LINUX environment and using the TCP and IP protocols. This model could also be applied into a similar situation. It is known, that HTTP and FTP have a similar behavior [17]. Therefore, this model could be expanded to estimate the performance measurements when using FTP. Also, the model could be used to estimate the performance measurements for other transport layer protocols. These possible future applications are left as future work.

CHAPTER 6
ETHICS

Today the field of Science and Engineering has been affected by conscious and unconscious decisions that have consequently ended in negligence. Negligence is traduced into harm to individuals, the society and especially, the environment. A responsible research scientist should be aware of the culture of where technology is going to be introduced. This refers to the ethnical and religious backgrounds that contribute to the creation of laws that are defined by the society where the results of the research findings are going to be introduced. Today new technologies could introduce negative effects as property damage, security issues, and profits for one at the expense of others. Research scientists and professionals have to be very sensible in the way they produced new technologies that could impact positively and also negatively the field of Science and Engineering. Been unaware of the impact that a special technology could introduce in our society is a risk that research scientist must not face. The scientist must have the moral responsibility when using its judgment for preparing and maintaining the trace goals. The goal of a responsible scientist is the development of usable technology that does not compromise the security of the social construction where the technology is going to be introduced.

6.1 Ethical Statements

6.1.1 Principle of Responsibility

The dissertation and articles publish are not only a technical documents. This documents should be consider also as responsible acts. From this perspective, research ethics have to be state as a subset inside the general moral issues but rather applied to other issues as the professional ethics. A research scientist must follow a politically correct conduct even if there exists a conflict between the intentional effect of its actions and the acquired results. Therefore, an ethical act is the one that is manifested as a responsible action. A responsible action is the one that eliminates or minimize the harms even if this harm is produced unconsciously.

6.1.2 Research Findings

Scientist must based its findings in reliable valid proofs that are of key importance in the process of making decisions, assumptions and conclusions. The research scientist must be aware of the assumptions taking in consideration in order not to fall into the manipulation of certain variables to create artificial context that bias its research out of its original scope. Research in Engineering per se is a process that seeks to comprehend new or updated technologies that their application ends in a real environment. Therefore, the analysis of the results obtained must be a responsible and moral act.

6.1.3 Plagiarism

The research scientist must be aware on the impact on science and engineering that plagiarism represents today. Copying literally a research project from other research scientists and present it as their own is a serious ethical fault that must be avoided at all cost. All texts extracted from articles, proceedings, textbooks

and other type of sources must be identified and properly cited. Another type of plagiarism is to attribute work that has been done by a research team as work achieved by an individual. All the collaborators that have taken part actively on the research project must be included or acknowledged and be considered as part of it.

6.1.4 Trimming and Result Forging

A responsible research scientist should not commit the act of results forging and trimming. These acts represent a direct attack to the fields of Science and Engineering. Reckless scientific methods should be avoided at all costs. Trimming and forging does not contribute to the state of the art of science and engineering. It is impossible to repeat research projects that have been carried out using reckless scientific methods. The harms that this type of research project could generate are incalculable and intractable. These type of actions are an open act of pure negligence.

6.2 Research Ethics

This dissertation presents a protocol offload framework that acts as a guideline for the research scientist to properly implement an OE. This research spans and opens a new trail on the investigation of a true TCP Offload Engine and their behavior on the scenarios exposed. The basic foundations of ethics have been part of the methodology of this research project. The principles of reversibility have been applied in order to minimize or eliminate harms and maximize goods to the field of Computing Information Science and Engineering and consequently to society.

APPENDICES

APPENDIX A
PROBABILITY THEORY CONCEPTS

A.1 Definitions

An experiment has a finite or infinite number of outcomes. Let Ω be the set of all possible outcomes $\{\omega_1, \omega_2, \ldots \omega_n\}$ of an experiment. A collection of these outcomes is called an event. Let A be an event, then $A \subseteq \Omega$. Therefore:

$$\bigcup_{x \in \mathscr{R}} A_x = \Omega \tag{A.1}$$

Let $p(\omega_j)$ be the the weight assigned to the likelihood that a given outcome ω_j will occur for each $j = 1, 2, \ldots, n$. This weight satisfies that $0 \leq p(\omega_j) \leq 1$ and:

$$\sum_{j=1}^{n} p(\omega_j) = 1 \tag{A.2}$$

This weight is called the probability [83]. Therefore, the probability that outcome ω_k will occur is $p(\omega_k)$ and $\omega_k \in \Omega$. The likelihood that a given event A_x will occur is $P(A_x)$.

A probability space is a triple $(\Omega, \mathscr{F}, P)$ where Ω is the set of all possible outcomes, $\mathscr{F}$ is a σ-algebra on Ω, and P is a measured from $\mathscr{F}$ to $\mathscr{R}$ satisfying that:

1. $P(\Omega) = 1$
2. $0 \leq P(A) \leq 1$ for all $A \in \mathscr{F}$

3. if $\{A_j\}$ is a finite or infinite disjoint sequence that defines an event [1] in $\mathscr{F}$ then:

$$P(\cup A_j) = \sum P(A_j) \tag{A.3}$$

A map $X : \Omega \to \mathscr{R}$ is a random variable if the range of X is a countable set $\{x_1, x_2, \ldots\}$, finite or infinte, and $\{\omega | X(\omega) \leq x_j\} \in \mathscr{F}$ for all $j \geq 1$. Then X is a function that maps the outcomes into the real numbers [70]. The notation $\{\omega | X(\omega) \leq x_j\}$ is written as $(X \leq x_j)$ and the event $\{\omega : a < X(\omega) \leq b\}$ is compressed as $(a < X \leq b)$ [70, 83].

A.2 Probability Distribution Concepts

Let X be a discrete random variable with range $\{x_1, x_2, \ldots\}$ and $\{\omega | X(\omega) = x_j\}$ for all $j \geq 1$. The probability mass function f is a function on the range of X defined by:

$$f(x_j) = P(X = x_j), \quad j = 1, 2 \ldots \tag{A.4}$$

P is a function on $\mathscr{F}$ with values in $\mathscr{R}$ [83]. The probability mass function has its domain consisting of the random variable X and its range in the close interval [0,1]. The probability that a value of the random variable X obtained on a performance of the experiment is equal to x. The following properties hold for f:

1. Since $p_X(x)$ is a probability then: $0 \leq p_X(x) \leq 1$ for all $x \in \mathscr{R}$.

[1] This means that every outcome ω_l is independent

2. Since the random variable assigns some value $x \in \mathscr{R}$ to each outcome $\omega \in \Omega$ then the sum of all $p_X(x)$ has to be equal to 1.
3. For a discrete random variable X, the set $\{x|p_X(x) \neq 0\}$ is a finite or countable infinite subset of real numbers. Let $\{x_1, x_2, x_3, \ldots\}$ be this set then:

$$\sum_i p_X(x_i) = 1 \tag{A.5}$$

The probability density function (pdf) is the equivalent probability mass function of a continuous random variable. Let X be a continuous random variable with range $\{x_1, x_2, \ldots\}$. The density function f is the real-valued function on the range of X defined by:

$$f(x_j) = P(X \leq x_j), \qquad j = 1, 2, \ldots \tag{A.6}$$

The probability density function f satisfies the following properties:

1. $f(x) \geq 0$ for all x
2. $\int_{-\infty}^{\infty} f(x)\ dx = 1$
3. $f(x) > 1$ is acceptable since the values of the probability density function are not probabilities[2] .

Let A be a collection of outcomes, then, the probability of the set $\{\omega|X(\omega) \in A\}$ is given by:

$$\{\omega|X(\omega) \in A\} = \bigcup_{x_j \in A} \{\omega|X(\omega) = x_j\} \tag{A.7}$$

[2] The probabilities are obtained by integrating the density function

For the sake of simplicity and as in [70], $\{\omega|X(\omega) \in A\}$ is written $[X \in A]$ and its probability $P(X \in A)$. If $p_X(\omega)$ denotes the probability mass function of random variable X then:

$$P(X \in A) = \sum_{x_j \in A} p_X(x_j) \tag{A.8}$$

Therefore, A is an interval with endpoints a and b and $-\infty < a < b < \infty$. The probability of this interval is written as $P(a < X < b)$. If A is the set of the interval $(a, b]$, then, A is written as $P(a < X \leq b)$. If A is the set of all probabilities in the interval from $(-\infty, x]$, then the probability of set A is written as $P(X \leq x)$.

The function $F_X(t)$ on the interval $-\infty < t < \infty$ is called the cumulative distribution function (CDF). The cumulative distribution function is the probability distribution of random variable X. The CDF is defined as:

$$F_X(t) = P(X \leq t) = \sum_{x \leq t} p_X(x), \quad -\infty < x < \infty \tag{A.9}$$

The probability distribution has several properties:

1. $0 \leq F(x) \leq 1$ for $-\infty < x < \infty$.
2. $F(x_1) < F(x_2)$ since the interval $(-\infty, x_1]$ is also in $(-\infty, x_2]$ assuming $x_1 < x_2$.
3. $P(-\infty < X \leq x_1) < P(-\infty < X \leq x_2)$ assuming $x_1 < x_2$
4. If random variable X is finite then $F(x) = 0$ for all x sufficiently small and $F(x) = 1$ for all x sufficiently large. For all other cases:

$$\lim_{x \to -\infty} F(x) = 0 \quad ; \quad \lim_{x \to \infty} F(x) = 1 \tag{A.10}$$

Moreover, $dF(x)/dx$ is equal to the probability density function $f(x)$ [70].

The probability density function and the cumulative distribution function are related. Since $f(x) = dF(x)/dx$, then $\int f(x) = F(x)$. Moreover:

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_X(t)\ dt, \quad -\infty < x < \infty \tag{A.11}$$

Let $A = (a, b]$, then $P(X \in A)$ is:

$$P(a < X \leq b) = P(X \leq b) - P(X \leq a) \tag{A.12}$$

Therefore, $P(a < X \leq b)$ is represented by the area under the curve f_X between a and b.

If X is a continuous random variable then $P[X = c] = 0$ since:

$$P(X = c) = P(c \leq X \leq c) = \int_{c}^{c} f(k)\ dk = 0 \tag{A.13}$$

Consequently:

$$P(a \leq X \leq b) = P(a < X \leq b) = P(a < X < b) = F_x(b) - F_x(a) \tag{A.14}$$

Including or excluding a or b of the interval is the same since $P[X = a] = P[X = b] = 0$.

A.3 Expectation,Variance and Moments

One of the most common calculations used in performance analysis is the average or mean value in which the experimentation converges. This is called the expected value. The expected value ($E[X]$) is the weighted sum of all possible values of a random variable X as defined in [83], and is given by :

$$E[X] = \begin{cases} \sum_i x_i p(x_i) & , \quad \text{if } X \text{ is discrete,} \\ \int_{-\infty}^{\infty} x f(x) dx \ , & \text{if } X \text{ is continuous.} \end{cases} \tag{A.15}$$

The expected value is also called the average or the mean of the distribution. It is a measure of the center of mass of the probability density [66].

The variance is a measure of dispersion. Variance measures the distance between the sample x_i and the mean $E[X]$. Let σ^2 be the variance then:

$$\sigma^2 = \begin{cases} \sum_i (x_i - E[X])^2 p(x_i) & , \quad \text{if } X \text{ is discrete,} \\ \int_{-\infty}^{\infty} (x_i - E[X])^2 f(x) dx \ , & \text{if } X \text{ is continuous.} \end{cases} \tag{A.16}$$

The square root of the variance is called the standard deviation and is denoted by σ. Another measure of dispersion commonly used is the moment of a distribution. The first moment of a probability distribution is the mean. The expected value of random variable X in X^2 and X^3 are the first, second and third moments denoted by $E[X], E[X^2]$ and $E[X^3]$. The second moment is also a measure of the *spread* of the distribution. The second moment is not the same as the variance. Let σ^2 be the variance of distribution F, then $\sigma^2 = E[X^2] - (E[X])^2$. Therefore, the second moment can be found if the variance of the distribution and the expected value are known since: $E[X^2] = \sigma^2 + (E[X])^2$.

A.4 Relevant Probability Distributions

In this section the distributions that are relevant for the research are presented. A summary of the Poisson, Exponential, Lognormal and Pareto distribution is presented.

A.4.1 Poisson Distribution

The Poisson distribution is used to calculate the probability of k customers arriving in the interval of duration t with a known expected rate λ. Customer arrivals are independent of each other during the interval observed [66, 70].

The probability mass function for the Poisson distribution with shape parameter λ, and $\lambda \in \mathscr{R}$ is:

$$f(k,\lambda) = \begin{cases} (\lambda^k e^{-\lambda})/k! & \text{if } k = 0, 1 \ldots \\ 0 & \text{otherwise} \end{cases} \tag{A.17}$$

Since $\{0, 1, \ldots\}$ is a countable set, then, $f(k) = 0$ for the complement of that set. This is the same as stating, $k \notin \{0, 1, \ldots\}$. Consequently:

$$\sum_{k=0}^{\infty} \frac{(\lambda^k e^{-\lambda})}{k!} = 1 \tag{A.18}$$

This is true since the Poisson distribution is mapped by a discrete random variable and the range of $X(\omega)$ is countable. Figure A–1 presents and example of the probability mass function for the Poisson distribution.

If parameter λ determine the expected arrivals in an interval of length t, then, λ is the success rate of an arrival on that interval [66]. Therefore, the probability of k successes given by discrete random variable X is:

$$P[X = k] = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \tag{A.19}$$

The expected value of the Poisson distribution is λ and also its variance.

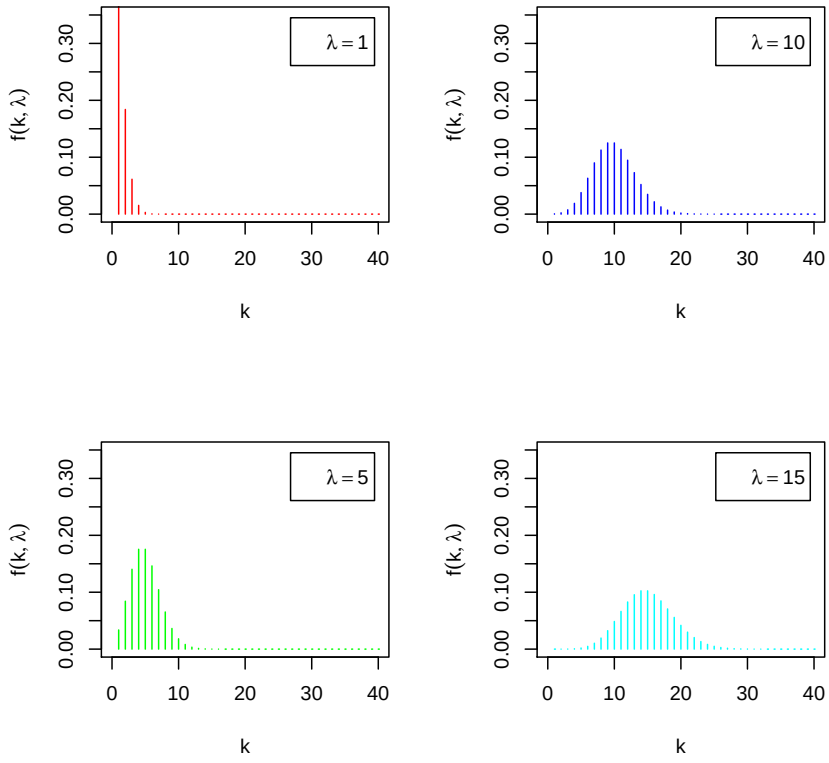


Figure A–1: Poisson Probability Mass Function

A.4.2 Exponential Distribution

The exponential distribution is used to describe the probability of the completion of a job at rate λ on an interval of time t [66]. The following random variables are modeled as exponential as presented in [70]:

1. Time between two successive job arrivals to a file server.
2. Service time at a server in a queueing network; the server could be a resource such as CPU, an I/O device, or a communication channel.
3. Time to failure or lifetime of a component.
4. Time required to repair a component that has malfunctioned.

The assertion that the above distribution are exponential is not a given fact but an assumption [70]. Experimentation should validate and verify that indeed the previous random variables could be modeled with an exponential distribution.

The exponential density function with parameters λ as the success rate in an interval of time t is given by:

$$f(t,\lambda) = \begin{cases} \lambda\ e^{-\lambda t}\ , & \text{if } t > 0, \\ 0 \qquad\quad , & \text{otherwise.} \end{cases} \tag{A.20}$$

The exponential distribution function is given by:

$$F(x,\lambda) = \begin{cases} 1 -\ e^{-\lambda x}\ , & \text{if } 0 \leq x \leq 1, \\ 0 \qquad\quad , & \text{otherwise.} \end{cases} \tag{A.21}$$

This distribution sometimes is called the negative exponential distribution. The exponential distribution its applied in queuing theory and reliability theory (see Section B). Reasons for its use include its memoryless property. This property is named memory less because no matter how long it has been since an event has started, if observed later, the distribution of time remaining since it is observed is precisely the same as it was observed since the start of the event. The process *forgot* that any time had been expended [66]. The exponential distribution has this property.

The expected value of the exponential distribution is $1/\lambda$, and its variance is $1/\lambda^2$. The second moment is then:

$$E[X^2] = \frac{1}{\lambda^2} + \left(\frac{1}{\lambda}\right)^2 = \frac{2}{\lambda^2} \tag{A.22}$$

If X is a continuous random variable whose logarithm is normally distributed, then $\ln(X)$ has a normal distribution. The lognormal distribution has a probability density function with parameters ν and σ and is given by:

$$f(t, \nu, \sigma^2) = \frac{1}{t\sigma\sqrt{2\pi}} e^{\left(-\frac{(\ln t - \nu)^2}{2\sigma^2}\right)} \tag{A.23}$$

for $t > 0$ where ν is the mean of the natural logarithm of the time to complete a job and σ is the standard deviation of the natural logarithm of the time to complete a job.

The cumulative distribution function is given by:

$$F(x, \nu, \sigma^2) = \Phi(\frac{\ln(x)}{\sigma}) \quad x \geq 0; \sigma > 0 \tag{A.24}$$

where Φ is the cumulative distribution function of the normal distribution.

The expected value for the lognormal distribution is $E[X] = e^{\nu+\sigma^2/2}$ [70]. The second moment is given by:

$$E[X^2] = e^{2(\nu+\sigma^2)} \tag{A.25}$$

A distribution is heavy-tailed if:

$$P[X > x] \sim \frac{1}{x^\alpha} \; ; \quad x \to \infty, \;\; 0 < \alpha < 2 \tag{A.26}$$

The Pareto distribution is a heavy-tailed distribution used to describe and approximate situations in which equilibrium is found in the distribution of the small to the large. On this research context, the Pareto distribution is used to approximate the file size distribution of Internet traffic. The probability density

function for the Pareto distribution with shape parameter α and scale parameter k:

$$f(t, \alpha, k) = \frac{\alpha k^{\alpha}}{t^{\alpha+1}} \quad \text{for} \quad t \geq k, \tag{A.27}$$

The Pareto distribution is power law over its entire range. The Pareto cumulative distribution function is:

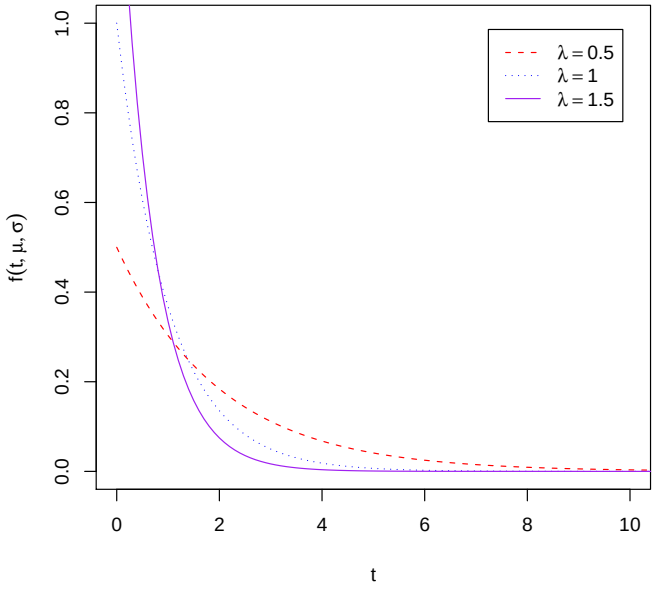
$$F(x, \alpha, k) = 1 - \left(\frac{k}{x}\right)^{\alpha} \tag{A.28}$$

The expected value of a random variable following a Pareto distribution is:

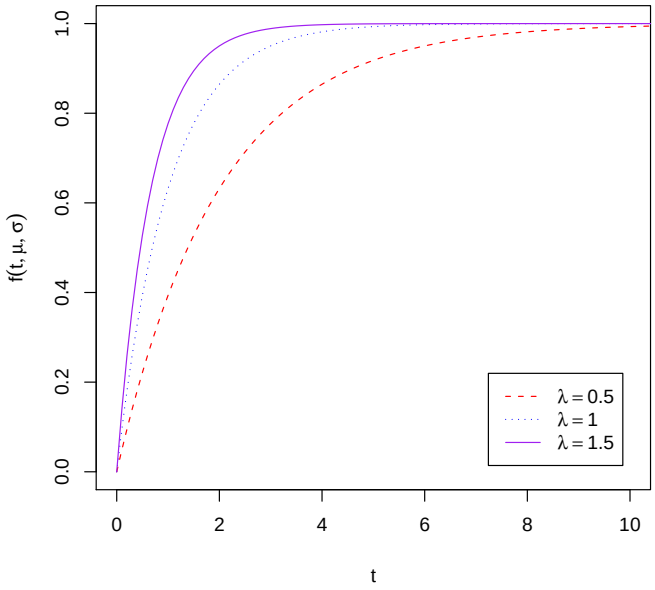
$$E[X] = \frac{\alpha k}{\alpha - 1} \tag{A.29}$$

If $\alpha \leq 1$ the expected value is infinite. Also, as said before, $E[X]$ is our first moment. The second moment of the Pareto distribution is given by:

$$E[X^2] = \frac{\alpha k^2}{\alpha - 2} \tag{A.30}$$

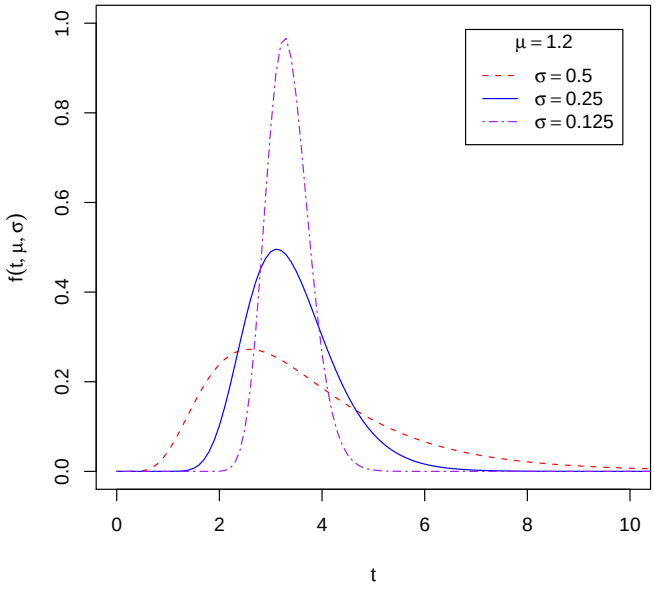


(a) Probability Density Function

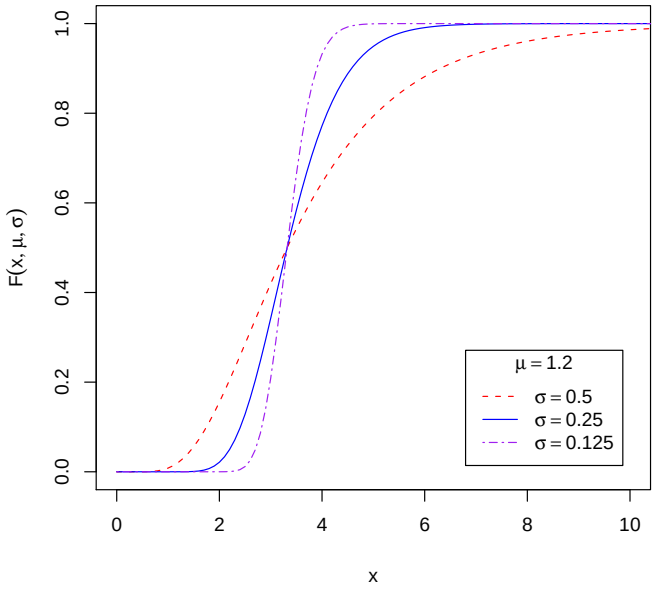


(b) Cumulative Distribution Function

Figure A–2: Exponential Distribution

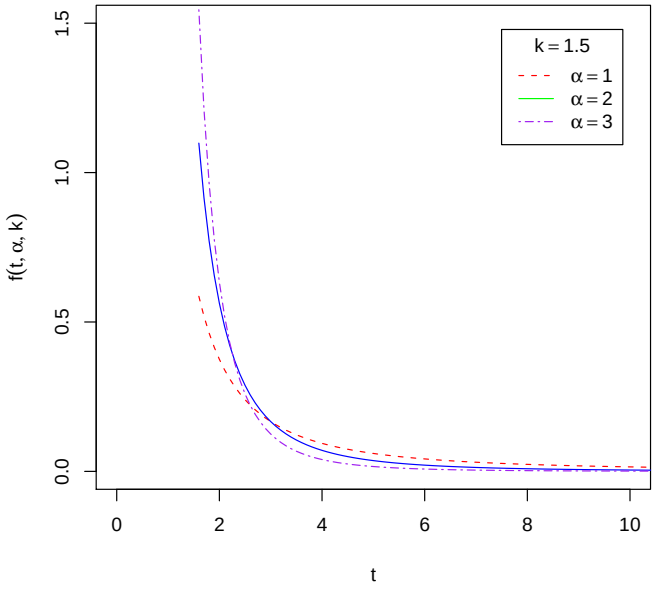


(a) Probability Density Function

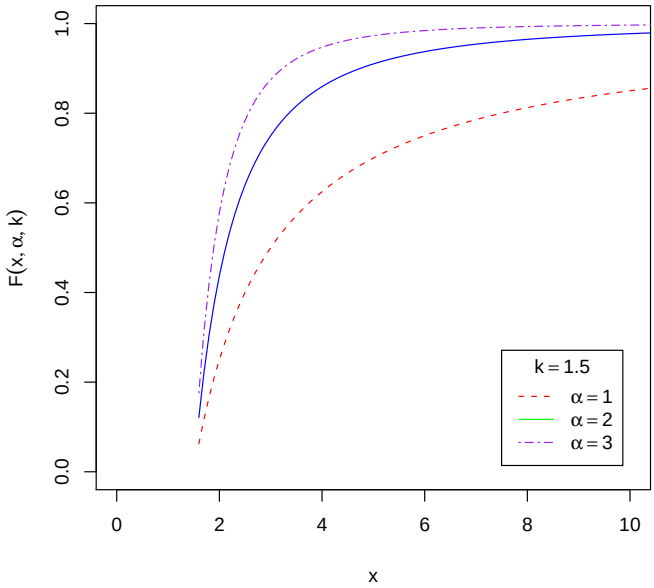


(b) Cumulative Distribution Function

Figure A–3: Lognormal Distribution



(a) Probability Density Function



(b) Cumulative Distribution Function

Figure A–4: Pareto Distribution with Location Parameter (1.5)

APPENDIX B
QUEUEING THEORY CONCEPTS

A queue models the basic concept of one or more customers that require services from one or more service centers. A model that uses Queuing Theory has also the capability of modeling the events encountered by a customer when the service centers are busy. The system is able to capture the actions performed by a customer that either waits and is serviced later or never enters the system. This type of model has a special notation that is based on five features of the queue [66] :

1. The distribution that describes the time between the arrival of customers.
2. The distribution that describes the time to service a customer.
3. The number of servers available to handle customers in the queue.
4. The number of *spaces* or slots reserved for customers that may actually wait in the queue.
5. The exact number of the population of customers that are available to enter the queue.
6. The order of removal of a customer from the queue.

This five features are the possible parameters used to model a system using queues. The notation uses letters and numbers to describe a specific queue. Let L be a letter and N be a natural number, then, the five feature notation of a queue

is of the form L/L/N/N/N. The letters are used for determining the probability distributions used to describe time. The letters used are: M for memoryless, D for deterministic, and G for general. A notation as M/G/... describes a queue with the inter-arrival time of customers described by a memory-less distribution (Poisson) and with a general distribution for describing the time it takes to service a customer.

The first N in the notation determines the amount of service centers available to handle a customer. Sometimes, an arriving customer could not be serviced immediately. This happens when the server is busy. The customer must wait until the customer been serviced by the center departs the system. The second N determines the number of available *slots* that a queue has for allocating waiting customers when the server is busy. If an arriving customer could not find an available slot, then, the customer never enters the system. The term *drop* is used to define this event. The last N in the notation give the total number of customers in the population. This last N is rarely used. The notation is simplified for some cases. If a queue is defined as an M/M/1 in fact is defined as an M/M/1/∞/∞/. However, ∞ is omitted [66, 70].

The service discipline specifies the order in which customers are selected from the queue. These are: first comes first serve (FCFS) or also called first in first out, last comes first serve (LCFS) or also called last in first out (LIFO), round robin (RR), processor sharing (PS), priority and more. The service discipline used for the probabilistic model presented is FCFS. The other disciplines can be found in [66, 70, 84].

A queue has a notation and a graphical representation. Figure B–1 presents the graphical representation of a queue. Customers arrive from the left directly

into the queue. The circle represents the service center and there could be more than one[1] . The arrow that exits the circle represents the exit point of the queue.

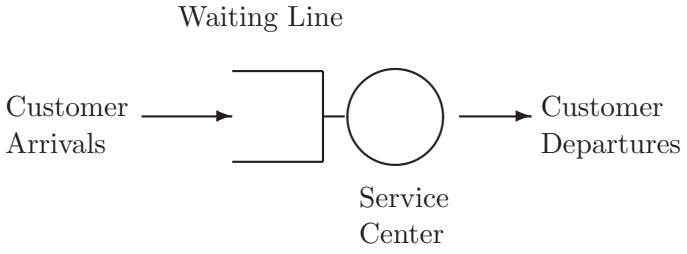


Figure B–1: Graphical Representation of a Queue

Queuing Theory is used to study performance measures based on input parameters. Among these measures are:

- The percent of time a server is busy.
- The mean time a customer waits in a system.
- The mean time a customer waits before been serviced.
- The average number of customers waiting to be serviced.
- The average number of customers inside a system.
- The average number of customers served by a system.
- The number of customers unable to be serviced.

The percent of time the server is busy is known as the utilization. Let B be this time and T the total time the system was observed. Then, the utilization is $U = B/T$ [84]. In queuing theory, utilization is calculated differently. Let λ be the arrival rate of *customers* to a service center. Let X be a random variable

[1] Figure B–1 only presents one service center

that represents the service time a *customer* spends on the service center. The utilization (ρ) is obtained by multiplying λ by the expected service time of center ($E[X]$), then, $\rho = \lambda E[X]$. The expected service time, by definition, is the inverse of the service rate and this rate is denoted by μ [66]. Then, $E[X] = 1/\mu$ and $\rho = \lambda E[X] = \lambda/\mu$ [66, 70].

The M/M/1 notation is a simplification used to described a queue with an arrival rate described by a Poisson distribution with service time described by an Exponential distribution and only one service center. The first M determines the distribution used to describe the inter-arrival times. The inter-arrival times of a Poisson arrival process are exponentially distributed with mean $1/\lambda$.

The queue is actually an M/M/1/∞/∞/FCFS queue. Therefore, the number of customers that can be waiting for service and the total population are infinite. The default service discipline is the FCFS.

The utilization of a service center in an M/M/1 queue is calculated as $\rho = \lambda E[X]$ where X is described by an Exponential distribution. Consequently, $E[X] = 1/\mu$ and μ is the service rate of the service center.

The expected time in the system on an M/M/1 queue is given by the equation:

$$W = \frac{1}{\mu - \lambda} = \left(\frac{1}{\lambda}\right)\frac{\rho}{1-\rho} \tag{B.1}$$

The expected time that a customer waits to be service after its arrival is obtained by subtracting the expected service time from Equation B.1:

$$W_q = W - E[X] = \frac{1}{\mu - \lambda} - \frac{1}{\mu} = \frac{\rho}{\mu - \lambda} \tag{B.2}$$

Using Little's Law the expected number of customers is:

$$N_s = \lambda W = \lambda \left(\frac{1}{\lambda}\right) \frac{\rho}{1-\rho} = \frac{\rho}{1-\rho} \tag{B.3}$$

The expected number of customers in waiting to be serviced is obtained also with Little's Law:

$$N_q = \lambda W_q = \frac{\rho^2}{1-\rho} \tag{B.4}$$

The M/M/... systems are tractable due to the memoryless property of the service times. There are situations where random variable X is not approximated by an Exponential distribution. The M/G/1 queue, like the M/M/1 queue has a Poisson arrival process, but it allows a general distribution for describing the service times. The distribution that describes the service time have to be identically distributed, mutually independent and independent of inter-arrival times.

The Pollaczek-Khinchin formula, also known as the P-K formula, provides some performance metrics for the M/G/1 queue [66]. The P-K formula is based on the analysis of the residual service time. This is the residual time seen by a customer β of another customer α that is been served upon the arrival of β. Let W_q be the average time a customer waits in line before entering service. Let N_q the average customers in the system waiting to be served. Let $E[X]$ be the expected service time. If R is the average residual time of the customer been in service, then: $W_q = R + N_q E[X]$. Applying Little's Law $N_q = \lambda W_q$, then: $W_q = R + (\lambda W_q)E[X]$. Then: $W_q - (\lambda W_q)E[X] = R$. Symplifying: $W_q(1 - \lambda E[X]) = R$, but $\lambda E[X] = \rho$. Substituting: $W_q(1-\rho) = R$. Finally:

$$W_q = \frac{R}{1-\rho} \tag{B.5}$$

The expected residual time (R) is obtained by analyzing the graph generated by the random variable that describes the service time (see Figure B–2). Let $r(t)$ be the residual time at time t and $m(t)$ is the number of service completions up to time t.

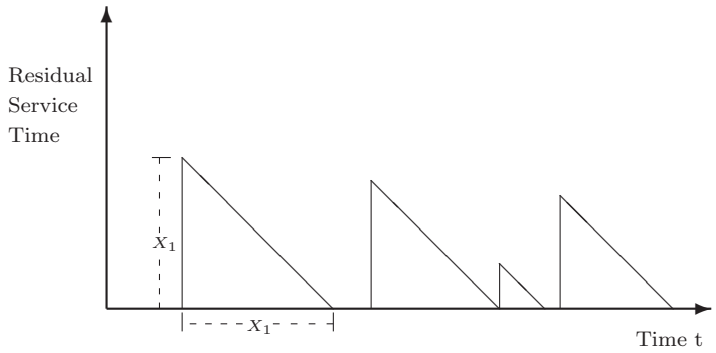


Figure B–2: Graphical Representation of Residual Time

The distribution of the residual time is obtained by computing the integral of the graph presented in Figure B–2. The time average of $r(t)$ in the interval $(0, t)$ is given by:

$$\frac{1}{t}\int_0^t r(s)ds = \frac{1}{t}\sum_{i=1}^{m(t)} \frac{1}{2}X_i^2 \tag{B.6}$$

The CDF of this distribution is obtained using the area bellow the curve. This area is represented by a series of triangles. The equation is then multiplied and divided by $m(t)$ to obtain:

$$\frac{1}{t}\sum_{i=1}^{m(t)} \frac{1}{2}X_i^2 = \frac{1}{2}\left(\frac{m(t)}{t}\right)\left(\frac{\sum_{i=1}^{m(t)} X_i^2}{m(t)}\right) \tag{B.7}$$

The expected residual time converges when $t \to \infty$. Therefore:

$$\lim_{t\to\infty}\frac{1}{t}\int_0^t r(s)ds = \frac{1}{2}\left(\lim_{t\to\infty}\frac{m(t)}{t}\right)\left(\lim_{t\to\infty}\frac{\sum_{i=1}^{m(t)}X_i^2}{m(t)}\right) \tag{B.8}$$

The first limit ($\lim_{t\to\infty}\frac{m(t)}{t}$) converges to λ when the system is in equilibrium ($\rho < 1$). The second limit ($\lim_{t\to\infty}\frac{\sum_{i=1}^{m(t)}X_i^2}{m(t)}$) converges to the average of all the X_i^2 and this is $E[X^2]$. Therefore:

$$R = \frac{1}{2}\lambda E[X^2] \tag{B.9}$$

The expected waiting time in the queue before been serviced is obtained by substituting in the P-K formula:

$$W_q = \frac{\lambda E[X^2]}{2(1-\rho)} \tag{B.10}$$

The second moment of random variable X is needed to calculate W_q. The expected total time in the system is obtained by adding the expected service time:

$$W = W_q + E[X] = E[X] + \frac{\lambda E[X^2]}{2(1-\rho)} \tag{B.11}$$

The expected number of customers in the queue is given by using the P-K formula and Little's Law:

$$N_q = \lambda W_q = \frac{\lambda^2 E[X^2]}{2(1-\rho)} \tag{B.12}$$

Consequently, the expected total number in the system is given by: $N = \lambda W = \lambda(E[X] + W_q)$ then:

$$N = \lambda E[X] + \frac{\lambda^2 E[X^2]}{2(1-\rho)} = \rho + \frac{\lambda^2 E[X^2]}{2(1-\rho)} \tag{B.13}$$

APPENDIX C
KOLMOGOROV-SMIRNOV TEST

The Kolmogorov-Smirnov test (K-S) is a non-parametric test of minimum distance estimation. The K-S test is used for computing the distance between the empirical distribution of the sample and a known cumulative distribution function. The K-S test is a distribution free test that can be used for computing the distance between the empirical distribution functions of two samples.

The empirical distribution function also called the cumulative fraction function is a step function defined by:

$$P[X_i < x] = F(x) = \frac{1}{n} \sum_{i=1}^{n} I(X_i < x)$$

where $I(X_i < x)$ is a characteristic function defined on a set X that indicates membership of the element x in A and $A \subseteq X$. This function is defined as:

$$I_A(x) = \begin{cases} 1 & : \quad \text{if } x \in A \\ 0 & : \quad \text{if } x \notin A \end{cases}$$

Figure C–1 presents the empirical distribution of A.

The null distribution of this statistic is calculated under the null hypothesis that the samples are drawn from the same distribution (in the two-sample case) or that the sample is drawn from the reference distribution (in the one-sample case).

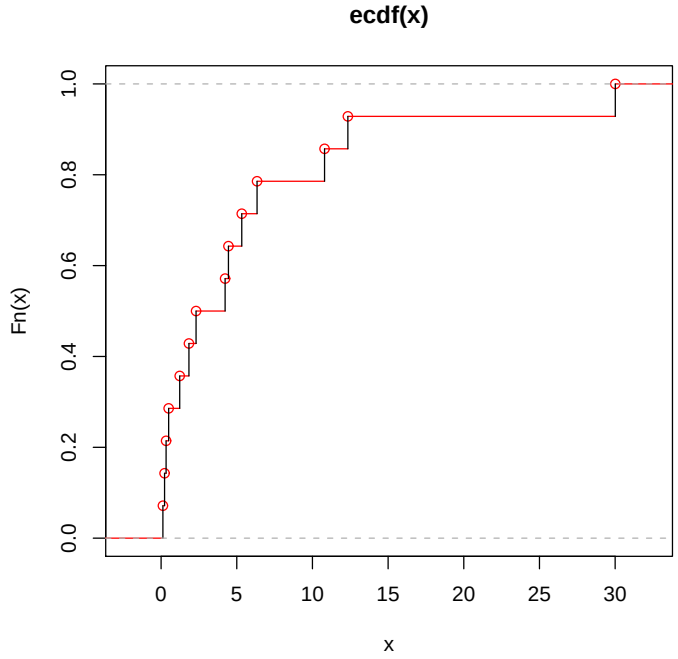


Figure C–1: Empirical Distribution Function Example

In each case, the distributions considered under the null hypothesis are continuous distributions but are otherwise unrestricted.

The probability distributions of these two curves, given a null hypothesis that both distributions were plotted from the same distribution, does not depend on what the hypothesized distribution is, as long as it is continuous. The KS-test is a robust test that cares only about the relative distribution of the data.

The KS-test uses the maximum vertical deviation between the two curves as the statistic D. This is obtained by finding the maximum vertical distance between two distributions. Let $F_a(x)$ and $F_b(x)$ be empirical cumulative distribution functions of two samples, then, the maximum vertical distance (D^+_-) is:

$$D_{-}^{+} = \sup_{x} |F_a(x) - F_b(x)|$$

Where sup is the supremo. The maximum vertical distance can be viewed graphically when the two empirical distributions are plotted. Figure C–2 presents graphically the maximum vertical deviation between the two distributions.

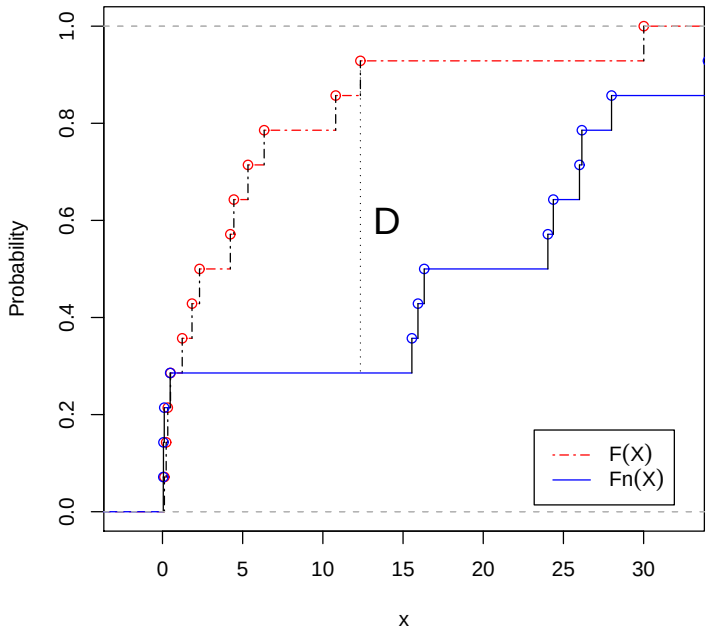


Figure C–2: Maximum Vertical Distance Between Empirical Distributions

The null hypothesis is accepted when the p-values obtained are greater than 0.20. The p-value is obtained by the following equation:

$$p = \sqrt{\frac{ab}{a+b}} D_{a,b} \tag{C.1}$$

.

Glossary of Terms

10GEA : the 10 Gigabit Ethernet Alliance. An alliance formed by corporations that aims to define the 10 Gigabit Ethernet standard.

Active Server Page (ASP) : is an HTML page that includes one or more scripts that are processed exclusively on a Microsoft Web server before the page is sent to the user. An ASP is somewhat similar to a server-side include or a common gateway interface (CGI) application in that all involve programs that run on the server, usually tailoring a page for the user.

Address : An integer value represented in hexadecimal numbers used to identify a particular node. The address must appear in every packet residing on the medium and is used to identify for whom and to whom the packet is destined for.

Application Layer : This is the upper layer protocol that defines the interface with the user. It is labeled as layer 7 of the OSI model.

bps : Bits per Second.

b/s : a formal notation for bits per second.

Broadcast : A way of transmitting a packet over a network that is capture by all the hosts attached to the network.

CGI : acronym for Common Gateway Interface.

Checksum : An error-detection code based on the summation of all the octets within a packet.

Circuit-Level Multi Processor (CMP) : An integrated circuit that combines two or more independent CPUs into a component that enclosed them together. This single component is called a die. A CMP is also called a chip-level multiprocessor.

Common Gateway Interface (CGI) : is a standard for external gateway programs to interface with information servers such as HTTP servers. A CGI program is any program designed to accept and return data using a Web server. The program could be written in any programming language been the most common Perl.

Congestion : A situation in which the load imposed by the amount of traffic on the network saturates it.

Connection : An abstraction provided by protocol software that logically links more than one host in an end-to-end communication.

Connection-oriented Protocol : A protocol that exchanged data by establishing a logical connection between the end-points involved.

Connectionless Protocol : A protocol that exchanged data informally without previous coordination between the parties involved.

CPU : stands for Central Processing Unit. A general-purpose processor that is responsible for running the operating system.

CR : acronym for Commander and Reader. This is the most important process of the TOE-Em and is used for reading and writing commands to and from the offload engine and the Web Server.

DDP : an acronym for Direct Data Placement. See *direct data placement*.

Datagram : An IP packet. The basic unit of information passed across the Internet. Each datagram contains the source and destination addresses as well as data.

De-capsulate : Is the process in which an agent extracts the data contents inside the packet destine to it.

De-multiplex : Is the process used by IP to extract incoming datagrams, sending each piece of information extracted to the appropriate upper layer protocol module or application program.

DHCP : Dynamic Host Configuration Protocol. Nodes running DHCP does not own a static IP address. These nodes consult the server and the server assigns an IP address usually at boot time. (See also *BOOTP*).

Direct Data Placement (DDP) : An Upper Layer Protocol proposed as a way for placing data directly into a buffer linked to a specific application. This mechanism place data directly into memory without CPU intervention. This mechanism is currently proposed by the RDMA consortium.

empirical : originating in or based on observation or experience. Relying on experience or observation alone often without due regard of system and theory. Capable of being verified or disproved by observation or experiment.

Encapsulation : The process in which a lower level protocol accepts a message from a higher level protocol and places it in the data portion of the low-level frame.

End-to-end : Feature of any delivery mechanism that operates only on the source and final destination.

Ethernet : Ethernet is a frame carrier protocol. Ethernet was design by XEROX at the Palo Alto Research Center and is a best-effort delivery system. Ethernet have a peak speed of 10 Mbps.

FCFS : an acronym for First Come First Serve. The first customer arriving at queue Q is the one who will be served first. A synonym for FIFO.

File Server : A hosts that provides access to files within the host file system to remote nodes that are connected to it.

Flow : A general term used to characterize a sequence of packets sent from a source to a destination.

Flow Control : Control of the rate at which hosts or routers inject packets into the Internet to avoid congestion.

Forwarding : The process of accepting an incoming packet and sending the packet to another node that is part of the path from source to destination.

Fragmentation : The technique of splitting a datagram into parts for the purpose of *fitting* them into the data portion of a lower layer that is smaller than the length of the whole datagram.

Frame : A formal term used to described a packet that is transmitted accross a serial line. A packet described in the data-link layer context.

Frame Carrier : The agent responsible of placing frames on the medium and extracting them when they match is physical address. Typically, the lower layer of a protocol stack. Examples: Ethernet, Token Ring, Appletalk.

FTP : The File Transfer Protocol. An upper layer protocol used to transfer files from source to destination on the end-to-end communication.

FP : acronym for Forker Process. Is a process of the TOE-Em entitled of spawning and removing slave process.

Full-duplex : A mechanism that allows simultaneous transfer of data in two directions.

Gateway : Any technology that interconnects two or more systems and translate among them. Dedicated nodes that route datagrams from one network to another. A synonym for *router*.

Gbps : Gigabits per second.

Gb/s : a formal notation of Gbps.

GPPs : acronym for General Purpose Processors.

Gigabit Ethernet : Ethernet with peak speeds of 1Gbps and 10Gbps

Hack : It is a term that refers to a clever fix to a computer program problem. A *hack* refers to a way of altering a computer program beyond of its original design goals.

Header : Information at the beginning of the packet that describes the contents and specifies the source and destination. Necessary overhead placed by the upper layer protocols to achieved the end-to-end communication.

Host : Any end-user processing node that connects to a network and provides a service. Is a formal term for describing a computer node rather than a router or gateway.

HTTP : acronym for Hypertext Transfer Protocol is an upper layer protocol (application protocol) that provides a standard for Web browsers and web server to communicate.

IETF : The Internet Engineering Task Force. A group of people who work on the design and engineering of TCP/IP and the global Internet.

Intranet : A private network consisting of hosts and routers that use TCP/IP as their primary protocol for achieving end-to-end communication.

IP : the Internet Protocol.

IPC : acronym for Inter-Process Communication (see System V IPC).

IP address : A 32-bit unique address assigned to each hosts that identifies each participant in a TCP/IP network.

iSCSI : Also called Remote SCSI. This is a term used for host-independent SCSI device with stand-alone communication capabilities. Most of the times the term refers to SCSI device that are capable of sending datagrams using a net device attached to them (see SAN).

iNIC : acronym for Intelligent Network Interface Card. An iNIC is a network interface with programmable capabilities than the traditional network interface card does not have.

Jumbo Frames : Frames with payloads greater than 1500 octets. This term is used formaly for defining the Gigabit Ethernet frames with 9000 octet payloads.

Kbps : 1000 bits per second. Acronym for Kilobits per second.

Kb/s : a formal notation of Kbps.

KBps : 1024 bytes per second. Acronym for Kilobytes per second.

KB/s : a formal notation of KBps.

LAN : Local Area Network. A private corporate network that only spans through a few buildings or blocks in a city.

LLP : acronym for Lower Layer Protocols.

MAC Address : A synonym for the physical address used by Ethernet, Fast Ethernet and Gigabit Ethernet.

MTU : Maximum Transmission Unit. The largest amount of data space within a frame that can be transfered across a given physical network. The MTU is determined by the physical layer.

Mbps : One million of bits per second.

Mb/s : a formal notation for Mbps.

MBps : 1,048,576 bytes per second.

MB/s : a formal notation for Mbps.

MSS : Maximum Segment Size. The MSS is the largest amount of data that can be transmitted inside a TCP segment. The MSS is negotiated between the sender and receiver on the initial handshake of the end-to-end communication.

Multi-homed Host : A host using TCP/IP that has more than two net devices and connects to various physical networks.

NIC : the Network Interface Card. Is a hardware device responsible of sensing the medium and extract any frame destine to it.

NP : acronym for Network Processors. A device that is capable of processing the network code either by its own ASICs or by using an embedded operating system.

Octet : An 8-bit unit of data. A synonym for an 8-bit byte but within the networking context. An octet either can be used to identify a data unit or a communication unit of 8 bits.

Offload Engine : An entity outside the main CPU of the host that is a specific purpose processor use for reducing the utilization of the CPU by processing specific commands. This aids the system in processing its workload.

Packet : Used to refer to a small block of data sent accross packet switching networks.

Path Length : A measure of the number of lines of code required to process every datagram through the IP stack code.

Payload : The data contents of any packet used by a protocol. This term is typically used for the data section of the frame carrier.

Process Identifier (PID) : A label used by some operating systems to idetify a process.

Protocol Offload Engine : An offload engine that reduces the utilization of the CPU by processing the protocols used in and end-to-end communication.

Physical Address : see *MAC Address*.

Port : The abstraction used by some protocols to distinguish among multiple *sockets* within a given host.

RDMA : Remote Direct Memory Access. A node capable of using RDMA can place data directly into the memory of a remote node without CPU intervention.

RFC : Request for Comments. The acronym for a series of notes that contains surveys, measurements, ideas, techniques and observations, as well as proposed and accepted TCP/IP protocol standards.

Route : A route is the path that a packet traverses from source to destination.

Router : see *gateway*.

Segment : A TCP packet. The unit of transfer sent from the TCP module at the sender and is destine to the TCP module of the receiver at some port.

Self-similar : is a statistical property in which the characteristics for the entire data set are the same for sub-sections of the data set. For example, the two

halves of the data set have the same statistical properties as the entire data set.

Server : A processing node that serves costumers inside a network or queues.

Servlet :A network application written on JAVA that runs in a Web server and provides server-side processing such as accessing a database and e-commerce transactions.

Sliding Window : Is the number of packets that the sender will transmit without waiting for the reception of an acknowledge.

SP : acronym for Slave Process. Is a subprocess of the TOE-Em that handles the socket interface layer for a matching process on the WS. It can only be triggered by the commander and reader process of the TOE-Em.

System V IPC : is a set of rules and functions for the exchange of data between one or more processes. IPC is divided into methods for message passing, synchronization, shared memory, and remote procedure calls.

TCP : Transmission Control Protocol. A transport layer protocol that provides the reliable, connection-oriented, full duplex, stream service on which many applications depends.

TCP/IP Protocol Suite : The formal name of the protocols forms by TCP and IP.

TELNET : The TCP/IP standard protocol for terminal service.

Three-way Handshake : The segment exchange used by TCP to reliably start or gracefully terminate a connection.

TOE : stands for TCP/IP Offload Engine. An entity outside main CPU that is capable of processing the TCP and IP protocols.

TOE-Em : a TCP Offload Engine Emulator. A program that mimics the functionality of an Offload Engine.

UDP : User Datagram Protocol. The protocol that allows an application program on one machine to send a datagram to an application program on another machine. UDP is simpler than TCP and does not provide congestion control mechanisms.

ULP : Upper Layer Protocol. High level protocol.

Web Server (WS) : A host that provides Web pages via the HTTP protocol.

Window : See *sliding window*.

Window Advertisement : A positive integer value used by the receiver TCP module to advertise to the sender the amount of data that can be received and stored in a buffer.

BIBLIOGRAPHY

[1] Patricia Gilfeather and Arthur Maccabe. Splintering TCP. In *Proceeedings of the 17th International Symposium on Computer and Information Sciences ISCIS '02*, pages 36–40, Orlando, FL (USA), October 2002.

[2] Patricia Gilfeather, Todd Underwood, and Arthur Maccabe. Fragmentation and high performance IP. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01) of the IEEE Computer Society*, pages 23–27, San Francisco, CA, April 2001.

[3] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinski, Andrew L. Schultz, and Steven K. Reinhardt. Performance analysis of system overheads in TCP/IP workloads. In *ACM PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 218–230, Washington, DC, USA, September 2005. IEEE Computer Society.

[4] HyongYoub Kim and Scott Rixner. Connection handoff policies for TCP offload network interfaces. *Operating Systems Design and Implementation*, 1(1):293–306, November 2006.

[5] Roland Westrelin, Nicolas Fugier, Erik Nordmark, Kai Kunze, and Eric Lemoine. Studying network protocol offload with emulation: approach and preliminary results. In *IEEE HOTI'04 Proceedings of the 12th Annual IEEE Symposium in High Performance Interconnects*, pages 84–90, Washington, DC, USA, 2004. IEEE/ACM Computer Society.

[6] Tim Brecht, G. (John) Janakiraman, Brian Lynn, Vikram Saletore, and Yoshio Turner. Evaluating network processing efficiency with processor partitioning and asynchronous I/O. *Operating Systems Review*, 40(4):265–278, October 2006.

[7] Yeh Eric, Herman Chao, Venu Mannem, Joe Gervais, and Bradley Booth. Introduction to TCP/IP offload engine (TOE). White Paper, 2002. affiliations : 10 GEA, Intel Corp., Hewlett Packard Coorp., Adaptec, Qlogic, Alacritech Inc.

[8] Juan Solá-Sloan and Isidoro Couvertier. A TCP offload engine emulator for estimating the impact of removing protocol processing from a host running Apache HTTP server. In *Proceedings of the ACM SIGSIM SCS CNS'09 12th Communications and Networking Simulation Symposium*, pages 1–5, March 2009.

[9] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of HostOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 25–30, Hawaii, May 2003.

[10] Piyush Shivam and Jeffrey S. Chase. On the elusive benefits of protocol offload. In *Proceedings of the ACM SIGCOMM on Network-I/O convergence*, pages 179–184, Karlsruhe, Germany, August 2003. ACM. Special Session of Promises and Reality.

[11] Krishna Kant. TCP offload performance for front-end servers. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM '03) 2003*, volume 6, pages 3242–3247, December 2003.

[12] Patricia Gilfeather and Arthur Maccabe. Modeling protocol offload for message-oriented communication. In *Proceedings of the IEEE International*

Conference on Cluster Computing, pages 1–10, Boston, MA (USA), September 2005.

[13] Michael Mitzenmacher. Dynamic models for file sizes and double Pareto distributions. *Internet Mathematics*, 1(3):305–333, March 2004.

[14] Allen B. Downey. The structural cause of file size distributions. In *IEEE MASCOTS 01 Proceedings 9th International Symposium Modeling Analysis and Simulation of Computer and Telecommunication Systems*, pages 361–370, Cincinnatti, OH, USA, August 2001. IEEE Computer Society.

[15] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. *ACM SIGMETRICS Performance Evaluation Review*, 26(1):151–160, 1998.

[16] Jan Hanning, Gennady Samorodnitsky, J.S. Marron, and F.D. Smith. Log-normal durations can give long range dependence. *(IMS) Institute of Mathematical and Statistics Journal*, 42(1):333–344, March 2005.

[17] Vern Paxson and Sally Floyd. Wide area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[18] Yiming Hu, Ashwini Nanda, and Qing Yang. Measurement, analysis and performance improvement of the Apache web server. In *Proceedings of the IEEE IPCCC '99 Performance Computing and Communications Conference*, pages 261–267, February 1999.

[19] Juan M. Solá-Sloan and Isidoro Couvertier. A parallel TCP/IP offloading framework for TOE devices. In *Proceedings of the SCS Applied Telecommunication Symposium (ATS'03) 2003*, pages 115–121, Orlando, FL, March 2003.

[20] Andrés Ortiz, Julio Ortega, Antonio F. Diaz, and Alberto Prieto. Analyzing the benefits of protocol offload by full-system simulation. In *IEEE PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 229–237, Napoli , Italy, February 2007. IEEE Computer Society.

[21] Roland Westrelin, Nicolas Fugier, Erik Nordmark, Kai Kunze, and Eric Lemoine. Studying network protocol offload with emulation: approach and preliminary results. In *HOTI '04: Proceedings of the High Performance Interconnects, 2004. on Proceedings. 12th Annual IEEE Symposium*, pages 84–90, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram A. Saletore, and Annie Foong. ETA: Experience with an Intel Xeon processor as a packet processing engine. *IEEE Micro Journal*, 24(1):24–31, 2004. Affiliations: Intel Corporation.

[23] Kalpana S. Banerjee. *TCP Servers: A TCP/IP Offloading Architecture For Internet Servers, Using Memory-Mapped Communication*. PhD thesis, Rutgers Universiry, Piscataway NJ, 2002. Department of Computer Science.

[24] Juan Solá-Sloan and Isidoro Couvertier. Are the benefits of TCP offload concealed? Under review [available upon request] http://www.geocities.com/juan_sola/publications.html, November 2008.

[25] Aleksandar Tudjarov, Dusko Temkov, Toni Janevski, and Ognen Firfov. Empirical modeling of Internet traffic at middle-level burstiness. In *IEEE MELECON '04: Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference*, volume 2, pages 535–538, May 2004.

[26] Mark E. Corvella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[27] Daniele Miorandi, Arzad A. Kherani, and Eitan Altman. A queueing model for HTTP traffic over IEEE 802.11 WLANs. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 50(1):63–79, 2006.

[28] Arzad A. Kherani and Anurag Kumar. On processor sharing as a model for TCP controlled HTTP-like transfers. *IEEE Communications*, 4(1):2256–2260, 2004.

[29] Liang Guo, Mark Crovella, and Ibrahim Matta. How does TCP generate pseudo-self-similarity? In *IEEE/ACM MASCOTS' 01 Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 215–223, August 2001.

[30] Jianhua Cao, Mikael Andersson, Christian Nyberg, and Maria Kihl. Web server performance modeling using an M/G/1/K*PS queue. In *IEEE ICT 2003: 10th International Conference on Telecomunications*, volume 2, pages 1501–1506, March 2003.

[31] Robert D. Van der Mei, Rema Hariharan, and Paul Resser. Web server performance modeling. *Journal of Telecommunication Systems*, 16(4):367–378, March 2001.

[32] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial Web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[33] Toni Janevski. *Traffic Analysis and Design of Wireless IP Networks*. Artech House, Inc., Norwood, MA, USA, 2003.

[34] Wei-Bo Gong, Yong Liu, Vishal Misra, and Don Townsley. Self-similarity and long range dependence on the Internet: a second look at the evidence, origins and implications. *Journal of Computer Networks*, 48(3):377–399, June 2005.

[35] Ronald G. Addie, Timothy D. Neame, and Moshe Zukerman. Performance evaluation of a queue fed by a Poisson Pareto Burst Process. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 40(3):377–397, 2002.

[36] David M. Nicol and Guanhua Yan. Simulation of network traffic at coarse timescales. In *IEEE PADS '05 Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, pages 141–150, Los Alamitos, CA, USA, June 2005. IEEE Computer Society.

[37] Konstantinos Christodoulopoulus, Emmanouel Varvarigos, and Kyriakos Vlachos. A new burst assembly scheme based on the average packet delay and its performance for TCP traffic. *Optical Switching and Networking*, 5(4):200–212, November 2007.

[38] Fatih Haciomeroglu and Tulin Atmaca. Impacts of packet filling in an optical packet switching architecture. In *AICT-SAPIR-ELETE '05 Proceedings of the Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/E-Learning on Telecommunications Workshop*, pages 103–108, Washington, DC, USA, July 2005. IEEE Computer Society.

[39] Michael Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, March 2004.

[40] A. Feldman, A. Gilbert, and W. Willinger. Data networks as cascades : Explaining the multi-fractal nature of Internet WAN traffic. In *Proceedings of ACM SIGCOMM '99*, 1999.

[41] Will E. Leland, Murad S. Taqqu, R Sherman, and Daniel V. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. In *Proceedings of ACM SIGCOMM '95*, pages 100–113, 1995.

[42] Mikael Andersson, Jianhua Cao, Maria Kihl, and Christian Nyberg. Performance modeling of an Apache Web server with bursty arrival traffic. In *IC '03 Proceedings of International Conference on Internet Computing*, pages 508–514, June 2003.

[43] Brent Tworetzky. *An Analysis of Web File Sizes: New Methods and Models.* PhD thesis, Hardvard Universiry, Cambridge, Massachusetts, 2003.

[44] Chang Heng Foh, Moshe Zukerman, and Juki Wirawa Tantra. A Markovian framework for performance evaluation of IEEE 802.11. *IEEE Transactions on Wireless Communications*, 6(4):1276–1285, April 2007.

[45] Carey Williamson. Internet traffic measurements. *IEEE Computer*, 5(6):70–71, December 2001.

[46] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server network scalability and TCP offload. In *ACM ATEC '05 Proceedings of the 2005 USENIX Annual Technical Conference*, pages 209–222, Berkeley, CA, USA, April 2005. ACM USENIX

Association.

[47] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Storage over IP: When does hardware support help? In *Proceedings of FAST '03 2nd Conference File and Storage Technologies*, pages 231–244, San Francisco, CA, March 2003.

[48] Patricia Gilfeather and Arthur Maccabe. Connection-less TCP. In *IEEE IPDPS'05 Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–15, Denver, Colorado USA, April 2005.

[49] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network I/O. *Computer Architecture*, 33(2):50–59, 2005.

[50] David Clark, Van Jacobson, John Romkey, and Howard Salwen. An anlysis of TCP processing overheads. *IEEE Communications*, pages 23–29, June 1989.

[51] Johnathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of ACM SIGCOMM '93*, pages 259–268, 1992.

[52] Douglas E. Comer. *Computer Newtworks and Internets*. Prentice Hall, Upper Saddle River, New Jersey, fifth edition, 2008.

[53] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of interrupt coalescence on network measurements. *Lecture Notes in Computer Science*, 30(15):247–256, 2004.

[54] Mohammad Peyravian, Gordon Davis, and Jean Calvignac. Search engine implications for network processor efficiency. *IEEE Network*, 17(4):12–20, August 2004. IBM Corporation.

[55] Erich Nahum, Tsipora Barzilai, and Dilip D. Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(1):2–11, February 2002.

[56] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP onloading for data center servers. *IEEE Computer*, 37(11):48–58, 2004. author affiliation: Intel Coorporation.

[57] Hyong youb Kim and Scott Rixner. TCP offload through connection handoff. *Journal of the ACM SIGOPS Special Interest Group in Operating System Review*, 40(4):279–290, 2006.

[58] Jeffrey Mogul, Lawrence Brakmo, David E. Lowell, Dinesh Subhraveti, and Justin Moore. Unveiling the transport. *ACM Computer Communication Review*, 34(1):99–106, 2004.

[59] G. Trent and M. Sake. Webstone: The first generation in HTTP server benchmarking. White Paper, February 1995.

[60] Windriver Corp. Tornado for intelligent network acceleration, 2001. [available online] http://www.windriver.com.

[61] Xubin He and Qing Yang. Performance evaluation of distributed Web server architectures under E-Commerce workloads. In *Proceedings of the First International Conference in Computing IC'2000*, pages 334–340, June 2001.

[62] Vern Paxson and Sally Floyd. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, August 2001.

[63] Tim O'Reilly. What is Web 2.0 : Design patterns and business models for the next generation of software. *Social Science Research Network Working Paper Series*, pages 1–5, March 2007.

[64] Xu Cheng, Cameron Dale, and Jiangchuan Liu. Statistics and social network of YouTube videos. In *IEEE IWQoS 2008 16th International Workshop on Quality Service*, pages 229–238, June 2008.

[65] M. Soraya, M. Zamani, and A. Abhari. Modeling of multimedia files on the Web 2.0. In *Proceedings of the 21th annual IEEE Canadian Conference on Electrical and Computer Engineering*, pages 100–105, May 2008.

[66] Michael K. Molloy. *Fundamentals of Performance Modeling*. Macmillian Publishing Company, New York, 1989.

[67] Glenn Judd and Peter Steenkiste. Repeatable and realistic wireless experimentation through physical emulation. *ACM SIGCOMM Computer Communication Review*, 34(1):63–68, January 2004.

[68] Sheetalkumar R. Doshi, Unghee Lee, and Rajive L. Bagrodia. Wireless network testing and evaluation using real-time emulation. *ITEA Journal of Test and Evaluation*, 3(7):1–16, June 2007.

[69] Doshi Sheetalkumar, Unghee Lee, Rajive Bagrodia, and Douglas McKeon. Network design and implementation using emulation-based analysis. In *IEEE MILCOM 2007 Proceedings of the 26th IEEE Military Communications Conference*, pages 1–8, Orlando, Florida USA, October 2007.

[70] Kishor S. Trivedi. *Probability and Statistics with Reliability Queuing and Computer Science Applications*. Wiley Inter-Science, New York, 2002.

[71] Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*, volume 1. Prentice Hall, Upper Saddle River, NJ, third edition, 1998.

[72] Kay A. Robbins and Steven Robbins. *UNIX Systems Programming: Communication, Concurrency and Threads*. Prentice Hall, Upper Saddle River

NJ, June 2003.

[73] Richard Stevens and Stephen Rago. *Advanced programming in the UNIX Environment*. Adison-Wesley, second edition, 2004.

[74] Juan Solá-Sloan. The TCP offload engine emulator, November 2008. http://www.geocities.com/juan_sola/.

[75] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, 1(4):88–90, July 1997.

[76] Thomas T. Kwan, Robert E. Mc Grath, and Daniel A. Reed. NCSA's World Wide Web server: Design and performance. *IEEE Computer*, 28(11):68–74, November 1995.

[77] The netcraft web server survey. [online] http://news.netcraft.com/archives/web_server_survey.html.

[78] Eugene Demidenko. Kolmogorov-Smirnov test for image comparison. *ICCSA Journal of Computational Science and Its Applications*, 3(46):933–939, April 2004.

[79] John Chambers. The R project for statistical computing. Project is online at http://www.r-project.org.

[80] Stephen T. Satchell and H.B.J. Clifford. *LINUX IP Stacks : Commentary In-Depth Code Annotation*. Coriolis Open Press, Scottsdale, Arizona, 2000.

[81] Jussara M. Almeida, Virgilio Almeida, and David J. Yates. Measuring the behavior of a World-Wide Web server. In *ACM HPN '97: Proceedings of the IFIP TC6 seventh international conference on High performance netwoking VII*, pages 57–72, London, UK, UK, 1997. Chapman & Hall, Ltd.

[82] Ang Boon. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. external, HP

Labs, January 2001. Hewlett Packard Laboratories [online] http://www.hpl.hp.com/techreports/2001/HPL-2001-8.html.

[83] Lester L. Helms. *Probability Theory with contemporary applications*. W.H. Freeman and Company, New York, 1997.

[84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative Performance Analysis : Computer System Analysis Using Queueing Network Models*, volume 1. Prentice Hall, Indianapolis, IN, 1984.

[85] Juan Solá-Sloan and Isidoro Couvertier. A queuing based approach for estimating CPU utilization after TCP protocol offload using a TOE emulator and Apache HTTP server as a case study. Under review [available online] http://www.geocities.com/juan_sola/publications.html, November 2008.

[86] Isidoro Couvertier and Juan M. Solá-Sloan. TCP/IP offloading framework for a TCP/IP offloading implementation. In *Proceedings of CRC '02, Computer Research Conference*, Mayaguez, Puerto Rico, March 2002.

[87] Juan M. Solá-Sloan and Isidoro Couvertier. A parallel TCP/IP offloading framework for a TCP/IP offloading implementation. In *Proceedings of IP System on Chip Based Design*, Grenoble, France, October 2002.

[88] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. TCP performance re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software ISPASS '03*, pages 70–79, Washington, DC, (USA), 2003. IEEE Computer Society.

[89] Daniel P Heyman and David Lucantoni. Modeling multiple IP traffic streams with rate limits. *IEEE/ACM Transactions on Networking*, 11(6):948–958, November 2001.

[90] F. Donelson Smith, Félix Hernández Campos, Kevin Jeffay, and David Ott. What TCP/IP protocol headers can tell us about the web. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):245–256, June 2001.

[91] Haiyong Xie, Li Zhao, and Laxmi Bhuyan. Architectural analysis and instruction-set optimization for design of network protocol processors. In *Proceedings of ACM CODES ISSS'03*, Newport Beach, CA, October 2003.

[92] Boris Tsybakov and Nicholas D. Georganas. On self similar traffic in ATM queues: Definitions, overflow probability bound, and cell delay distribution. *IEEE/ACM Transactions on Networking*, 5(3):397–409, 1997.

[93] A Erramilli, O Narayan, and W Willinger. Experimental queuing analysis with long range dependent packet traffic. *IEEE/ACM Transactions on Networking*, 4(2):209–223, 1996.

[94] John Ousterhout. Why aren't operating systems getting faster as fast as hardware. Technical report, Digital Equipment Coorporation, Palo Alto Research Center, Palo Alto, CA, October 1989.

[95] Douglas Comer. *Internetworking with TCP/IP Principles, Protocols, Architectures*, volume 1. Prentice Hall, Upper Saddle River, New Jersey, fifth edition, 2006.

[96] William Stallings and Richard Van Slyke. *Business Data Communications*. Prentice Hall, Upper Saddle River, NJ USA, fourth edition, 2000.

[97] Timothy S. Ramteke. *Networks*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 2001.

[98] Alberto León-García and Indra Widjaja. *Communication Networks Fundamental Concepts and Key Architectures*. Mc Graw-Hill, Boston Burr Ridge, IL, second edition, 2003.

[99] Allyn Romanov, Jeff Mogul, Tom Talpey, and Stephen Bailey. Remote direct memory access (RDMA) over IP problem statement. Internet draft, April 2005.

[100] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. R. Stevens. Basic socket interface extensions for IPv6. Technical report, Internet Engineering Task Force, 2003. RFC-3494 Internet Engineering Task Force: Intransa Inc., Cisco Co., Hewlett-Packard Co.

[101] M Allman, Vern Paxson, and W Stevens. TCP congestion control. RFC-2581 Internet draft, 2001. NASA Glenn/Steerling Software.

[102] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC-1323 Internet draft, 1992. http://www.ietf.org.

[103] J. Satran, C. Sapuntzakis, K. Meth, E. Zeider, and M. Chadalapaka. iSCSI. RFC 3720 Internet Draft, April 2004. http://www.ietf.org.

[104] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanov. TCP selective acknowledgment options. RFC-2018 Internet draft, 1996. http://www.ietf.org.

[105] John Postel. User Datagram Protocol. RFC-768 Internet draft, 1980. http://www.ietf.org.

[106] John Postel. Internet Protocol. RFC-791 Internet draft, 1981. http://www.ietf.org.

[107] John Postel. Transmission Control Protocol. RFC-793 Internet draft, 1981. http://www.ietf.org.

[108] Angus Telfer. Front-end communications processors and their place in an IP world. Technical report, INTECO Systems Limited, February 2002. [available online at] http://www.inteco.com.

[109] John Valley. *UNIX Programmer's Reference*. Que Coorporation, Carmel, Indiana, 1991. QUE Programming series.