ABSTRACT

KRISHNAMURTHY, RAMESH . A Framework for Evaluating Server Performance: Application to SIP Proxy Servers. (Under the direction of Dr. George N. Rouskas.)

The growing number of applications that use the Session Initiation Protocol (SIP) to manage media sessions over IP is placing increasing demands on the SIP proxy servers (SPS) that make up the core of the SIP network. In this work we investigated the performance of OpenSIPS, an open source SIP proxy server. We collected a large set of experimental data to characterize the performance of the SPS under various call arrival rates and inter-arrival time distributions. Based on these measurements, in the first part, we studied a single SPS server thread on a single-core CPU hardware and modeled the SIP proxy server as an *M/G/1* queue. A key component of the model is a parameter that captures the *interrupt overhead*, i.e., the impact of interrupts and the resulting cache-misses on socket queue service times.

For the second part, we studied the performance of multiple SPS server threads on a single-core CPU hardware. We measured the call rate where the SPS server starts experiencing losses greater than 1% and developed a prediction model for the drop probability as a function of call rate and number of server threads. We also introduced a new parameter to capture the overhead of multiple server threads, in addition to the interrupt overhead.

For the third part, we investigated the impact of the Linux scheduler settings on the performance of single-core, multi-threaded SIP proxy servers, in terms of packet service time, waiting time, and packet drop rate (PDR) to capture the impact on end-user experience. Based on the results of a large set of experiments across a wide range of traffic loads and number of server threads, we developed a methodology to configure the scheduler parameters such that it resulted in significant gains in SPS performance compared to industry-recommended "server" mode operation. Importantly, the gains in performance were the result of setting the scheduler parameters to appropriate values, without the need for adding server capacity or other capital expenditures.

For the final phase of our research, we investigated the impact of the Linux scheduler's load-balancing algorithm on the performance of multi-threaded SIP proxy server running on a multi-core processor system. We conducted extensive experiments and developed a practical guidelines for tuning the scheduler to a "enhanced multi-core server mode" that results in significant gains in performance, thus addressing one of the most crucial needs in today's data center: to extract performance gains from the existing computing infrastructures. We further developed a capacity planning model that provides a good first-order approximation of the total capacity of the SPS system in terms of the call arrival rate that may be supported without affecting user experience in terms of dropped call.

Our measurement and modeling methodology is general, and can be applied to characterize the performance of a wide range of network application protocols.

A Framework for Evaluating Server Performance:
Application to SIP Proxy Servers

by
Ramesh Krishnamurthy

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2016

APPROVED BY:

Dr. Rudra Dutta	Dr. Khaled Harfoush
Dr. David Thuente	Dr. George N. Rouskas Chair of Advisory Committee

DEDICATION

To my parents, for their support and instilling in me the value of education and hard work.
To my wonderful wife, Sujatha for her patience, love and encouragement.
To my sons, Prem and Raj for the joy and happiness they bring to our lives.

BIOGRAPHY

Ramesh Krishnamurthy grew up in the central Indian city of Nagpur. He received his Bachelor's degree in Computer Science from Visvesvaraya National Institute of Technology (formerly called Visvesvaraya Regional College of Engineering), Nagpur, India. He was awarded the graduate merit fellowship from University of Maryland Baltimore County, Baltimore, where he pursued his Master's in Computer Science.

Ramesh joined Cisco at Research Triangle Park, North Carolina as a Software Engineer and has since worked on several software development projects at Cisco. Ramesh's primary role is centered around design, development and architecture of software. Ramesh has also been awarded two patents from the U.S. Patent office for developing novel methods in the area of network protocols.

Ramesh pursued his Ph.D in Computer Science at North Carolina State University, Raleigh, while working full time at Cisco. He was selected for the Preparing for Professoriate (PTP) fellowship at NC State. As part of PTP he underwent extensive training to improve his skills and effectiveness as a teacher. He worked with Dr. Rouskas in updating the course content, projects and lecturing for Data Structures and Algorithms course (CSC 316).

Ramesh's extra-curricular activities include, helping coach his sons recreational soccer and basketball teams. Ramesh is an active volunteer in the community. Ramesh enjoys gardening and spending time with family and friends. Ramesh is married and lives in Raleigh with his wife and twin sons.

ACKNOWLEDGEMENTS

I consider myself very fortunate to have Dr. George Rouskas as my advisor. Throughout my research Dr. Rouskas was very supportive, encouraging, treated me with respect and made himself available whenever I needed him. I especially admire his ability to identify key technical points and help prioritize items so as to maintain focus on tasks that mattered most. Open and candid communication with Dr.Rouskas created an environment of trust and empowerment and brought out the best in me. For this I am very grateful to him. This dissertation would not have been possible without his constant support.

I would like to thank committee members Dr. Rudra Dutta, Dr. Khaled Harfoush and Dr. David Thuente for their time in the research advisory committee. Trying to find answers to their challenging question made me more diligent and prepared. Their insightful comments have significantly raised the quality of my research.

This work is dedicated to my parents for their support and instilling in me the value of hard work and education. I am grateful to my wonderful wife Sujatha for her patience, love and encouragement and to my sons, Prem and Raj for the joy and happiness they bring to our lives. I want to thank my in-laws, especially my father-in-law for helping with the editing of the thesis. I want to thank my sisters, brother-in-law and their families and my nieces and nephews for their support and encouragement.

Finally, I want to thank Cisco for providing financial support for my graduate study at NC State in the form of tuition reimbursement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The Session Initiation Protocol (SIP) [1] is widely used as a signaling protocol for managing media sessions over IP. The ubiquitous presence of IP over the past decade has spawned a new set of non-traditional service providers, including Vonage, Skype, and Lingo, that offer voice over IP (VoIP) services with advanced features (e.g, PC-to-phone calling, enhanced voice-mail, multi-ring capability, etc.) at lower cost to users. Established providers, including both telcos and cable companies, are increasingly providing VoIP service so as to remain competitive and to take advantage of the resulting increased efficiencies. Furthermore, with the advent of smart phones, there has been a proliferation of applications that enable users to make voice and video calls using any available WiFi hot-spot, thus avoiding the more expensive 3G/4G data networks. Most of these providers and their applications use SIP as the underlying signaling protocol for setting up and managing the voice and video calls.

The growing number of customers and devices that make use of SIP is placing an increased demand on the SIP proxy servers (SPS) that make up the core of the SIP network. For service providers to deal effectively with the demand growth, they must develop a good understanding of current usage patterns, forecast and plan upgrade needs, and be able to configure a robust service capability for new users. In [2] the requirements for management of overload in SIP is discussed and poor capacity planning was cited as one of the leading causes of overload in SIP. Ultimately, all these considerations require accurate estimates of the performance capability of the SPS.

Hence, we have the following main objectives for our research:

1. Develop tools and techniques that can be easily adapted to carry out similar experimental studies for other SPS configurations as well as different protocol suites.

2. Conduct a comprehensive set of experiments to understand how individual SIP packets are processed and measure their processing and waiting times (within the kernel and the

SIP protocol)

3. Develop a guideline for extracting performance gains from existing computing infrastructures without additional capital expenditures.

4. Develop a parametrized model that can be used to estimate the performance and the capacity of the SPS over a range of offered loads, and a range of SPS hardware and software configurations.

In this work we investigate the performance of OpenSIPS [3], an open source SIP proxy server, and make several contributions.

1. We have modified the Linux kernel and the OpenSIPS source code to obtain packet-level measurements for each SIP message, from which the service and waiting times within the kernel and the SIP layer can be easily obtained. In particular, the kernel modifications can be used for collecting such measurements for *any* protocol, while the OpenSIPS modifications may be easily adapted to other application servers.

2. We also enhanced SIPp [4], a SIP traffic generator tool, to generate calls with inter-arrival times that follow any user-specified distribution. The modified versions of the kernel, OpenSIPS, and SIPp are made available in the Appendix and also as media files as part of this thesis.

3. We have collected a large set of experimental data to characterize the performance of the SPS under various call arrival rates and inter-arrival time distributions.

4. Based on these measurements, in the first part, we study a single SPS server thread on a single-core CPU hardware and model the SIP proxy server as an *M/G/1* queue. A key component of the model is a parameter that captures the Interrupt overhead, i.e., the impact of Interurpts and the resulting cache misses on socket queue service times. Our measurement and modeling methodology is general, and can be applied to characterize the performance of a wide range of network application protocols.

5. For the second part, we study the performance of multiple SPS server threads on a single-core CPU hardware. We measure the call rate where the SPS server starts experiencing losses greater than 1% and develop a prediction model for the drop probability as a function of call rate and number of server threads. We also introduce a new parameter to capture the overhead of multiple server threads, in addition to the interrupt overhead.

6. For the third part, we investigated the impact of the Linux scheduler settings on the performance of single-core, multi-threaded SIP proxy servers, in terms of packet service time,

waiting time, and packet drop rate (PDR) to capture the impact on user performance. We identify the key scheduler parameters of the Linux scheduler and provide concrete guidelines for tuning these parameters that we identify as 'enhanced server mode' to achieve significant performance improvement.

7. For the final phase of our research, we investigated the impact of the Linux scheduler's load-balancing algorithm on the performance of multi-threaded SIP proxy server running on a multi-core processor system. We conducted extensive experiments and develop practical guidelines for tuning the scheduler to a 'enhanced multi-core server mode' that results in significant gains in performance, thus addressing one of the most crucial needs in today's data center: to extract performance gains from the existing computing infrastructures without additional capital expenses. We further develop a capacity planning model that provides a good first-order approximation of the total capacity of the SPS system in terms of the call arrival rate that may be supported without affecting user experience in terms of dropped call.

1.1 Thesis Organization

The remainder of this thesis is organized as follows.

In Chapter 2, we present an introduction to the SIP architecture and its main components: the SIP user agent (UA), the SIP proxy server (SPS), the SIP Registrar server and the SIP Re-direct server. We provide an example of the SIP call setup and tear-down processes, and explain the message flow for a typical call. We also discuss related research.

In Chapter 3, we provide the details of the kernel modifications that we made as part of the research, starting with an overview of the kernel and the packet processing within Linux kernel. We then provide a description of service time and waiting time components and then describe measurement methodology for obtaining these components.We end this chapter with a discussion of work that studied the impact of increased packet arrivals.

In Chapter 4, we present the experimental testbed that we used for obtaining the measurements. We present the details of the SPS server and the SIPp tool that is used in the experiments. We then present a large set of data collected and from which we develop a queuing model for the SPS proxy server.

In Chapter 5, we investigate the performance of SPS server in a multi-threaded environment. We measure the performance of SPS as a function of number of server threads and call arrival rates. We expand our earlier finding and introduce a new parameter to capture the overhead due to resource contention among multiple server threads, and develop a drop-probability model for the multi-threaded system.

In Chapter 6, we investigate the impact of the Linux Completely Fair Scheduler (CFS) on the performance of SPS, as a function of the number of threads and the call arrival rate. We characterize the impact of the scheduler on the performance of a multi-threaded SPS, in terms of waiting time and packet drop rate. Further, we identified key scheduler parameters of CFS scheduler and developed concrete guidelines on tuning these parameters to achieve significant performance improvement.

We present the impact of the Linux scheduler's load-balancing algorithm on the performance of multi-threaded SIP proxy server running on a multi-core processor system in Chapter 7. We conducted extensive experiments, developed practical guidelines for tuning the scheduler to an 'enhanced multi-core server mode' and were able to obtain significant performance gains in the SPS. We further develop a capacity planning model that provides a good first-order approximation of the total capacity of the SPS system in terms of the call arrival rate that may be supported without affecting user experience in terms of dropped call.

We conclude in Chapter 8 by summarizing our motivation, goals of our research and list the various contributions we have made. We also identify possible areas where the research can be expanded further.

Chapter 2

SIP Architecture

SIP is an application layer signaling protocol that can establish, modify, and terminate multimedia sessions such as Internet telephony calls [1, 5]. SIP is independent of the transport layer and can work over any transport protocol including UDP, TCP, stream control transmission protocol (SCTP), and transmission layer security (TLS). SIP is based on an HTTP-like *request/response* transaction model. Each transaction consists of a request that invokes a particular method, or function, on the server and at least one response for the request. The most common SIP requests are as follows:

- `INVITE:` Used by a client to initiate or update a media session.
- `ACK:` Indicates the client has received a final response for an `INVITE` and is ready for media exchange.
- `BYE:` This message is used to terminate a session.
- `CANCEL:` Cancels any pending requests. For instance, if an `INVITE` has been sent but the user has not received a final response, `CANCEL` is sent to terminate the pending `INVITE`.
- `REGISTER:` This message is used by the client to indicate its contact address, so that it can receive calls.

The response to a SIP request can be of one of the following types:

- *1xx:* Provisional – request received, continuing to process the request; e.g., `100 Trying`, `180 Ringing`.
- *2xx:* Success – the action was successfully received, understood, and accepted; e.g., `200 OK`, `202 Accepted`.

- *3xx:* Redirection – further action needs to be taken in order to complete the request; e.g., `301 Moved Permanently`, `302 Moved Temporarily`.
- *4xx:* Client Error – the request contains bad syntax or cannot be fulfilled at this server; e.g., `403 Forbidden`, `404 Not Found`.
- *5xx:* Server Error – the server failed to fulfill an apparently valid request; e.g., `500 Internal Server Error`, `503 Service Unavailable`.
- *6xx:* Global Failure – the request cannot be fulfilled at any server; e.g., `600 Busy Everywhere`, `604 Does Not Exist Anywhere`.

2.1 Main Components

The SIP infrastructure consists of four key components: the SIP user agent (UA), the SIP proxy server (SPS), the SIP Registrar server, and the SIP Re-direct server. We now briefly describe the main functions of these components.

- **SIP User Agent (UA).** A SIP UA is an end-point device such as a VoIP phone, a cell phone, a PC, etc. The UA initiating a SIP request becomes the user agent client (UAC) for this particular session, while the end-point device which responds to the request is referred to as the user agent server (UAS) of the SIP session. A UAC is capable of generating a request based on some external stimulus (e.g., a user clicking a button) and processing a response. A UAS is capable of receiving a request and generating a response based on user input (or some other external stimulus).
- **SIP Proxy Server (SPS).** The main function of an SPS is to route SIP messages between the UAC and UAS. A SIP request from the UAC may traverse through several proxies on its way to a UAS, with each proxy making routing decisions and possibly modifying the request before forwarding it to the next. A response is forwarded through the same set of proxies traversed by the corresponding request, but in the reverse order. When a SIP request arrives, the SPS may respond on its own as well.

  An SPS can operate in either a *stateful* or *stateless* mode. In stateless mode, a proxy acts as a simple forwarding element. A stateless proxy does not maintain any information about a SIP message after the message has been forwarded. A stateful proxy, on the other hand, remembers transaction state about each incoming request. For example, when a proxy receives an `INVITE`, it will maintain the transaction state until a final response is received or the transaction is canceled. Another feature present only in stateful proxies is the ability to *fork* a request to a number of locations at the same time. When using

TCP as the transport protocol, an SPS has to operate in stateful mode, as the UA relies on the proxy to perform retransmissions for any UDP hops in the signaling path.

- **SIP Registrar Server.** Before a SIP UA can receive a call, it needs to make its reachability information available so that an SPS can forward SIP requests to the UA. A SIP UA makes itself discoverable by registering with (i.e., submitting a `REGISTER` message to) a special UAS known as the SIP Registrar server. The Registrar acts as the front end to the location service for a given domain. The SPS queries the Registrar to determine how to forward a request for that domain. Note that the Registrar and SPS are *logical* roles; typically, both functionalies are implemented in the same physical device. The `REGISTER` message contains the address-of-record (AOR) whose registration is to be created or modified, and the corresponding contact binding to a device.

- **SIP Re-direct Server.** This is a special class of UAS that generates a response of type *3xx* (e.g., `302 Moved Temporarily`) to a request it receives. The server populates the *3xx* response with a list of alternative locations and provides the time during which the new contact data is valid. The Re-direct server allows the SPS to direct SIP requests to external domains.

As we can see, the SPS is the key component of the SIP infrastructure: it handles all SIP messages generated by the UAC and UAS during setup, modification or termination of media session; hence, in case of overload, the SPS may become a performance bottleneck that limits the ability of users to establish SIP sessions. Consequently, we only focus on the SPS in this thesis.

2.2 Call Setup and Teardown

Recall that SIP is mainly used to signal the establishment and termination of multimedia sessions between user agents. Once a call has been established, no SIP messages are exchanged until either party decides to modify or terminate the session. Therefore, understanding the flow of messages during call setup and teardown is crucial in evaluating the performance of the SPS.

Figure 2.1 illustrates the basic call setup process and the resulting media session between two users within the same domain. In the figure, Alice and Bob are two users in the domain `XYZ.com` with phone numbers 1111 and 2222, respectively.

Suppose that Bob has a soft-phone on his laptop. When the soft-phone application is started, it sends a `REGISTER` message to the SIP Registrar server in the `XYZ.com` domain (the Registrar's IP address is usually available via configuration). The `REGISTER` message includes two important fields:

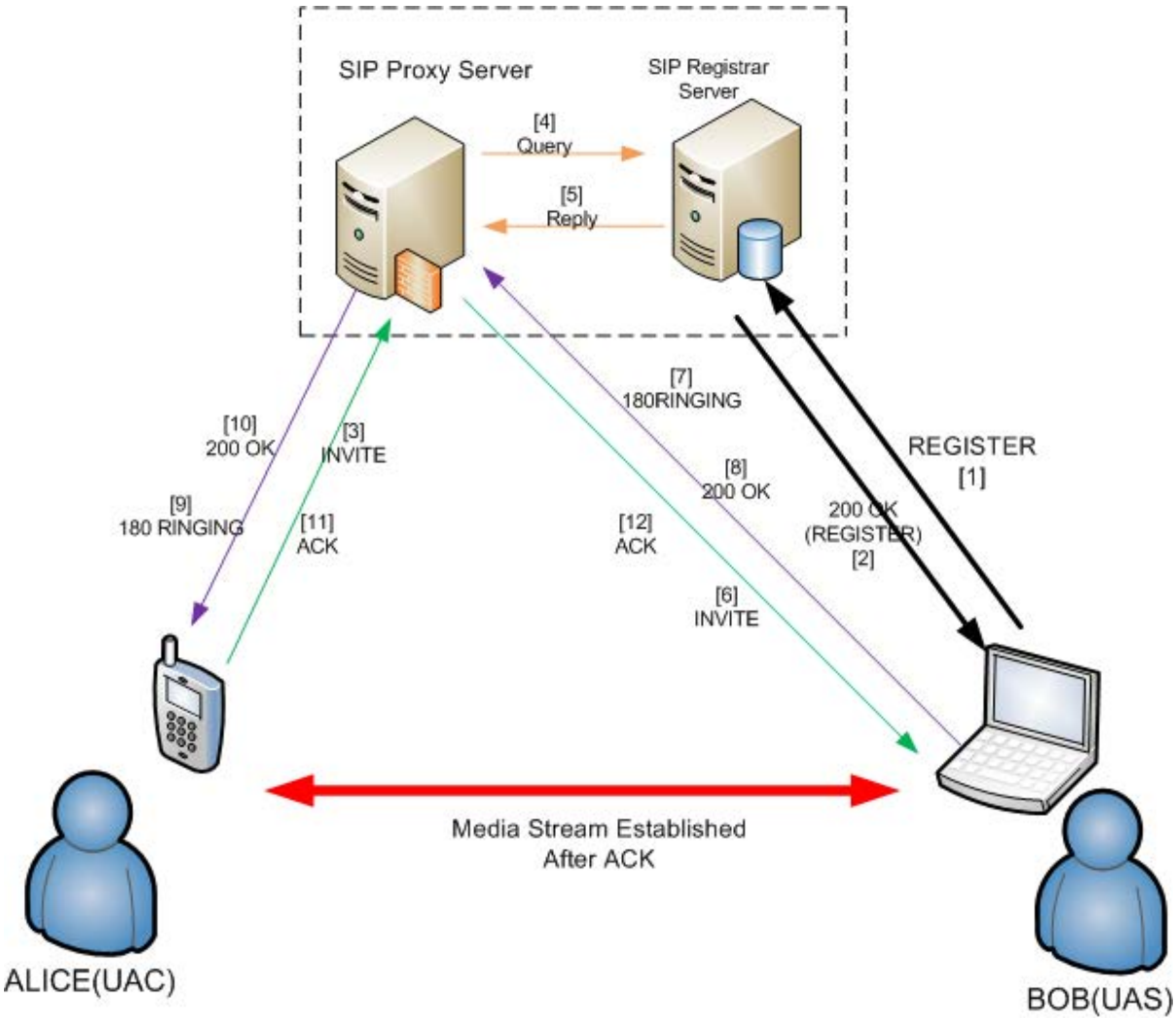

Figure 2.1: SIP call setup within the same domain

- the `To:` field contains the value *sip:2222@XYZ.com*, which corresponds to Bob's address-of-record (AOR), and
- the `Contact:` field is set to the value *sip:bob@<IP Address of Bob's laptop>*, binding the AOR to the current IP address of the device where Bob may be reached.

Upon successful registration, the Registrar responds with a `200 OK` message.

When Alice (having the role of a UAC) wishes to call Bob (who, in this case, acts as a UAS), Alice sends an `INVITE` message to the SPS in the `XYZ.com` domain; this message provides Bob's AOR (i.e., *sip:2222@XYZ.com*) in the value of its `To:` field. Upon receipt of this message, the

SPS queries the Registrar to obtain the corresponding contact address, and the latter replies with the contact address that Bob provided in its earlier `REGISTER` message. The SPS then forwards the `INVITE` message to this contact address. Once Bob (the UAS) answers the call, the SPS forwards the `200 OK` response message from Bob back to Alice. Alice (the UAC) acknowledges Bob's acceptance of the call with an `ACK` message. Once Bob receives this last message, the media session between the two parties is established.

To terminate the session, either the UAC or the UAS sends a `BYE` request, and the other party responds with a `200 OK` message.

Figure 2.2 shows the exchange of SIP messages between the UAC and UAS through an SPS, for both the call setup and teardown operations. This is the message flow that we use in our experimental data collection and in modeling the SPS performance.

2.3 SIP Protocol Family

SIP is one part of the IETF protocol suite that makes up the complete multimedia architecture. For the sake of completeness, we briefly discuss two other protocols that are commonly used with SIP, namely, the Real-time Transport Protocol (RTP, defined in RFC 1889) and the Session Description Protocol (SDP, defined in RFC 2327).

SDP is used with SIP for describing the multimedia session. When setting up a voice or video session, SDP information is included in the `INVITE` request by the UAC and the `200 OK` response from the UAS. The SDP media session information typically includes: an IP address and port number, the media type (e.g., audio, video, interactive whiteboard, etc.), and the media encoding scheme (e.g., G711 audio, H264 video, etc.). Once the SIP session is established, the UAC and UAS use the SDP information to encode the media, and use RTP for transporting real-time voice and/or video data.

SIP must be used in conjunction with SDP and RTP in order to provide complete services to the users. However, the basic functionality and operation of SIP, as well as the performance of the SPS, does not depend on any of these protocols.

2.4 Related Work

RFC 6076 [6] defines metrics and parameters used to evaluate the end-to-end performance of SIP for telephony service. Call related metrics discussed in this RFC include the session request delay (SRD), the session disconnect delay (SDD), and the session duration time (SDT). While the latter only depends on the users' behavior and characteristics (e.g., how long they wish to stay connected), the performance of the system in terms of the former two metrics is clearly

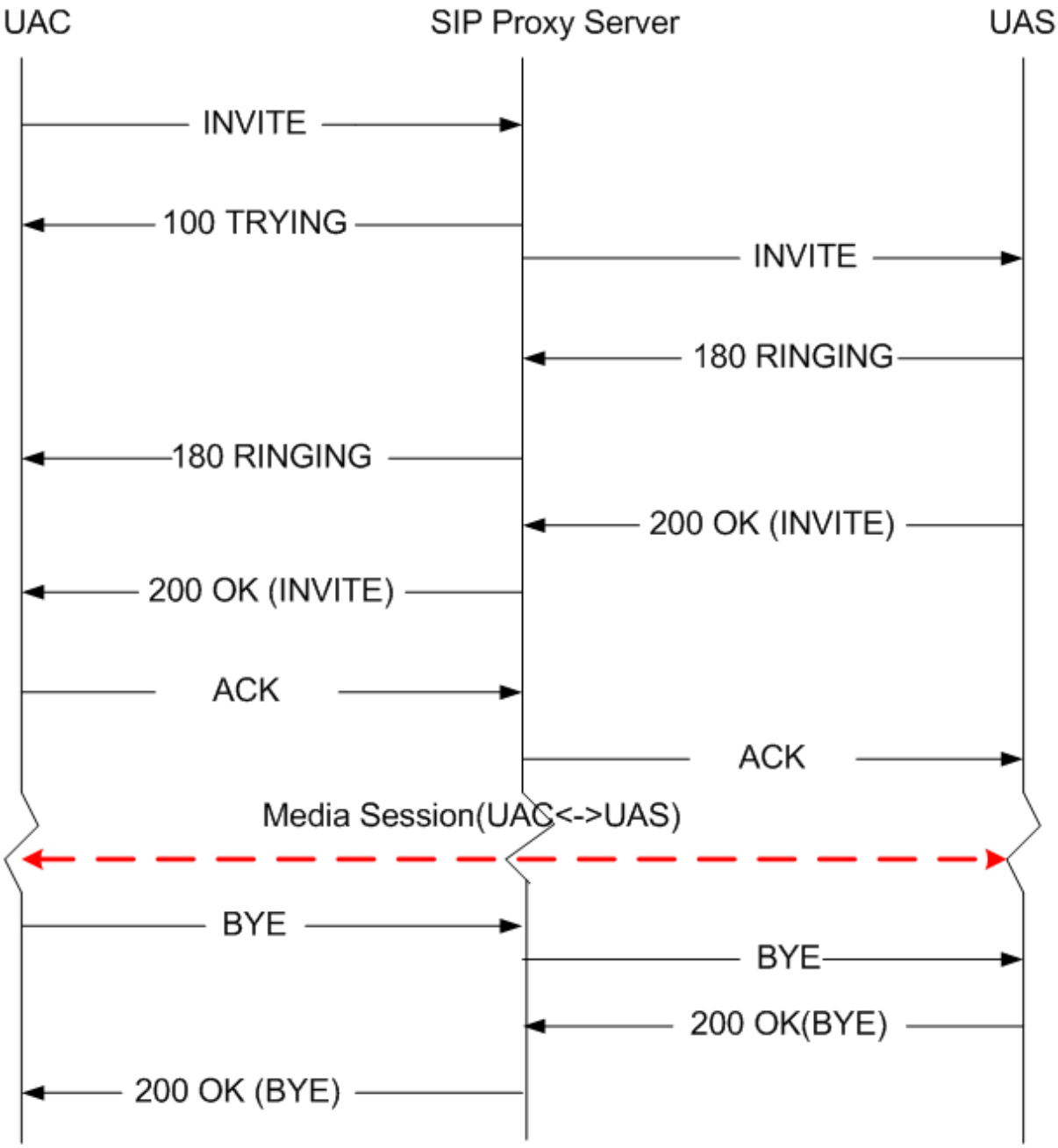

Figure 2.2: SIP message exchange for call setup and teardown

determined by the processing capacity and load of the SPS. There have been several attempts in the literature to characterize the queuing behavior of the SPS.

An analytical model to estimate the mean response time for call setup, as measured from

the UAC perspective, was developed in [7]. In this study, the mean response time consists of the processing and queuing delays at the SPS and UAS. Specifically, the mean response time is defined as the difference between the instant an `INVITE` request from the UAC arrives at the SPS until the instant the corresponding final response is forwarded by the SPS to the same UAC , excluding the call duration and the propagation delays between the SPS and the UAC and UAS. The SPS and UAS are modeled as a queuing network with six $M/M/1$ queues, each queue corresponding to the processing of one message type at either the SPS or UAS, including failure messages not shown in Figure 2.2.

Based on the observation that the processing of messages of a specific type does not follow an exponential distribution but, rather, it is close to a constant, the queuing network of [7] was analyzed in [8] under the assumption that each of the six queues are of the $M/D/1$ type.

While these studies provide some insight into the mean response time, the six-node queuing network is not an accurate model of the way the SPS and UAS operate in practice. In [9], the SPS is modeled as a single $M/M/c$ queue, and the analytical results are compared to experimental data collected from an SPS with three threads. This study was further extended in [10]. Specifically, the experiments were modified to incorporate a call holding time for each call, and the CPU and memory utilization and queue size were measured to investigate the scalability of the SPS model.

An experimental evaluation of the OpenSER [11] proxy server under various configurations was carried out in [12] using the SIPp tool [4] (we also utilize the SIPp tool and discuss it in the next chapter). It was observed that, depending on the configuration (authentication enabled/disabled, UDP or TCP, stateful or stateless), the measured throughput may vary from hundreds to thousands of operations per second, and that more complex configurations result in a lower throughput, as expected.

The impact of UDP versus TCP transport on the performance of OpenSER was investigated in [13]. The main conclusion of this study was that the lower performance observed when using TCP is mainly due to the design of the OpenSER server (which is better suited to handle a connectionless protocol), while the overhead of TCP compared to UDP is not a significant factor affecting the performance.

The objective of the work in [14] was to improve the performance of a cluster of SIP proxy servers. A number of load balancing algorithms were developed that exploit the properties of SIP, namely, the session-oriented nature of the protocol which mandates that all SIP messages of a given session be handled by the same SPS. The proposed algorithms have the load-balancing server employ session-aware policies to assign requests so as to improve throughput and response times.

An algorithm that uses the type of incoming SIP request, the CPU usage and the retransmit count to dynamically decide if the incoming message needs to be handled in a stateful or

stateless manner by the SPS was proposed in [15]. It was shown via IP multimedia system (IMS) simulation that this hybrid approach that combines the performance advantage of stateless functionality with the higher reliability of a stateful server may lead to better scalability properties for the SPS.

In [16] the authors studed multi-core scalability of the OpenSER SIP server, not only on a Linux on Intel platform, but also using Solaris system (developed by Sun Microsystems). They additionally consider TCP as a transport protocol. They encountered scalability bottlenecks in both the Linux and Solaris operating systems, such as those caused by a single lock protecting access to a socket, and proposed using multiple sockets (and port numbers) in response to avoid the problem. They also noticed a problem involving contention in the shared memory segment used by OpenSER. Their response is to hash a shared state by call ID and use multiple shared memory segments to partition the global state and have a lock per segment, reducing the contention. They improved scalability by up to a factor of four, depending on the scenario.

In [17] the authors evaluated the performance and scalability of an open-source SIP proxy on three different multi-core platforms: AMD Santa Rosa, Intel Harpertown, and IBM POWER6. Two performance and scalability bottlenecks were identified using whole-system profiling: One was the OpenSER use of hash table for user location lookup. By increasing the width of hash table the performance was improved by a factor of four. The workload used in the study was user based and required Database lookup, storing the data in memory improved performance. The scalability was reduced when using eight cores.

In [18] the impact of TLS on SIP server performance is studied and a measurement-driven cost model is developed to predict the incremental cost of using TLS over TCP for SIP signaling compared to SIP-over-UDP. In the experiments conducted, the authors found that SIP over TLS/TCP can reduce the performance by a factor of 17 compared to SIP-over-UDP.

Chapter 3

Linux Kernel Modifications and Measurement Methodology

The kernel forms the core part of an operating system and provides an abstraction layer between the applications and the hardware present in the system. In this chapter, we first provide an overview of the kernel function, then we present the steps a packet takes as it moves from the device layer through the Linux kernel to the application layer and back to the device layer; Note that these steps are common for any application layer service. We describe the kernel modifications we have implemented and the measurement methodology that we used for collecting the data. We also discuss the impact of interrupts and cache misses on system performance as a function of load, and survey related research.

In our research, we carry out a large set of experiments with the objective of obtaining a precise measurement of the time a packet spends within the SIP Proxy Server (SPS) System. The total time is defined as the length of time from the arrival instant (i.e., the time it is received by the device layer) to the departure instant (i.e., the time it is transmitted by the device layer after it has undergone processing at the application layer (SIP in this instance)). Therefore, to measure the total time the packet spent in the system, it is necessary to first understand the processing that a packet undergoes in the kernel.

3.1 Kernel Overview

The kernel provides an abstraction layer for the hardware present in the system. The main function of the kernel is to perform resource management of the available resources (including CPU, memory and devices). Kernel allows other user processes to utilise these resources. Some of the functions that a kernel can perform are:

1. **Process Management and Scheduling**.

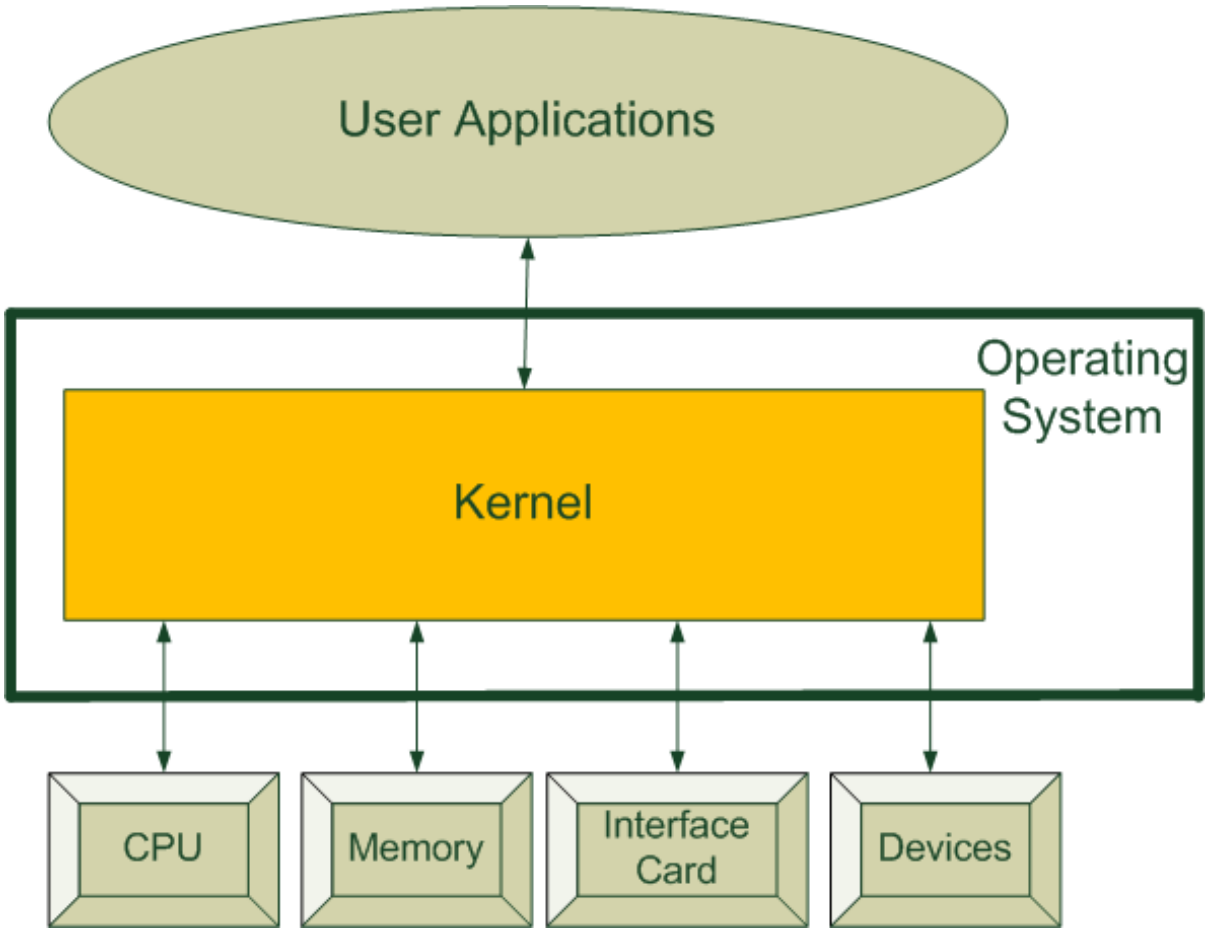

Figure 3.1: Kernel: An abstraction layer for available resources in a system

To run any process, the kernel has to setup address space for the process and load the code for the process into memory and allocate the process, CPU time, so that the process can execute. All modern systems use multi-tasking, i.e, the kernel gives each process a slice of CPU time before switching to another process so as to give an illusion of running multiple processes simultaneously.

2. **Memory Management**.

   The kernel is responsible for managing all the memory in the system that is currently in use by various processes. The kernel needs to ensure that each process uses only its own memory space and does not interfere with memory dedicated to other processes.

3. **Interrupt Handling**.

When a hardware devices triggers an interrupt, it is handled by the kernel. The kernel will typically handles interrupts by suspending the currently executing process, saving its context, and then running the code to service the interrupt. Once the interrupt is serviced, the kernel will restore the interrupted process's context and allow it to continue execution. System calls made by a process also result in interrupts and the control coming back to kernel. The kernel will handle the request and return the results and control back to the process.

3.1.1 Kernel Design

There are two main kernel design approaches: monolithic kernel and the other is the micro Kernel. Each type is briefly described here.

In monolithic kernel all the OS services reside in the same memory space and use the same shared memory. All the services run along with the main kernel thread. This allows all the services easier access to data from other services with less overhead. The implementation is also simpler. However, monolithic kernel has some drawbacks. First, a lack of protection between services can result in the crash of the entire system, when a bug is encountered in one service. Second, as the size of kernel becomes larger, the lack of modularization makes maintenance of the kernel difficult. Examples of monolithic kernel implementations include Linux, BSD (and its flavors), AIX, HP-UX, Windows 9x series, etc.

The approach taken in micro-kernel design is to provide the most fundamental and primitives functions including process scheduling, inter-process communication and low-level address space management. All other services like network protocol stack, device drivers, file systems etc. run in user space. Micro kernel design addresses the drawbacks of monolithic kernel, by providing address space protection between services and making it easier to maintain the kernel due to its smaller size. However, the benefits come at the expense of performance due to the large number of context switches and additional overhead in sharing data between services. Examples of micro kernel implementations are: QNX, Symbian, L4Linx, etc.

All the experiments that were conducted as part of this thesis, used the Linux kernel, which is a monolithic kernel. For the remainder of this thesis, any reference to the Linux kernel or Operating System in the context of our experiments should be considered as referring to version 2.6 of the Linux kernel, unless stated otherwise.

3.2 Packet Processing within the Linux Kernel

Let us refer to Figure 3.2 which illustrates the packet receiving and sending operations within the Linux kernel network stack. Based on this figure, there are two distinct entities involved in processing a packet at the Linux kernel, as discussed next.

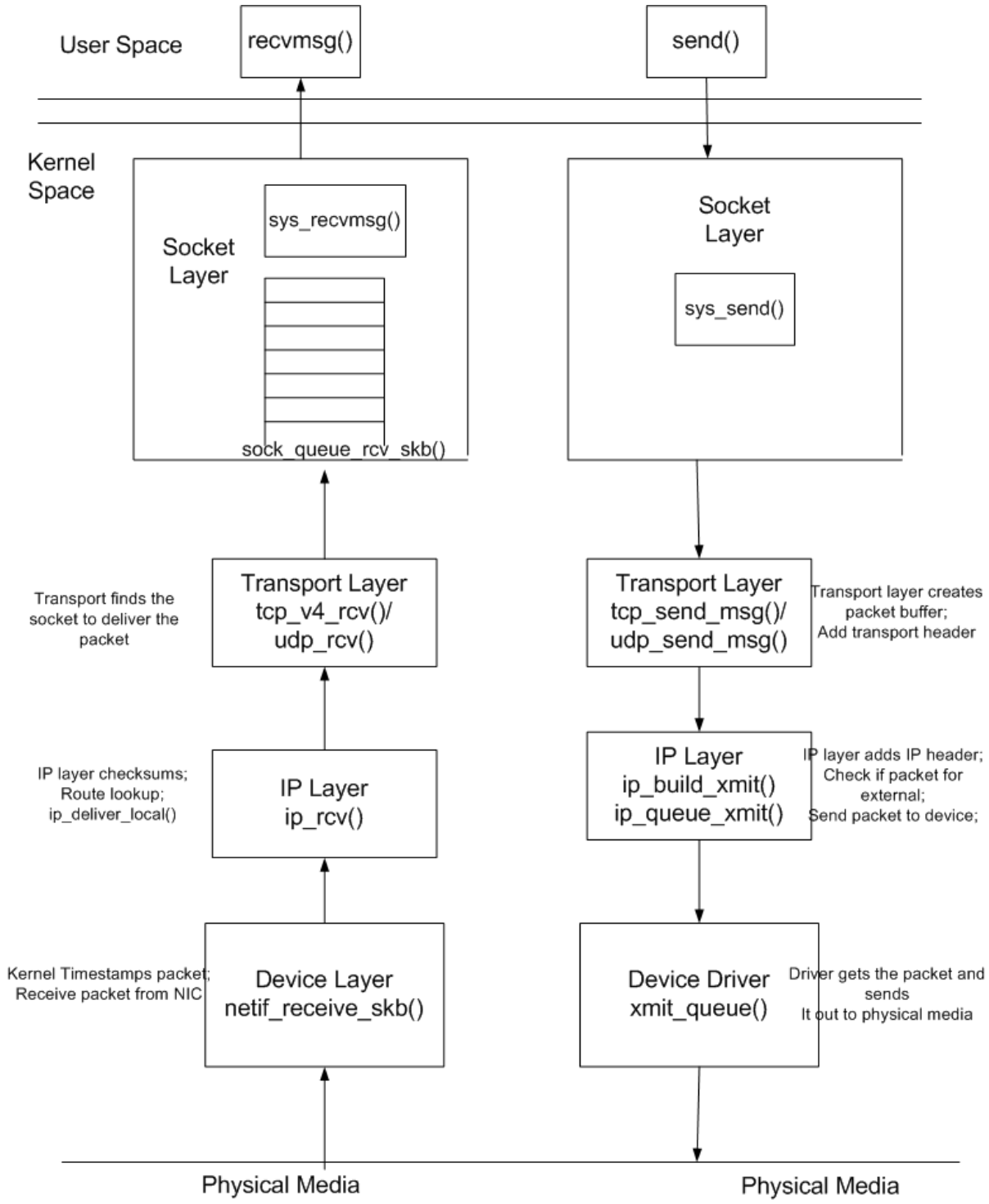

Figure 3.2: Linux network stack:UDP packet receiving/sending operations

3.2.1 Linux Kernel Network Stack – Packet Receiving

As soon as a packet is received at the Network Interface Card (NIC) it is transferred to a ring buffer that is in kernel space. The packet then undergoes the following operations within the Linux kernel stack before it is handed to the application layer [19, 20, 21, 22, 23]:

1. **Kernel Device Layer**. The kernel device driver interface receives the packet via interrupt from the NIC. The `netif_receive_skb()` is the main receive data processing function of the kernel. This function is called from softirq (an interruptible kernel event) context and with interrupts enabled. In this function, the kernel time-stamps the packet, checks the payload type and calls any registered handler for that type. For IP traffic the registered handler is ip_rcv().

2. **IP Layer**. `ip_rcv()` handles the packet from the netif_receive_skb(). The IP layer computes the check-sum and carries out a number of other checks, it performs route look-up, and delivers the packet to the transport layer.

3. **Transport Layer**. The transport layer receives the packet, performs certain checks, and finds the socket to which it must be delivered. It then passes the packet to the kernel socket.

4. **Socket Layer**. The `socket_queue_rcv_skb()` call inserts the packet at the tail of the socket's receive queue; once the packet reaches the head of the queue, it is made available to the application layer socket.

3.2.2 Application Layer Packet Processing

The application layer process waits for packets to arrive on the socket. Once a packet is received from the socket, application specific processing is performed on the packet. If the application needs to forward the packet to a different network node, the socket function to send the packet to the transport layer is called.

3.2.3 Linux Kernel Network Stack – Packet Sending

Once a packet completes processing at the application layer, it is passed on to the kernel for forwarding and undergoes the following operations [19, 20, 21, 22]: (refer also to Figure 3.2):

1. **Socket Layer**. The application writes the data to the kernel socket via the `send()` `and/or write()` socket calls. The kernel socket layer then sends the data to the transport layer

2. **Transport Layer**. The transport layer creates the packet buffer and adds the transport header.

3. **IP Layer**. The IP layer receives the packet and adds the IP header in the `ip_build_xmit()` call. Since the packet is destined for external delivery, the IP layer calls `ip_queue_xmit()` to pass the packet to the device driver.

4. **Device Layer**. The device driver receives the packet and inserts it in its transmit queue; when the device driver is scheduled to send the packet, it transmits the packet on the physical medium.

3.3 Packet Service and Waiting Time Components

Based on the previous discussion, three main components are identified that make up the total time a packet spends within the System for an application-layer protocol:

1. K_{rcv}: This component represents the time spent within the kernel from the instant the packet is received at the kernel device layer until the instant it is handed off to the Application layer. Note that this component consists of four distinct sub-components:

   (a) K_{stack}: The time it takes for the kernel network stack to process the packet, i.e., to perform various checks and look up the forwarding table. This part involves network and transport layer processing time, hence this component is not part of the service time of the packet for application layer but part of the time the packet spends in the system.

   (b) K^w_{sockq}: The time the packet spends waiting at the socket queue before it can be delivered to the application layer.

   (c) K^s_{sockq}: The time it takes the kernel to process the packet while the packet is in the queue. This includes the time to wake the receiving user level process to indicate the availability of data and handling the dequeue request from the user process. This component is part of the service time for the packet within application layer.

   (d) K_{copy}: The time needed to copy the data from the kernel space to user space.

   Clearly, $K_{rcv} = K_{stack} + K^w_{sockq} + K^s_{sockq} + K_{copy}$.

2. T_{app}: This component represents the time that the packet undergoes processing within the application layer. The application layer typically receives one packet at a time from the socket, processes it, and passes it to the kernel for forwarding before it receives the next packet from the socket. Therefore, T_{app} reflects the service time of the packet within the application layer, and does not include any waiting time.

3. K_{snd}: This component represents the time it takes the packet to traverse the kernel on the sending side, until it is transmitted to the physical medium. The application layer passes one packet at a time to the kernel and then is blocked till the kernel has processed the packet and sent it to the device layer. Hence, K_{snd} is considered to be part of the packet service time within the application layer.

3.4 Impact of Interrupts and Cache Misses

As the packet arrival rate increases, there is an impact on the Linux kernel because of increased interrupts, context-switching and the resulting cache misses. We now present a literature survey of some of the research that investigated the impact of these factors.

In [24] the author describes the condition of "receive livelock" which results when the interrupt rate is high enough to cause the system to spend all of its time responding to interrupts. In this case the system does not have any time to perform any other task, and as a result the throughput drops to zero. Under such conditions the system is not deadlocked, but it makes no progress on any of its tasks. The author suggests some techniques to overcome this problem including: using interrupts only to initiate polling, using round-robin polling to fairly allocate resource among different event sources and temporarily disable interrupts and dropping packets early rather than later to avoid wasted work.

The performance of TCP/IP network stack in Linux kernel 2.4 and 2.5 was studied in [25]. It was observed that the Linux kernel (version 2.4 and 2.5) TCP/IP stack was not efficient in handling high bandwidth network traffic of a gigabit network interface. The Linux TCP/IP stack needed to mimic the interrupt mitigation techniques that network interfaces adopt. The techniques that would accomplish this effect in the TCP/IP stack was explored.

To address the issue raised in [24] and studied in [25] and to handle the increased traffic introduced by Gigabit networks, a modification to the device driver packet processing framework was done for Linux called New API (NAPI) [26]. NAPI works through interrupt mitigation, NAPI allows drivers to run with (some) interrupts disabled during times of high traffic with a corresponding decrease in system load. Another technique used is packet throttling, where the system is overwhelmed, it was found to be better to dispose these packets before much effort goes into processing them. NAPI-compliant drivers can often cause packets to be dropped in the network adaptor itself even before the kernel sees them. NAPI is present in version 2.6 of the Linux kernel.

In [27] the authors looked at the performance handling of the Linux kernel using the NAPI framework and modified NAPI configurable parameters Budget B and MAX_SOFTIRQ_RESTART to determine values that would give better performance than the default setting for the Snort application. The authors further try to model the performance of Snort as an open

tandem queuing network comprised of two M/G/1/B queues in series.

In [28] a performance study of memory reference behavior in network protocol processing was carried out. Some of the statistics derived were cache miss rates and percentage of time spent waiting for memory. The interesting finding for the cold cache case i.e. no instruction or data referenced, is cache resident, was that latencies were roughly six times longer for UDP and four times longer for TCP without check-summing.

Context switching introduces high overheads directly and indirectly. Direct context switch overheads include saving and restoring processor registers, flushing the processor pipeline, and executing the OS scheduler. Indirect context switch overheads include the perturbation of the cache and transaction look-ahead buffers (TLB) states. When a process/thread is switched out, another process/thread runs and brings its own working set to the cache. When the switched-out process/thread resumes execution, it has to refill the cache and TLB to restore the state lost due to the context switch. Prior research has shown that indirect context switch overheads, mainly the cache perturbation effect, are significantly larger than direct overheads. In [29], the goal was to understand how cache parameters and application behavior influence the number of context switch misses from which the application suffers. The main findings were that context switch misses can contribute to a significant increase in the total (Level 2) L2 cache misses, and they tend to increase along with the cache size up until the cache size is large enough to hold the entire combined working sets. Re-ordered misses tend to contribute to an increasing fraction of context switch misses as the cache size increases. The maximum number of reordered misses occurs when cache perturbation displaces roughly a half of the total cache blocks.

In [30], the relationship between cache misses and software performance was investigated. It was pointed out that one of the missing pieces in the existing models is the relationship between cache misses and timing penalties. Potential causes for the weak relationship between cache misses and timing penalties included (a)competition for hardware prefetch, (b)competition for request handling capacity of the shared cache, and (c)competition for request handling capacity of the memory bus was investigated. It was shown that workloads differ significantly in their sensitivity to cache misses.Besides cache misses, workloads can also be sensitive to other aspects of cache sharing. This is seen especially when the ability of the cache to handle multiple requests, or the ability of the cache to initiate prefetch requests, is stressed.

3.5 Measurement Methodology

The main objective of our experimental study is to obtain precise measurements of the time a SIP packet spends within the SPS, from the arrival instant (i.e., the time it is received by the device driver) to the departure instant (i.e., the time it is transmitted by the device driver after it has undergone processing at the SIP layer). In Section 3.2 we presented an overview

of the steps a packet takes as it moves from the device driver through the Linux kernel to the SIP layer and back to the device driver. Here we describe the modification we introduced to the kernel and the methodology we used to measure all the components of the packet service and waiting times.

3.5.1 Measuring the Time Components: Kernel and OpenSIPS Modifications

In order to capture all the time components, we modified the OpenSIPS and the Linux kernel source code to obtain and log certain information about each packet as it moves through the system. Each log entry contains a time-stamp along with the source IP address, call id, command sequence, and method type [1] of the packet, i.e., all the information necessary to uniquely identify the message type and related call. Time-stamps are recorded with microsecond precision. As we explain shortly, multiple log entries are recorded for each packet, e.g., when a packet crosses the kernel-user boundary. At the end of the experiment, the log file is parsed and the times-tamps associated with a given packet are processed to determine the time components. Time-stamps are recorded at the various instances as shown in Figure 3.3.

The K_{rcv} Component

We log three time values for each packet in the kernel and a fourth one as soon as it enters the SIP layer:

- t_{arr}: the arrival time of the packet to the SPS, i.e., the time it was received at the kernel device layer. We obtain this value by making a system call with the `SIOCGSTAMP` flag on the socket that returns the time-stamp attached to the packet by the kernel at the time it arrived.

- t_{stack}: A new field was added to the kernel packet structure to record the time of completion of processing by the kernel network stack for that packet. The time-stamp was recorded just before the packet is added to the socket queue i.e just before (Figure 3.3). `socket_queue_receive_skb()` , the time-stamp is then stored in this new field. To obtain this value in the user space, a new ioctl command `SIOCGSTAMPUDP` was defined on the same lines as SIOCGSTAMP.

- t_{sockq}: Another time-stamp field was added to the socket structure, to get a precise measurement of the time at which the application is ready to dequeue the packet via the socket recvmsg() call. The stored time-stamp is then accessed via another new ioctl command `SIOCGSTAMPRECV`.

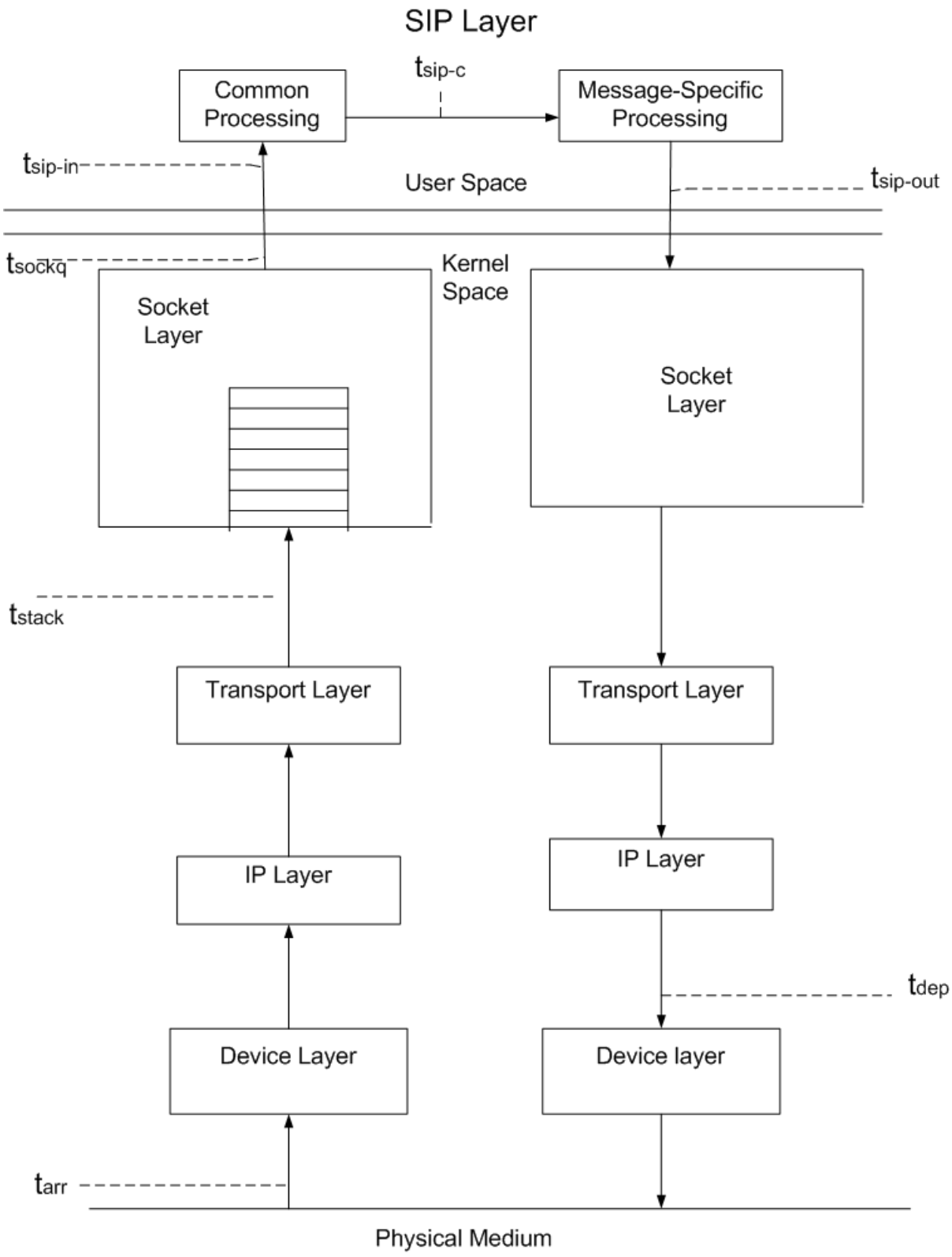

Figure 3.3: Time-stamps recorded at the instances shown in the Kernel and SIP layer

- t_{sip-in}: this is the current time, i.e., the time that the packet enters the SIP layer for processing.

Given these time values, we may calculate K_{rcv} and its various components as:

$$\begin{aligned} K_{stack} &= t_{stack} - t_{arr}, \\ K_{sockq} = K^s_{sockq} + K^w_{sockq} &= t_{sockq} - t_{stack}, \\ K_{copy} &= t_{sip-in} - t_{sockq}, \\ K_{rcv} &= t_{sip-in} - t_{arr}. \end{aligned} \tag{3.1}$$

The T_{sip} Component

The T_{sip} component is the time that the packet undergoes processing at the SIP layer. T_{sip} corresponds to the T_{app} component of Section 3.2, when the Application layer protocol is SIP.

For each packet, we log two additional time values within the SIP layer:

- t_{sip-c}: this is the instant at which the packet processing part that is common to all packet types is complete.
- $t_{sip-out}$: this is the instant at which the SIP layer has completed the processing of the packet and is ready to pass the packet back to the kernel (i.e., just before the `msg_send()` call that send the packet to the transport layer returns).

From these values, the SIP service time and its sub-components can be calculated as:

$$\begin{aligned} T^1_{sip} &= t_{sip-c} - t_{sip-in}, \\ T^2_{sip} &= t_{sip-out} - t_{sip-c}, \\ T_{sip} &= T^1_{sip} + T^2_{sip}. \end{aligned} \tag{3.2}$$

The K_{snd} Component

Here we describe the steps taken to determine the time K_{snd}, a packet spent traversing the kernel after being processed at the SIP layer. The socket sendto() call completes only when the kernel layer has completed processing and transferred the packet to the device driver for transmission. We record another time-stamp at the SIP layer as soon as this call returns (at which time the SIP layer is ready to fetch the next packet from the socket for processing).

- T_{dep}: the instant at which the packet departs the kernel layer and control is transferred back to SIP layer. Although the time-stamp is taken at the SIP layer, logically, it corresponds to the time shown in Figure 3.3.

Using this value and the $T_{sip-out}$ value, the K_{snd} value is calculated as:

$$K_{snd} \;=\; T_{dep} - T_{sip-out} \tag{3.3}$$

These equations and measurements points are referenced in later chapters when we describe the experiments and the performance measurements that were collected. The research related to the impact of increased packet load on the kernel is referenced in later chapters as well, where the service and waiting times of SPS server modeling are described.

Chapter 4

Evaluation of SIP Proxy Server Performance: Packet-Level Measurements and Queuing Model

In this chapter, we present the findings of our investigation of the performance of OpenSIPS [3], an open source SIP proxy server. Our work makes several contributions.

- As described in Chapter 3, we modified both the Linux kernel and the OpenSIPS server source code to obtain packet-level measurements for each SIP message, from which the service and waiting times within the kernel and the SIP layer can be easily obtained. In particular, the kernel modifications can be used for collecting such measurements for *any* protocol, while the OpenSIPS modifications may be easily adapted to other application servers.
- We enhanced SIPp [4] a SIP traffic generator tool, to generate calls with inter-arrival times that follow any user-specified distribution. The modified versions of the kernel, OpenSIPS, and SIPp are attached as Appendix.
- We have collected a large set of experimental data to characterize the performance of the SPS under various call arrival rates and inter-arrival time distributions.
- Based on these measurements, we model the SIP proxy server as an *M/G/1* queue. A key component of the model is a parameter that captures the interrupt overhead, i.e., the impact of interrupts on socket queue service times.

Our measurement and modeling methodology is general, and can be applied to characterize the performance of a wide range of network application protocols.

4.1 Experimental Setup

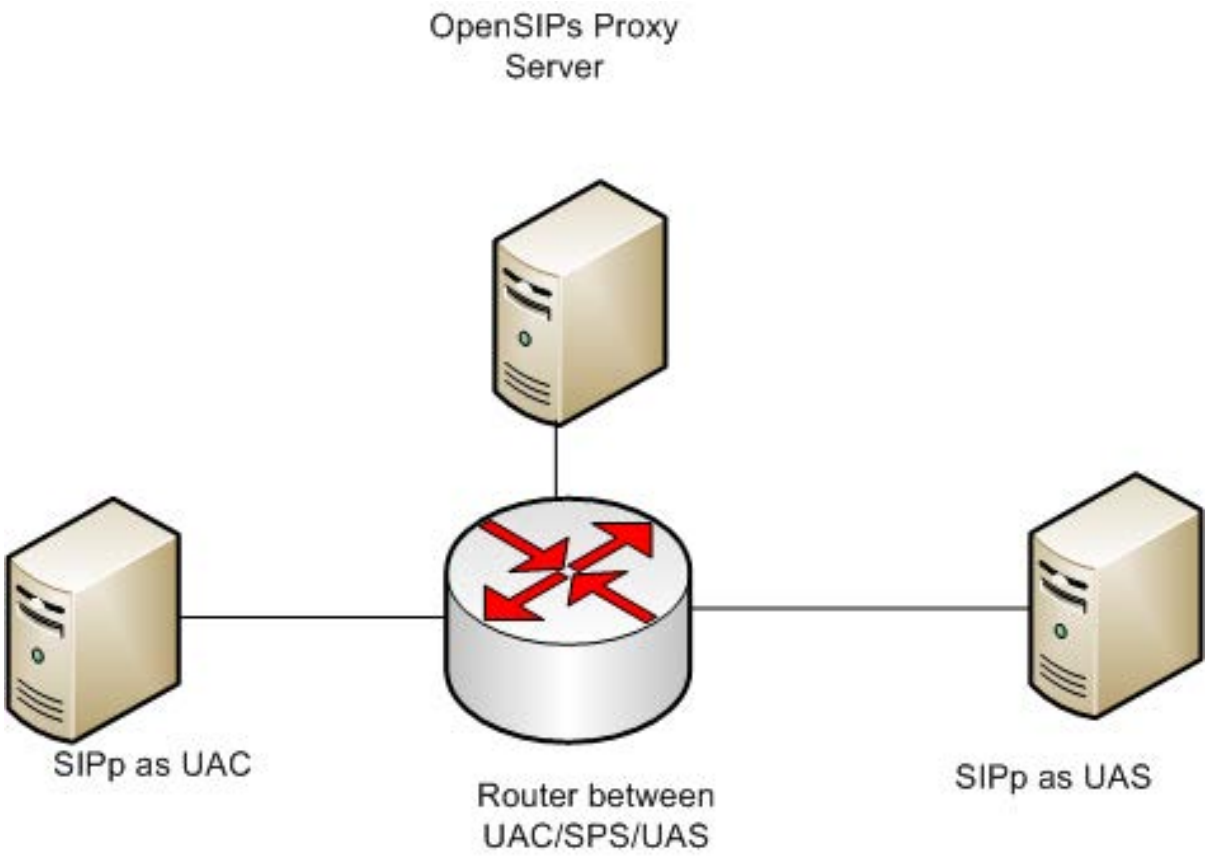

Figure 4.1: Testbed for performance measurements of OpenSIPS SPS

Figure 4.1 shows the network testbed that we used to generate SIP calls and collect measurement data so as to characterize the performance of the SPS as a function of traffic load. The hardware setup consists of:

- **OpenSIPS SPS.** OpenSIPS [3] is an open source implementation of a SIP proxy server, and a continuation of the OpenSER [11] project. In our testbed, the OpenSIPS SPS was installed on an HP workstation with dual-core processors and 4 GB RAM, running on a Debian 5.0.6 Linux distribution (2.6.26 Linux operating system). Each of the processors is an Intel® Core™ i3 CPU 530 @2.93GHz with 4096 KB cache size. All the experiments we report in this chapter were conducted after setting the number of cores to one (i.e., disabling three cores of the workstation) so as to emulate a single processor environment.

- **SIP*p* UAC.** This is an HP workstation with dual Intel® Xeon® CPU 3050 @2.13GHz processors, running Redhat Linux 4.1.2-44 (Linux kernel version 2.6.18) OS. The SIP*p* [4] tool was run on this machine configured as a UAC, as we describe shortly.

- **SIP*p* UAS.** This is an HP workstation with quad Intel® Xeon® CPU E5540 @2.53GHz processors, running Redhat Linux 4.1.2-44 (Linux kernel version 2.6.18) OS. The SIP*p* tool was run on this machine configured as a UAS.

- **Router.** Each of the three workstations above (SPS, UAC, and UAS) is attached to a different port of a Cisco 7206 VXR series router through 1 GigE interfaces. The router simply forwards SIP messages between the three workstations, and due to its high switching capacity, it does not introduce any material delay for any of the traffic loads we generated.

4.1.1 OpenSIPS as SPS

We installed the Debian 5.0.6 Linux Distribution and the OpenSIPS server (specifically, the `opensips-1.6.2-notls` version of the source code) on the workstation acting as the SPS, following the instructions in [31]. All the experiments were carried out by configuring the server in stateful mode such that the state of the transaction is maintained until a final response is received.

We configured UDP as the transport layer protocol, and we modified the default configuration file (i.e., `opensips.cfg`) to set the number of child processes to one, so as to have a single SIP layer process handling all incoming messages. The configuration was also modified to disable authentication and loose routing for SIP packets.

We start the OpenSIPS SPS using the command

opensips -f /etc/opensips/opensips.cfg -m 2048

where the "-m" option increases the shared memory to 2048 MB when OpenSIPS runs.

4.1.2 SIP*p* as UAC

SIP*p* is a free open source test tool and traffic generator for the SIP protocol. It includes a few basic user agent scenarios (including UAC and UAS). SIP*p* can be used to establishe and release multiple calls with the `INVITE` and `BYE` methods, respectively. It may also import custom XML scenario files describing call flows of various levels of complexity.

A major limitation of the SIP*p* tool is that it does not provide a mechanism to generate call inter-arrival times that follow general distributions. Specifically, the tool only provides users the option to specify either the call arrival rate, in calls per second (cps), or a fixed call rate period (with a 1 ms granularity). Consequently, calls are generated at fixed intervals, i.e., in a

deterministic fashion, making it impossible to test the performance of the SPS under various distributions of call inter-arrival times.

To address this limitation, we enhanced the source code of the SIP*p* tool to generate arrivals based on a user specified input file. Specifically, we added a new option "*iat_file <filename>*" to the latest stable release (version 3.1) of the tool available from the source repository [4] at the time of this writing. The user-specified file is expected to contain a single number per line, representing the inter-arrival time between consecutive calls. When invoked with this new option, the modified version of the tool processes this input file one line at a time, generating one call and waiting for the specified amount of time before proceeding to the next line. As a sanity check, we tested the modified version of the tool on a input file having deterministic inter-arrival times (i.e., the same number in each line), and verified that its functionality was not affected (i.e., the results were identical to those produced by the original, unmodified tool invoked with the corresponding call arrival rate).

We implemented another enhancement to overcome the 1 ms limitation in the granularity of call inter-arrival times imposed by the original SIP*p* tool. To this end, we reduced the polling period of the tool to 200 μs. This new feature allows the input file specified with the new "*iat_file*" option to contain inter-arrival times expressed at this finer granularity, making it possible to generate values that accurately represent general time distributions.

For the experiments, we start the modified SIP*p* tool to act as the UAC using the following command:

./sipp -sf <uac.xml> <SPS IP addr>:5060
-s <username> -m <#calls> -iat_file <filename>

The command specifies the IP address of the SPS workstation and the standard SIP port (i.e., 5060) to send the generated traffic, and includes four options:

- *-sf:* Run the specified XML file as the scenario, acting as a UAC to originate SIP calls. The `uac.xml` file we used was modified slightly from the one embedded in the `sipp` executable. Specifically, we disabled the retransmission of SIP messages so as to avoid processing the same call again.
- *-s:* Set the username part of the request URI (required so that the SPS can forward the call).
- *-m:* Specify the total number of calls to generate.
- *-iat_file:* Specify the file of inter-arrival times.

4.1.3 SIPp as UAS

The latest available stable release of the SIPp tool available at the time of this writing (version 3.1) has a software bug that prevents the processing of call rates above 300 calls per second (cps). Therefore, for the UAS, we use the most recent unstable release dated 2009-07-09 for which this error was not observed.

We start the SIPp tool to act as the UAS using the command

./sipp -sf <uas.xml> -rsa <SPS IP address>

with the two options:

- *-sf:* Run the specified XML file as the scenario. The `uas.xml` we used was modified from the built-in UAS scenario to (1) remove the pause after the `BYE` message is sent and (2) disable the retransmission of SIP messages (as with the UAC).
- *-rsa:* Provide the remote sending address where the message needs to be sent.

4.2 Experiments and Performance Measurements

We conducted a large set of experiments to measure the components K_{rcv}, K_{stack}, K_{sockq}, K_{snd} and T_{sip}, that we described in Chapter 3.5, of the time each SIP packet spends at the SPS. For each experiment, the UAC initiates 1M (million) calls to the UAS. Each call is accepted by the UAS, resulting in the message exchange shown in Figure 2.2. Therefore, for each call, six different messages are generated by either the UAC or the UAS and are forwarded to the other party via the SPS. In other words, for each experiment, the SPS may process up to 6M SIP messages, i.e., up to 1M messages of each type seen in Figure 2.2. Each experiment is characterized by two parameters:

- *Call arrival rate.* We varied the call arrival rate from 100 cps to 1200 cps. At a rate of 1200 cps, the SPS crashes frequently as it cannot handle the message volume, indicating that the server is severely overloaded.
- *Call inter-arrival time distribution.* The call inter-arrival time is the time between two consecutive `INVITE` messages generated by the UAC. For each call arrival rate, we generated inter-arrival times using exponential and deterministic distributions. Note that, since the router in Figure 4.1 is not a bottleneck, `INVITE` message arrivals at the SPS follow the distribution from which these message are generated by the UAC. However, the overall packet inter-arrival time distribution at the SPS (i.e., across all packet types) depends on the processing and waiting times at the UAC, SPS, and UAS, and, in general, is unknown and different from the distribution the generates `INVITE` messages.

Table 4.1: Measured Mean Values and Confidence Intervals for K_{rcv}, and Poisson Inter-Arrivals (μS)

Message Type	Call Arrival Rate								
	100cps	200cps	400cps	600cps	700cps	800cps	900cps	1000cps	1200cps
`INVITE`	24.5	38.74	82.35	201.4	340.3	672.61	9747.0	29092.5	40207.7
`180 Ringing`	33.0	53.46	114.7	266.55	428.22	788.19	6228.9	23851.6	29868.36
`200 OK (INVITE)`	36.2	64.62	131.15	262.99	398.8	717.72	13183	34085.47	44259.25
`ACK`	21.7	38.96	100.45	240.15	382.28	712.72	6567.6	24315.56	32157.11
`BYE`	54.1	92.08	162.02	284.06	411.21	725.27	12545	33088.71	38225.25
`200 OK (BYE)`	29.5	55.43	134.13	294.85	444.74	775.41	6431.23	23810.67	31375.82
Overall Mean	33.17	57.2	120.80	258.33	400.93	731.99	9116.5	28065.01	36262.29
Conf. Int. (Half-Width)	0.304	.649	1.71	3.48	5.595	19.39	3332.16	3164.98	6129.90
# Messages to SIP layer	5,999,931	5,999,940	5,999,943	5,999,931	5,999,943	5,999,940	5,823,173	5,359,200	3,457,644

For each packet processed by the SPS, we log the seven time values t_{arr}, t_{stack}, t_{sockq}, t_{sip-in}, t_{sip-c}, $t_{sip-out}$ and t_{dep}. We then process the log files to obtain the mean values for quantities K_{rcv}, K_{stack}, K_{sockq} T_{sip}, and K_{snd}; we record both the overall mean (i.e., across all packets) and the mean per packet type. We also use the method of batch means to estimate 95% confidence intervals around the overall mean.

The call holding time in all experiments was set to 250 ms. Referring to Figure 2.2, the *call holding time* is defined as the interval between the instant the UAC acknowledges the `200 OK` message until it closes the session by sending a `BYE`. Note that the SPS is not involved after a session has been established, as no SIP messages are exchanged during the call until the time the session is closed with a `BYE`. A short call holding time (1) ensures that the experiments do not take an extraordinarily long time, and (2) provides a worst-case scenario in that the SPS is observed under higher than normal packet arrival rates. This is because, with longer holding times the same number of response messages would arrive at the SPS but over a longer time period.

4.2.1 Experimental Results: Measurement Data for K_{rcv}, K_{stack}, and K_{sockq}

Tables 4.1 and 4.2 list the measured values for quantity K_{rcv}, under exponentially and deterministically distributed inter-arrival times and for various call rates. These tables present the mean value for each message type and the overall mean value, as well as the half-width of the confidence interval around the overall mean; all values are expressed in μs. Tables 4.1 and 4.2 also list the number of messages (out of 6M generated) that the kernel passes to the SIP layer at each call rate; the remaining messages are lost due to congestion or server crashes (at higher call rates).

Table 4.2: Measured Mean Values and Confidence Intervals for K_{rcv}, Deterministic Inter-Arrivals (μs)

Message Type	Call Arrival Rate								
	100cps	200cps	400cps	600cps	700cps	800cps	900cps	1000cps	1200cps
`INVITE`	13.4	98.1	141.5	256.4	337.3	466.1	751.8	25500.9	27021
`180 Ringing`	14.3	19.22	46.6	133.7	212.9	342.98	623.42	24139	25335
`200 OK (INVITE)`	17.0	19.05	71.4	176.27	209.9	363.99	590.85	25965	28293
`ACK`	13.4	19.04	62.3	197.29	187.13	434.52	604.44	24358	25997
`BYE`	24.8	31.07	86.1	159.48	211.16	293.54	548.35	26198	28542
`200 OK (BYE)`	14.3	21.2	59.97	210.2	262.17	418.4	656.77	24656.9	25662
Overall Mean	16.2	34.6	78.0	188.9	236.76	386.6	629.27	25137	26813
Conf. Int. (Half-Width)	0.294	0.48	1.59	5.28	6.38	8.17	24.7	628.13	639
# Messages to SIP layer	5,999,940	5,999,940	5,999,939	5,999,940	5,999,940	5,999,940	5,999,940	4,984,808	4,046,503

Figures 4.2 and 4.3 present the K_{sockq} values we have obtained for various messages and exponential and deterministic distribution inter-arrivals in the stable region (for call rate up-to 800cps). From the figures 4.2 and 4.3 we can observe that as the arrival rate increases, there is a corresponding increase in the K_{sockq}. As described in Chapter 3, K_{sockq} comprises of the time the packet spends waiting at the socket queue before it can be delivered to the application layer and the time it takes the kernel to process the packet while the packet is in the queue.

Figures 4.4 and 4.5 present the K_{stack} values we have obtained. K_{stack} is the time taken by kernel network stack to process the packet, from the point the packet is received from the device, till the point the packet is placed in the socket queue. From these figures we can observe that kernel stack processing time K_{stack} is mostly constant, with the average value of 2 μs, independent of message type and call arrival rate.

The values of K_{copy}, obtained from the these data as $(K_{rcv} - K_{stack} - K_{sockq})$, are also constant at around 2 μs.

We observe that all mean values of K_{rcv} and K_{sockq} increase rapidly with the call arrival rate up to 1000 cps, but level off beyond that rate. The contributing factor to the increase in these values are:

- *Queuing delay.* As the call arrival rate increases, packets arriving at the SPS are buffered at the socket queue and experience increasing waiting times K^{w}_{sockq} before being delivered to the SIP layer.

- *Interrupt overhead.* Recall that in Chapter 3 we discussed the impact of increased packet arrival on the kernel. The number of interrupts increases in direct proportion to the packet arrival rate. Interrupts introduce overhead in the form of the time needed to handle each interrupt, the context-switching operations, and the increase in processing time as a result

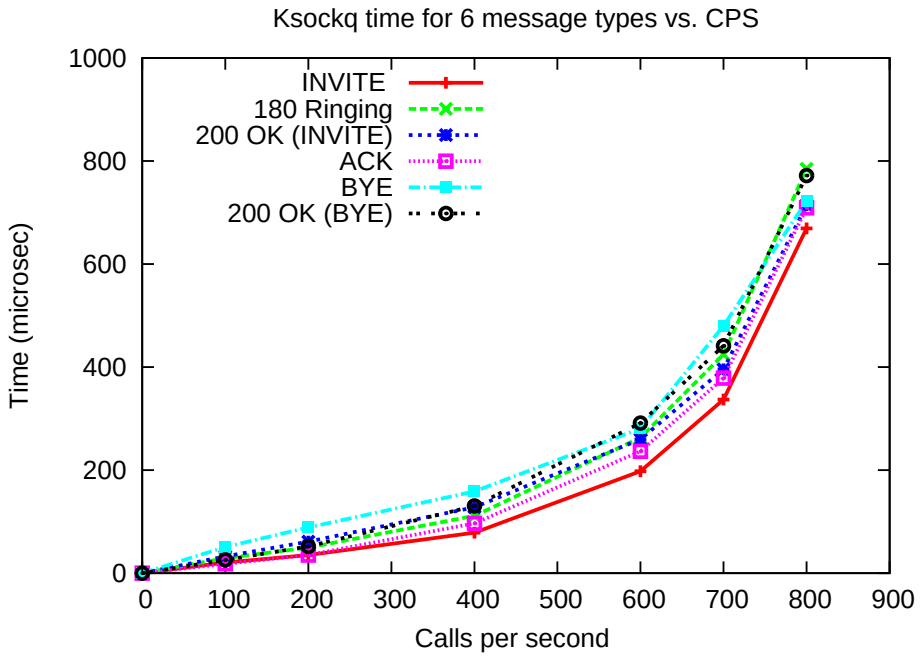

Figure 4.2: Mean value of K_{sockq} in the stable region, Poisson Arrivals

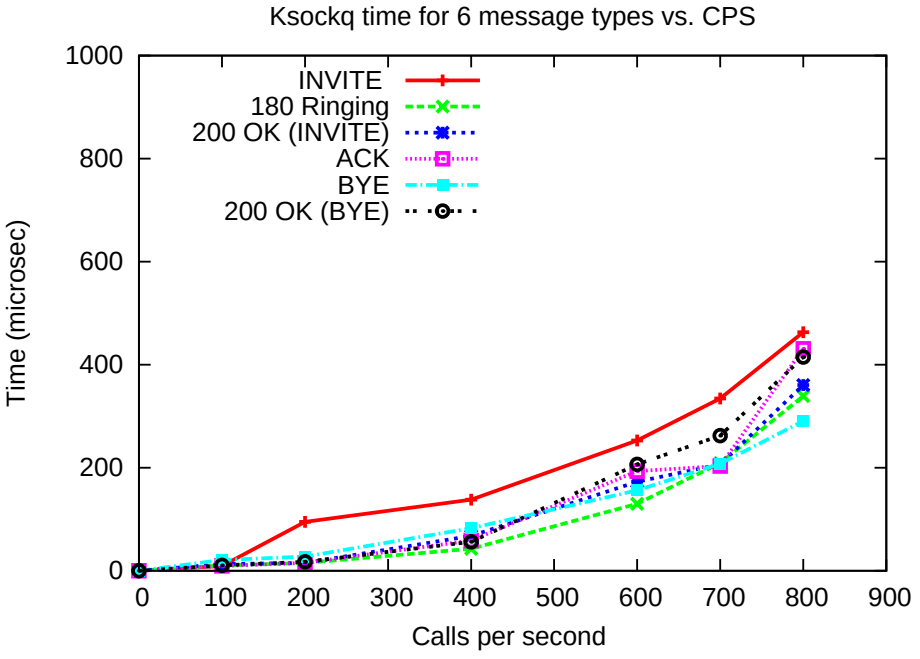

Figure 4.3: Mean value of K_{sockq} in the stable region, Deterministic Arrivals

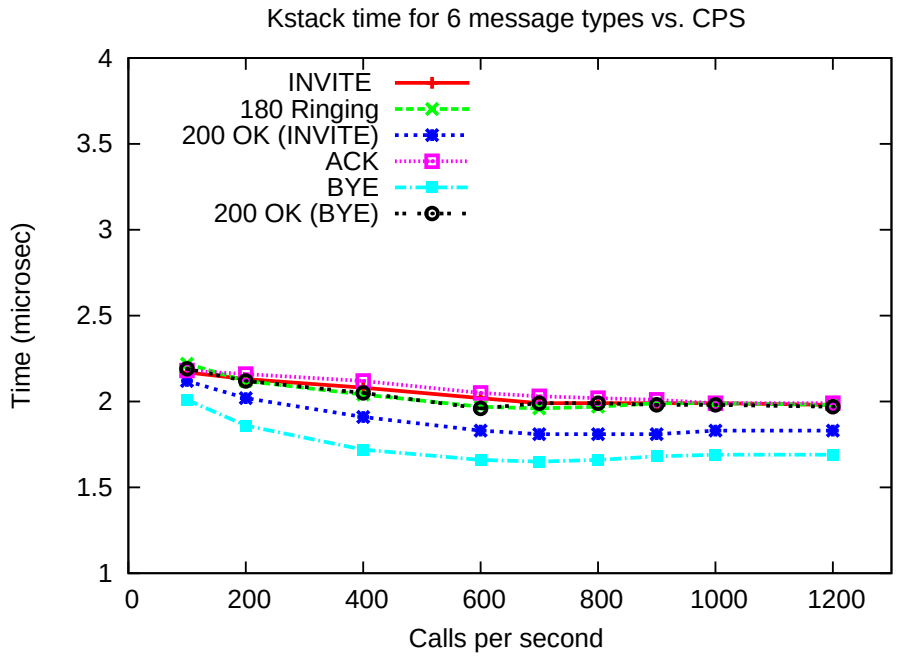

Figure 4.4: Mean value of K_{stack}, Poisson Arrivals

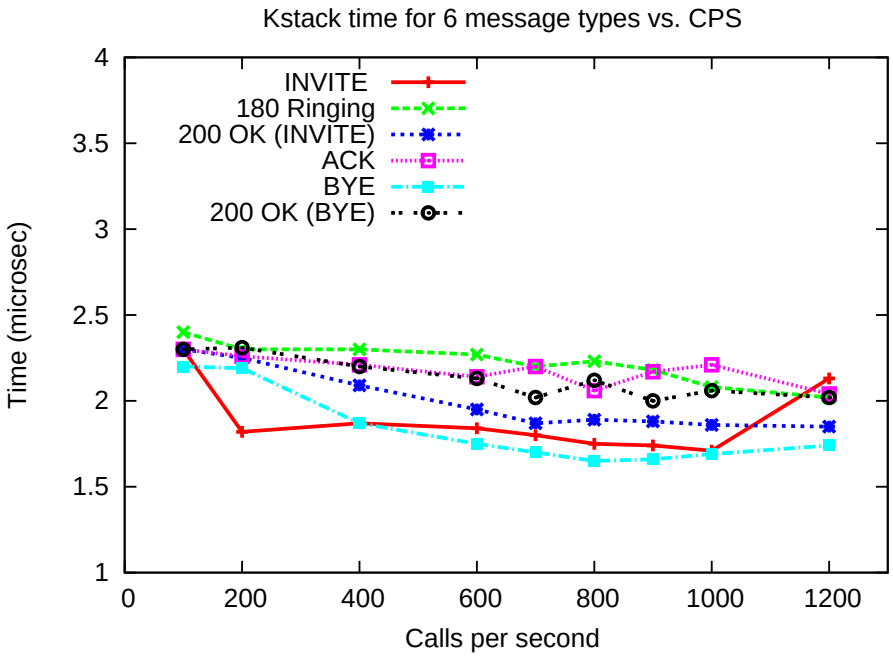

Figure 4.5: Mean value of K_{stack}, Deterministic Arrivals

of cache misses due to these interrupts. A cache miss causes the number of CPU cycles consumed by the network stack receiving process to increase, as the memory access cycles of main memory are several times that of L2 caches.

We also observe that, up to a rate of 800 cps for exponentially distributed inter-arrivals, the number of packets dropped at the kernel is extremely small (about 0.001% of the 6M packets generated) and does not vary much with the arrival rate, for deterministic inter-arrival this is valid till 900 cps.

However, at 900 cps for exponential distribution and at 1000 cps for deterministic distribution, packet losses increase substantially, and at 1200 cps the loss rate is about 42%. The leveling off of the measured K_{rcv} values at 1200 cps is due to the fact that the dropped packets are not observed at the SIP layer where statistics are logged, hence they are not taken into account in the average values shown in the tables.

Finally, while the trends seen in the data in the tables are similar for both distribution, there are some differences. Specifically, with the exception of the `INVITE` message, the mean values (including the overall mean) under deterministic inter-arrivals are lower than under exponential inter-arrival times in the stable region (i.e., up to 900 cps). On the other hand, `INVITE` message experience higher times in the kernel for arrival rates between 200 and 800 cps under deterministic inter-arrivals. We believe that this relative behavior is due to the fact that, with exponential inter-arrival times, the system experiences bursts of call (i.e., *INVITE* message) arrivals, that, in turn, lead to higher waiting times overall. With deterministic arrivals, calls are spaced apart evenly, hence most messages experience lower delays in the stable region. However, due to the fact that call holding times were fixed to 250 ms, for medium (deterministic) arrival rates, new `INVITE` messages may consistently arrive at the SPS while the previous call's tear down messages are being processed, leading to higher waiting times for the former.

4.2.2 Experimental Results: Measurement Data for T_{sip}

Figures 4.6 and 4.7 present the SIP layer processing times T_{sip} under exponential and deterministic distributed call inter-arrival times and for various call rates. These figures present the mean value for each message type, the overall mean value, and the half-width of the confidence interval around the overall mean; all values are expressed in μs. As we can see, the mean SIP service time varies with the message type across all call arrival rates. As we described earlier, SIP processing consists of a common component and a message-specific component, and the differences in the latter component account for the difference in service times among the various message types. For instance, processing an `INVITE` message that initiates a new session requires more operations than other messages (e.g., to verify that this is a new transaction and create a new table entry), and this fact is reflected in the data.

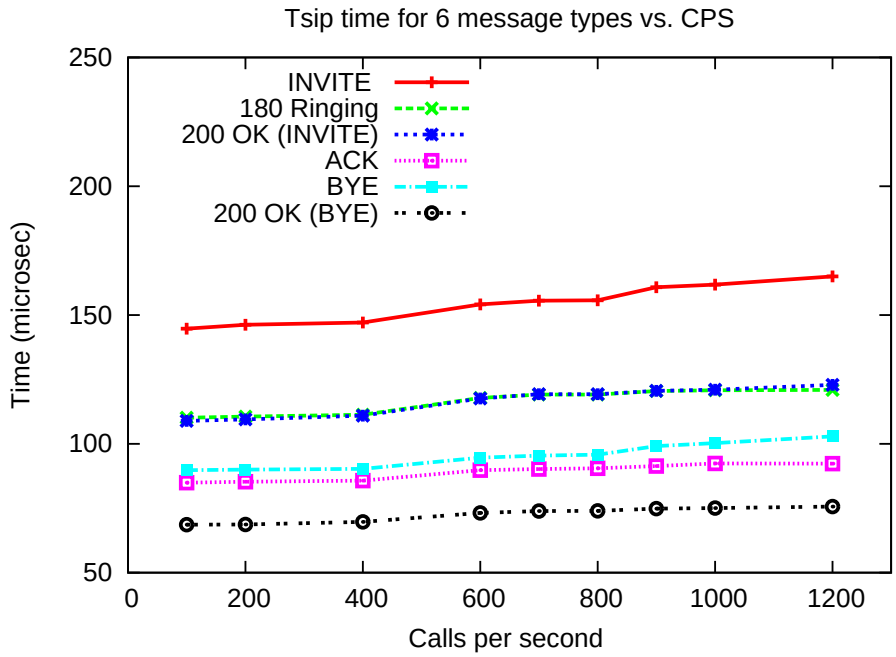

Figure 4.6: Mean value for T_{sip}, Poisson Arrivals

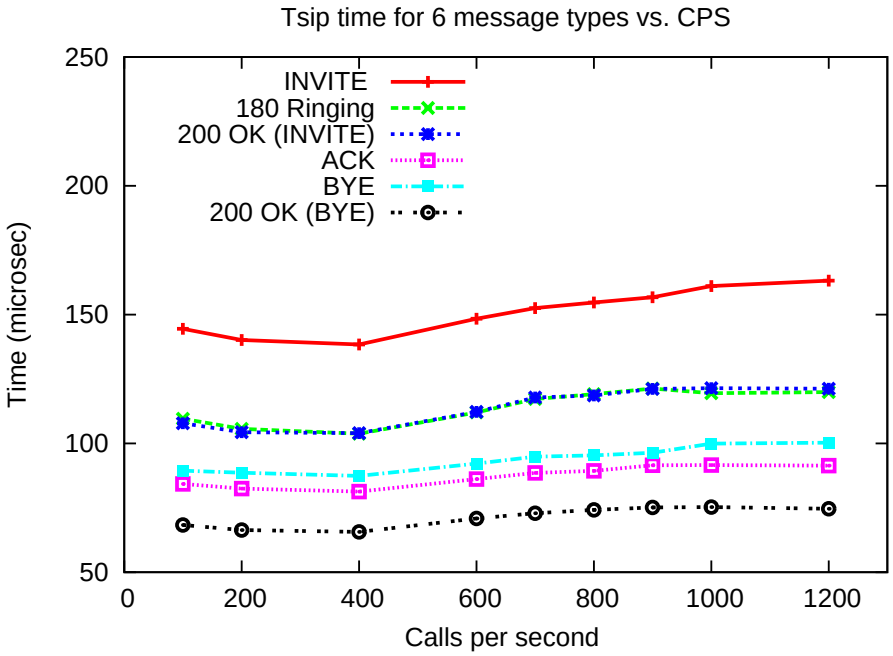

Figure 4.7: Mean value for T_{sip}, Deterministic Arrivals

Another important observation is that mean SIP service times for all message types increase almost linearly with the call rate. Recall that an SPS operating in stateful mode needs to perform table look-ups for each incoming message, so as to match an existing transaction or create a new one. As the call rate increases, the number of transactions in the system also increases, resulting in larger tables at the SIP layer and, hence, longer look-up and overall service times. Part of the service time increase may also be attributed to the higher logging rate at higher call rates; this overhead cannot be avoided, but we believe that it is not a significant factor.

4.2.3 Experimental Results: Measurement Data for K_{snd}

Figures 4.8 and 4.9 present values of the kernel service time K_{snd} incurred for sending a packet received from the SIP layer down to the device driver for exponential and deterministic inter-arrivals respectively. We observe that this service time varies only slightly for each message type. However, across various call arrival rates, the value remains fairly constant for a given message type.

4.2.4 Experimental Results: Overall mean for K_{sockq} and T_{sip}

Figure 4.10 shows the overall average and the confidence interval over all the message types for K_{sockq} for both exponential and deterministic inter-arrivals. In the earlier figures for K_{sockq} we showed the data in the stable region. This figure shows the point where the waiting time vastly increases as the call rate continues to increase and the SPS reaches a point, where the system is not stable (call rates above 900cps).

Figure 4.11 presents the overall mean and confidence interval over T_{sip} over all six message types, for both exponential and deterministic inter-arrivals. An important observation is that mean SIP service times for both inter-arrival types increase almost linearly with the call rate. Also, the service times for both types are almost equal for various call rates.

4.3 $M/G/1$ Queuing Model for the SPS

We now develop an analytical model for predicting the packet-level performance of the SPS, and specifically, the packet waiting time. The experimental data we presented in the previous section indicate that the SPS packet service time distribution exhibits six modes, corresponding to the six SIP message types. Therefore, we model the SPS as an $M/G/1$ queue [32] as shown in Figure 4.12. Specifically, this single queue models the performance of the SPS from the time the packets arrive at the socket queue until they depart from the kernel after undergoing SIP

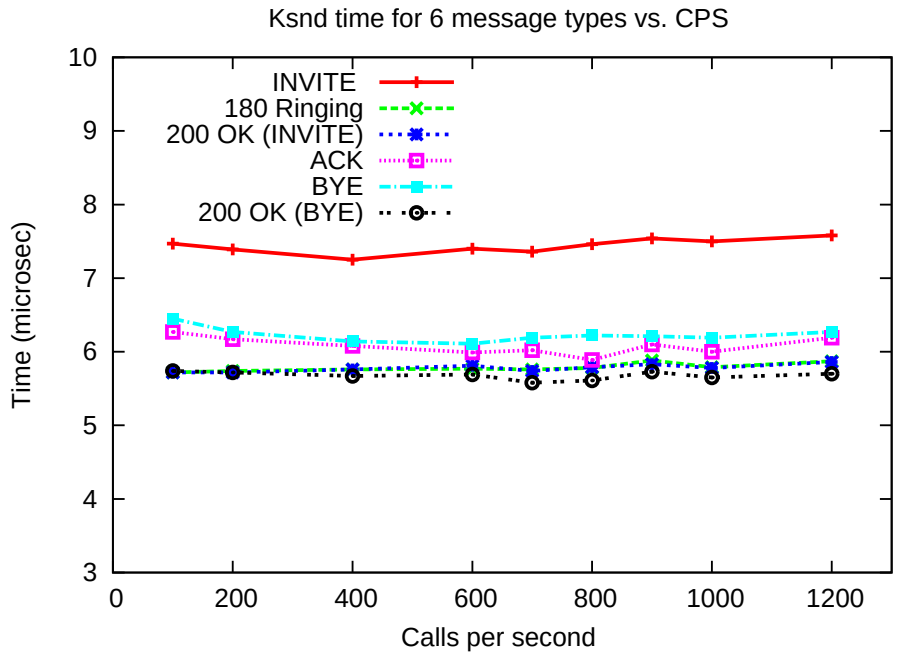

Figure 4.8: Mean value of K_{snd}, Poisson Arrivals

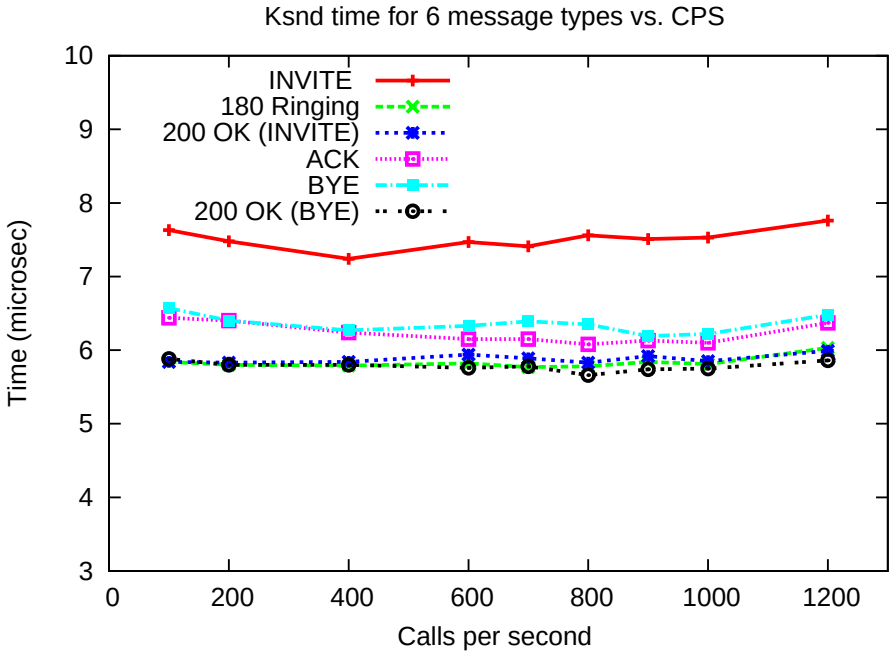

Figure 4.9: Mean value of K_{snd}, Deterministic Arrivals

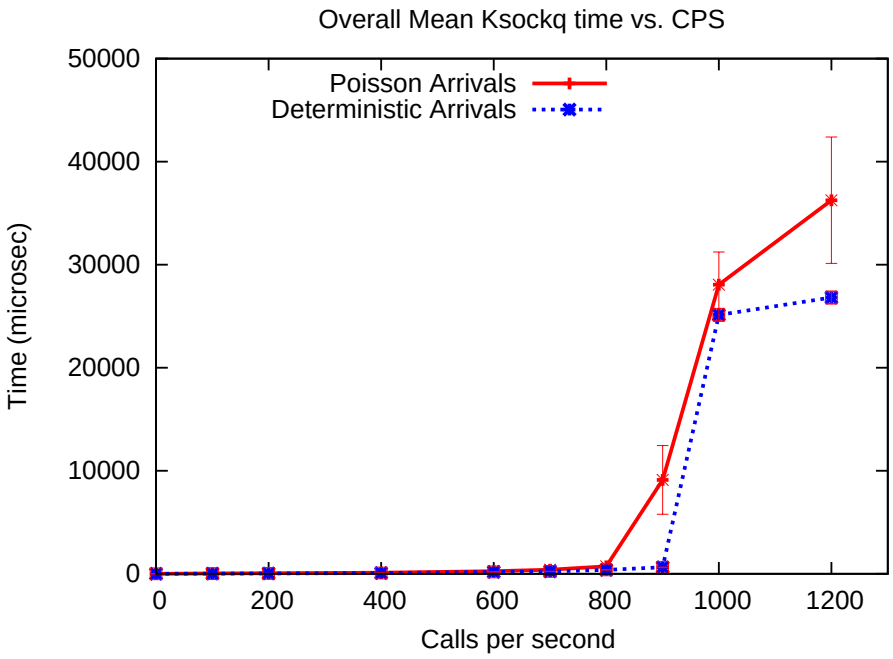

Figure 4.10: Overall Mean value of K_{sockq} and Confidence Interval

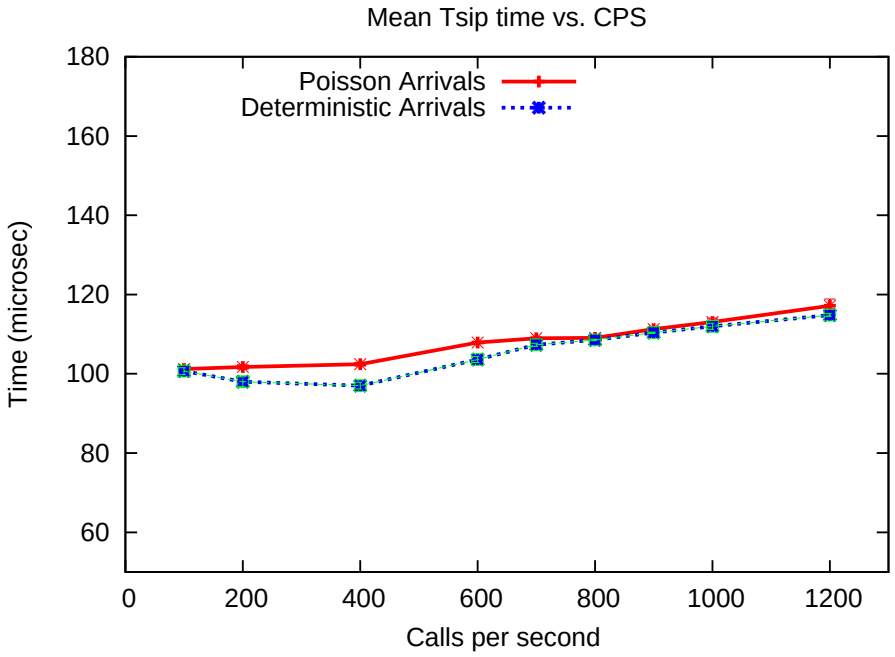

Figure 4.11: Overall Mean value of T_{sip} and Confidence Interval

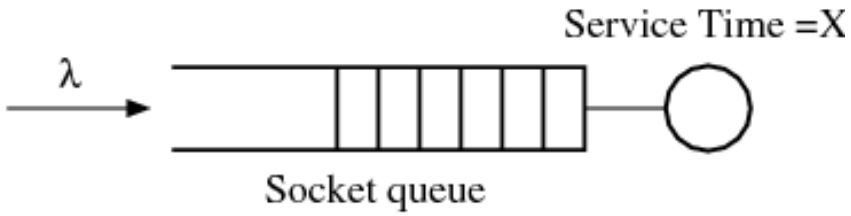

Figure 4.12: $M/G/1$ queuing model of the SPS

processing. From our earlier discussion, the service time X of a packet may be expressed as:

$$X = K^s_{sockq} + K_{copy} + T_{sip} + K_{snd}, \tag{4.1}$$

where K^s_{sockq}, K_{copy} and K_{snd} are the socket, copy and send service times, respectively, in the kernel, and T_{sip} is the service time at the SIP layer.

Despite its simplicity, this model is sufficiently accurate for capturing the performance of the SPS. First, recall that the kernel stack processing times K_{stack} on the receive side are constant around 2 μs across the various arrival rates. Therefore, queuing times at the ring buffer of the device driver are negligible compared to the queuing times at the socket queue and the SIP service time, as shown in Figures 4.2 and 4.6, respectively. Hence, we believe that a single queue model accounting for the socket queue is sufficient. Second, it is clear that a model that assumes an infinite queue (such as the one in Figure 4.12) will not be valid at loads that result in substantial packet losses. Nevertheless, service providers are highly unlikely to operate the SPS at the loads at which material losses occur. Therefore, it is sufficient for the $M/G/1$ model to predict accurately the load at which such losses will start to occur through the estimation of average waiting times.

The waiting time for the $M/G/1$ queue is calculated using the well-known Pollaczek-Khinchin formula [32]:

$$W = \frac{\lambda E[X^2]}{2(1-\rho)}. \tag{4.2}$$

In this expression:

- λ is the packet arrival rate, expressed in packets per unit time;
- $\rho = \lambda E[X]$ is the server utilization;
- $E[X]$ is the packet service time at the SPS; and
- $E[X^2]$ is the second moment of the packet service time. Let $E[X_i], i = 1, 2, \ldots, 6$, denote the mean service time of the six SIP message types. Since each call generates exactly six

messages, one of each type, the second moment of the service time may be obtained as:

$$E[X^2] \;=\; \frac{1}{6}\;\sum_{i=1}^{6}(E[X_i^2]). \tag{4.3}$$

Therefore, we use the estimates of $E[X]$ and $E[X_i^2]$ from the measurement data to obtain the mean packet waiting time using expression (4.2).

4.3.1 Estimating the K^s_{sockq} Component of the Service Time X

The service time of a packet consists of the four components in expression (4.1). In our experiments, we have measured directly three of the components, K_{copy}, T_{sip}, and K_{snd}. The fourth component, K^s_{sockq}, represents the processing time that is incurred by the packet from the instant, t_{stack}, it is added to the socket queue until the instant, t_{sockq}, it is removed from the queue (refer to Figure 3.2). K^s_{sockq} is one of the two components of K_{sockq}, as seen in expression (3.1); the other component, K^w_{sockq}, represents the waiting time of the packet at the socket queue. Although we have directly measured K_{sockq}, it is important to have a accurate measurement of K^s_{sockq} (and, consequently, K^w_{sockq}) to apply the $M/G/1$ model. To this end, we first obtain a baseline measurement of K_{sockq} under conditions of no queuing, and then we adjust this baseline value for higher call rates by accounting for the interrupt overhead.

The K_{sockq} Component Under No Queuing

We conducted a separate experiment to obtain the time values t_{arr}, t_{stack}, t_{sockq} and t_{sip-in} under conditions that ensured no queuing at the kernel socket as the packets move through the network stack on the way to the SIP layer. Let us denote this quantity as $K_{sockq-noq}$. We use this quantity as the baseline value $K^{s,base}_{sockq}$, i.e., $K^{s,base}_{sockq} \;=\; K_{sockq-noq}$.

To measure the $K_{sockq-noq}$, we conducted an experiment in which we (1) generated SIP calls at a rate of 1 cps, since at this rate packets belonging to different calls do not interfere with each other; (2) modified the UAS configuration to add a 250 ms pause between sending the `180 Ringing` and `200 OK` messages (for the same call), so as to avoid having the latter message queued behind the former one at the SPS; and (3) modified the UAC configuration to add a 250 ms pause between sending the `ACK` and `BYE` messages to end the call, again to avoid queuing of the latter message at the SPS. In this experiment, we generated 10,000 calls and computed the average value of $K_{sockq-noq}$ for each packet type. The results are shown in Table 4.3. We observe that there is little difference in the kernel processing time across the six message types. This result is expected as (1) all message types undergo identical processing inside the kernel, and (2) the packet size does not vary significantly across messages for the

Table 4.3: Measured Mean Values (in μs) at 1 cps

Message Type	Packet Size (B)	K_{stack}	$K_{sockq}^{s,base}$	K_{snd}
INVITE	624	2.6	8.3	6.15
180 Ringing	375	2.2	7.6	5.2
200 OK (INVITE)	542	2.5	8.5	5.5
ACK	466	2.2	7.4	5.6
BYE	466	2.5	8.1	5.9
200 OK (BYE)	367	2.2	7.5	5.2
Overall Mean		2.4	7.9	5.6
Conf. Interval (Half-Width)		0.014	0.196	0.013

data copying operations to make a difference in the processing time.

However, it is not reasonable to set the value of K_{sockq}^{s} in expression (4.1) equal to the baseline value $K_{sockq}^{s,base}$ shown in Table 4.3, as doing so would result in an underestimation of the actual K_{sockq}^{s} values under higher call arrival rates. Specifically, at 1 cps there is no overhead due to interrupts, whereas as the call arrival rate increases, this overhead becomes substantial, as we discussed earlier. The interrupt overhead effectively increases the service time of each packet within the socket queue, and this factor must be taken into account explicitly in order to provide a robust estimate of the overall packet service time to be used in the waiting time formula (4.2).

Modeling the Interrupt Overhead

From the basic computer architecture principles [33], the execution time of an operation is given as the product of (number of instructions) $\times$ (cycles per instruction) $\times$ (time per cycle). Interrupts pollute the cache, increasing cache misses, and in turn increasing the number of cycles per instruction; the other two values in the product are constants for a given operation. We model this interrupt overhead by expressing the K_{sockq}^{s} service times as a function of the baseline value $K_{sockq}^{s,base}$ and the server utilization ρ, as follows:

$$K_{sockq}^{s}(\lambda) \;=\; \alpha(\lambda)\; K_{sockq}^{s,base}. \tag{4.4}$$

In the above expression, parameter α, given as a function of λ, adjusts the service time $K_{sockq}^{s,base}$ under no queuing delay (i.e., under minimal interrupt overhead) to account for the cache misses due to interrupts under a given packet arrival rate λ.

Based on our experimental results, we model parameter $\alpha(\lambda)$ as a piece-wise linear function of the arrival rate λ, expressed in units of packets/sec, such that: $\alpha(0) = 1$, $\alpha(600) = 3$, $\alpha(1200) = 5$, and $\alpha(\lambda) = 8$ for $\lambda \geq 2400$ for exponentially distributed inter-arrivals. This

Table 4.4: Waiting Times (in μs): Measured vs. Analytical, Poisson Inter-Arrivals

Model Parameters	Call Arrival Rate								
	100cps	200cps	400cps	600cps	700cps	800cps	900cps	1000cps	1200cps
λ (packets/sec)	600	1200	2400	3600	4200	4800	5400	6000	7200
$\alpha(\lambda)$	3.0	5.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0
$E[X] = K^s_{sockq}(\lambda) + K_{snd} + K_{copy} + T_{sip}$	133.4	149.88	174.53	179.98	181.09	181.185	183.49	185.23	189.5
$\rho = \lambda E[X]$	0.080	0.1798	0.4188	0.6479	0.7606	0.8698	0.9908	1.111	1.3644
W (model)	6.0	16.89	64.21	169.19	293.98	617.77	10171.9	N/A	N/A
W (measured) $= K^w_{sockq}$	5.485	13.59	53.27	190.92	333.52	664.57	9049.07	27997	36194

Table 4.5: Waiting Times (in μs): Measured vs. Analytical, Deterministic Inter-Arrivals

Model Parameters	Call Arrival Rate								
	100cps	200cps	400cps	600cps	700cps	800cps	900cps	1000cps	1200cps
λ (packets/sec)	600	1200	2400	3600	4200	4800	5400	6000	7200
$\alpha(\lambda)$	1.5	3.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0
$E[X] = K^s_{sockq}(\lambda) + K_{snd} + K_{copy} + T_{sip}$	121.03	130.22	152.97	159.89	163.54	164.77	166.59	168.2	171.29
$\rho = \lambda E[X]$	0.0726	0.156	0.3671	0.5756	0.686	0.7909	0.8995	1.009	1.233
W (model)	4.93	12.45	45.39	111.12	183.89	319.68	765.55	N/A	N/A
W (measured) $= K^w_{sockq}$	0.31	6.87	26.4	137.35	185.39	335.17	577.93	25085.8	26761

function reflects our observations that (1) as the packet arrival rate increases, the cache becomes more polluted resulting in higher memory access times, (2) the marginal rate at which cache pollution increases diminishes with increasing packet arrival rates, and (3) after a point, the cache will always be polluted, hence there would be no further deterioration due to further increases in the packet arrival rate. For deterministic inter-arrivals the value of alpha are $\alpha(0) = 1$, $\alpha(600) = 1.5$, $\alpha(1200) = 3$, and $\alpha(\lambda) = 6$ for $\lambda \geq 2400$. For deterministic inter-arrivals it is possible that there is a bigger percentage of remnants of data in the cache that are re-usable and this results in a lower value of alpha compared to exponentially distributed inter-arrivals

Using the values for α from this function, Tables 4.4 and 4.5 compare the measured waiting times to the ones obtained through the $M/G/1$ model for exponential and deterministic distribution respectively. We observe that there is a good match between analytical and measured values in the stable region, i.e., up to 800 cps for exponential distribution. At arrival rates of 900 cps or higher, the system becomes unstable, losses increase sharply; hence the $M/G/1$ model is not valid. In fact, in this overload region, the estimated value of ρ is higher than 1; hence the Pollaczek-Khinchin formula (4.2) cannot be applied.

Despite their simplicity, the $M/G/1$ and interrupt overhead models are sufficiently accurate in two aspects that are important to service providers: jointly they (1) correctly predict the

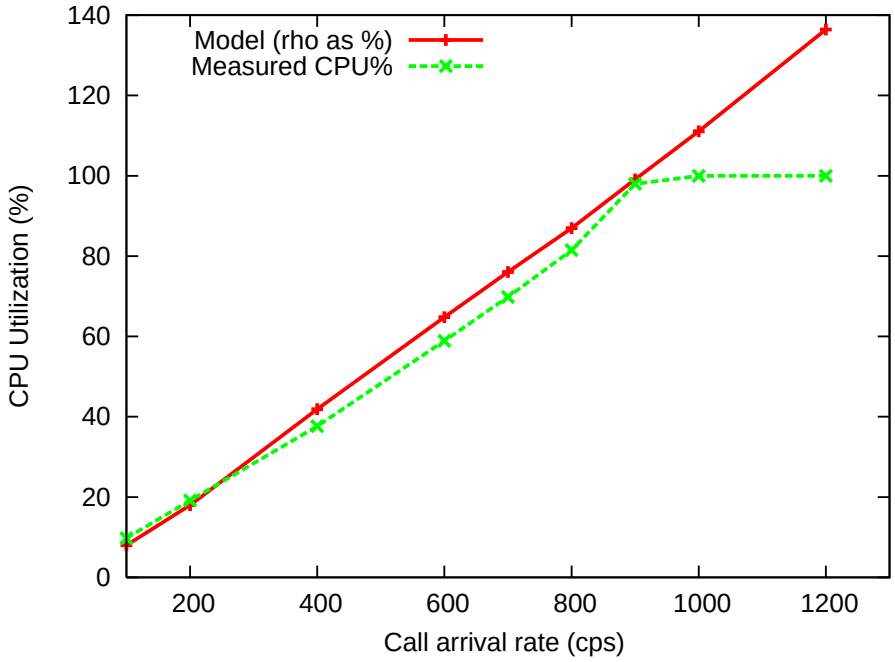

Figure 4.13: Server utilization: CPU% vs. ρ (as %), exponential inter-arrival times

measured packet waiting times at the socket queue within the stable region, and (2) accurately predict the transition to the overload region through the value of ρ. Using this model, service providers only need to monitor the packet arrival rate λ at the SPS to be able to estimate the packet waiting times, as well as detect whether congestion is imminent; In the latter case, providers could take actions to mitigate the incipient congestion, e.g., by activating another SPS and by applying load balancing techniques.

As a further validation of the $M/G/1$ model, Figure 4.13 compares the experimental and analytical values of server utilization for exponential inter-arrival times. The experimental values were measured using the `mpstat` command for about 80% of the duration of the experiment, while the analytical value corresponds to the offered load ρ predicted by the $M/G/1$ model.

We observe a good agreement between analytical and measured values up to full load, beyond which the infinite-queue $M/G/1$ model is not valid.

4.4 Conclusions

We have presented a measurement methodology and carried out a large set of experiments to characterize the performance of the OpenSIPS SPS as a function of call arrival rate. We have

also presented an $M/G/1$ model of the SPS that takes into account the interrupt overhead. The tools and methodology we have developed can be adapted to investigate the performance of a range of application servers. In the next chapter, we investigate the impact of multiple threads on the performance of the SPS.

Chapter 5

Performance Evaluation of Single-Core, Multi-Threaded SIP Servers

Multi-threading is a widely used program execution model, where each thread executes independently, while sharing some of the process resources. Multi-threaded processes are used for a range of network application servers including web-servers, mail-servers, and for SIP proxy servers (SPS) for voice over IP (VoIP). In this chapter, we investigate the performance of OpenSIPS, an open source SIP proxy server, in a multi-threaded environment.

In Chapter 4, we used waiting time as the metric for evaluating the performance of the SPS in a single-server, single-core system. We observed that the waiting time increases several orders of magnitude from few microseconds to about 35 milliseconds, as a function of the call-arrival rate. Though there is a significant change in waiting time with call-arrival rate, the values are low and end-users may not notice any delay. To provide a better performance measure of SPS for service providers, we focus on packet drop-probability as the key performance metric, in this chapter. The reasons for focusing on drop-probability are:

- Using drop-probability service providers can derive a single point where the SPS starts experiencing packet loss, for a given number server threads and call-arrival rates. Service providers can adjust the threshold that represents an acceptable loss-rate.
- Since we are focusing on the control-plane performance measure of SPS, packet-loss of control packets translates to call-establishment problems that will be experienced by end-users.

The contributions we have made in this chapter are enumerated here:

1. We have collected a large set of experimental data, in a methodical fashion to characterize the performance of SPS under increasing server threads and under increasing call arrival rates.

2. We develop a queuing model to predict the drop probability of the multi-threaded system.

3. In addition to the Interrupt-overhead we introduced in Chapter 4, we identify a new parameter to capture the overhead due to resource contention among multiple server threads.

5.1 Related Work

There have been several studies in the literature that have investigated the impact of multi-threading. The most relevant to our current research are [34], [35] and [36]. In [34], a simulation study was done to see the impact of multi-threading on cache performance. In [36] the authors developed techniques to determine optimal allocation of threads for a specific Quality of Service (QoS) objective and used realistic workloads on a typical web server to show the efficacy of the new methodology. In [35], the complexity of optimal configuration of a multi-threaded web server for different workloads was explored. There are also several studies that explore the impact of multi-threading in a multi-core environment; this is an area we cover in Chapter 7 of our thesis. These studies, though they investigate multi-threaded performance, significantly differ from our study. In our study we measure the performance in terms of packet drop-rate probability as a function of number of SPS threads. The kernel and the SPS was modified to get the packet waiting time in the queue, we then develop a drop-probability model based on our results that can be applied to predict performance of any SPS system.

5.2 Experiments and Performance Measurements

In this section we describe the experiments and the corressponding performance measurements.

We use the same testbed that we have described in Chapter 4, and shown in Figure 4.1. The testbed contains the OpenSIPS SPS Proxy server running on a quad-core Intel® Xeon™ E5540 @2.53GHz processor with 8192 KB cache size. The other components of the testbed UAC, UAS and router are as described in chapter 4. All the experiments are performed with UDP as the transport protocol and with default kernel configurations.

The measurement methodology described in Chapter 3 is used to measure the components K_{rcv}, K_{stack}, K_{sockq}, K_{snd} and T_{sip} of the time each SIP packet spends at the SPS. In addition, for each experiment, the overall drop rate is measured by using the 'netstat -su' command. The command provides the statistics for UDP packets for the system.

For each experiment, the UAC initiates more than 100,000 calls to UAS. Each call is accepted by the UAS, resulting in the message exchange shown in Figure 2.2. Therefore, for each call, six different messages are generated by either the UAC or the UAS and are forwarded to the other party via the SPS. In other words, for each experiment, the SPS may process more than 600,000 SIP messages, i.e., up to 100,000 messages of each type shown in Figure 2.2. To measure the performance of SPS under multiple threads, two cores of the quad-core processor were enabled such that, all the SPS threads were run on one core and 'syslogd' was run on another core. There was no other user-initiated process in the system.

Each experiment is characterized by three parameters:

- *Call arrival rate:* We varied the call arrival rate from 200 cps to 4000 cps. The maximum call rate was set to the point where the SPS crashes frequently as it cannot handle the message volume, indicating that the server is severely overloaded.

- *Call inter-arrival time distribution:* The call inter-arrival time is the time between two consecutive `INVITE` messages generated by the UAC. For each call arrival rate, we generated inter-arrival times using exponential distributions.

- *Number of server threads:* The total number of SPS server threads that are running as part of SPS server. The experiments were conducted for 1,2,4,6,8 and 16 server threads.

For each packet processed by the SPS, we log the seven time values t_{arr}, t_{stack}, t_{sockq}, t_{sip-in}, t_{sip-c}, $t_{sip-out}$ and t_{dep}. We then process the log files to obtain the sample mean values for quantities K_{rcv}, K_{stack}, K_{sockq} T_{sip}, and K_{snd}; we record both the overall mean (i.e., across all packets) and the mean per packet type. We use the method of batch means to estimate 95% confidence intervals around the overall mean.

5.2.1 Experimental Results: Measurement Data for K_{rcv}, K_{stack}, and K_{sockq}

The measured values for quantity K_{rcv}, under exponentially distributed inter-arrival times and for various call rates and for various number of server threads are shown in Figure 5.1. The figure shows the cumulative mean value over the mean values of all message types.

The K_{sockq} and K_{stack} values we have obtained are similar to the data listed in Chapter 4. We have observed that kernel stack processing times K_{stack} are largely constant, averaging 2 μs independent of message type and call arrival rate (refer also to Table 4.3 for the K_{stack} values obtained through a different experiment). The values of K_{copy}, obtained as $(K_{rcv} - K_{stack} - K_{sockq})$, are also constant at around 2 μs. Hence, K_{rcv} values are about 4 μs higher than the K_{sockq} values.

We observe that all mean values of K_{rcv} and K_{sockq} increase rapidly with the call arrival after a certain point for all cases. Specifically, a single SPS server process has the least K_{rcv} value.

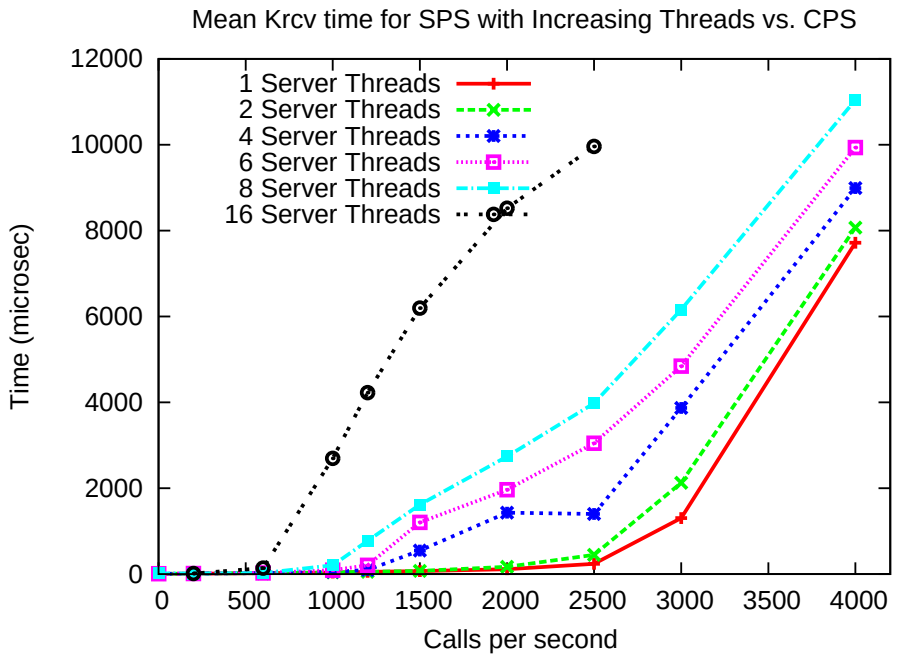

Figure 5.1: Krcv value for SPS as a function of the number of server threads

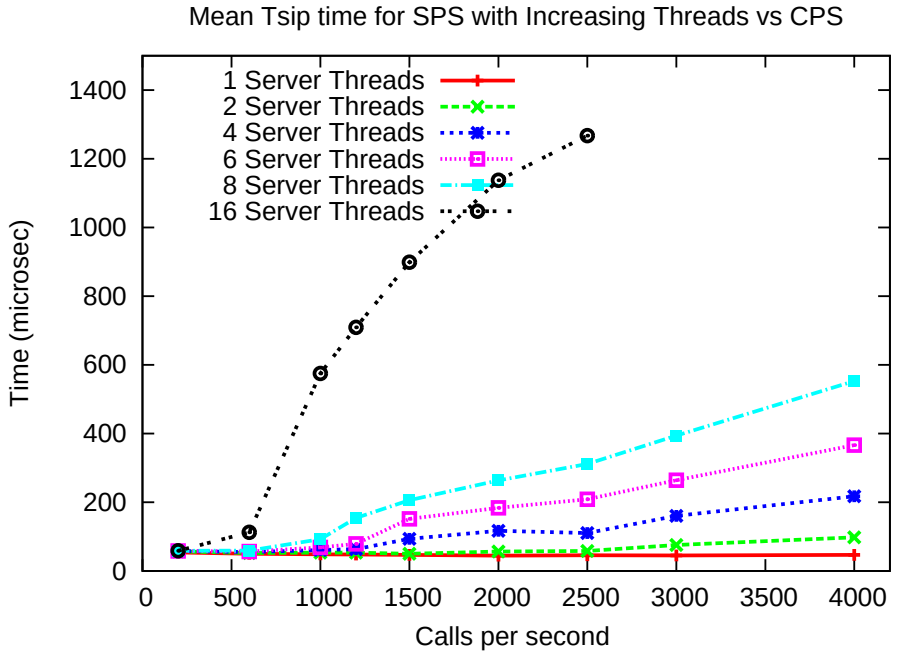

Figure 5.2: Tsip value for SPS as a function of the number of server thread

As server threads are added, the overhead due to the additional threads results in increased service time for the same call rate. This further matches what we know about SPS, i.e, each server thread processes a packet completely after receiving it from the queue and then hands it for transmission. As only one server thread can run at a given instance due to only one core being available, we see that, increasing the number of server threads, leads to increasing overhead in switching the process context and results in higher waiting time.

The figure for K_{rcv} can be explained by the following factors:

- *Queuing delay.* As the call arrival rate increases, packets arriving at the SPS are buffered at the socket queue and experience increasing waiting times (K^{w}_{sockq}) before being delivered to the SIP layer. This is the biggest factor that causes the increase in K_{rcv} as the call rate increases.

- *Interrupt overhead.* The number of interrupts increases in direct proportion to the packet arrival rate. Interrupts introduce overhead in the form of the time needed to handle each interrupt, the context-switching operations, and the increase in processing time as a result of cache misses due to these interrupts. A cache miss causes the number of CPU cycles consumed by the network-stack receiving process to increase, as the memory access cycles of main memory are several times that of L2 caches. The impact of cache misses on the network-stack was described in Chapter 3.

- *Thread overhead.* As the number of server threads increases, the overhead associated with the context switching for each thread increases and this causes additional misses in the cache for packets that are in the queue.

5.2.2 Experimental Results: Measurement Data for T_{sip}

Figure 5.2 presents the SIP layer processing times T_{sip} under exponentially distributed call inter-arrival times and for various call rates and varying number of server threads.

The figure presents the overall mean value for all message types. All values are expressed in μs. The overall trend for T_{sip} value is that it increases proportional to the number of threads. For 16 server threads this value is significantly higher compared to one and two threads. This increase can be attributed to the additional context switching and cache miss overhead.

5.2.3 Experimental Results: Measurement Data for K_{snd}

We have observed that the values of the kernel service time K_{snd} incurred for sending a packet received from the SIP layer down to the device driver varies only slightly for each message type. However, across various call arrival rates and server threads, the value remains fairly constant

for a given message type. The overall mean value is around 6 μs (refer also to Table 4.3 for the K_{snd} values obtained through a different experiment).

5.3 Drop-Probability Model for the SPS

As we mentioned at the beginning of this chapter, our focus is on drop-probability as the key performance metric of SPS. The reasons are:

- Service providers will have single point of focus for the performance of SPS.
- End-users start experience call-establishment problems due to packet drops.

Based on our survey of industry standards, the threshold for acceptable drop-rate in VoIP environment is about 1% [37]. This is used in industry as the point below which the voice quality can still be considered toll quality. Though this data is for the voice data-plane, we use 1% as the threshold for packet-loss in control plane to reflect the impact on end-users, that service providers can use as a trigger to add more capacity to the system.

In Chapter 4, we modeled the single server SPS process as an $M/G/1$ queue. In the experiments we conducted for multiple server threads, we see that as the number of server threads increases, the SPS suffers from packet loss as call rate increases. Therefore, we enhance the queuing model to be a $M/G/c/K$ model as shown in Figure 5.3, to reflect the multiple server threads and also the limited buffer of the socket queue. This single queue, models the performance of the SPS, from the time packets arrive at the socket queue until they depart from the kernel after undergoing SIP processing. From our earlier discussion, the service time X of a packet may be expressed as:

$$X = K^s_{sockq} + K_{copy} + T_{sip} + K_{snd}, \tag{5.1}$$

where K^s_{sockq}, K_{copy} and K_{snd} are the socket, copy and send service times, respectively, in the kernel, and T_{sip} is the service time at the SIP layer.

Despite its simplicity, this model is sufficiently accurate for capturing the drop probability of the SPS. First, recall that the kernel stack processing times K_{stack} on the receive side are constant around 2 μs across the various arrival rates. Therefore, queuing times at the ring buffer of the device driver are negligible compared to the queuing times at the socket queue and the SIP service time, as shown in Figures 5.1 and 5.2, respectively. Hence, we believe that a single queue model accounting for the socket queue is sufficient. The additional server threads that are run as part of the SPS server all process packets from this single queue.

The queue size K is determined by the socket buffer size SK_RMEM_MAX allocated by the system, based on the packet size of the various SIP messages for SPS. Size of the queue for

the SPS process is approximated as $K = 200$.

An exact solution for the $M/G/c/K$ model is only possible for special cases, such as for exponential service time, a single server, or no waiting room at all. Given the complexity of this problem, a robust and efficient approximations for the blocking probabilities of these systems is developed by Smith in [38]. In that paper, the author developed an expression for p_K, the drop probability of an $M/G/c/K$ queue. The expression was derived by first looking at the optimal buffer size of the system, and then inverting the buffer expression to yield the expression for p_K. The approximation formula for the p_K value for the $M/G/c/K$ system is based on a weighted combination of the formulas for the $M/M/c/K$ optimal buffers. Given that for our SPS system, we have K approximated at 200, the queue is sufficiently large.

The blocking probability p_K for both $M/M/c/K$ and $M/G/c/K$ as $K \to \infty$ can be given by:

$$lim_{K\to\infty}(p_K) = 0 \tag{5.2}$$

- p_K is the long run probability that arrivals are rejected, i.e blocking probability of finite queue of size K.

This is because as the queue size increases the number of packet drops decreases for the queues. Given the large size of queue for the SPS system and $M/M/c/K$ being a special case of $M/G/c/K$, we now use the blocking probability of $M/M/c/K$ to model the SIP proxy servers blocking probability. Taking this approach serves two purpose:

- Using the blocking probability of $M/M/c/K$, we can use a closed form equation to obtain the blocking probability of SPS.
- It will provide flexibility in approximating the blocking probability of various combinations of number of servers c and buffer size K. This will greatly aid service providers in capacity planning, which is one of our primary goals.

The analytical results for $M/M/c/K$, has the following expression for p_K

$$p_K = \frac{1}{c^{K-c}c!}\left(\frac{\lambda}{\mu}\right)^K p_0 \tag{5.3}$$

The expression for p_0 is given by:

$$p_0 = \left[\sum_{n=0}^{c-1}\frac{1}{n!}\left(\frac{\lambda}{\mu}\right)^n + \frac{(\lambda/\mu)^c}{c!}\frac{1-(\lambda/c\mu)^{K-c+1}}{1-(\lambda/c\mu)}\right]^{-1} \tag{5.4}$$

The various terms in Eq.(5.3) and(5.4) are:

- p_0 is the unconditional probability that there are no customers in the queue or are being serviced.
- λ is the packet arrival rate, expressed in packets per unit time;
- μ is the packet service rate, this is obtained from $E[X]$, the packet service time at the SPS; where $\mu = 1 \ / \ E[X]$
- $\rho = \frac{\lambda E[X]}{c}$ is the server utilization;
- c is the number of SPS server threads running on the system
- K is the queue length of the SPS server, in terms of number of packets the queue can hold.

As we have explained in Chapter 4 and shown again in expression (5.1), the service time of the packet processed by SPS consists of four components. In our experiments, we have measured directly three of the components, K_{copy}, T_{sip}, and K_{snd}. The fourth component, K^s_{sockq}, represents the processing that is incurred by the packet from the instant, t_{stack}, it is added to the socket queue until the instant, t_{sockq}, it is removed from the queue (refer to Figure 3.2). K^s_{sockq} is one of the two components of K_{sockq}, as seen in expression (3.1). The other component, K^w_{sockq}, represents the waiting time of the packet at the socket queue. Although we have directly measured K_{sockq}, it is important to have an accurate estimate of K^s_{sockq} (and, consequently, K^w_{sockq}) to apply the $M/M/c/K$ model. To this end, we use the baseline measurement of K_{sockq} under conditions of no queuing as described in Chapter 4, and then we adjust this baseline value for higher call rates by accounting for the overhead caused by cache misses and server thread overhead.

5.3.1 Modeling the Interrupt Overhead:

In Chapter 4, for the single single server thread case, the queuing model captured the Interrupt overhead parameter $\alpha(\lambda)$ as the packet arrival increased. Interrupts pollute the cache, increasing cache misses, and in turn increasing the number of cycles per instruction; the other two values in the product are constants for a given operation. The interrupt overhead still applies as additional threads are added to the system and we continue to use this parameter as before in our new model.

5.3.2 Modeling the Server-Thread Overhead:

For modeling the overhead introduced by additional threads we look at the underlying process scheduling policy used by the Linux OS. The main purpose of the scheduler is to provide fairness

among different processes, maintain high-throughput for the system and achieve maximum utilization of the CPU. Some of these parameters can be conflicting with each other. Linux Kernel uses a scheduling mechanism called Completely Fair Scheduling (CFS) [39].CFS is a variant of Weighted-Fair Queuing (WFQ) and the main idea behind the CFS is to maintain balance (fairness) in providing processor time to tasks. CFS uses Red-Black trees to get the next process to run based on a 'virtual run-time'. CFS has a concept called `group scheduling` (introduced with the 2.6.24 kernel). Group scheduling is another way to bring fairness to scheduling, particularly in the face of tasks that spawn many other tasks as is the case with SPS process with multiple server threads. The server process that spawns many task share the same 'virtual run-time' (parameter used by CFS to schedule process to run). As the system load increases, in order to achieve some useful work by the system, CFS relaxes the fairness policy to reduce the context switching overhead of the threads and also to achieve some throughput for the system.

As the number of threads in the system increases, there is additional process scheduling overhead, and additional pollution of the instruction cache, in addition to the data cache pollution with increasing packet arrival rate. The additional server threads add to the time for context switching for the next available server thread to remove the packet from the queue to process the packet. To capture this additional overhead we introduce a new parameter β as a factor of number of threads c to be $\beta(\lambda, c)$.

To capture the impact due to CFS's group scheduling and behavior under load with the parameter β, we update the service time of additional threads to reflect the reduced capacity for each thread as follows:

$$E[X_c] \ = \beta(\lambda, c)(E[X_1] * c) \tag{5.5}$$

Where:

- $E[X_1]$ is the service time with one server thread, with the α parameter captured in it.
- $E[X_c]$ is the service time with c server threads.
- $\beta(\lambda, c)$ is the parameter to capture the additional overhead due to increased number of threads
- c is the number of server threads

Based on our data, we can model β to predict the point where we can start seeing drop-rate exceeding 1%, as a multi-linear function of two parameters, the call-arrival rate λ and the number of server thread.

$$\beta \ = (0.174)c + (0.923) * 10^{-4}\lambda - 0.60 \tag{5.6}$$

Table 5.1: Waiting Time(μs): Measured vs Analytical, Exponential Inter-Arrival Times SPS on 1 core **1-Server**, K=200

Model Parameters	Call Arrival Rate								
	200cps	600cps	1000cps	1200cps	1500cps	2000cps	2500cps	3000cps	4000cps
λ (packets/sec)	1200	3600	6000	7200	9000	12000	15000	14800*	14250*
$\alpha(\lambda)$	1.3	2.0	2.5	3.2	3.2	3.2	3.2	3.2	3.2
$\beta(\lambda, c)$	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$E[X]$	61.65	60.66	63.12	67.01	65.61	65.16	65.78	67.09	69.842
$\rho = \lambda E[X]$	0.074	0.218	0.378	0.482	0.591	0.782	0.987	0.992	0.995
W (model)	2.57	8.86	20.03	32.28	49.24	121.75	2566.36	4691.1	7679.62
W (measured) $= K^w_{sockq}$	2.60	10.30	26.53	66.69	57.81	123.36	413.29	1743.26	8309.5
$TotalMessages$	1,200,000	1,200,000	1,200,006	1,083,794	1,080,783	933213	835461	530772	556420
$RCVErrors$	0	0	0	0	0	22	4730	12079	91209
$Droprate$	0	0	0	0	0	0	0.00566	0.02275	0.1639

Using the above equation, we can interpret that, for smaller number of server threads, the β value is more dependent on the call-arrival rate, indicating that the losses will be at a higher rate. As the number of server thread increases, the beta value is more dependent on the number of servers.

Using the β value above, Service Providers would need to only obtain the $E[X_1]$ value for the system they would be using. Then, apply the β value for a given number of server threads and call arrival rates, to obtain an approximate service time of the system. This can then be applied to the p_K formula in Eq.(5.3) to obtain a first approximation of the capacity of their system.

5.3.3 Modeling Results

We now present the results for the SPS using the model we just described. Table 5.1 presents the waiting time comparison for modeling vs. measured data as described in chapter 4. For the one-server thread case, we keep β=1.

Tables 5.2, 5.3, 5.4 , 5.5 and 5.6 show the comparison of the drop probability obtained analytically versus the measured drop rate. From the tables we observe that as the server thread count increases, SPS begins to experience packet losses at lower call rates. The β value at the point where the SPS starts experiencing 1% packet drop-rate increases gradually from about 1.08 to 2.69 as the number of server threads increases from 2 to 16.

Figure 5.4 presents the drop probability as a function of the number of servers and call-arrival rates, at the point where the drop probability exceeds 1% and compares the model values to measured values. From the figure 5.4 we can see that the as the number of server thread increases for single-core system, the system begins to experience drops at lower call rates. The model we have developed captures this accurately.

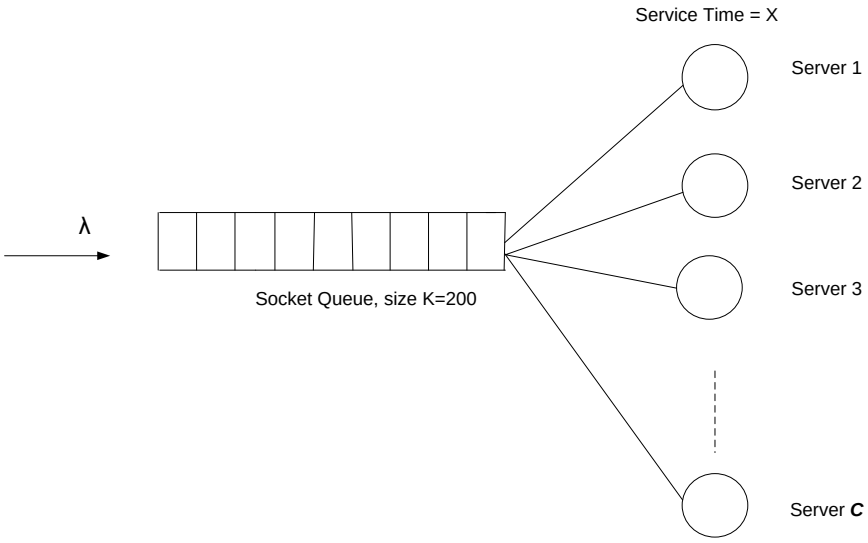

Figure 5.3: $M/G/c/K$ queuing model of the SPS

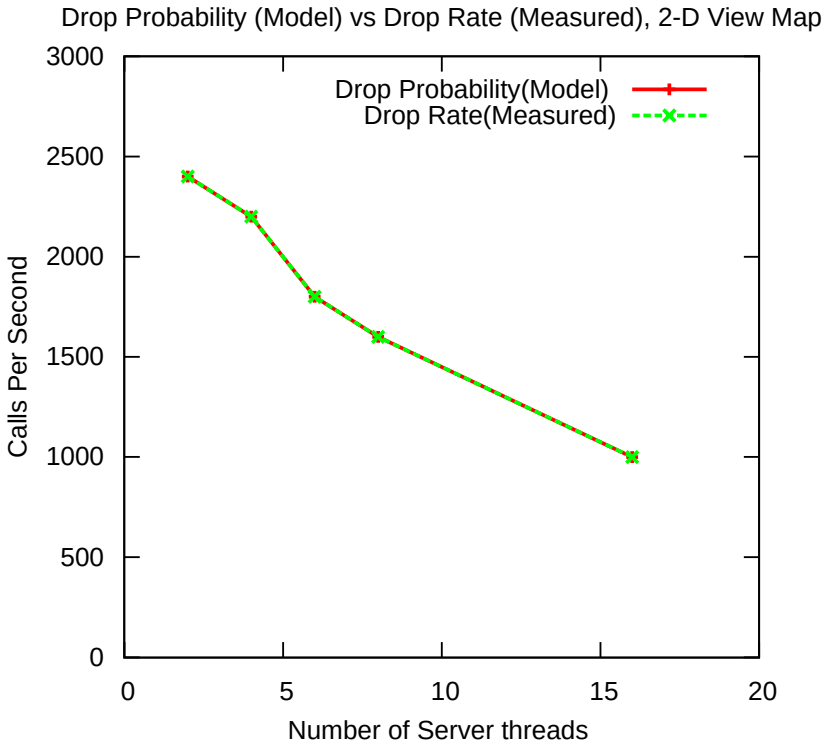

Figure 5.4: Drop probability(Model) vs. Drop-Rate(Measured), Point where Drop rate starts exceeding 1%; 2D-map view as Function of Number of Server threads and Call-Arrival rate

Table 5.2: Drop probability Analytical vs. Drop Rate Measured, SPS with **2-Server, K=200**

Model Parameters	Call Arrival Rate		
	2000cps	2200cps	2400cps
λ (packets/sec)	12000	13200	14400
$\beta(\lambda, c)$	1.158	1.158	1.077
$E[X]$	151.0	151.0	140.37
$E[X_2]$	130.32	130.32	130.32
$DropProbability(model)$	0.00	0.0035	0.0119
$TotalMessages$	927,216	1,462,853	1,207,041
$RCVErrors$	0	5787	16890
$Droprate(measured)$	0	0.0039	0.0139

Table 5.3: Drop probability Analytical vs. Drop Rate Measured, SPS with **4-Server, K=200**

Model Parameters	Call Arrival Rate		
	1800cps	2000cps	2200cps
λ (packets/sec)	10800	12000	13200
$\beta(\lambda, c)$	1.417	1.284	1.314
$E[X]$	372	337.0	342.57
$E[X_4]$	260.64	260.64	260.64
$DropProbability(model)$	0.0075	0.0110	0.115
$TotalMessages$	878,218	942,046	927,456
$RCVErrors$	6,344	9537	14791
$Droprate(measured)$	0.00722	0.0101	0.01595

Table 5.4: Drop probability Analytical vs. Drop Rate Measured, SPS with **6-Server, K=200**

Model Parameters	Call Arrival Rate		
	1400cps	1600cps	1800cps
λ (packets/sec)	8400	9600	10800
$\beta(\lambda, c)$	1.806	1.598	1.4408
$E[X]$	711.0	629.0	563.31
$E[X_6]$	393.66	393.66	393.66
$DropProbability(model)$	0.0031	0.0088	0.0147
$TotalMessages$	1029177	1010682	1053671
$RCVErrors$	3,399	8483	10611
$Droprate(measured)$	0.0033	0.0084	0.01

Table 5.5: Drop probability Analytical vs. Drop Rate Measured, SPS with **8-Server, K=200**

Model Parameters	Call Arrival Rate		
	1200cps	1400cps	1600cps
λ (packets/sec)	7200	8400	9600
$\beta(\lambda, c)$	2.076	1.789	1.678
$E[X]$	1113	959.0	874.75
$E[X_8]$	536.08	536.08	536.08
$DropProbability(model)$	0.0059	0.0093	0.047
$TotalMessages$	977624	1102910	1057955
$RCVErrors$	5852	10556	16577
$Droprate(measured)$	0.0059	0.0096	0.0156

Table 5.6: Drop probability Analytical vs. Drop Rate Measured, SPS with **16-Server, K=200**

Model Parameters	Call Arrival Rate		
	800cps	900cps	1000cps
λ (packets/sec)	4800	5400	6000
$\beta(\lambda, c)$	3.298	2.956	2.737
$E[X]$	3325	2980	2758
$E[X_{16}]$	1008	1008	1008
$DropProbability(model)$	0.00425	0.0087	0.0334
$TotalMessages$	787008	643863	687914
$RCVErrors$	3745	5627	13023
$Droprate(measured)$	0.00475	0.00873	0.0189

5.4 Conclusions

In this chapter, we have presented a large set of experiments that we conducted for the multi-threaded SIP proxy server system. The experiments were conducted as a function of number of server threads for the OpenSIPS SPS, where the SPS was assigned a single CPU-core. We then presented a drop-probability model for the SPS, to predict the point where the SPS starts experiencing more than 1% packet drops. We compared this model against the measured packet drops due to buffer overflow and a good match was found between the model values and the measured data. Based on our results, we can conclude that for a one-core system, the performance of SIP proxy server is best with single-thread. As additional server threads are added, the performance degrades due to lower capacity for each thread and due to resource contention among multiple threads. This matches with the queuing theory that single server with larger capacity performs better than N-servers with $1/N$ capacity of single server.

Chapter 6

Performance of Multi-threaded SIP Servers: The Impact of Scheduler Parameters

Multi-threading can increase the performance of a system in terms of responsiveness by concurrent execution of these threads. The *process scheduler* is a core part of the operating system, and determines which process can execute on the system and the duration it runs. The scheduling policy needs to account for several objectives, including fairness, throughput and response time, which often may be contradictory. In this chapter, we investigate the impact of the Linux Completely Fair Scheduler (CFS) on the performance of OpenSIPS, as a function of the number of threads and the call arrival rate.

In Chapter 5, we used the SIP control packet drop-rate as the key performance metric for the SPS. To provide a better metric to quantify the impact of the control packet loss at SPS on user experience, we continue to use Packet Drop-rate (PDR) as the key performance metric in this phase of our research. We collected a large set of experimental data, in a methodical fashion, to characterize the performance of SPS as a function of number of server threads and increasing call arrival rates.

Our key contributions presented in this chapter are:

- Characterization of the impact of the scheduler on the performance of a multi-threaded SPS, in terms of waiting time and packet drop rate; and
- Identification of the key scheduler parameters of CFS scheduler and concrete guidelines on tuning these parameters to achieve significant performance improvement.

6.1 Related Work

Several studies investigate the impact of the process scheduler in various contexts. An approach to improve the interactivity of user tasks in an Android smartphone environment by passing information about the user task from the Android framework layer to the underlying Linux CFS is implemented in [40]. In [41], the Linux scheduler was analyzed under the presence of network I/O and a certain parameter was tuned to mitigate starvation experienced by some processes. In [42], the authors designed a feedback method between the user process and the Linux scheduler so as to lower the global power budget. Our work differs from these studies in that: (1) we focus on the performance of SPS in terms of SIP control packet drop rate; (2) we modify the kernel and SPS code to obtain accurate packet measurements; and (3) we study the impact of the scheduler and tune its parameters so as to improve the SPS performance.

6.2 Experiments and Performance Measurements

We use a testbed that was described in Chapter 4, and shown in Figure 4.1. The testbed contains the OpenSIPS SPS Proxy server that was described in Chapter 5 running on a quad-core Intel® Xeon™ E5540 @2.53GHz processor with 8192 KB cache size. The other components of the testbed UAC, UAS and the router are as described in chapter 4. All the experiments are performed with UDP as the transport protocol and default kernel configurations. Multi-threading in OpenSips server is implemented as a completely autonomous worker model. Each thread represents a light-weight process in Linux OS. In OpenSIPS, a master process spawns these autonomous worker threads based on a configuration parameter in OpenSIPS.

We measure the various time components of each SIP control packet as it traverses via the SPS. The measurement methodology described in Chapter 3 is used to measure the components K_{rcv}, K_{stack}, K_{sockq}, K_{snd} and T_{sip} SIP packet. In addition, for each experiment, the overall drop rate is measured by using the 'netstat' command.

The experiments conducted are same as those described in Chapter 5. For each experiment, the UAC initiates more than 100,000 calls to UAS. In each experiment, the SPS may process more than 600,000 SIP messages, i.e., up to 100,000 messages of each type seen in Figure 2.2. To measure the performance of SPS under multiple threads, two cores of the quad-core processor were enabled such that all the SPS threads were run on one core and 'syslogd' was run on another core. There were no other user initiated process in the system.

Each experiment was characterized by three parameters as in Chapter 5: *Call-arrival rates*, *Call inter-arrival distribution* and *Number of server threads*

According to [43], the two primary factors operators consider when designing their network is the service availability to end-users and the cost of operating the network. Furthermore,

from an economic standpoint, network operators aim to achieve high server utilization in order to maximize the return on their capital investment. These observations motivate us to develop guidelines for tuning the scheduler parameters that will balance these two conflicting objectives: availability of SIP service and SPS server utilization. As call arrival rate increases, the key is to know the threshold where CPU is getting overloaded resulting in excessive loss of SIP call setup packets and service availability is negatively affected. Therefore, we consider the packet drop rate (PDR) as the key performance metric of interest. Note that the only packets seen by the SPS are SIP call setup and teardown messages. Since the loss of any of these messages affects the call establishment process, we consider PDR as the primary metric that captures the impact of server overload on end-user experience.

In this Chapter, We carry out a large number of experiments to measure the impact of the scheduler parameters on PDR. We use the `netstat` command in each experiment to obtain the number of SIP messages dropped at the SPS. This number is provided by the `RcvErrors` counter that is incremented by the Linux kernel when the receive buffer is full (note that in our experiments only the SIP process is active, hence the `RcvErrors` counter provides an accurate count of dropped SIP messages). Therefore, we estimate the PDR metric as follows:

$$\texttt{PDR} = \frac{\texttt{RcvErrors}}{\texttt{Total SIP MSGs}}$$

6.3 Impact of Process Scheduler on SPS Performance

The main purpose of the process scheduler is to provide fairness among different processes, maintain high-throughput for the system and achieve maximum utilization of the CPU. The Linux kernel uses a scheduling mechanism called completely fair scheduling (CFS) [39]. CFS is a variant of weighted fair queuing (WFQ), and its objective is to maintain fairness in providing processor time to tasks. CFS uses red-black trees to get the next process to run based on the concept of a "virtual run-time".

The main design principle of CFS is to model an ideal, precise multi-tasking CPU. Note that a CPU can run only a single task at a given time, while other tasks are waiting. To ensure fair access to the CPU across all tasks, CFS tracks a task's "fairness imbalance" via a per-task variable referred to as `wait_runtime`, that captures the task's waiting time. The `wait_runtime` is the amount of time the task should be allowed to run on the CPU under completely fair and balanced scheduling. CFS tries to enforce fairness among all its runnable tasks by scheduling the task that has the maximum `wait_runtime` value and, thus, is most in need of CPU time.

CFS also encompasses the concept of `group scheduling`, introduced with the 2.6.24 kernel. Group scheduling allows the scheduler to provide fair access to CPU time across all tasks in the system, and enforces hierarchical fairness among tasks, when a task spawns multiple child

tasks.

CFS uses nanosecond granularity to account for the process times. Linux provides several parameters for tuning the behavior of the CFS scheduler, including the following three that we considered in our study:

- `sched_latency_ns`: A period in which each task runs once.
- `sched_min_granularity_ns`: The minimum time after which a task becomes eligible to be preempted. The scheduler tries to maintain this equality:

$$\texttt{sched\_min\_granularity\_ns} = \frac{\texttt{sched\_latency\_ns}}{\texttt{nr\_tasks}}$$

  where `nr_tasks` is the number of tasks in the queue. If the equality is not met, the CFS scheduler tries to increase the `sched_latency_ns` time to match the increased number of tasks in the queue.

- `sched_wakeup_granularity_ns`: This parameter represents the ability of the task being awakened to preempt the current task. A larger value for this parameter makes it difficult for other tasks to force preemption. Therefore, the function of this is parameter is to reduce over-scheduling.

The above parameters may be used to tune the behavior of schedulers to "desktop" workloads (where the objective is low latency) or "server" workloads (where the goal is to achieve good batching of jobs). The scheduler defaults to a setting suitable for desktop workloads. The values of these parameters are a function of the number of CPUs in the system. For the two-core system used for our experiments, the default values are: `sched_latency_ns` = 10,000,000 (10 ms), `sched_min_granularity_n` = 2,000,000 (2 ms), and `sched_wakeup_granularity_ns` = 2,000,000 (2 ms).

6.3.1 Baseline Server Mode

In our preliminary experiments, we determined that configuring the scheduler in "desktop" mode results in poor performance for the SPS. Therefore, following the recommendations in [44], we modified the values of the three scheduler parameters to move the default scheduler policy to "server" mode, as follows: `sched_latency_ns` = 1,000,000 (1 ms), `sched_min_granularity_n` = 100,000 (100 μ s), and `sched_wakeup_granularity_ns` = 25,000 (25 μ s). With these values, the scheduler allows the threads that are spawned as part of the server to be scheduled more often, improving the overall system performance. We will refer to this configuration as the *baseline* "server" mode; we note that these parameter values are recommended in [44] as a

generic server configuration and do not take into account characteristics or workloads specific to SPS.

Consider an SPS process with multiple server threads. As the system load increases, context switching overhead and data cache pollution increases, and performance suffers. Similarly, as the number of threads in the system increases, there is further process scheduling overhead, as well as pollution of the instruction cache, in addition to the data cache pollution. Therefore, it is clear that no set of fixed values for the scheduler parameters will work well across the range of system loads and the number of threads under which a server is expected to operate. As we mentioned above, CFS attempts to adapt to an increase in the system load (i.e., an increase in the number of tasks in the queue) by increasing the value of parameter `sched_latency_ns`. By doing so, in effect, CFS relaxes the fairness policy so as to reduce the context switching overhead. As a result, the CPU is better utilized and system throughput increases.

6.3.2 Enhanced Server Mode

Based on the above observations, we have carried out a large number of experiments to measure the impact of the three scheduler parameters identified above on two performance metrics: (1) the service time, T_{sip} of a packet in the SIP layer, and (2) the kernel time, K_{rcv} that includes the waiting time at the socket queue.

Our findings are as follows:

1. *sched_latency_ns*: Setting this parameter to a fixed value 800,000, independent of the number of threads, gave the best results. Recall that this parameter controls the latency of CPU bound tasks, and is dynamically adjusted by the scheduler in response to variations in the system load (in our case, packet arrival rate). Therefore, using a low value for this parameter provides the scheduler with significant flexibility in adjusting this value upwards to control the context switching overhead following an increase in system load.

2. *sched_min_granularity_ns*: This parameter controls the amount of time that tasks may run without preemption. Therefore, it is desirable to set it to a value that corresponds to the amount of time needed to complete a task (in this case, we process a SIP packet), so as to minimize context switching overhead. To this end, we set this parameter to a value that roughly corresponds to the measured value of mean service time T_{sip} at the point where the system starts experiencing overload. The specific values we used for this parameter were 100,000, 150,000, 200,000, 250,000, and 400,000, for 2, 4, 6, 8 and 16 server threads, respectively.

3. *sched_wakeup_granularity_ns*: This parameter controls the wake-up latency of a task, i.e., the amount of time it must lapse before it can preempt the current task. Since we

set the amount of time that a task may run without preemption to a value that ensures that most tasks will complete before being preempted (see the discussion on the second parameter above), it follows that we should allow a new task to immediately preempt the current task. Indeed, setting the value of this parameter to zero achieved the best results across all thread configurations.

We will refer to the configuration of the CFS scheduler with these parameter values as the *enhanced* "server" mode.

6.4 Experimental Results

We now present the results of experiments we have conducted to compare the performance of the SPS under the baseline and enhanced "server" mode configuration for the scheduler. In the experiments, we vary both the number of SPS threads and the traffic load, expressed as number of calls per second (cps).

6.4.1 Average Service and Waiting Times

Figures 6.1 and 6.2 plot the value of K_{rcv} and T_{sip}, respectively, as a function of traffic load (in cps), for 2, 4, 6, 8 and 16 server threads and with the scheduler parameters tuned to baseline "server" mode. Figures 6.3 and 6.4 present results with the scheduler parameters tuned to enhanced "server" mode. Recall that K_{rcv} primarily represents the waiting time of a SIP packet, and T_{sip} is the service time within the SIP layer.

From the figures 6.1 and 6.2 we can observe that the K_{rcv} value increases rapidly beyond 1500cps. This increase is due to three factors:

- *Queuing delay.* As the call arrival rate increases, packets arriving at the SPS are buffered at the socket queue and experience increasing waiting times, K^{w}_{sockq} before being delivered to the SIP layer.

- *Interrupt Overhead.* The number of interrupts increases in direct proportion to the packet arrival rate. Interrupts introduce overhead in the form of the time needed to handle each interrupt, the context-switching operations, and the increase in processing time as a result of cache misses due to these interrupts. A cache miss causes the number of CPU cycles consumed by the network stack receiving process to increase, as the memory access cycles of main memory are several times that of L2 caches.

- *Multi-Threading Overhead.* Multi-threading adds another layer of overhead by requiring additional context-switching, resulting in instruction-cache misses. This is most clearly seen by results of the 16-server threads compared to the results of lower number of threads.

Regarding the T_{sip} curves shown in Figures 6.2 and 6.4, we observe that the T_{sip} value (i.e., service time) is similar for low loads and smaller number of server threads under both modes. An important observation is that the mean SIP service times for all message types increases almost linearly with the call rate. As the SPS is operating in stateful mode, it needs to perform table look-ups for each incoming message, so as to match an existing transaction or create a new one. As the call rate increases, the number of transactions in the system also increases, resulting in larger tables at the SIP layer and, hence, longer lookup and overall service times. The rapid increase in T_{sip} time for increased number of threads can be explained by the additional lock contention among threads for shared data, including lookup tables at the SIP layer.

However, as the number of server threads and the load increase, we observe a larger improvement for the enhanced "server" mode; this improvement is particularly evident in the 16-thread system for which the service time is notably lower under the enhanced "server' mode. This is because of fine tuning of the scheduler to allow each thread to run to completion by adjusting the *sched_min_granularity_ns* parameter to match the T_{sip} time.

In Figures 6.5 and 6.6, we see that between 1500 and 2500 cps, the "knee" of the K_{rcv} curves shifts to a higher load under the enhanced "server" mode relative to the baseline "server" mode; clearly, this behavior represents an improvement with respect to the load value where the server starts becoming overloaded.

6.4.2 Packet Drop Rate

To provide a better measure of the impact on user experience as a function of load we use the packet drop rate (PDR) as the metric to capture this.

Our goal is to identify the traffic load (in cps) at which the SIP control packet drop rate starts to exceed a certain threshold, implying that end users may experience call establishment problems due to packet drops. Service providers may use this traffic load as a trigger to add more capacity to the system *before* significant losses of SIP call setup packets start to occur.

Based on our survey of industry standards, the threshold for acceptable drop-rate in a VoIP environment is about 1% [37]. This value is considered in the industry as the threshold below which voice calls have quality comparable to toll quality. Note that this threshold is typically applied to voice packets to characterize the performance of the *data plane*. In this study, we consider three different threshold values, 1%, 2% and 5%, as reasonable values above which the SIP PDR will negatively impact the operation of the *control plane*; however, our methodology can be applied with any other appropriate threshold value.

Tables 6.1 and 6.2 present the measured values of T_{sip}, K_{rcv}, and SIP PDR under the baseline and enhanced "server" mode, respectively. Each column in these tables presents the average values of thirty experiments for the stated number of threads and load (cps). For each

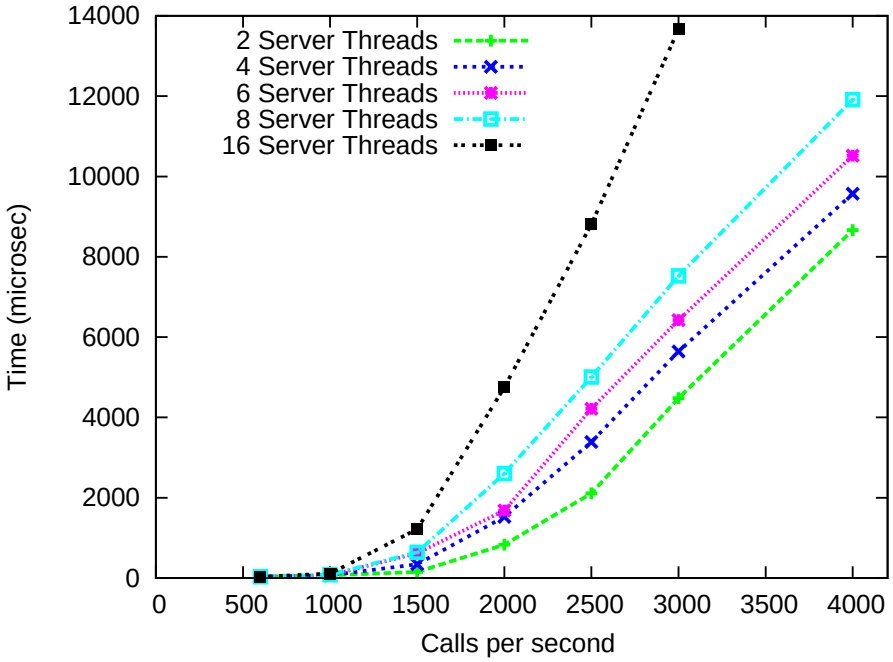

Figure 6.1: Mean K_{rcv} values vs. load, baseline "server" mode

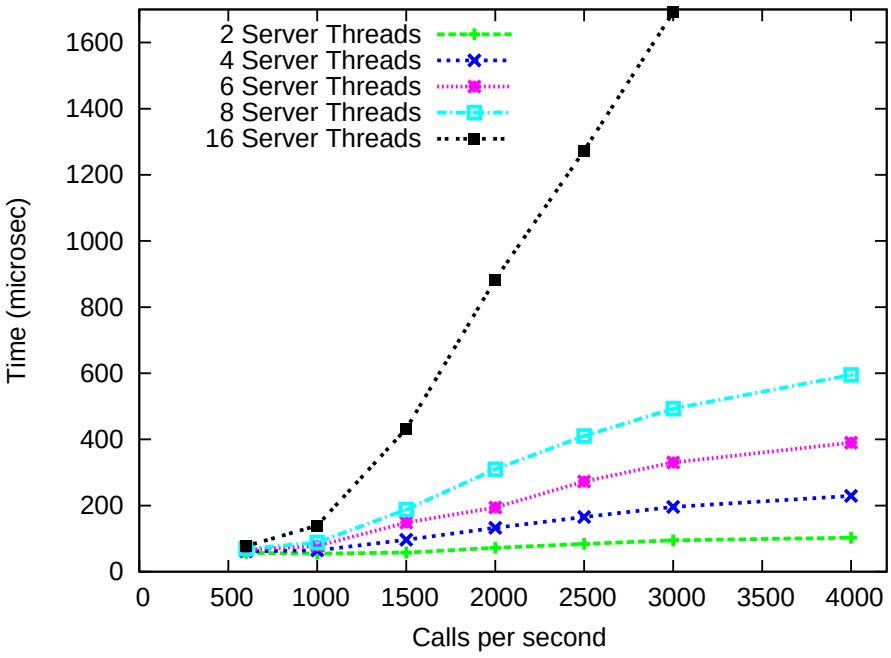

Figure 6.2: Mean T_{sip} values vs. load, baseline "server" mode

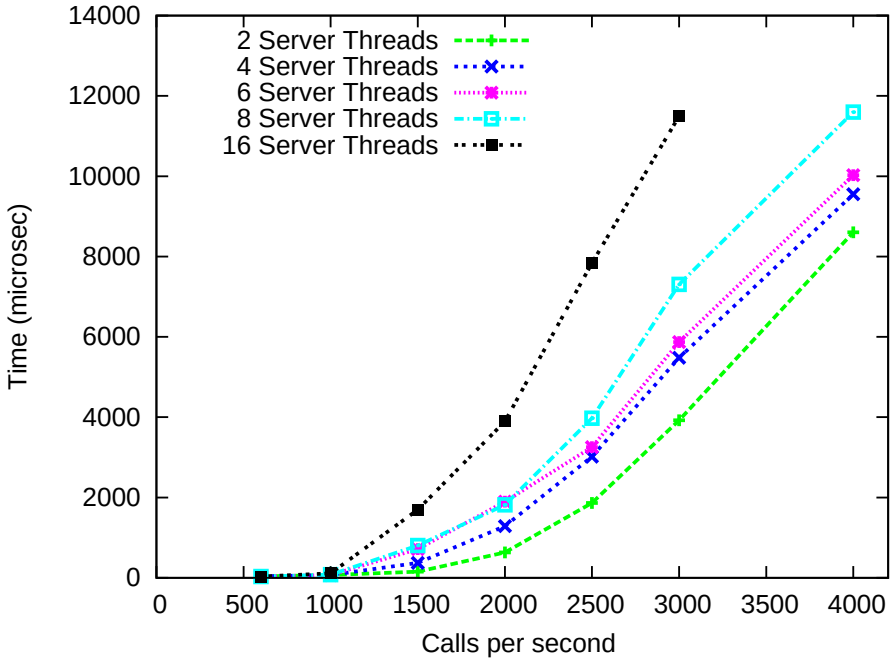

Figure 6.3: Mean K_{rcv} values vs. load, enhanced "server" mode

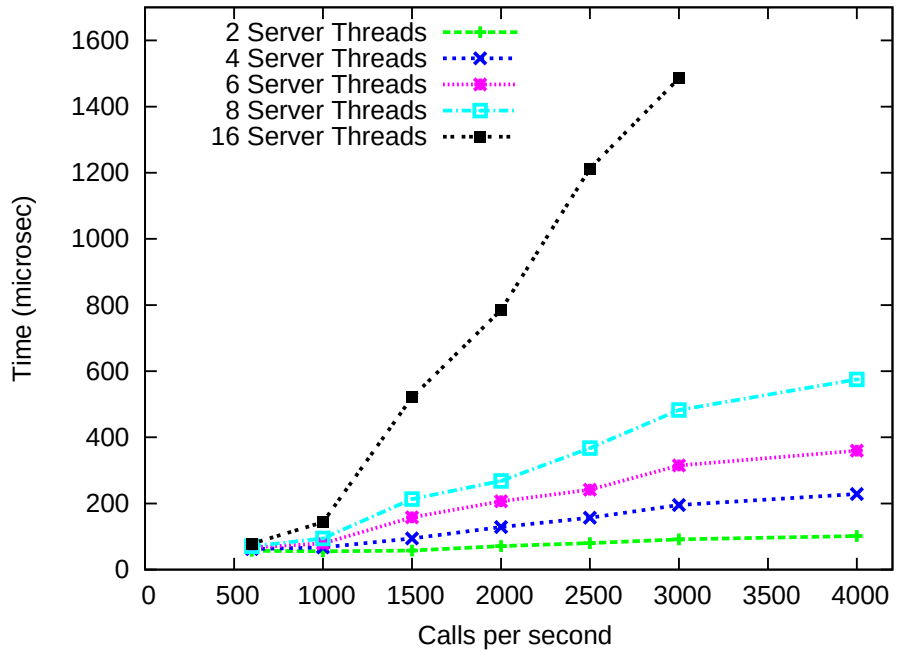

Figure 6.4: Mean T_{sip} values, enhanced "server" mode

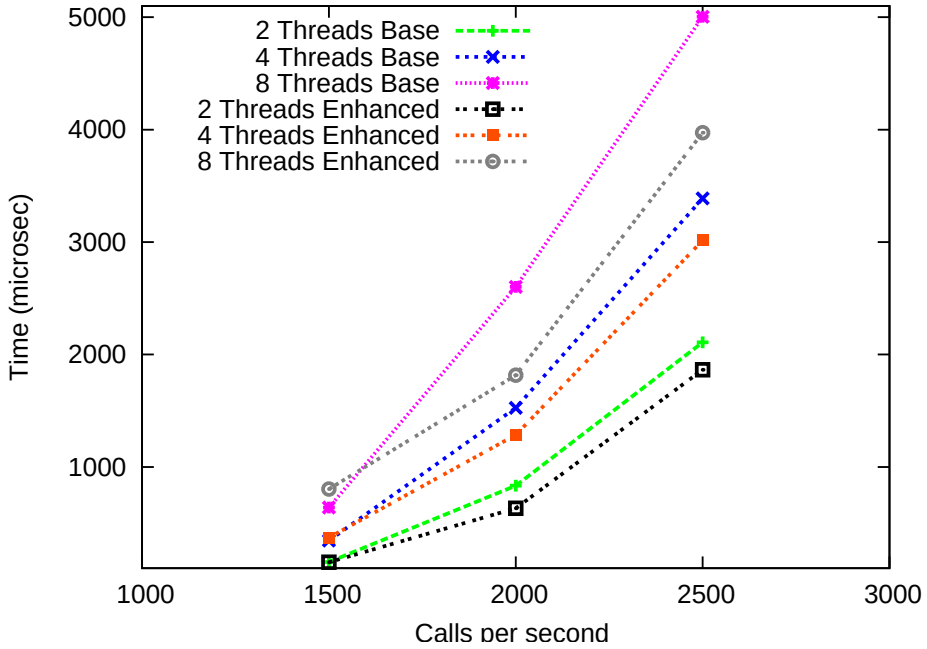

Figure 6.5: Mean K_{rcv} values, Comparison, Base vs enhanced Mode

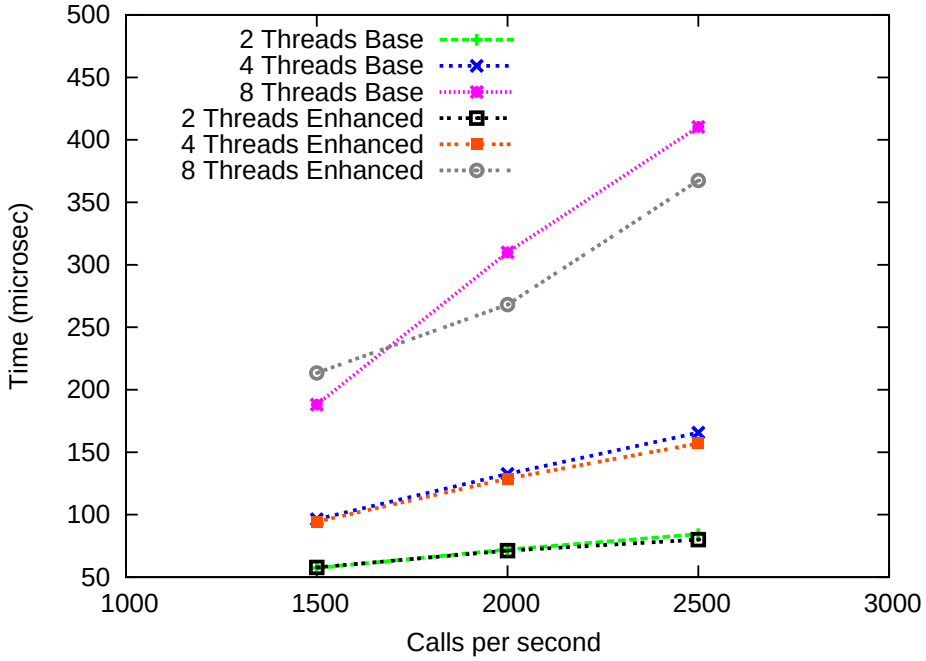

Figure 6.6: Mean T_{sip} values, Comparison, Base vs enhanced Mode

value of the number of threads, we present results for two load values: the load value at which the drop rate exceeds 1% for the first time, and the immediately lower value (in our experiments, we used an increment of 200 cps for load values). Two observations can be made from these tables. For the same load value, configuring the scheduler parameters to the enhanced "server" mode, always improves the SPS performance in terms of drop rate and waiting time (which is included in the kernel time, K_{rcv}). Furthermore, in some cases (i.e., for two and 16 threads), the load at which the PDR crosses the 1% threshold we imposed is higher for the enhanced mode as compared to the baseline mode.

Table 6.1: Measured SPS Performance, 1% PDR, Baseline Server Mode

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	2200cps	2400cps	2000cps	2200cps	1800cps	2000cps	1600cps	1800cps	1200cps	1400cps
Arrival rate (packets/sec)	13200	14400	12000	13200	10800	12000	9600	10800	7200	8400
T_{sip} (μs)	76.9	79.58	130.81	146.25	167.97	203.9	213.47	279.36	303.00	429.83
K_{rcv} (μs)	1309.16	1752.26	1473.32	2184.27	1042.66	1981.73	893.26	1974.84	809.76	1394.47
Call-Setup Drops	3039	6353	3106	6270	2351	5956	1895	6053	1788	3637
Call-Setup Messages	490479	476002	374480	399158	327973	373430	313721	334716	242858	264716
Total Messages	633487	598877	494568	513137	441154	490867	417454	447876	354911	372133
PDR	0.0062	**0.0133**	0.0083	**0.0157**	0.0072	**0.0159**	0.0052	**0.018**	0.0074	**0.0137**

Table 6.2: Measured SPS Performance, 1% PDR, Enhanced Server Mode

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	2400cps	2600cps	2000cps	2200cps	1800cps	2000cps	1600cps	1800cps	1400cps	1600cps
Arrival rate (packets/sec)	14400	15600	12000	13200	10800	12000	9600	10800	8400	9600
T_{sip} (μs)	77.21	85.11	129.28	138.76	178.11	186.85	189.37	242.29	392.25	540.13
K_{rcv} (μs)	1624.2	2322.44	1363.77	1805.35	1116.93	1418.35	594.44	1321.15	918.58	1864.63
Call-Setup Drops	4957	8840	3452	4688	2454	3878	1233	3717	2112	6238
Call-Setup Messages	501871	520479	382147	408507	330213	350419	314944	327789	259611	267954
Total Messages	636057	642238	504670	525251	445583	461109	433703	440549	367064	363775
PDR	0.0098	**0.0169**	0.0090	**0.0115**	0.0074	**0.0110**	0.0039	**0.0113**	0.0081	**0.0233**

To better illustrate the performance improvement under the enhanced mode, the PDR and K_{rcv} values for specific number of threads and load pairs are compared in Table 6.3. These

Table 6.3: PDR at 1% And Kernel Time (K_{rcv}) Comparison

Server Threads and CPS	PDR comparison			Kernel time, K_{rcv}, comparison (μs)		
	Baseline Mode	Enhanced Mode	%*Lower*	Baseline Mode	Enhanced Mode	%*Lower*
2 Threads, 2400cps	0.0133	0.0098	26.31%	1752.26	1624.2	7.3%
4 Threads, 2200cps	0.0157	0.0115	26.4 %	2184.27	1805.35	17.3 %
6 Threads, 2000cps	0.0159	0.0110	30.8%	1981.73	1418.35	28.4%
8 Threads, 1800cps	0.018	0.0113	37.22%	1974.84	1321.15	33.08%
16 Threads, 1400cps	0.0137	0.0081	40.87 %	1394.47	918.58	34.1%

pairs were selected as they correspond to the scenarios where the drop rate under the baseline mode exceeds 1%. As one can see, the performance improvement is between 26 and 40% for the drop rate and between 7 and 34% for K_{rcv}. In fact, the enhanced mode results in better performance in all the scenarios, and the degree of improvement increases with the number of threads for the 1% threshold.

Tables 6.4, 6.5 and 6.6 and Tables 6.7, 6.8 and 6.9 are similar to Tables 6.1 6.2 and 6.3 and show results for the 2% and 5% drop thresholds, respectively. As we can see, the performance improvement of the enhanced mode over the baseline mode remains significant and ranges from 3% to 18% for the drop-rate and 3% to 21% for the K_{rcv} values for the 2% threshold. For the 5% threshold the improvements range from from about 0.5% to 17% for the drop-rate and 4.5% to 16% for the K_{rcv} value.

The improvements due to enhanced "server" mode seen in data is a result of tuning the scheduler to lower the multi-threading overhead. This in-turn results in better utilization of the cache memory leading to lower waiting times. Further, lower waiting times in the socket queue, results to a lower packet drop-rate as the socket buffer doesn't get filled as quickly in the enhanced "server" mode.

Overall, the results we have presented indicate that, by adjusting the scheduler parameters to align with the packet processing time at various degrees of multi-threading, the enhanced "server" mode is capable of reducing the context switching overhead associated with larger thread numbers. We emphasize that *the benefits of the enhanced mode are available for free* (other than the one-time cost of carrying out the off-line experiments), in that they are achievable simply by setting the scheduler parameter to appropriate values and do not involve any resource trade-offs. Finally, we note that our methodology for optimizing the scheduler mode, although carried out in the context of SPS, is independent of the application layer, and hence can be applied to a spectrum of servers.

Table 6.4: Measured SPS Performance, 2% PDR, Baseline Server Mode

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	2600cps	2800cps	2200cps	2400cps	2000cps	2200cps	1800cps	2000cps	1600cps	1800cps
Arrival rate (packets/sec)	15600	16800	132000	14400	12000	13200	10800	12000	9600	10800
T_{sip} (μs)	85.43	89.71	148.46	160.36	203.9	230.92	279.36	287.14	429.83	574.27
K_{rcv} (μs)	2268.29	3047.18	2155.71	2947.3	1981.73	2658.6	1974.84	2156.87	1394.47	2320.68
Call-Setup Drops	8615	14479	6642	11247	5956	10334	6053	7655	3637	7239
Call-Setup Messages	546939	510773	417568	434796	373430	388947	334716	356045	264716	270279
Total Messages	674413	620083	535172	547052	490867	498053	447876	465978	372133	366678
PDR	0.0157	**0.028**	0.0159	**0.0258**	0.0159	**0.0265**	0.018	**0.0215**	0.0137	**0.0267**

Table 6.5: Measured SPS Performance, 2% PDR, Enhanced Server Mode

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	2600cps	2800cps	2200cps	2400cps	2000cps	2200cps	2000cps	2200cps	1400cps	1600cps
Arrival rate (packets/sec)	15600	16800	13200	14400	12000	13200	12000	13200	9600	10800
T_{sip} (μs)	85.11	87.94	138.76	155.67	186.85	217.39	277.22	308.29	392.25	537.54
K_{rcv} (μs)	2322.44	2930.03	1805.35	2769.65	1418.35	2216.36	1872.53	2624.5	918.58	1823.79
Call-Setup Drops	8840	13969	4688	10586	3878	8876	6369	11567	6238	6056
Call-Setup Messages	520479	561100	408507	425738	350419	400161	351093	384919	267954	278801
Total Messages	642238	687893	525251	538929	461109	513095	459549	491256	367064	379054
PDR	0.0169	**0.0249**	0.0115	**0.0248**	0.0110	**0.0222**	0.0182	**0.030**	0.0081	**0.0217**

Table 6.6: PDR at 2% And Kernel Time (K_{rcv}) Comparison

Server Threads and CPS	PDR comparison			Kernel time, K_{rcv}, comparison (μs)		
	Baseline Mode	Enhanced Mode	%*Lower*	Baseline Mode	Enhanced Mode	%*Lower*
2 Threads, 2800cps	0.028	0.0249	11.07%	3047.18	2930.03	3.84%
4 Threads, 2400cps	0.0258	0.0248	3.87%	2947.3	2769.65	6.04%
6 Threads, 2200cps	0.0265	0.0222	16.23%	2658.6	2216.36	16.6%
8 Threads, 2000cps	0.0215	0.0182	15.34%	2156.87	1872.53	13.17%
16 Threads, 1800cps	0.0267	0.0217	18.7%	2320.68	1823.79	21.4 %

Table 6.7: Measured SPS Performance, 5% PDR, Baseline Server Mode

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	3000cps	3200cps	2600cps	2800cps	2400cps	2600cps	2200cps	2400cps	1800cps	2000cps
Arrival rate (packets/sec)	18000	19200	15600	16800	14400	15600	13200	14400	10800	12000
T_{sip} (μs)	93.26	97.77	171.96	186.42	258.99	292.53	328.84	382.28	666.79	909.43
K_{rcv} (μs)	4240.97	5634.44	3927.19	4850.92	3828.7	4876.98	3139.58	4364.83	2908.09	4902.85
Call-Setup Drops	24494	37581	22200	25563	20223	30836	16263	26337	12251	23421
Call-Setup Messages	553364	562299	454056	495007	415974	437796	391284	391631	300485	304924
Total Messages	664214	669725	558083	601043	523968	537479	499464	487814	394028	386162
PDR	0.044	**0.0668**	0.0489	**0.0666**	0.0486	**0.0704**	0.0416	**0.067**	0.0407	**0.0768**

Table 6.8: Measured SPS Performance, 5% Drop-Rate, Enhanced Server Mode

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	3000cps	3200cps	2600cps	2800cps	2400cps	2600cps	2200cps	2400cps	1800cps	2000cps
Arrival rate (packets/sec)	18000	19200	15600	16800	14400	15600	13200	14400	10800	12000
T_{sip} (μs)	93.15	98.66	167.96	180.84	262.92	278.15	308.29	373.27	633.17	826.29
K_{rcv} (μs)	4147.52	5373.29	3648.16	4235.35	3735.95	4359.41	2624.5	3943.79	2535.36	4125.02
Call-Setup Drops	23242	39067	17715	25613	19745	27726	14598	22589	11530	20396
Call-Setup Messages	532064	586615	460049	479114	420969	443106	381532	398049	306690	321400
Total Messages	641179	695193	567401	581137	528531	546988	491256	497423	404023	410442
PDR	0.0437	**0.0665**	0.0385	**0.0534**	0.0453	**0.0626**	0.030	**0.0567**	0.0376	**0.0634**

Table 6.9: PDR at 5% And Kernel Time (K_{rcv}) Comparison

Server Threads and CPS	PDR comparison			Kernel time, K_{rcv}, comparison (μs)		
	Baseline Mode	Enhanced Mode	%*Lower*	Baseline Mode	Enhanced Mode	%*Lower*
2 Threads, 3200cps	0.0668	0.0665	0.45%	5634.44	5373.29	4.63%
4 Threads, 2800cps	0.0666	0.0534	19.82 %	4850.92	4235.35	12.69 %
6 Threads, 2600cps	0.0704	0.0626	11.08%	4876.98	4359.41	10.61%
8 Threads, 2400cps	0.067	0.0567	15.37%	4364.83	3943.79	9.65 %
16 Threads, 2000cps	0.0768	0.0634	17.45%	4902.85	4125.02	15.86%

6.5 Concluding Remarks

We have investigated the impact of the Linux scheduler settings on the performance of single-core, multi-threaded SIP proxy servers. The metrics we used were packet service time, waiting time, and packet drop rate (PDR) to capture the impact on user performance. Based on the results of a large set of experiments across a wide range of values for the number of server threads and traffic load, we have developed a methodology to configure the scheduler parameters that results in significant gains in SPS performance compared to industry-recommended "server" mode operation. Our methodology is not limited to SPS and may be applied to any application-layer server. Importantly, the gains in performance are the result of simply setting the scheduler parameters to appropriate values, without the need for adding server capacity or other capital expenditures. In the next chapter, we investigate the performance of multi-threaded SPS on multiple CPU cores.

Chapter 7

Performance Evaluation of Multi-Core, Multi-Threaded SIP Servers

Multi-core processors are ubiquitously used in all areas of computing, ranging from hand-held devices to laptops/desktops and to server farms in data centers. Most multi-core processors in use currently follow the symmetric multi-processor (SMP) paradigm, whereby all cores are identical, and are controlled by a single instance of the OS, and share a common main memory. With SMP, processes that do not need to share data with each other may be run on independent CPU cores to improve the performance of each process. The OS scheduler performs load balancing so as to ensure that some cores do not become overloaded if other cores have processing capacity available. Given the proliferation of multi-core systems, the characterization of the performance of multi-threaded applications on such systems is of practical importance.

Multi-threading support in various applications provides improved performance by using multiple CPU cores available in these processors. Process schedulers that are part of the core functionality of an operating system (OS), have been enhanced over the years to account for multiple cores in the processors and to support multi-threaded applications. One of the enhancements in the Linux scheduler is the support for load-balancing among various CPU cores available to the OS.

In this chapter, we investigate the impact of the Linux scheduler's load-balancing algorithm on the performance of multi-threaded OpenSIPS (an open source SIP proxy server, SPS) running on a multi-core processor system.

7.1 Related Work

The scalability of Linux on a multi-core system was analyzed in [16] by examining seven system applications. It was determined that all applications except one trigger a scalability bottleneck in the Linux kernel, and several modifications to the kernel were introduced to reduce this bottleneck. In [17], the scalability of a multi-core web server was examined, and it was observed that the capacity of the address bus in the eight-core system was the limiting factor in performance scaling. The performance of a SIP server on multi-core systems was studied in [45]. This study analyzed the performance of a realistic SIP workload on three different multi-core architectures and suggested improvements to certain operations, including garbage collection and lock contention, to improve performance. In this work, we look at the impact of migration cost of a process to a processor that is going idle. The subject of migration cost has been studied in the context of Virtual Machines (VM). In [46] the authors evaluated the effects of live migration on virtual machines on the performance of applications running inside Xen VMs. Their findings was, the migration overhead was acceptable for most cases. However, for systems where service availability and responsiveness are governed by strict Service Level Agreements (SLAs), the migration cost cannot be disregarded. In [47] the authors introduced an efficient algorithm that performed a live migration with minimal cost possible when certain conditions were met. The authors also developed a simpler algorithm for scenarios where optimal algorithm was not applicable and a fixed amount of bandwidth was available.

Several studies have specifically investigated the performance of SIP proxy servers (SPS). A load balancing algorithm for processing SIP messages in server clusters was developed in [48] and led to improvements in response time. The deployment of a multimedia service involving SIP sessions and MGCP connections was studied in [49], and strategies consisting of resource allocation and configuration in a virtualized environment were proposed to provide an optimal deployment.

In Chapters 4, 5 and 6, we presented results related to comprehensive packet-level measurements performed to obtain an accurate characterization of SPS performance in terms of service time, waiting time and packet drop rate. In these chapters, investigatons pertaining to single-core systems with a single and multiple server threads, respectively were discussed.

In this chapter, we investigate the impact of the load-balancing parameters of the Linux completely fair scheduler (CFS) policy on the performance of multi-threaded SPS on multi-core systems, as a function of the number of cores, the number of server threads and the call arrival rate. In our study we use the OpenSIPS SIP proxy server, which provides a transactional service. As seen from the above studies, there are several challenges in efficiently using multi-core systems for these type of services. Our work differs from these in that we specifically focus on the Linux CFS scheduler, and we provide guidelines for configuring the scheduler to optimize

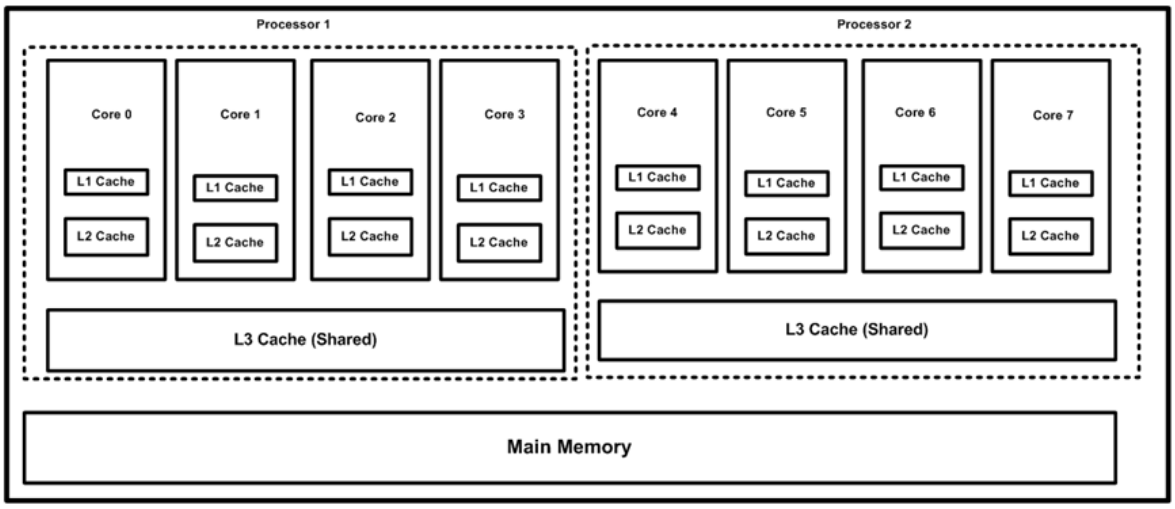

Figure 7.1: Dual quad-core processor hosting the OpenSIPS server for the experiments

load-balancing among the CPU cores. Specific contributions of our work include: (1) collection of extensive experimental data to characterize the packet-level performance of multi-threaded SPS running on multiple cores; (2) insight into the impact of CFS scheduler settings on SPS packet drops and and packet waiting times; (3) practical guidelines for tuning various CFS parameters to optimize SPS performance; and (4) a capacity planning model for estimating the scalability of SPS on a multi-core system.

7.2 Testbed and Experimental Setup

We use the same testbed and setup consisting of an OpenSIPS SPS and SIPp UAC and UAS, as in earlier chapters. For the performance evaluation of multi-threaded SPS on multiple cores, we upgraded the OpenSIPS SPS server to use Linux OS version 3.2.51 that is run on a system with two quad-core Intel Xeon CPU E5540 @ 2.53 GHz processors. Hence, a total of 8 CPU cores were available for our experiments, as shown in Figure 7.1.

7.3 Measurement Methodology and Experiments

In our experiments, we initiated calls between the UAC and UAS via the SPS. For each experiment, 100,000 calls were started. For each call, the messages exchanged between the UAC and UAS are the SIP call setup messages INVITE, 180 RINGING, 200OK and ACK, and the call

teardown messages, BYE and 200OK. For each message processed by the SPS, we measured the time components described above to determine the waiting and service times of the message through the SPS; we also kept a count of any messages dropped. Each experiment was characterized by three parameters:

1. *Number of CPU cores.* The experiments were conducted with the SPS server configured as a 2-, 4-, or 6-core system. In the experiments in which SPS was run on two cores, one additional core was used for running `syslogd`, and the remaining cores were disabled. For the 4-core and 6-core experiments, two cores were used for `syslogd`, and in the 4-core case, the two unused cores were disabled.

2. *Number of server threads.* Experiments were conducted with 2, 4, 6, 8, and 16 server threads for each of the multi-core systems above.

3. *Call arrival rate.* We varied the call rate starting at 200 calls per second (cps), with an increment of 200 cps, up to a maximum call rate beyond which the SPS is overloaded and the drop rate exceeded a certain threshold. As we show later, this maximum call rate depends on the experimental parameters and drop threshold.

Upon completion of an experiment for a specific call rate and number of cores and server threads, we process the logged data and calculate the sample mean values for K_{rcv} and T_{sip} as we have mentioned in earlier chapters. We obtain these mean values for each SIP message type as well as the overall mean across all six message types. We also estimate 95% confidence intervals around the overall mean. In addition, we use the `netstat` command to measure the packet drop rate.

7.4 Impact of Process Scheduler on Multi-core SPS Performance

In the 3.0-based Linux version, two configurable scheduler parameters are available specifically for tuning the multi-core operation and performance of the system.

- `sched_migration_cost`: A tunable parameter used to specify the "cost" of migrating a task from the current CPU to a CPU that is becoming idle. The scheduler load-balancing algorithm uses this parameter to allow a CPU going idle to pull tasks from another CPU.

- `sched_tunable_scaling`: This parameter allows the various scheduler parameters to be *scaled* as a function of the number of CPUs in the system. There are three options for this parameter: "no scaling" (i.e., the values of other scheduler parameters are used

without modification); "logarithmic scaling" (i.e., other parameter values are multiplied by $1 + \log(ncpus)$); and "linear scaling" (i.e., parameter values are multiplied by $ncpus$).

7.4.1 Baseline Multi-Core Server Mode

The default value scheduler parameters that we just described are, for `sched_migration_cost` it is 500,000 (i.e., 500 μs), and for `sched_tunable_scaling` it is set to "logarithmic scaling." However, the study in [50] found that setting the value of `sched_migration_cost` to 5,000,000 (5 ms) instead of the default value of 500,000 (500 μs) results in better performance. This is due to the fact that specifying a very high migration cost forces the scheduler to keep a task in the current CPU, thereby increasing cache utilization.

In chapter 6 for a single-core system, we used the following values for three other scheduler parameters to move the scheduler policy to "server" mode, as recommended in [44]: `sched_latency_ns` = 1,000,000 (1 ms), `sched_min_granularity_n` = 100,000 (100 μs), and `sched_wakeup_granularity_ns` = 25,000 (25 μs). In this study, we denote as the *baseline* "multi-core server" mode the scheduler configuration in which, in addition to the above three parameter values, the values of the multi-core specific parameters are set to: `sched_migration_cost` = 5,000,000 (5 ms), and `sched_tunable_scaling` is "logarithmic scaling."

7.4.2 Enhanced Multi-Core Server Mode

We note that the parameter value recommendations in [44] are for a generic server configuration and do not take into account workloads or operation characteristics specific to SPS. In this section, we develop a methodology for configuring the scheduler parameters specifically for SPS servers.

According to [43], the two primary factors that operators consider when designing their network is the service availability to end-users and the cost of operating the network. Furthermore, from an economic standpoint, network operators aim to achieve high server utilization in order to maximize the return on their capital investment. These observations motivate us to develop guidelines for tuning the scheduler parameters so as to balance these two conflicting objectives: availability of SIP service and SPS server utilization. To illustrate the tradeoffs involved, consider the scheduler parameter `sched_migration_cost`. Increasing the value of this parameter considerably higher than the default value of 500,000 (500 μs), e.g., as suggested in [50], will allow SPS threads to remain resident in a single CPU. However, doing so may cause the CPU to become overloaded. The increase in call arrival rates, will lead to excessive loss of SIP call setup packets, negatively affecting service availability. Therefore, we consider the packet drop rate (PDR) as the key performance metric of interest.

Our findings regarding the impact of each scheduler parameter are summarized below.

- *sched_latency_ns*: Setting this parameter to a fixed value of 800,000, independent of the number of threads, achieved the best results for the single-core SPS system as we discussed in Chapter 6. In experiments with multi-core systems, scaling this value linearly with the number of CPU-cores resulted in the best performance in terms of PDR. Therefore, we used the values $800,000 \times ncpus$, where $ncpus = 2, 4, 6$, for the 2-, 4-, and 6-core systems, respectively.

- *sched_min_granularity_ns*: As in chapter 6, we set this parameter to a quantity that corresponds to the measured value of mean service time T_{sip} at the point where the system starts experiencing overload. Note that the minimum value allowed for this parameter is 100,000. The specific values we used for this parameter were as follows: *2-core system:* 100,000 for 2, 4, and 6 threads, 150,000 for 8 threads, and 200,000 for 16 server threads; *4- and 6-core systems:* 100,000 for 2, 4, 6 and 8 threads and 200,000 for 16 server threads.

- *sched_wakeup_granularity_ns*: Setting the value of this parameter to zero achieved the best results across all threads and CPU core configurations, as explained in the previous chapter.

- *sched_migration_cost*: For 2- and 4-core systems we set this parameter to zero, while for 6-core systems a value of 500,000 provided the best results. In the next section we present experimental results that justify this choice of values.

- *sched_tunable_scaling*: Recall that this parameter may be used to scale the values of other scheduler parameters either logarithmically or linearly with the number of cores. Since our findings indicate that there is no common scaling factor for the other four parameters above, we have set the value of this parameter to "no scaling."

We will refer to the configuration of the CFS scheduler with these parameter values as the *enhanced* "multi-core server" mode.

7.5 Experimental Results

7.5.1 Impact of `sched_migration_cost`

Figures 7.2 and 7.3 show the impact of varying the value of *sched_migration_cost* on the packet waiting time (K_{rcv} and PDR, respectively, as a function of the number of CPU cores. In these experiments we used the following values for the call arrival rate and number of threads: 2-core system – 4200 cps, 4 threads; 4-core system – 5400 cps, 6 threads; 6-core system – 6200 cps, 8 threads. Results for other call arrival rate and thread values are very similar and are omitted. Also, the scheduler was configured in the enhanced "multi-core" mode described

in the previous section, in that all parameters were set to values dictated by that mode, except the *sched_migration_cost* parameter whose value was varied as shown in the figures.

As we can see, the results confirm our choice of values for this parameter, as we discussed in the previous section. Consider the CPU architecture shown in Figure 7.1, and recall that the value of *sched_migration_cost* is used by the load balancing algorithm to determine whether to move tasks to an idle CPU, with a low value allowing an idle CPU to pull tasks more easily. Also note that the main operation of the SPS involves processing a packet and forwarding it to the UAC or UAS. For 2- and 4-core systems, a value of zero provided the best results: in these systems, all cores are part of the same processor, and task migration incurs minimal cache penalty; hence setting this cost to zero allows for better load balancing. For the 6-core system, the most effective value is 500,000. In such systems, the cores are distributed across two processors, thus the penalty of migrating the task in terms of cache miss is, on average, higher. Therefore, using a low but non-zero value for the migration cost in an attempt to keep tasks on the same CPU provided an appropriate balance between the cost of cache misses and load balancing across CPUs. In all cases, setting the migration cost to the high value recommended in [50] results in low or no thread migration; hence, increased load in a single core leads to high packet drop rates even if other CPUs have available capacity. These results indicate that scheduler configuration is highly application-dependent and settings that work well for some applications may result in poor performance for others.

7.5.2 SPS Performance

Tables 7.1 and 7.2 present the measured values of T_{sip} (service time of SIP messages), K_{rcv} (waiting time), and PDR under the baseline and enhanced multi-core server mode, respectively, for the 2-core system and various server thread configurations. Tables 7.4 and 7.5 present the same data for the 4-core system, while Tables 7.7 and 7.8 show data for the 6-core system. Each column in these tables present the average of thirty experiments for the stated number of threads and load (in calls per second, cps). For each value of the number of threads, we present results for two load values: the value at which the PDR exceeds 1% for the first time, and the immediately lower value (recall that in our experiments, we used an increment of 200 cps).

We make two observations from these tables. For the same load value, configuring the scheduler parameters to the enhanced server mode always improves the SPS performance over the baseline mode, in terms of PDR and waiting time. Furthermore, in all cases the load at which the PDR rate crosses the 1% threshold we imposed is higher for the enhanced mode compared to the baseline mode in terms of CPS. We also see that, as the number of cores increases, this threshold load is higher for 4-core compared to 2-cores, and increases further for 6-core; this result is expected given the higher capacity available with additional cores.

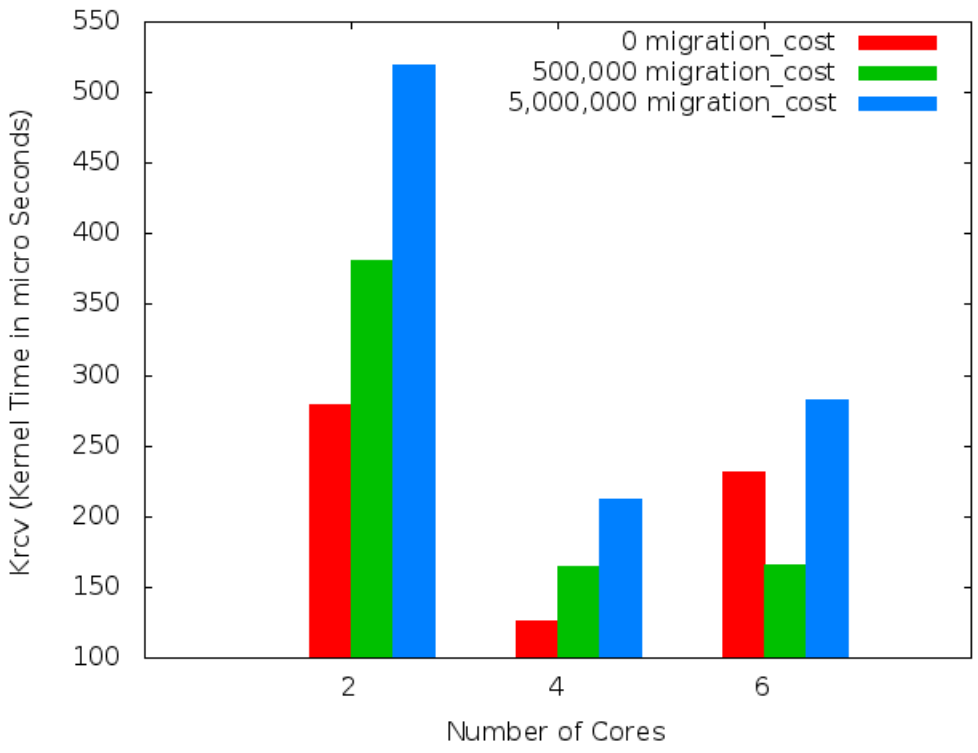

Figure 7.2: Impact of *sched_migration_cost* on K_{rcv} (waiting time)

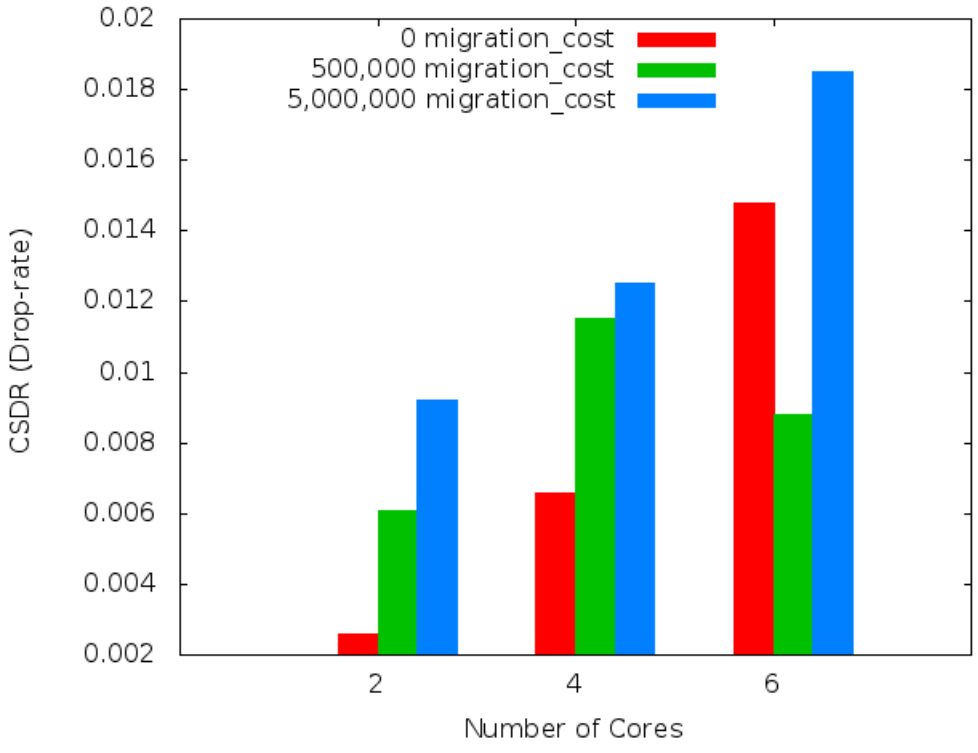

Figure 7.3: Impact of *sched_migration_cost* on PDR

To better illustrate the performance improvement under the enhanced mode, Tables 7.3, 7.6, and 7.9 compare the PDR and K_{rcv} values for 2-, 4-, and 6-core systems, respectively. The results shown are for a specific load for each thread value; this load value corresponds to the scenario where the PDR rate under the baseline mode exceeds the 1% threshold for the given number of threads.

As we can see, the performance improvement is between 4 and 53% for the PDR rate, and ranges from 1 to 72% for K_{rcv} for the 2-core system. For the 4-core (respectively, 6-core) system, the improvement range is from 14 to 42% (respectively, 33 to 58%) for the PDR and between 2 and 42% (respectively, 12-45%) for K_{rcv}. In fact, the multi-core enhanced server mode shows improvements across all core and thread configurations.

Table 7.1: Measured SPS Performance, Baseline Multi-core Server Mode, SPS on 2-Core

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	3000cps	3200cps	3600cps	3800cps	3400cps	3600cps	3000cps	3200cps	2400cps	2600cps
Arrival rate (packets/sec)	18000	19200	21600	22800	20400	21600	18000	19200	14400	15600
T_{sip} (μs)	58.05	60.31	74.92	78.22	96.78	104.48	114.64	124.6	204.04	218.29
K_{rcv} (μs)	474.97	684.87	292.39	373.72	319.04	436.66	308.36	412.81	352.09	395.01
Call-setup Drops	2762	4202	2206	3792	2883	3614	2850	4452	2804	3364
Call-setup Messages	327213	322174	322212	318275	323412	320313	328611	323481	339835	335899
Total Messages	390029	379763	373528	366842	379274	372628	394732	384077	427449	417470
PDR	0.0084	**0.0130**	0.0053	**0.0119**	0.0089	**0.0113**	0.0087	**0.0137**	0.0083	**0.0100**

Table 7.2: Measured SPS Performance, Enhanced Multi-core Server Mode, SPS on 2-Core

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	3800cps	4000cps	4400cps	4600cps	3800cps	4000cps	3200cps	3400cps	2600cps	2800cps
Arrival rate (packets/sec)	22800	24000	26400	27600	22800	24000	19200	20400	15600	16800
T_{sip} (μs)	54.41	53.55	83.62	85.34	101.28	116.15	125.78	139.26	218.07	262.32
K_{rcv} (μs)	241.28	283.81	469.17	591.99	348.17	569.58	358.59	492.15	398.73	622.95
RCV Errors	3087	8017	3119	4245	3348	5366	3798	5586	3986	6830
Call-setup Drops	2682	7002	2736	3734	2879	4646	3205	4781	3228	5624
Call-setup Messages	320141	635594	317922	315676	321645	316459	324962	318851	334935	326550
Total Messages	368496	727720	362463	358886	373972	365455	385030	372492	413491	396564
PDR	0.0084	**0.011**	0.0086	**0.0118**	0.0089	**0.0147**	0.0098	**0.0149**	0.0096	**0.0172**

Table 7.3: Drop rate And Kernel Time (K_{rcv}) Comparison, SPS on 2-Core

Server Threads and CPS	Drop Rate comparison			Kernel time, K_{rcv}, comparison (μs)		
	Baseline Mode	Enhanced Mode	%*Lower*	Baseline Mode	Enhanced Mode	%*Lower*
2 Threads, 3200cps	0.0130	0.0060	53.84%	684.87	186.83	72.72%
4 Threads, 3800cps	0.0119	0.0082	31.09%	373.72	253.45	32.18 %
6 Threads, 3600cps	0.0113	0.0058	48.67%	436.66	264.13	39.51 %
8 Threads, 3200cps	0.0137	0.0098	28.47%	412.81	358.59	13.13%
16 Threads, 2600cps	0.0100	0.0096	4%	395.01	388.73	1.6%

Table 7.4: Measured SPS Performance, Baseline Multi-core Server Mode, SPS on 4-Core

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	4000cps	4200cps	4800cps	5000cps	5000cps	5200cps	4200cps	4400cps	3800cps	4000cps
Arrival rate (packets/sec)	24000	25200	28800	30000	30000	31200	25200	26400	22800	24000
T_{sip} (μs)	63.25	61.06	86.88	79.61	94.18	94.86	120.77	121.99	177.37	188.18
K_{rcv} (μs)	427.16	518.30	220.13	212.46	165.59	192.17	212.6	221.57	209.9	259.36
RCV Errors	3318	4877	3387	3962	3276	3617	3145	3748	3101	3769
Call-setup Drops	2891	4267	3000	3542	2910	3214	2750	3305	2684	3279
Call-setup Messages	319102	315952	315704	313908	315739	315209	318907	316519	320946	318663
Total Messages	366171	361079	356395	351104	355482	354681	364633	358892	370774	366263
PDR	0.0091	**0.0135**	0.0095	**0.0112**	0.0092	**0.0102**	0.0086	**0.0104**	0.0084	**0.0103**

7.6 Capacity Planning Model

Several studies have investigated aspects of capacity planning in various contexts. A new methodology was presented in [51] for the efficient analytic solution to account for the burstiness of workload so as to develop a model for capacity planning. In [52], time-series analysis techniques were used to automatically adjust the number of users for an on-demand streaming service and the server bandwidth demand; the proposed mechanism was evaluated on a dataset collected from a video-on-demand service provider. The study in [53] compared the static vs. dynamic resource allocation of virtual machines (VMs) in corporate clouds to evaluate the energy efficiency of each mechanism. The authors concluded that dynamic resource allocation and associated migration overhead may cost more than static VM allocation and does not increase energy efficiency.

As we described earlier, the PDR metric captures the impact on user experience as a result of the SPS becoming overloaded. We have obtained experimental measurements of PDR as a function of the number of threads and the number of CPU cores in the system. We have also adopted a drop rate of 1% as the point of impact. Based on the data we have collected from our experiments, we now develop a capacity planning model that cloud providers and service providers may use to obtain a first-order approximation of the load (in calls per second) that

Table 7.5: Measured SPS Performance, Enhanced Multi-core Server Mode, SPS on 4-Core

Model Parameters	Number of Server Threads									
	2 Threads		**4 Threads**		**6 Threads**		**8 Threads**		**16 Threads**	
	4200cps	4400cps	5000cps	5200cps	5400cps	5600cps	4600cps	4800cps	4600cps	4800cps
Arrival rate (packets/sec)	25200	26400	30000	31200	32400	33600	27600	28800	27600	28800
T_{sip} (μs)	61.03	60.49	80.74	79.59	89.59	96.5	114.33	109.6	184.16	203.75
K_{rcv} (μs)	386.12	402.52	181.61	221.73	125.61	272.06	193.29	206.95	255.31	395.69
RCV Errors	3339	5001	3411	4642	2365	5580	3156	4215	2328	4916
Call-setup Drops	2918	4400	3038	4124	2098	4987	2789	3745	2044	4342
Call-setup Messages	318842	315232	315041	313723	316880	312176	316979	314595	318870	313507
Total Messages	364836	358278	353704	353137	357111	349292	358660	354077	363074	354953
PDR	0.0092	**0.0139**	0.0096	**0.013**	0.0066	**0.0159**	0.0088	**0.0119**	0.0064	**0.0138**

Table 7.6: Drop rate And Kernel Time (K_{rcv}) Comparison, SPS on 4-Core

Server Threads and CPS	Drop Rate comparison			Kernel time, K_{rcv}, comparison (μs)		
	Baseline Mode	Enhanced Mode	$\%Lower$	Baseline Mode	Enhanced Mode	$\%Lower$
2 Threads, 4200cps	0.0135	0.0092	31.85%	518.3	386.12	25.5%
4 Threads, 5000cps	0.0112	0.0096	14.28%	212.46	181.61	14.5%
6 Threads, 5200cps	0.0102	0.0059	42.16%	192.17	110.16	42.67%
8 Threads, 4400cps	0.0104	0.0057	45.19%	221.57	132.69	40.11%
16 Threads, 4000cps	0.0103	0.0063	38.83%	259.36	252.64	2.6%

can be supported by their SPS servers without exceeding the 1% drop rate threshold.

Referring to Figure 7.1, let p and c be the number of independent processors and CPU cores, respectively, in the system that are available to run SPS threads. Also, let $T_{p,c}$ be the number of threads that provides the best performance (i.e., the highest call arrival rate for which the PDR under the enhanced multi-core server mode does not exceed 1%) for the given number of CPU cores and processors. In our experiments, we have found that in systems with only $p = 1$ processor with c cores, setting $T_{1,c} \approx c + 2$ provides a good balance between the conflicting factors of cache locality, multi-threading overhead and process migration cost, and offers the best results. However, for $p \geq 2$ of processors whereby threads execute on cores that are in different processors, keeping the number of threads close to the total number of cores provides the best performance as it avoids the additional migration cost of moving threads from one processor to another. In this case, we let $T_{p,c} \approx c, p \geq 2$. Indeed, referring to Tables 7.2, 7.5, and 7.8, we see that the above formula accurately predicts the number of threads that gives the best results for the 2-, 4-, and 6-core systems as 4, 6, and 6 threads, respectively (note that $p = 1$ for 2- and 4-core systems, while $p = 2$ for the 6-core system).

Now let $C_{1,1}$ be the baseline capacity of a single-core, single-thread system; in earlier chapters we established $C_{1,1} = 1000$ cps for our system. Then, the capacity of the multi-core, multi-threaded system can be estimated as:

Table 7.7: Measured SPS Performance, Baseline Multi-core Server Mode, SPS on 6-Core

Model Parameters	Number of Server Threads							
	4 Threads		**6 Threads**		**8 Threads**		**16 Threads**	
	5800cps	6000cps	6000cps	6200cps	6000cps	6200cps	3800cps	4000cps
Arrival rate (packets/sec)	34800	36000	36000	37200	36000	37200	22800	24000
T_{sip} (μs)	71.78	72.12	83.03	93.07	100.29	110.13	181.02	193.93
K_{rcv} (μs)	161.10	207.9	124.3	277.4	165.35	214.04	187.3	278.03
RCV Errors	2832	4366	2489	6235	3295	5204	2640	4673
Call-setup Drops	2505	3853	2200	5408	2852	4650	2283	4058
Call-setup Messages	317172	315447	317769	316568	320129	312013	321594	317896
Total Messages	358497	357417	359417	364948	369834	349180	371789	366019
PDR	0.0079	**0.012**	0.0069	**0.017**	0.0089	**0.0149**	0.0071	**0.0127**

$$C_{p,c} = T_{p,c} \times C_{1,1} \tag{7.1}$$

Expression (7.1) provides estimates of the CPS capacity of 4000 cps for the 2-core system and 6000 for the 4- and 6-core systems, a good first-order approximation of the experimental results.

7.7 Concluding Remarks

We investigated the performance of multi-core, multi-threaded SPS and the impact of Linux CFS scheduler tuning on SPS in terms of packet drop rates and waiting times. We conducted extensive experiments over a wide range of offered load and determined the CFS scheduler settings that result in significant gains in performance. This addressed one of the most crucial needs in today's data center, which is to extract performance gains from the existing computing infrastructures without additional capital expenses. Our methodology is expected to lead to a better price-performance metrics for the SPS so that costly infrastructure expansion may be avoided. We also developed a capacity planning model that provides a good first-order approximation of the total capacity of the SPS system in terms of the call arrival rate that may be supported without affecting user experience in terms of dropped call.

Table 7.8: Measured SPS Performance, Enhanced Multi-core Server Mode, SPS on 6-Core

Model Parameters	Number of Server Threads							
	4 Threads		**6 Threads**		**8 Threads**		**16 Threads**	
	6000cps	6200cps	6600cps	6800cps	6400cps	6600cps	4200cps	4400cps
Arrival rate (packets/sec)	36000	37200	39600	40800	38400	39600	25200	26400
T_{sip} (μs)	72.25	73.59	91.09	92.26	111.71	112.86	195.16	211.05
K_{rcv} (μs)	182.19	221.16	180.26	215.09	224.81	278.99	245.88	386.42
RCV Errors	2946	4586	2777	4435	3417	5509	3406	5656
Call-setup Drops	2573	3994	2344	3759	2897	4698	2978	4949
Call-setup Messages	319534	317889	325325	322247	321044	320477	318556	314689
Total Messages	365891	364950	385418	380141	378616	375724	364293	359605
PDR	0.0080	**0.013**	0.0072	**0.0117**	0.0090	**0.0146**	0.00935	**0.0157**

Table 7.9: Drop rate And Kernel Time (K_{rcv}) Comparison, SPS on 6-Core

Server Threads and CPS	Drop Rate comparison			Kernel time, K_{rcv}, comparison (μs)		
	Baseline Mode	Enhanced Mode	%*Lower*	Baseline Mode	Enhanced Mode	%*Lower*
4 Threads, 6000cps	0.012	0.008	33.33 %	207.9	182.19	12.36%
6 Threads, 6200cps	0.017	0.0071	58.23%	277.4	151.4	45.4%
8 Threads, 6200cps	0.0149	0.0086	42.28%	214.04	177.89	16.88 %
16 Threads, 4000cps	0.0127	0.0081	36.22%	278.03	193.68	30.33 %

Chapter 8

Summary and Future Work

The primary considerations for service providers and cloud operators when designing their networks are:(a) service availability to end-users, and (b) the cost of operating the network. Furthermore, from an economic standpoint, network operators aim to achieve high server utilization in order to maximize the return on their capital investment. For service providers to deal effectively with the demand growth, they must develop a good understanding of current usage patterns, be able to forecast and plan upgrade needs, and be able to configure a robust service capability for new users. Ultimately, all these considerations require accurate estimates of the performance capability of the SPS that forms the core of the SIP network and a deeper understanding of various factors that can impact the performance of the SPS.

Hence, we had the following main objectives for our research:

1. Develop tools and techniques that can be easily adapted to carry out similar experimental studies for other SPS configurations as well as different protocol suites.

2. Conduct a comprehensive set of experiments to understand how individual SIP packets are processed and measure their processing and waiting times (within the kernel and the SIP protocol)

3. Develop a guideline for extracting performance gains from existing computing infrastructures without additional capital expenditures.

4. Develop a parametrized model that can be used to estimate the performance and the capacity of the SPS over a range of offered loads, and a range of SPS hardware and software configurations.

In this work we investigated the performance of OpenSIPS [3], an open source SIP proxy server, and made several contributions, as follows.

- **Measurement Methodology and tools:**
  - We have modified the Linux kernel and the OpenSIPS source code to obtain packet-level measurements for each SIP message, in order to obtain the service and waiting times within the kernel and the SIP layer. In particular, the kernel modifications can be used for collecting these measurements for *any* protocol, while the OpenSIPS modifications may be easily adapted to other application servers.
  - We also enhanced SIP*p* [4], a SIP traffic generator tool, to generate calls with inter-arrival times that follow any user-specified distribution. The modified versions of the kernel, OpenSIPS, and SIP*p* are made available in the Appendix and also as media files as part of this thesis.
  - We have collected a large set of experimental data to characterize the performance of the SPS under various call arrival rates and inter-arrival time distributions and various SPS configurations.
- **Single-core, single threaded SPS and Queuing Model:** For a single SPS server thread on a single-core CPU hardware we conducted large set of experiments and modeled the SIP proxy server as an *M/G/1* queue. A key component of the model is a parameter that captures the Interrupt overhead, i.e., the impact of Interrupts and resulting cache-misses on socket queue service times.
- **Single-core, multi-threaded SPS and Drop-Probability Model:** We studied the performance of multiple SPS server threads on a single-core CPU hardware. We measure the call rate where the SPS server starts experiencing losses greater than 1% and developed a prediction model for the drop probability as a function of call rate and the number of server threads. We also introduced a new parameter to capture the overhead of multiple server threads, in addition to the interrupt overhead.
- **Impact of Process Scheduler:** We investigated the impact of the Linux scheduler settings on the performance of single-core, multi-threaded SIP proxy servers, in terms of packet service time, waiting time, and packet drop rate (PDR) to capture its effect on user performance. We identified the key scheduler parameters of the Linux scheduler and provide concrete guidelines for tuning these parameters that we identify as 'enhanced server mode' to achieve significant performance improvement.
- **Multi-core, multi-threaded SPS and Capacity Model:** We expanded our study to investigate the impact of the Linux scheduler's load-balancing algorithm on the performance of multi-threaded SPS running on a multi-core processor system. We conducted extensive experiments and collected data to characterize the packet-level performance of

multi-threaded SPS running on multiple cores. Further, we developed practical guidelines for tuning various CFS parameters to optimize SPS performance on a multi-core system; and developed a capacity planning model for estimating the scalability of SPS on a multi-core system.

8.1 Future Work

Performance evaluation of servers is a broad area of research and in our research we developed a specific methodology and made several key contributions. Our research ranged from specific application layer protocol, to tuning underlying OS software to configuring the hardware to a specific settings. This research can serve as the foundation for this topic. Some of the area's where this research can be further expanded are the following:

- **Transport Layer:** In our study we configured the SPS to use UDP as the transport layer. In future research, using our tools and measurement methodology the impact of using TCP as transport protocol for the SPS can studied. Security is one of the main area of concern for service providers and as such impact of using Transport Layer Security (TLS) over TCP on SPS can be further studied. TLS is used for encapsulating SIP packets for providing a secure control path session for call setup and teardown.

- **Application Protocols:** As we mentioned earlier, our measurement and modeling methodology is general, and can be applied to characterize the performance of a wide range of network application protocols. One area of future work is to apply similar methodology to measure and model the performance of other protocols such as HTTP, SMTP and XMPP. This will provide data on the variations across different network application protocols.

- **Hyper-Threading:** Intel's Hyper-Threading technology is enabled on most server class Intel processors. Hyper-threading allows one physical CPU core to be divided into two logical cores, allowing number of cores seen by the operating system to be doubled. However, each logical core's cache size is reduced by half. This trade-off between cache size and increased number of processors presents an interesting area of research on the impact of overall server performance. This trade-off can be studied using our methodology to quantify the impact in terms of waiting time and total capacity of the system. The Hyper-threading can be controlled via a setting in the BIOS. In our study Hyper-threading was disabled.

- **CPU tuning:** The performance capacity of a given processor itself can have significant impact on the overall performance of the server. In all our experiments we used the default

clock frequencies of the CPU. Another area that can be investigated is the impact of CPU clock frequency in terms of waiting time and drop rate. The CPU clock frequency can be set at artificially lower rate, the default rate and then over-clocked. The waiting time and drop-rate for these variations can be compared to see the performance gain obtained in terms of CPU clock frequency.

In our research we developed new measurement methodology and tools that enabled us to measure service time with high accuracy and provide performance metrics in terms of waiting time and drop-rate. We applied this technique to a SPS for various settings and our research enhanced the knowledge in several areas related to server performance.

REFERENCES

[1] J. Rosenberg *et al.* SIP: Session Initiation Protocol. RFC 3261, June 2002.

[2] J. Rosenberg. Requirements for management of overload in the session initiation protocol. RFC 5390, December 2008.

[3] OpenSIPS: Open source implementation of a SIP server. http://www.opensips.org/.

[4] SIP*p*. http://sipp.sourceforge.net/.

[5] A. Johnston. *SIP: Understanding Session Initiation Protocol*. Artech House Publishers, 2nd Edition, 2004.

[6] D. Malas *et al.* Basic telephony SIP end-to-end performance metrics. RFC 6076, January 2011.

[7] V.K. Gurbani, L. Jagadeesan, and V.B. Mendirittam. Characterizing the Session Initiation Protocol (SIP) network performance and reliability. In *Proceedings of ISAS 2005*, pages 196–211, April 2005.

[8] S.V. Subramanian and R. Dutta. Comparative study of M/M/1 and M/D/1 models of a SIP proxy server. In *Proceedings of the Australasian Telecommunications Networking and Application Conference (ATNAC)*, pages 397–402, December 2008.

[9] S.V. Subramanian and R. Dutta. Measurements and analysis of M/M/1 and M/M/c queueing models of the SIP proxy server. In *Proceedings of the 18th International Conference on Computer Communications and Networks (ICCCN 2009)*, pages 1–7, August 2009.

[10] S.V. Subramanian and R. Dutta. Performance and scalability of M/M/c based queueing model of the SIP proxy server - a practical approach. In *Proceedings of the Australasian Telecommunications Networking and Application Conference (ATNAC)*, pages 1–6, December 2009.

[11] SIP express router. http://www.iptel.org/ser/.

[12] E. Nahum, J. Tracey, and C.P. Wright. Evaluating sip server performance. Technical Report Research Report RC24183, IBM T. J. Watson Research Center, February 2007.

[13] K.K. Ram, I.C. Fedeli, A.L. Cox, and S. Rixner. Explaining the impact of network transport protocols on SIP proxy performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 75–84, April 2008.

[14] H. Jiang, A. Iyengar, E. Nahum, W. Segmuller, A. Tantawi, and C.P. Wright. Load balancing for SIP server clusters. In *Proceedings of IEEE INFOCOM*, pages 2286–2294, April 2009.

[15] M. Cortes, J.O. Esteban, and H. Jun. Towards stateless core: Improving SIP proxy scalability. In *Proceedings of IEEE Globecom*, pages 1–6, December 2006.

[16] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[17] Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 57–66, New York, NY, USA, 2007. ACM.

[18] Charles Shen, Erich Nahum, Henning Schulzrinne, and Charles Wright. The impact of tls on sip server performance. In *Principles, Systems and Applications of IP Telecommunications*, IPTComm '10, pages 59–70, New York, NY, USA, 2010. ACM.

[19] C. Benvenuti. *Understanding Linux Network Internals*. O'Reilly, 2006.

[20] S.P. Bhattacharya and V. Apte. A measurement study of the Linux TCP/IP stack performance and scalability on SMP systems. In *Proceedings of the 1st International Conference on Communication Systems software and middleware (COMSWARE)*, pages 1–10, 2006.

[21] G. Chuanxiong and Z. Shaoren. Analysis and evaluation of the TCP/IP protocol stack of LINUX. In *Proceedings of the International Conference on Communication Technology Proceedings (ICCT 2000)*, 2000.

[22] W. Wu, M. Crawford, and M. Bowden. The performance analysis of Linux networking – packet receiving. *Computer Communications*, 30:1044–1057, March 2007.

[23] Red-Hat Linux Performance tuning guide. Linux Packet Path. https://access.redhat.com/documentation/enUS/Red_Hat_Enterprise_-Linux/6/htmlsingle/Performance_Tuning_Guide/index.html.

[24] J.C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[25] V. Anand and B. Hartner. TCP/IP Network Stack Performance in Linux Kernel 2.4 and 2.5. Proc. of Linux Symposium, 2008.

[26] Linux Foundation. NAPI. http://www.linuxfoundation.org/collaborate/workgroups/networkingnapi,.

[27] K. Salah and A. Kahtani. Improving snort performance under linux. *IET Communications*, 3(12):1883–1895, Dec. 2009.

[28] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache behavior of network protocols. In *Proceedings of ACM SIGMETRICS*, volume 25, pages 169–180, June 1997.

[29] Fang Liu, Fei Guo, Yan Solihin, Seongbeom Kim, and Abdulaziz Eker. Characterizing and modeling the behavior of context switch misses. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 91–101, New York, NY, USA, 2008. ACM.

[30] V. Babka, L. Marek, and P. Tuma. When misses differ: Investigating impact of cache misses on observed performance. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 112–119, Dec 2009.

[31] F.E. Goncalves. *Building Telephony Systems with OpenSIPS 1.6*. PACKT publishing, 2004.

[32] Leonard Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[34] Ben Lee *et al.* Hantak Kwak. Effects of multithreading on cache performance. *IEEE Transactions on Computer*, 48(2):176–184, Feb. 1999.

[35] V Beltran, J Torres, and E Ayguade. Understanding tuning complexity in multithreaded and hybrid web servers. pages 1–12, April. 2008.

[36] H. Jamjoom, C.T. Chou, and K.G. Shin. Impact of concurrency gains on the analysis and control of multi-threaded internet services. pages 827–837, March 2004.

[37] Packet Drop Rate. http://www.telecompute.com/voip.asp/.

[38] J. MacGregor Smith. $M/G/c/K$ blocking probablity models and system performance, 2002.

[39] Linux CFS Proces Scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[40] Jonghun Yoo Sungju Huh and SeongSoo Hong. Improving interactivity via vt-cfs and framework-assisted task characterization for linux/android smartphones. 2012.

[41] S.Zeaddally K.Salah, A.Manea and Jose M.Alcaraz Calero. On liux starvation of cpu-bound processes in the presense of network i/o. *Computer and Electrical Engineeting*, 30, 2011.

[42] Ajoy K. Datta and Rajesh Patel. Cpu scheduling for power/energy management on multi-core processors using cache misses and context switch data. *IEEE Transactions on Parallel and Distributed Sys*, pages 1190–1199, May 2013.

[43] Cisco White Paper. Capacity Management and Optimization of Voice Traffic. https://www.cisco.com/en/US/technologies/tk869/tk769/technologies_white_paper0900aecd8070329d.html.

[44] CFS Tuning by IBM. http://tinyurl.com/CFSTuning.

[45] C.P. Wright, E.M. Nahum, D. Wood, J.M. Tracey, and E.C. Hu. Sip server performance on multicore systems. *IBM Journal of Research and Development*, 54(1):7:1–7:12, January 2010.

[46] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 254–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[47] David Breitgand, Gilad Kutiel, and Danny Raz. Cost-aware live migration of services in the cloud. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, New York, NY, USA, 2010. ACM.

[48] Hongbo Jiang, A. Iyengar, E. Nahum, W. Segmuller, A.N. Tantawi, and C.P. Wright. Design, implementation, and performance of a load balancer for sip server clusters. *Networking, IEEE/ACM Transactions on*, 20(4):1190–1202, Aug 2012.

[49] M. Femminella, F. Giacinti, and G. Reali. Optimal deployment of open source application servers providing multimedia services. *Network, IEEE*, 28(5):54–63, September 2014.

[50] Cfs load balancing tuning for sql. http://tinyurl.com/CFSSQLLoad.

[51] G. Casale, Ningfang Mi, and E. Smirni. Model-driven system capacity planning under workload burstiness. *Computers, IEEE Transactions on*, 59(1):66–80, Jan 2010.

[52] Di Niu, Zimu Liu, Baochun Li, and Shuqiao Zhao. Demand forecast and performance prediction in peer-assisted on-demand streaming systems. In *INFOCOM, 2011 Proceedings IEEE*, pages 421–425, April 2011.

[53] A. Wolke, M. Bichler, and T. Setzer. Planning vs. dynamic control: Resource allocation in corporate clouds. *Cloud Computing, IEEE Transactions on*, PP(99):1–1, 2015.

APPENDICES

Appendix A

Kernel, OpenSIPS and SIPp Modification

Here we are attaching the modification introduced to the Linux kernel, OpenSIPS and SIPp. The attachments are the README and 'diffs' compared to the base code. The complete files will be provided as media file.

A.1 Kernel Modification

List of files :

```
/usr/include/asm_sockios.h
include/asm-x86/sockios.h

include/linux/skbuff.h
include/net/sock.h

net/ipv4/af_inet.c
net/ipv4/udp.c
net/core/sock.c
```

Description of Changes:

- Defines for SIOCGSTAMPUDP and SIOCGSTAMPRECV are:
  /usr/include/asm_sockios.h include/asm-x86/sockios.h
- The fields to store the new stamps in:
  include/linux/skbuff.h include/net/sock.h

- Handle the ioct call from user and provide the timestamp to user space in: $net/ipv4/af\_inet.c$
- Set the timestamps after udp stack processing and in recvfrom (currently checks if port is a SIP port 5060) in: (for other UDP apps, this check can be removed, for TCP similar change will be needed) net/ipv4/udp.c
- Define $sock\_get\_timestamp\_recvfrom()$ and $sock\_get\_timestamp\_udp()$ that copies the timestamp to socket structure, called from $inet\_ioctl()$ in $af\_inet.c$

  net/core/sock.c

A.1.1 Diffs

```
include_asm-x86_sockios.h
--------------------------

11a12,13
> #define SIOCGSTAMPRECV 0x890a /* Get stamp (timeval) */
> #define SIOCGSTAMPUDP 0x1234 /* Get stamp (timeval) */

/usr/include/asm_sockios.h
--------------------------
11a12,13
> #define SIOCGSTAMPRECV 0x890a /* Get stamp (timeval) */
> #define SIOCGSTAMPUDP 0x1234 /* Get stamp (timeval) */


include/linux/skbuff.h
----------------------


257a258,259
>  ktime_t tstamp_udp;
>  ktime_t tstamp_sockrecv;


include/net/sock.h
----------------------
```

```
265a266,269
>
>  ktime_t sk_stamp_recvfrom;
>  ktime_t sk_stamp_udp;
>
1256a1261
>  ktime_t kt_udp = skb->tstamp_udp;
1261a1267,1268
>
>         sk->sk_stamp_udp  = kt_udp;
1323a1331,1334
> extern int sock_get_timestamp_recvfrom(struct sock *, struct timeval __user *);
> extern int sock_get_timestamp_udp(struct sock *, struct timeval __user *);
>
>

net/ipv4/af_inet.c
----------------------

810a811,818
>  case SIOCGSTAMPRECV:
>  err = sock_get_timestamp_recvfrom(sk,
>   (struct timeval __user *)arg);
>  break;
>  case SIOCGSTAMPUDP:
>  err = sock_get_timestamp_udp(sk,
>   (struct timeval __user *)arg);
>  break;
net/ipv4/udp.c
----------------------

107a108,109
> #include <linux/hrtimer.h>
>
904a907,909
>         if (ntohs(udp_hdr(skb)->dest) ==  5060) {
>             sk->sk_stamp_recvfrom  = ktime_get_real();
```

```
>         }
1189a1195,1196
>
>
1191,1193c1198,1204
<  if (!sock_owned_by_user(sk))
<  ret = udp_queue_rcv_skb(sk, skb);
<  else
---
>          if (ntohs(udp_hdr(skb)->dest) ==  5060) {
>                     skb->tstamp_udp = ktime_get_real();
>          }
>
>  if (!sock_owned_by_user(sk)) {
>       ret = udp_queue_rcv_skb(sk, skb);
>          } else
1216a1228,1229
>
>
net/core/sock.c
----------------------


300a301,302
>        // skb->tstamp_udp = ktime_get_real();
>
1741a1744,1745
>  sk->sk_stamp_udp = ktime_set(-1L, 0);
>  sk->sk_stamp_recvfrom = ktime_set(-1L, 0);
1796a1801,1828
> int sock_get_timestamp_recvfrom(struct sock *sk, struct timeval __user *userstamp)
> {
>  struct timeval tv;
>
>  tv = ktime_to_timeval(sk->sk_stamp_recvfrom);
>
>  if (tv.tv_sec == -1) {
>  return -ENOENT;
```

```
>  }
>
>  return copy_to_user(userstamp, &tv, sizeof(tv)) ? -EFAULT : 0;
> }
> EXPORT_SYMBOL(sock_get_timestamp_recvfrom);
>
> int sock_get_timestamp_udp(struct sock *sk, struct timeval __user *userstamp)
> {
>  struct timeval tv;
>
>  tv = ktime_to_timeval(sk->sk_stamp_udp);
>
>  if (tv.tv_sec == -1) {
>  return -ENOENT;
>  }
>
>  return copy_to_user(userstamp, &tv, sizeof(tv)) ? -EFAULT : 0;
> }
> EXPORT_SYMBOL(sock_get_timestamp_udp);
>
```

A.2 OpenSIPS Modification

Description of the modification performed on OpenSIPS SPS server.

- receive.c: Implement the callid, and cseq number function here for logging with the packet in *receive_msg*() get current time. after the parsing, make ioctl call with SIOCGSTAMP, SIOGGSTAMPUDP and SIOCGSTAMPRECV and use these timestamps to get relevant data. log the data to syslog.
- $forward.c, modules/tm/twd.c, modules/tmi\_t\_reply.c$: Get the current timestamp just before the packet is sent, get another timestamp after packet is sent. Log the timestamp and the diff to syslog

```
receive.c:
-----------
```

```
47a48
> #include <sys/ioctl.h>
63a65
>
82a85,132
> static int
> get_callid(struct sip_msg* msg, str *cid)
> {
>     if (msg->callid == NULL) {
>         if (parse_headers(msg, HDR_CALLID_F, 0) == -1) {
>             LM_ERR("cannot parse Call-ID header\n");
>             return 0;
>         }
>         if (msg->callid == NULL) {
>             LM_ERR("missing Call-ID header\n");
>             return 0;
>         }
>     }
>
>     *cid = msg->callid->body;
>
> //    trim(cid);
>
>     return 1;
> }
>
> static int
> get_cseq_number(struct sip_msg *msg, str *cseq, int *method_id)
> {
>     if (msg->cseq == NULL) {
>         if (parse_headers(msg, HDR_CSEQ_F, 0)==-1) {
>             LM_ERR("cannot parse CSeq header\n");
>             return 0;
>         }
>         if (msg->cseq == NULL) {
>             LM_ERR("missing CSeq header\n");
>             return 0;
```

```
>         }
>         }
>
>         *cseq = get_cseq(msg)->number;
>         *method_id  = get_cseq(msg)->method_id;
>
>     if (cseq->s==NULL || cseq->len==0) {
>         LM_ERR("missing CSeq number\n");
>         return 0;
>     }
>
>     return 1;
> }
>
>
>
88a139,161
>         str callid, cseq;
>         int method_id;
>         struct timeval tim1, tim2, wtime,  wtime_recv;
>         struct timeval wtime_udp ;
>         double time1 =0 ;
>         double time2 =0 ;
>         double time3 =0 ;
>         double time4 =0 ;
>         double time5 =0 ;
>         double time6 =0 ;
>         double time9 =0 ;
>         double time10 =0 ;
>         double time11 =0 ;
>         double time12 =0 ;
>         double prev = 0, now = 0, waittime = 0;
>         double sock_recvtime =0;
>         double udp_recvtime =0;
>  int werror = 0;
>  int werror_recvfrom = 0;
>  int werror_udp = 0;
```

```
>
>  memset(&tim1, 0, sizeof(struct timeval));
>         gettimeofday(&tim1, NULL);
114a188,229
>     if  ((get_callid(msg, &callid)) && (get_cseq_number(msg, &cseq, &method_id))) {
>
>  memset(&wtime, 0, sizeof(struct timeval));
>  memset(&tim2, 0, sizeof(struct timeval));
>
>  memset(&wtime_recv, 0, sizeof(struct timeval));
>  memset(&wtime_udp, 0, sizeof(struct timeval));
>
>         werror  = ioctl(rcv_info->bind_address->socket, SIOCGSTAMP, &wtime);
>         werror_recvfrom  = ioctl(rcv_info->bind_address->socket, SIOCGSTAMPRECV,
>       &wtime_recv);
>         werror_udp  = ioctl(rcv_info->bind_address->socket, SIOCGSTAMPUDP,
>       &wtime_udp);
>         time1 = tim1.tv_sec;
>         time2 = tim1.tv_usec;
>         prev = time1 * 1000000  + time2;
>
>         time5 = wtime.tv_sec;
>         time6 = wtime.tv_usec;
>         waittime = time5 *  1000000  + time6;
>
>
>         time9 = wtime_recv.tv_sec;
>         time10 = wtime_recv.tv_usec;
>         sock_recvtime = time9 *  1000000  + time10;
>
>         time11 = wtime_udp.tv_sec;
>         time12 = wtime_udp.tv_usec;
>         udp_recvtime = time11 *  1000000  + time12;
>
>         gettimeofday(&tim2, NULL);
>         time3 = tim2.tv_sec;
>         time4 = tim2.tv_usec;
```

```
>         now = time3 *  1000000  + time4;
>
>     LM_ERR("ramekris  src_ip %d parse_time=%lf  wait_time=%lf stack_time=%lf sock_recvtim
>       *(rcv_info->src_ip.u.addr32),   now - prev, prev - waittime,
>        udp_recvtime  - waittime, sock_recvtime - udp_recvtime, (long long ) now,
>        msg->REQ_METHOD , callid.len, callid.s,
>         cseq.len, cseq.s, method_id
>        );
>      }
```

forward.c:

```
60a61
> #include <sys/time.h>
376a378,386
>         struct timeval tim2;
>         struct timeval tim3;
>         double time3 =0;
>         double time4 =0;
>         double now =0;
>         double time5 =0;
>         double time6 =0;
>         double ip_out =0;
>
397a408,410
>
>
>
431a445,449
>                 gettimeofday(&tim2, NULL);
>                 time3 = tim2.tv_sec;
>                 time4 = tim2.tv_usec;
>                 now = time3 *  1000000  + time4;
>
```

```
436a455,464
>                 gettimeofday(&tim3, NULL);
>                 time5 = tim3.tv_sec;
>                 time6 = tim3.tv_usec;
>                 ip_out = time5 *  1000000  + time6;
>
>             LM_ERR("ramekris now=%lld ip_out=%lld request %d callid=%.*s cseq=%.*s",
>            (long long) now, (long long) (ip_out - now),
>                   msg->REQ_METHOD, msg->callid->body.len, msg->callid->body.s,
>                   msg->cseq->body.len, msg->cseq->body.s);
>
451a480
>
619,620c648,652
<  LM_DBG("reply forwarded to %.*s:%d\n", msg->via2->host.len,
<  msg->via2->host.s, (unsigned short) msg->via2->port);
---
>  LM_ERR("reply forwarded to %.*s:%d method %d callid %.*s\n",
>                 msg->via2->host.len,
>  msg->via2->host.s, (unsigned short) msg->via2->port,
>                 msg->REQ_METHOD , msg->callid->body.len, msg->callid->body.s
>                  );
```

```
t_fwd.c:
--------

62a63
> #include <sys/time.h>
605a607,614
>  struct timeval tim2;
>  struct timeval tim3;
>  double time3 =0;
>  double time4 =0;
>  double now =0;
```

```
>         double time5 =0;
>         double time6 =0;
>         double ip_out =0;
699a709,712
>  gettimeofday(&tim2, NULL);
>          time3 = tim2.tv_sec;
>          time4 = tim2.tv_usec;
>          now = time3 *  1000000  + time4;
700a714,730
>
>      gettimeofday(&tim3, NULL);
>              time5 = tim3.tv_sec;
>              time6 = tim3.tv_usec;
>              ip_out = time5 *  1000000  + time6;
>
>
>                                              LM_ERR("ramekris now=%lld ip_out=%lld request
>                    (long long) now, (long long)
>      (ip_out - now),
>                                             p_msg->REQ_METHOD,
>     p_msg->callid->body.len,
>     p_msg->callid->body.s,
>                                             p_msg->cseq->body.len,
>                                             p_msg->cseq->body.s);
>
>


t_reply.c
---------

80a81
> #include <sys/time.h>
92a94
> #include <sys/time.h>
289a292,300
>         struct timeval tim2;
```

```
>         double time3 =0;
>         double time4 =0;
>         double now =0;
>         struct timeval tim3;
>         double time5 =0;
>         double time6 =0;
>         double ip_out =0;
>
306a318,322
>         gettimeofday(&tim2, NULL);
>         time3 = tim2.tv_sec;
>         time4 = tim2.tv_usec;
>         now = time3 *  1000000  + time4;
>
307a324,334
>
>         gettimeofday(&tim3, NULL);
>         time5 = tim3.tv_sec;
>         time6 = tim3.tv_usec;
>         ip_out = time5 *  1000000  + time6;
>
>         LM_ERR("ramekris  now=%lld ip_out=%lld request %d callid=%.*s cseq=%.*s",
>  (long long) now, (long long) (ip_out - now),
>                 rpl->REQ_METHOD, rpl->callid->body.len, rpl->callid->body.s,
>                 rpl->cseq->body.len, rpl->cseq->body.s);
>
1110a1138,1146
>         struct timeval tim2;
>         double time3 =0;
>         double time4 =0;
>         double now =0;
>         struct timeval tim3;
>         double time5 =0;
>         double time6 =0;
>         double ip_out =0;
>
1239a1276,1281
```

```
>
>                 gettimeofday(&tim2, NULL);
>                 time3 = tim2.tv_sec;
>                 time4 = tim2.tv_usec;
>                 now = time3 *  1000000  + time4;
>
1240a1283,1294
>
>                 gettimeofday(&tim3, NULL);
>                 time5 = tim3.tv_sec;
>                 time6 = tim3.tv_usec;
>                 ip_out = time5 *  1000000  + time6;
>
>                 LM_ERR("ramekris  now=%lld ip_out=%lld request %d callid=%.*s cseq=%.*s",
>      (long long) now, (long long) (ip_out - now),
>                    p_msg->REQ_METHOD, p_msg->callid->body.len,
>                    p_msg->callid->body.s,
>                    p_msg->cseq->body.len, p_msg->cseq->body.s);
>
```

A.3 SIPp Modification

The modifications introduced to SIPp (version 3.1 SIPp code base), file modified 'sipp.cpp'

1. Add the Command-Line change for $'iat\_file'$
2. Read the inter-arrivals and delay the next INVITE for that period
3. Make the clock more granular 200uS from 1 mS

```
66a67
>
116c117,118
< #define SIPP_OPTION_INDEX_FILE    33
---
> #define SIPP_OPTION_IAT_FILE      33
> #define SIPP_OPTION_INDEX_FILE    34
```

```
159a162
>         {"iat_file", "Input file specifying the Inter-arrival time for next Outgoing Invi
348c351,352
<   return getmicroseconds() / 1000LL;
---
>   //return getmicroseconds() / 1000LL;
>   return getmicroseconds() /200LL;
3277a3282,3290
>
>   /*
>    * Variables used for reading IAT file and
>    * using the read data for next call time
>    */
>   unsigned long iat_msec;
>   unsigned int  next_call_time;
>   FILE        *f_iat;
>
3280d3292
<
3288a3301,3306
>
>   f_iat = fopen(iat_file, "r");
>   if (f_iat == NULL ) {
>     ERROR("\n Unable to Open IAT file - it is required!");
>   }
>
3347,3350c3365,3366
<           while((calls_to_open--) &&
<                 (!open_calls_allowed || current_calls < open_calls_allowed) &&
<                 (total_calls < stop_after))
<             {
---
>           while( total_calls < stop_after )   //Modified to support IAT
>           {
3357a3374,3377
>              /*
>               * Check Added to Support IAT file
```

```
>               */
>             if (getmilliseconds() >= next_call_time) {
3383c3403,3407
<         call_ptr -> run();
---
>          call_ptr -> run();
>                 fscanf(f_iat, " %u", &iat_msec);
>                 next_call_time = getmilliseconds() + iat_msec;
>
>               }    //End of check block for IAT
3390a3415,3417
>
>
>
3392c3419,3420
<         if (new_time > (first_open_tick + (timer_resolution < 2 ? 1 : (timer_resolution / 1
---
>
>         if (new_time > first_open_tick ) {  //check modified for IAT
3398c3426
<      set_rate(rate);
---
>      //set_rate(rate);  //rate not set for IAT
3562c3590
<  set_rate(rate);
---
> // set_rate(rate);  // rate not set for IAT
3565a3594
>   fclose(f_iat);
4186d4214
<
4364a4393,4403
>
>  case SIPP_OPTION_IAT_FILE:
>    REQUIRE_ARG();
>    CHECK_PASS();
>    /* By default, the first file is used for IP address input. */
```

```
>    if (!iat_file) {
>      iat_file = argv[argi ];
>             //ERROR("File name found for -iat_file: %s", argv[argi ]);
>    }
>    break;
>
```

Appendix B

Call Arrival Rate and Number of Interrupts

As the call arrival rate increases, the rate of packets arrving at the SPS increases in proportion. The increased packet rate causes an increase in the number of interrupts. In Chapter 4 we introduced the parameter α to capture the impact of increased interrupt on the waiting time. The outputs below shows the average number of interrupts per second seen by the kernel for various Calls per second.

- 1000 cps

```
Average:      CPU    intr/s
Average:      all  11288.53
Average:        0   8849.51
Average:        1      0.00
Average:        2    0.00
Average:        3    0.00
```

- 1500 cps

```
Average:      CPU    intr/s
Average:      all  19340.27
Average:        0  11657.27
Average:        1      0.00
Average:        2    0.00
Average:        3    0.00
```

- 2000 cps

```
Average:      CPU    intr/s
Average:      all  22908.33
Average:        0  13221.17
Average:        1      0.00
Average:        2    0.00
Average:        3    0.00
```

- 2500 cps

```
Average:      CPU    intr/s
Average:      all  26946.75
Average:        0  14514.25
Average:        1      0.00
Average:        2    0.00
Average:        3    0.00
```