STREAM CONTROL TRANSMISSION PROTOCOL SUPPORT IN SESSION INITIATION PROTOCOL PROXY SERVER

by

Thomas Kwok-Cheong Pang
B.A.Sc. (Computer Engineering), Simon Fraser University, 1995

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENT FOR THE DEGREE OF

MASTER OF ENGINEERING

in the School

of

Engineering Science

SIMON FRASER UNIVERSITY

May 2003

Approval

Name: Thomas Kwok-Cheong Pang

Degree: Master of Engineering

Title of report: Stream Control Transmission Protocol Support In Session Initiation Protocol Proxy Server

Examining Committee:

Chair: Dr. John Bird
Professor

_____
Dr. Ljiljana Trajkovic
Senior Supervisor
Associate Professor

_____
Dr. James K. Cavers
Supervisor
Professor

_____
Mr. Patrick Leung
Supervisor
Senior Lecturer

_____
Mr. Trevor Cooper
External Supervisor
Project Manager
Intel Corporation

Date approved: _____

Abstract

In recent years, Session Initiation Protocol (SIP) developed by the Internet Engineering Task Force (IETF) has gained significant popularity in the Voice-over-IP (VoIP) arena and is competing with the Internet Multimedia protocol H.323. SIP is also selected by Third Generation Partnership Project (3GPP) as a standard signaling protocol for service control in Third Generation (3G) wireless network.

SIP is a communication control protocol capable of running on different transport layers, e.g., Transport Control Protocol (TCP), User Datagram Protocol (UDP), or Stream Control Transmission Protocol (SCTP). Today's SIP application is mostly operating over the unreliable transport protocol UDP. In lossy environment such as wireless networks and congested Internet networks, SIP messages can be lost or delivered out of sequence. The SIP application then has to retransmit the lost messages and re-order the received packets. This additional processing overhead can degrade the performance of the SIP application. To solve this problem, researchers are looking for a more suitable transport layer for SIP. SCTP, a transport protocol providing acknowledged, error-free, non-duplicated transfer of messages, has been proposed to be an alternative to UDP and TCP [1] [2]. The multi-streaming and multi-homing features of SCTP are especially attractive for applications that have stringent performance and high reliability requirements. An example is the SIP proxy server.

In this project, I successfully implemented a SIP system running on SCTP. The SIP system is comprised of two SIP user agents called *linphone* connecting via a SIP proxy server called *Partysip*. Both Linphone and Partysip were modified to run on SCTP. All components are running on Intel Pentium platform with Redhat Linux operating system. The network protocol analyzer *Ethereal* was used to monitor SIP and SCTP packets in the network and to measure the latency of SIP transactions. *NIST Net* was used to simulate packet loss in the network. Both SIP proxy server and user agent are publicly available under GNU Public License. To encourage future benchmarking activities, academic work and research on SCTP support for SIP, the modification to Partysip has been released to the public.

Acknowledgments

I would like to give my special thanks to my advisor, Dr. Ljiljana Trajkovic, for her guidance and support throughout my project. I would like to express my gratitude to Dr. James Cavers, Mr. Patrick Leung, and Mr. Trevor Cooper for serving on my supervisory committee, and Dr. John Bird for being the chair of my project presentation. I also thank Intel Inc. for lending me equipment for my development. Above all, I dedicate this report to my wife Michelle for her encouragement, patience, support and love.

Table of Contents

APPROVAL II

ABSTRACT III

ACKNOWLEDGMENTS V

TABLE OF CONTENTS VI

LIST OF FIGURES VIII

LIST OF TABLES XI

ACRONYMS AND ABBREVIATIONS XII

1 INTRODUCTION 1

1.1 CIRCUIT-SWITCHED TELEPHONE NETWORK 1
1.2 IP TELEPHONY 2
1.3 PROJECT MOTIVATION 5
1.4 PROJECT SCOPE 6

2 SESSION INITIATION PROTOCOL 7

2.1 BASIC CONCEPTS 7
2.2 SIP ARCHITECTURE 8
2.3 SIP MESSAGES 11
2.4 SAMPLE MESSAGE FLOW 13

3 STREAM CONTROL TRANSMISSION PROTOCOL 16

3.1 SCTP OVERVIEW 16
3.2 SAMPLE MESSAGE FLOW 17
3.3 CONGESTION CONTROL 20
3.4 MULTI-STREAMING 20
3.5 MULTI-HOMING 23
3.6 SCTP IMPLEMENTATION 25
3.6.1 Siemens SCTP Library (SCTPLIB) *26*
3.6.2 Randall's SCTP Kernel (LKSCTP) *29*

4 HIGH LEVEL DESIGN OF SCTP PLUGIN 31

4.1 HIGH LEVEL ARCHITECTURE 31
4.2 SCTP PLUGIN 35
4.2.1 Data Structures *36*
4.2.2 Function Prototypes *38*

5 EXPERIMENTAL SETUP AND RESULTS 41

5.1 SOFTWARE CONFIGURATION 41

5.1.1 SCTP Kernel *41*
5.1.2 SIP Proxy Server *43*
5.1.3 SIP User Agent *44*
5.1.4 Ethereal *44*
5.1.5 NIST Net *45*
5.1.6 TTCP *46*
5.2 COMPARISON OF TRANSPORT PROTOCOL THROUGHPUT 46
5.3 SINGLE-SESSION PERFORMANCE TEST 51
5.3.1 SIP Registration *52*
5.3.2 SIP Call Setup *55*
5.3.3 SIP Call Termination *57*
5.4 MULTI-SESSION PERFORMANCE TEST 58
5.5 MULTI-HOMING 61
5.6 TEST RESULT SUMMARY 63
6 AREAS FOR IMPROVEMENT 64
7 CONCLUSIONS 65
LIST OF REFERENCES 66
APPENDIX A – DISCLAIMER OF PARTYSIP 68
APPENDIX B - SCTP PLUGIN RELEASE NOTICE 69
APPENDIX C – MODIFICATIONS TO PARTYSIP 70
APPENDIX D – MODIFICATIONS TO LINPHONE 80
APPENDIX F – TEST PROGRAMS 84

List of Figures

FIGURE 1.1: CIRCUIT-SWITCHED NETWORKING BETWEEN TELEPHONE NETWORK AND CELLULAR NETWORK. 2

FIGURE 1.2: PACKET-SWITCHED NETWORKING BETWEEN TELEPHONE AND CELLULAR NETWORKS. IN THIS MODEL, THE SIGNALING AND VOICE PATHS ARE SEPARATED. THE MSC AND PBX IN CIRCUIT-SWITCHED NETWORK SHOWN IN FIGURE 1.1 ARE REPLACED BY SOFTSWITCHES. 4

FIGURE 2.1: INTERNET MULTIMEDIA PROTOCOL STACK. H.323 IS OPERATING ON TCP; SIP ON TCP, SCTP ON UDP; AND RTP ON UDP. 8

FIGURE 2.2: SIP ARCHITECTURE MAINLY CONSISTS OF SIP USER AGENT, SIP NETWORK SERVER (PROXY OR REDIRECT SERVER), AND LOCATION SERVICE 9

FIGURE 2.3: SIP REGISTRAR TO KEEP TRACK OF SIP USER AGENT CURRENT LOCATION. 11

FIGURE 2.4: SIP REGISTRATION MESSAGE FLOW DIAGRAM 13

FIGURE 2.5: BASIC CALL SETUP AND TEARDOWN MESSAGE FLOW DIAGRAM 14

FIGURE 3.1: SS7 SIGNALING GATEWAY. IT ACTS AS A BRIDGE BETWEEN PSTN AND IP NETWORK. 17

FIGURE 3.2: SCTP ASSOCIATION INITIALIZATION PROCEDURES 18

FIGURE 3.3: SCTP USER DATA PASSING. 19

FIGURE 3.4: SCTP HEARTBEAT MECHANISM. 19

FIGURE 3.5: SCTP ASSOCIATION SHUTDOWN PROCEDURES. 19

FIGURE 3.6: SCTP MULTI-STREAMING FEATURE. PACKET LOSS IN ONE STREAM WILL NOT AFFECT OTHER STREAMS. THIS SOLVES THE HEAD-OF-LINE BLOCKING PROBLEM IN TCP. 21

FIGURE 3.7: TCP AND SCTP PERFORMANCE COMPARISON IN SINGLE-SESSION ENVIRONMENT. TCP PERFORMS BETTER THAN SCTP. SOURCE: RAJAMANI ET AL., 2002 [13], BY PERMISSION. 22

FIGURE 3.8: SCTP AND TCP PERFORMANCE COMPARISON IN MULTI-SESSION ENVIRONMENT. SCTP PERFORMS BETTER THAN TCP BECAUSE

OF ITS MULTI-STREAMING FEATURE. SOURCE: RAJAMANI ET AL., 2002 [13], BY PERMISSION.....22

FIGURE 3.9: SCTP MULTI-HOMING. IN THIS SETUP, WE HAVE FOUR POSSIBLY INDEPENDENT PATHS IN AN ASSOCIATION BETWEEN HOST 1 AND HOST 2. SOURCE: BRENNAN ET AL., 2001 [14], BY PERMISSION.....23

FIGURE 3.10: SINGLE-HOMED WITH PACKET LOSS. FIRST PACKET IS DROPPED AT AROUND 1.5 SECONDS. THE RETRANSMITTED PACKET FOR THE FIRST LOST PACKET IS ALSO LOST. AS A RESULT, THE TRANSMISSION HALTS FOR MORE THAN 3 SECONDS BEFORE RECOVERY. SOURCE: BRENNAN ET AL., 2001 [14], BY PERMISSION.....24

FIGURE 3.11: MULTI-HOMED WITH PACKET LOSS. IT TAKES ABOUT 0.5 SECONDS TO RECOVER. SOURCE: BRENNAN ET AL., 2001 [14], BY PERMISSION.....24

FIGURE 4.1: PROGRAM FLOW DIAGRAM OF THE MAIN THREAD IN PARTYSIP.....32

FIGURE 4.2: INTERACTIONS BETWEEN TLP AND IMP. SIP REQUESTS AND RESPONSES RECEIVED BY TLP ARE RESPECTIVELY PLACED IN SIP_TRAFFIC_FIFO AND SIP_ACK_FIFO QUEUES. TLP THEN SIGNALS IMP ABOUT THE INCOMING SIP MESSAGES FOR PROCESSING. 34

FIGURE 5.1: EXPERIMENTAL SETUP FOR THROUGHPUT TEST. 47

FIGURE 5.2: THROUGHPUT COMPARISON AMONG DIFFERENT TRANSPORT PROTOCOLS. THE THROUGHPUT PERFORMANCE IS MEASURED IN TERMS OF MBPS AGAINST VARIOUS PACKET SIZE IN BYTES. THE RESULTS SHOW THAT UDP HAS THE BEST THROUGHPUT PERFORMANCE FOLLOWED BY TCP, SIEMENS SCTP, AND RANDALL'S SCTP KERNEL.....49

FIGURE 5.3: FRAMING STRUCTURES OF UDP, TCP, AND SCTP. 50

FIGURE 5.4: EXPERIMENTAL SIP NETWORK. UA1 AND UA2 ARE LOCATED IN DIFFERENT SUBNETS AND ARE CONNECTED VIA A SIP PROXY SERVER. 52

FIGURE 5.5: SIP REGISTRATION TEST RESULT. THIS IS THE MESSAGE FLOW CAPTURED BY ETHEREAL. PLEASE NOTE THE ADDITIONAL OVERHEAD CAUSED BY SCTP HANDSHAKING. THE MESSAGE OF THE REGISTRATION FROM UA2 SHOULD BE THE SAME. 53

FIGURE 5.6: CALL SETUP TEST RESULT. THIS MESSAGE FLOW IS CAPTURED BY *ETHEREAL*. T1 IS THE TIME TAKEN FOR UA2 TO ANSWER THE CALL AFTER THE PHONE RINGS. 56

FIGURE 5.7: SIP CALL TERMINATION TEST RESULT CAPTURED BY *ETHEREAL*. SACK MESSAGES ADD OVERHEAD TO THE CALL TERMINATION DURATION. 57

FIGURE 5.8: MULTI-STREAMING TEST SETUP. THE LINUX MACHINE INSTALLED WITH NIST NETWORK EMULATOR IS THE DEFAULT GATEWAY OF MACHINES A, B AND THE SIP PROXY SERVER. EIGHT IDENTICAL SIP USER AGENT PROGRAMS ARE RUNNING SIMULTANEOUSLY ON MACHINE A AND B. 58

FIGURE 5.9: EXPERIMENTAL SETUP FOR MULTI-HOMING. 62

List of Tables

TABLE 4.1: THREADS IN PARTYSIP. 33

TABLE 4.2: SCTP FUNCTIONAL REQUIREMENTS....35

TABLE 5.1: COMMONLY USED CNISTNET COMMANDS....46

TABLE 5.2: TEST PROGRAM EXECUTION PROCEDURES....47

TABLE 5.3: MULTI-STREAMING PERFORMANCE (NUMBER OF SIP CALL TRANSACTIONS). 60

TABLE 5.4: PACKET LOSS BETWEEN DIFFERENT GEOGRAPHICAL LOCATIONS. SOURCE: STANFORD LINEAR ACCELERATOR CENTER (SLAC), 2003 [24], BY PERMISSION....61

TABLE 5.5: TEST RESULT SUMMARY. 63

Acronyms and Abbreviations

3G	Third Generation cellular network
3GPP	3rd Generation Partnership Project
API	Application Programming Interface
BSS	Base-station Subsystem
CELP	Code Excited Linear Prediction
FIFO	First-In-First-Out
FRR	Fast retransmit and recovery
GPL	GNU Public License
HTTP	Hyper Text Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ISUP	ISDN User Part
LKSCTP	Randall's Linux SCTP kernel
MSC	Mobile Switching Center
OS	Operating System
PC	Personal Computer
PDA	Personal Digital Assistant
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RFC	Request For Comments

RSVP	Resource Reservation Setup Protocol
RTP	Real-Time Transport Protocol
SIP	Session Initiation Protocol (IETF RFC 3261)
SCTP	Stream Control Transmission Protocol (IETF RFC 2960)
SCTPLIB	Siemens SCTP library
SDP	Session Description Protocol (IETF RFC 2327)
TCP	Transport Control Protocol
TTCP	Test TCP - Benchmarking Tool for Measuring TCP and UDP Performance
UDP	User Datagram Protocol
ULP	SCTP Upper Layer Protocol
URL	Uniform Resource Identifiers (RFC 2396)
UTP	Unshielded Twisted Pair
VoIP	Voice-over-IP

1 Introduction

Telecommunication service providers are transforming their traditional circuit-switched networks into packet-based networks that facilitate new services that combine data, voice, and video information. The key driving force of this deployment is the lower cost associated with converged data and voice networks. In this Chapter, we first describe a traditional circuit-switched telephone network, followed by describing a packet-based network, known as IP Telephony. Next, the project motivation and the project scope are presented.

1.1 Circuit-switched Telephone Network

Figure 1.1 illustrates the circuit-switched networking between a private branch exchange (PBX) and a cellular network. A PBX is a private telephone network used within an enterprise. Users of the PBX share a certain number of outside lines for placing telephone calls external to PBX. A cellular network consists of a base-station subsystem (BSS) and a mobile switching center (MSC). The BSS takes care of the radio-related tasks, while the MSC routes incoming and outgoing calls and assigns user channels. The MSC connects to the PBX using dedicated T1 or T3 circuits for carrying both signaling (SS7) and voice (PCM) messages.

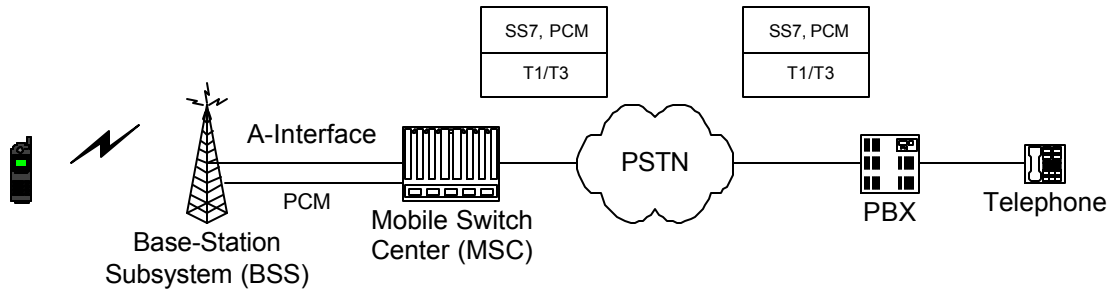


Figure 1.1: Circuit-switched networking between telephone network and cellular network.

In traditional telephone networks, when a call is made between two parties, the connection is maintained for the entire duration of the call and the connection is called a circuit. The network is called circuit- switched network. It is the foundation of the Public Switched Telephone Network (PSTN) [3]. One drawback of circuit-switched network is the wasted bandwidth and the limited capacity of the network. In typical phone conversation, when one party is talking, another party is listening. Therefore, only half of the connection is in use at any given time. Furthermore, there are times when both parties are not talking.

1.2 IP Telephony

IP Telephony, known as Voice-over-IP (VoIP), is the transmission of voice calls over a packet-switched IP network. The main advantages of VoIP include more efficient use of bandwidth, relative low capital and operating costs for the service providers or carriers, and the possibility of value added services.

In packet-switched network, many users share the same physical connection. This is achieved by breaking communication into small units of data called packets (in IP networks) or cells (in ATM networks). This project focuses on IP networks only. IP packets are routed through a network based on the destination address contained within the packet header. Data compression techniques such as Code Excited Linear Prediction (CELP) can further reduce the size of each packet. Besides better utilization of network resources, by integrating voice and data on a single converged network, companies save long distance changes, wiring fees, and administrative costs. IP telephony allows service providers to offer next-generation applications and advanced services that boost revenue. For example, unified messaging that provides voice mail, email and fax access from one centralized location using multiple devices such as Personal Computer (PC), phone, or a Personal Digital Assistant (PDA).

Next generation network separates the control signal path from the voice path (bearer path) as shown in Figure 1.2. The MSC and PBX in the circuit-switched network are replaced by softswitches. A softswitch is a software-based entity that provides call control functionality [4]. A new entity called media gateway is introduced into the network. The media gateway acts as a bridge between a circuit-switched network and the packet-switched network, and is controlled by softswitches using Media Gateway Control Protocol (MGCP). It converts the voice (PCM) signals into packets and vice versa. In IP network, voice packets are transported using Real-time Transport Protocol (RTP). The

call control signaling protocol can be either SS7 or SIP protocols. This project uses SIP for call control signaling.

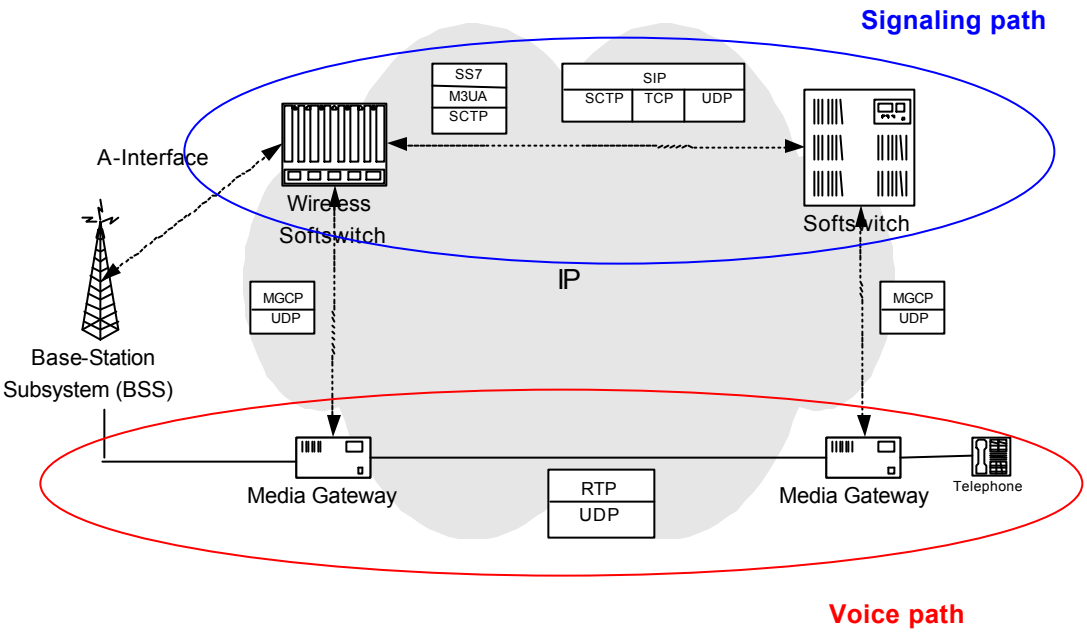


Figure 1.2: Packet-switched networking between telephone and cellular networks. In this model, the signaling and voice paths are separated. The MSC and PBX in circuit-switched network shown in Figure 1.1 are replaced by softswitches.

The separation of the signaling path and the voice path provides the following benefits:

1. Independent control and maintenance: Signaling and voice traffic have different network requirements (e.g., reliability, throughput, latency) and should be implemented independently.

2. Load balancing: Signaling and voice traffic can be distributed among multiple softswitches and media gateways respectively based on traffic load.

3. Redundancy: Signaling messages can be detoured from the active softswitch to the standby softswitch if the active softswitch fails. Similarly, voice traffic can be detoured from one media gateway to another during system failure.

4. Scalability: Softswitches and media gateways can be added or removed from the network based on the number of subscribers in the network, without interrupting the ongoing services.

1.3 Project Motivation

SIP can operate on User Datagram Protocol (UDP), Transport Control Protocol (TCP), or Session Control Transmission Protocol (SCTP). UDP provides unreliable datagram service, and relies on the application layer for error control, detection of message duplication, and retransmission of lost messages. TCP, on the other hand, provides error and flow control. However, its strict byte-order delivery poses performance issues. It also suffers from other drawbacks as mentioned in [5]. SCTP overcomes some of the limitations of TCP and provides a reliable datagram transport mechanism. SCTP also provides features required by a SIP system such as multi-stream message passing for performance, cookie mechanism for security, and multi-homing for fault-tolerance and high-availability. A SIP system, consisting of SIP user agents and servers, will be described in details in Chapter 2.2.

1.4 Project Scope

SIP user agents and servers available for public use include Kphone [6], Linphone [7], Vovida [8], and Partysip [9]. However, none of them support SCTP. Well-known SCTP implementations for public use include Siemens SCTPLIB and Randall's LKSCTP [10]. In this project, I successfully modified Partysip's proxy server and Linphone to use SCTP as the transport protocol. The modified proxy server was tested on Randall's LKSCTP. I measured the transaction response time of the proxy server under no loss condition and single session environment. I compared the performance of the modified proxy server and the original proxy server under lossy and multi-session environment. The modifications of the proxy server have been released to the public under GNU Public License (GPL). This should encourage future benchmarking activities, academic work, and research on SCTP support for SIP.

This report is organized as follows. Chapter 2 gives an overview of SIP. Chapter 3 presents SCTP studies from other institutes or companies. Two public SCTP implementations are also described and the pros and cons of each approach are examined. The design of SCTP support for Partysip software is detailed in Chapter 4. The test environment and the test results are presented in Chapter 5. Chapter 6 suggests areas for improvement, while Chapter 7 concludes our findings.

2 Session Initiation Protocol

2.1 Basic Concepts

Session Initiation Protocol (SIP), developed by the Internet Engineering Task Force (IETF), is a control protocol for creating, modifying and terminating session with one or more participants. A session can be an Internet call, multimedia conference session, or multimedia distribution. The protocol is defined in IETF RFC 3261 [11]. SIP is a lightweight protocol because it requires very few messages, called methods, for managing a basic session. These methods are INVITE, BYE, ACK, REGISTER, OPTIONS, CANCEL and INFO. The INVITE method is used to invite a user to join a session. It is similar to IAM message in SS7 ISDN User Part for Public Switched Telephone Network. The BYE method is used to terminate an established session. ACK confirms that a caller has received a final response to an INVITE. A user agent uses the REGISTER method to notify a SIP network of its current location. The OPTIONS method is used to query a user agent or server about its capabilities and discover its current availability. The CANCEL method is used to end a pending request. The INFO method is used to carry mid-call information.

SIP uses Session Description Protocol (SDP) to describe the session. In the case of video, audio, or multimedia session, the session information will be used for setting up an RTP stream, running on UDP that in turn operates on IP. SIP is independent of the transport layer, i.e., it can be used over UDP, TCP, or SCTP. For SIP using UDP, messages may be lost or received out of sequence.

SIP, therefore, uses its own reliable mechanisms via retransmission timers, command sequence (CSeq) numbers, and positive acknowledgments. The reliable mechanisms will be discussed later in the report. Figure 2.1 depicts the commonly used Internet multimedia protocol stack.

<table>
<tr><td>Application Layer</td><td>H.323</td><td colspan="2">SIP</td><td>RTP</td></tr>
<tr><td>Transport Layer</td><td colspan="2">TCP</td><td>SCTP</td><td>UDP</td></tr>
<tr><td>Internet Layer</td><td colspan="4">IP</td></tr>
</table>

Figure 2.1: Internet multimedia protocol stack. H.323 is operating on TCP; SIP on TCP, SCTP on UDP; and RTP on UDP.

2.2 SIP Architecture

A SIP system has two components: user agents and network servers. A user agent is an end system that acts on behalf of someone who wants to participate in calls. A user agent contains both a protocol client called user agent client (UAC), and a protocol server called user agent server (UAS). The UAC is used to initiate a call, while the UAS is used to answer a call. The presence of both in a user agent enables peer-to-peer operations.

SIP provides two different types of network servers: proxy and redirect. A SIP proxy server receives requests, determines where to send the requests, and then forwards the request to the next server on behalf of the user. A redirect server receives requests, but rather than passing these onto the next server, it sends a response to the caller indicating the address of the called user. The caller contacts the called party at the next server directly. The main function of a SIP network server is to provide name resolution and user location, i.e., very

much like Domain Name Server (DNS). Like Hyper Text Transfer Protocol (HTTP), SIP user is identified using Uniform Resource Locator (URL). The SIP network server looks up the URL either from the local database or remote location server, and finds the exact location (IP address) of the user.

A proxy server can either be stateful or stateless. When stateful, a proxy server remembers the incoming requests that generate outgoing requests. A stateless proxy forgets all information once an outgoing request is generated. Proxies that accept TCP connections or SCTP associations must be stateful. Otherwise, if the proxy were to lose a request, the TCP or SCTP client would never retransmit it. Therefore, in this project we assume a stateful proxy server.

Figure 2.2 depicts SIP architecture and illustrates the basic message flow for setting up a session.

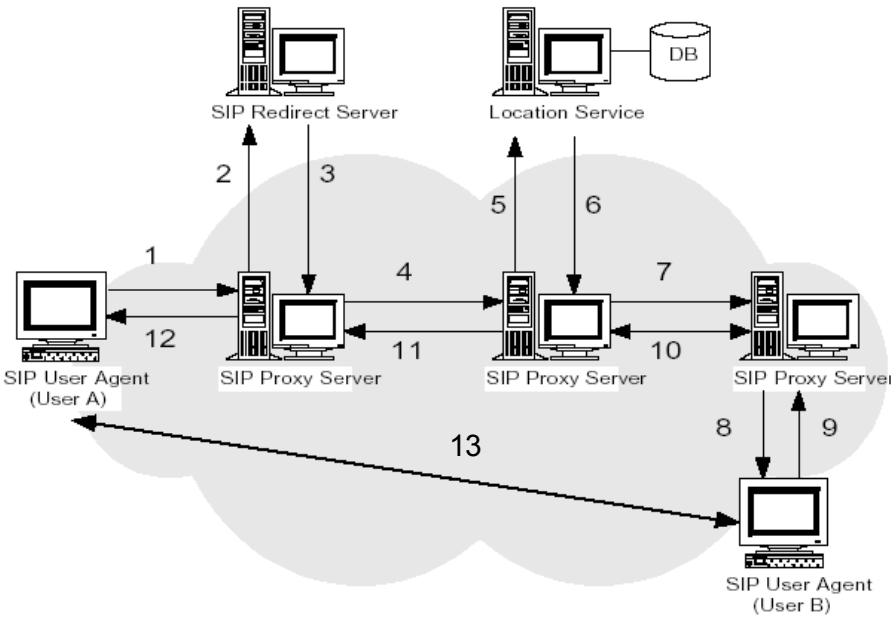


Figure 2.2: SIP architecture mainly consists of SIP user agent, SIP network server (proxy or redirect server), and location service.

The procedures of establishing a SIP session shown in Figure 2.2 are explained below:

1. User A sends INVITE request to SIP proxy server.
2. If the SIP proxy server cannot resolve the called party's location, it forks the request to the redirect server.
3. Redirect server returns the address of the next proxy server.
4. Based on the returned address from the redirect server, the proxy server forks the request to the next SIP proxy server.
5. The proxy server consults the location server for the address of the next hop.
6. The location server returns the address of the next hop.
7. The proxy server forks the request to the next hop.
8. This time, the proxy knows the location of the called party and forks the invitation request to User B.

9-12. The acknowledgment message from User B will be returned to User A using the same path as the request message.

13. Once the call setup procedure is completed, the session will be directly established between User A and B. For IP network, the voice/audio stream can be carried over IP using RTP.

Another entity, that comprises SIP network, is SIP Registrar shown in Figure 2.3. SIP registrar allows mobility support within the SIP network. When a SIP user agent moves to a new location, it will register its new location with the Registrar, which in turns updates the location database via Location Server. Upon consulting the Location Server, the network server will then know how to route the new incoming calls to the new location. SIP Network Server and SIP Registrar are usually implemented on the same machine.

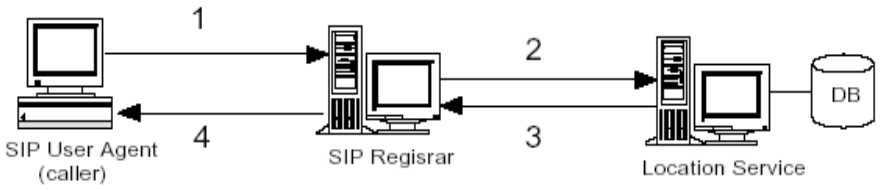


Figure 2.3: SIP registrar to keep track of SIP user agent current location.

The procedures of registering a SIP user agent to the SIP registrar shown in Figure 2.3 are explained below:

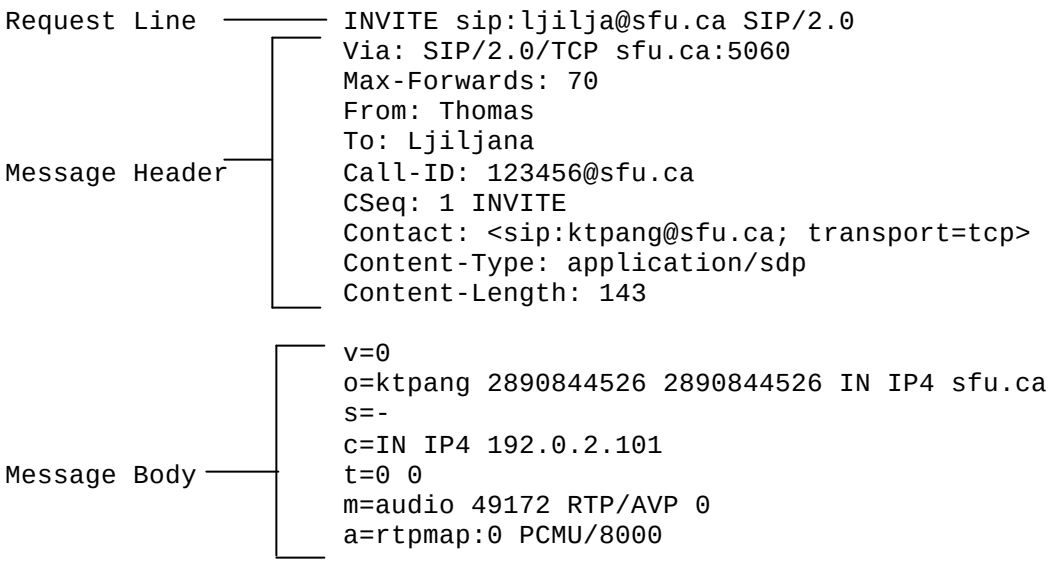
1. The user agent sends a registration message to the SIP Registrar
2. The Registrar stores the registration information in a location service.
3, 4. Once the information is stored, the Registrar sends the appropriate response back to the user agent.

2.3 SIP Messages

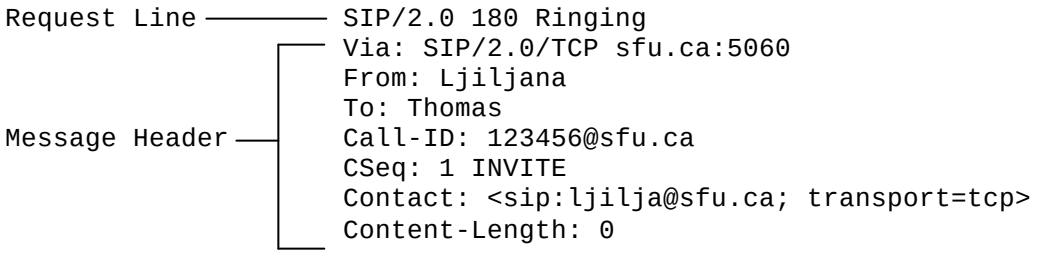
A SIP message is either a request from a client to a server, or a response from a server to a client. A SIP request message begins with a request line, followed by header fields, and an optional message body. Similarly, a SIP response message begins with a status line, followed by header fields, and an optional message body. The request line and header field define the nature of the call in terms of services, addresses, and protocol features. The message body is independent of the SIP protocol and can have an arbitrary content. For multimedia application, the message body usually contains Session Description Protocol (SDP) that defines the session information. Below are sample SIP

request and response messages. The descriptions of each field in the SIP messages are defined in SIP specifications [11].

Sample SIP Request Message:



Sample SIP Response Message:



In this example, there is no message body.

2.4 Sample Message Flow

This section gives examples of SIP message flows. When a user agent starts up or moves to a new location, it must register its location with the SIP Proxy Server before accepting any incoming calls. Figure 2.4 depicts a sample SIP registration call flow. Both UA 1 and UA 2 register their location with the Proxy Server. Upon successful registration, the Proxy Server sends the success response “401 OK” to the user agents.

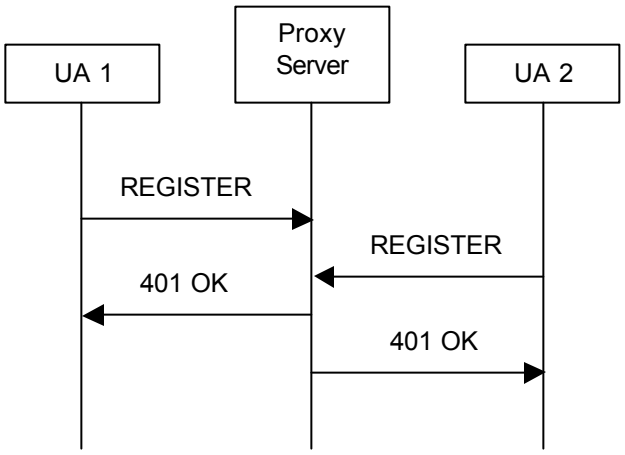


Figure 2.4: SIP registration message flow diagram.

If UA 1 knows the IP address of UA 2, it can send the INVITE directly to UA 2. In reality, this may not be the case because IP addresses are often dynamically assigned due to the shortage of IP version 4 (IPv4) addresses. Furthermore, the peer may move to different locations from time to time. Therefore, a proxy server is required to route the requests.

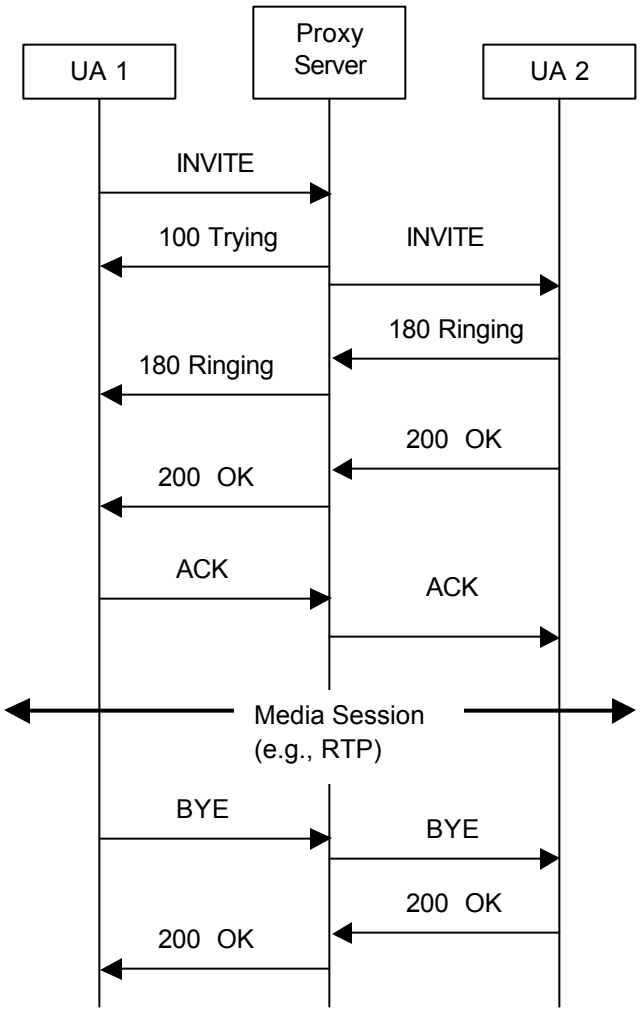


Figure 2.5: Basic call setup and teardown message flow diagram.

Figure 2.5 depicts a sample SIP call establishment and termination via SIP proxy server. UA 1 does not know exactly where UA 2 is currently logged in, and therefore sends the INVITE method with UA 2's SIP URL to the SIP Proxy Server. The Proxy Server sends TRYING response to UA 1 as it takes time to perform a name resolution of UA 2's SIP URL. After obtaining the IP address (current location) of UA 2, the Proxy Server route the INVITE request to UA 2. UA 2 returns RINGING response to indicate that the INVITE has been received

and that alerting has taken place. The Proxy Server routes the RINGING response to UA 1, and UA 1 should play the ringing tone. UA 2 sends OK to the Proxy Server to accept the call. UA 1 acknowledges the final responses to the INVITE request by sending ACK to UA 2 via the proxy server. UA 1 and UA 2 can now start the media session, for example, sending and receiving RTP streams.

To terminate the call, UA 1 sends BYE to UA 2 via proxy server. UA 2 responds with OK, and the media session will be closed. To distinguish the second OK response (for BYE request) from the first OK response (for INVITE request), the BYE and the INVITE requests have different command sequence number (CSeq), but they have the same Call-ID because they belong to the same call session.

3 Stream Control Transmission Protocol

This Chapter describes the basic concepts of the Stream Control Transmission Protocol (SCTP) and summarizes other literature survey on SCTP performance for message-oriented protocol such as Session Initiation Protocol (SIP). The key finding is that SCTP is a better transport for message-oriented protocol, compared to TCP and UDP. This especially holds under the lossy and multiple-session environment. This finding was the reason for understanding this project.

3.1 SCTP Overview

Stream Control Transmission Protocol (SCTP) was originally designed by the Signaling Transport (SIGTRAN) group of IETF for Signaling System 7 (SS7) transport over IP-based networks. It is a reliable transport protocol operating on top of unreliable connectionless service, such as IP. It provides acknowledged, error-free, non-duplicated transfer of messages through the use of checksums, sequence numbers, and selective retransmission mechanism. SCTP supports multiple streams known as multi-streaming within an association, and hosts with multiple network addresses known as multi-homing. Both multi-streaming and multi-homing features will be explained in Section 3.3 and Section 3.4 respectively. The first application of SCTP is the SS7 signaling gateway bridging a circuit-switched network (PSTN) with a packet-switched network (IP network) as shown in Figure 3.1.

The ISDN User Part (ISUP) is protocol used in the establishment and tear down of voice and data calls over the PSTN. ISUP corresponds to the combination of transport layer, session layer, presentation layer, and application layer in Open System Interconnect (OSI) model. In circuit-switched network, ISUP message is transported over the network layer Message Transfer Part Level 3 (MTP3), link layer MTP2, and, finally, the physical layer MTP1 (T1/E1). At the IP network side, SS7 ISUP is transported over SCTP via MTP3 User Adaptation layer (M3UA), IP network layer, Ethernet link layer, and, finally, the physical layer cable or unshielded twisted pair (UTP).

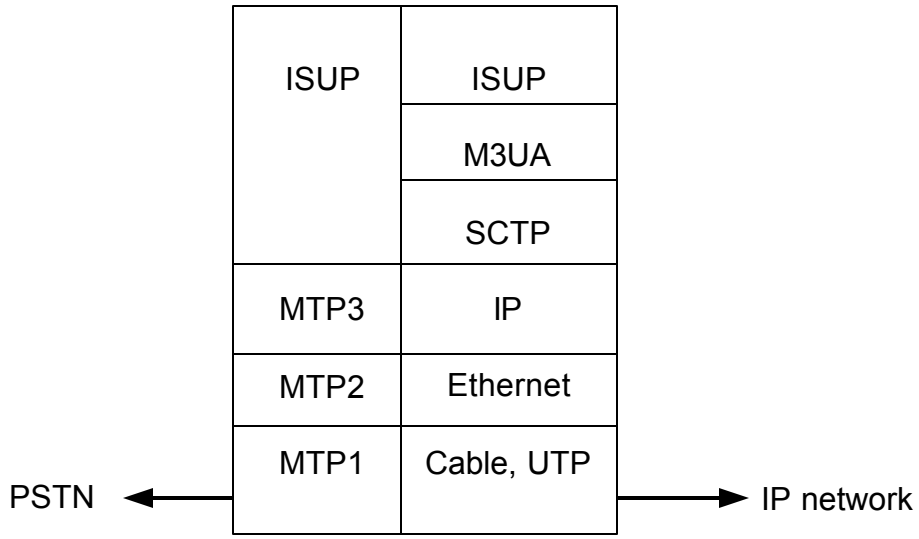


Figure 3.1: SS7 signaling gateway. It acts as a bridge between PSTN and IP network.

3.2 Sample Message Flow

This section gives examples of SCTP message flows. Before peer SCTP users can send data to each other, a connection must be established between two endpoints. This connection is called association in SCTP context. A cookie

mechanism is employed during the initialization to provide protection against security attacks. Figure 3.2 shows a sample SCTP association initialization message flow. User 1 initiates an association by sending User 2 an INIT message. User 2 acknowledges the initiation of an SCTP association by returning User 1 an INIT_ACK message. User 1 then sends User 2 a COOKIE_ECHO message that contains the cookie to be used for subsequent data message passing. User 2 acknowledges the receipt of COOKIE_ECHO message by returning User 1 a COOKIE_ACK message.

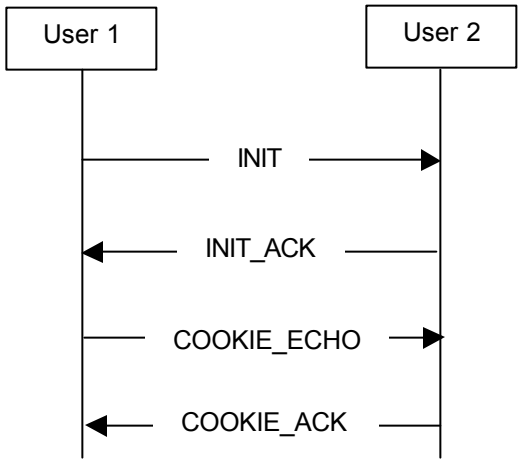


Figure 3.2: SCTP association initialization procedures.

Once an association is successfully established, the SCTP user can send its user data using SCTP DATA message. The peer acknowledges the receipt of the user data by returning SCTP SACK message as shown in Figure 3.3. SCTP monitors the reachability of the peer by periodically sending HEARTBEAT messages. The peer acknowledges the receipt of HEARTBEAT message by returning HEARTBEAT_ACK message as shown in Figure 3.4. A user can shutdown the association by sending SHUTDOWN message and the peer

acknowledges the shutdown request by returning SHUTDOWN_ACK message as shown in Figure 3.5.

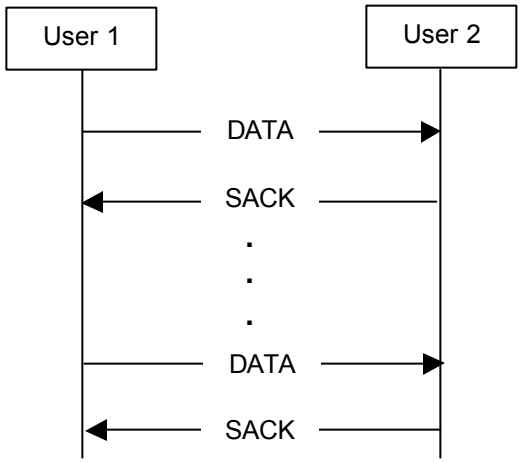


Figure 3.3: SCTP user data passing.

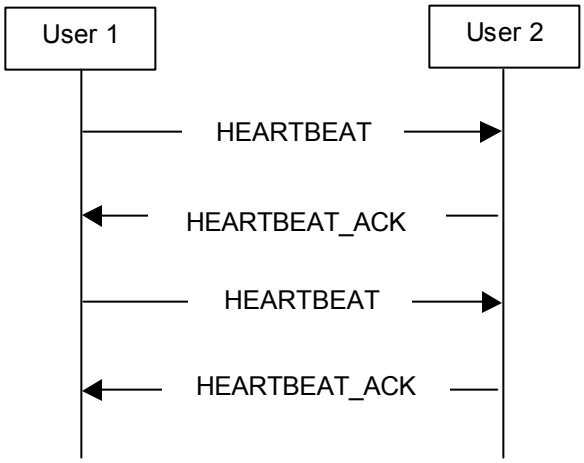


Figure 3.4: SCTP heartbeat mechanism.

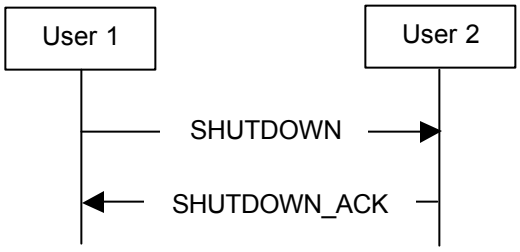


Figure 3.5: SCTP association shutdown procedures.

3.3 Congestion Control

SCTP congestion control mechanism is very similar to TCP. It includes slow start, congestion avoidance, and fast retransmit. The details of SCTP congestion control algorithms are given in [5]. In SCTP, the initial congestion window (cwnd) is set to the double of the maximum transmission unit (MTU). In TCP, it is usually set to one MTU. In SCTP, cwnd increases based on the number of acknowledged bytes, rather than number of acknowledgments in TCP. The larger initial cwnd and the more aggressive cwnd adjustment contribute the larger average congestion window and, hence, better throughput performance of SCTP than TCP [12].

3.4 Multi-streaming

To take advantage of the congestion control mechanisms built in TCP, we can bundle multiple SIP sessions in a single TCP session. However, TCP does not have any mechanism to distinguish which application session a TCP segment is originated from. SCTP allows multiple streams within an association. This multiplexing/de-multiplexing capability is called multi-streaming. Multi-streaming is achieved by introducing a field called stream identifier that is used to differentiate segments in different streams. Figure 3.6 illustrates the multi-streaming feature of SCTP. When the message with TSN=3 arrives to the receiver, the receiver knows that TSN=2 is lost. However, it also knows that TSN=3 is the next packet belonging to stream id=0. Therefore, it delivers the

packet to the application without waiting to receive TSN=2 that belongs to stream id=1. In the case of TCP, the receiver will not deliver TSN=3 until it receives TSN=2. This is known as the head-of-line blocking problem.

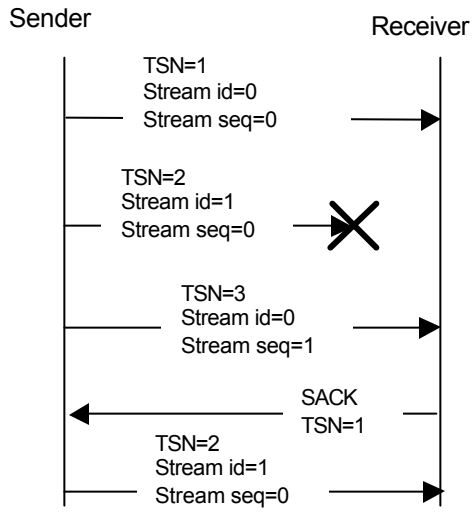


Figure 3.6: SCTP multi-streaming feature. Packet loss in one stream will not affect other streams. This solves the head-of-line blocking problem in TCP.

The Hypertext Transfer Protocol (HTTP) is one of the most widely used protocols on the World Wide Web today. Rajamani et al. [13] compare the performance between SCTP and TCP as the transport protocol for HTTP. Since SIP is based on HTTP, the results presented in [13] provide a good reference of the expected performance comparison of proxy server using SCTP and the one operating over TCP. First, Rajamani et al. [13] performed a single file transfer test with network bandwidth of 10 Mbps under no loss and measured the transfer time for both TCP and SCTP. The result shown in Figure 3.7 indicates that TCP performs better than SCTP.

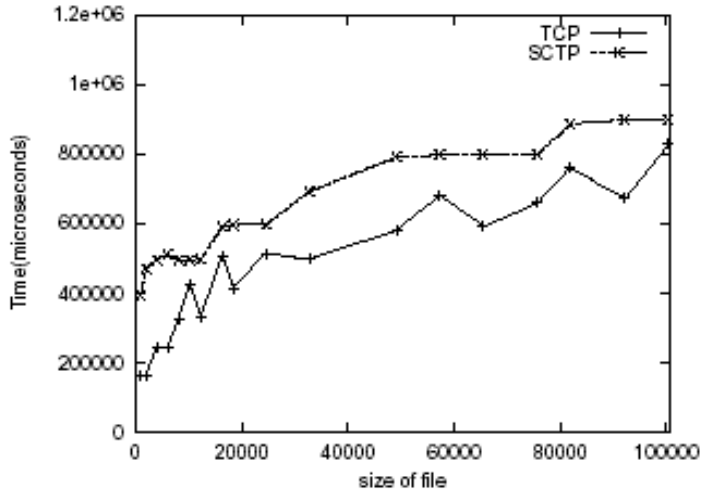


Figure 3.7: TCP and SCTP performance comparison in single-session environment. TCP performs better than SCTP. Source: Rajamani et al., 2002 [13], by permission.

Next, Rajamani et al. [13] performed multiple file transfer test (8 files) with identical network bandwidth of 10 Mbps under loss, and measured the transfer time. The results shown in Figure 3.8 indicate that SCTP performs better than TCP for multi-session application and under lossy condition.

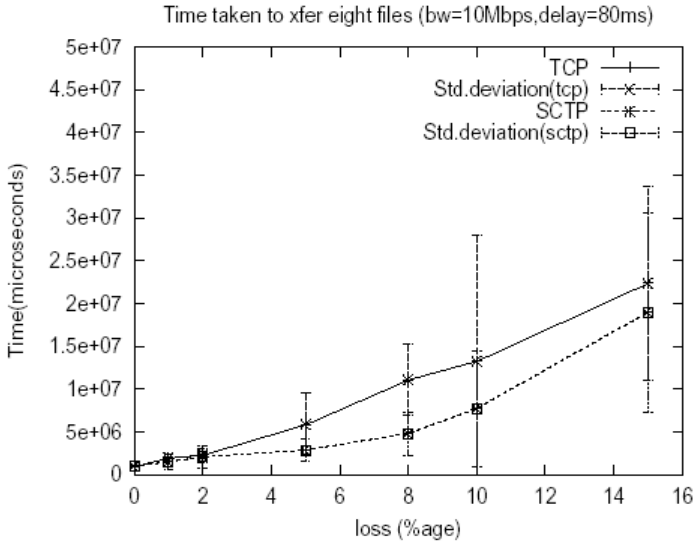


Figure 3.8: SCTP and TCP performance comparison in multi-session environment. SCTP performs better than TCP because of its multi-streaming feature. Source: Rajamani et al., 2002 [13], by permission.

3.5 Multi-homing

SCTP provides several source and destination addresses within an association. This is known as multi-homing. Multi-homing provides alternative paths to be used in case of network failures, similar to the redundancy provided in SS7 network. SCTP detects link failure through its heart-beating mechanism [5]. Figure 3.9 illustrates the SCTP multi-homing in which we have four possibly independent paths in an association between host 1 and host 2. This feature increases the reliability of an association [14].

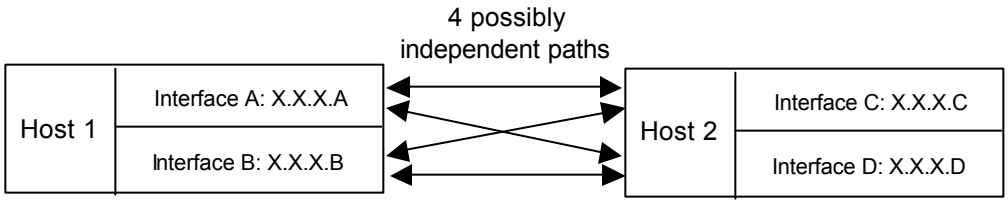


Figure 3.9: SCTP multi-homing. In this setup, we have four possibly independent paths in an association between host 1 and host 2. Source: Brennan et al., 2001 [14], by permission.

To evaluate SCTP multi-homing performance, Brennan et al. [14] compared the single-homed system with the multi-homed system similar to the setup in Figure 3.9 under the packet loss rate of 15%. The Transmission Sequence Number (TSN) of the received packets is monitored. As mentioned in Section 3.3, SCTP also uses slow start mechanism similar to TCP. After approximately 1.5 seconds of transmission, the slow start phase is interrupted by a first lost packet. The retransmitted packet for the first lost packet is also lost.

As a result, the transmission is halted for more than 3 seconds in the case of single-homed system as shown in Figure 3.10.

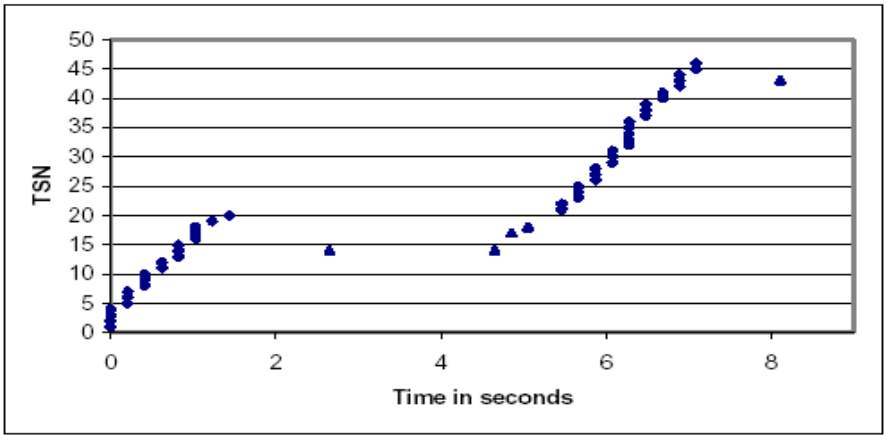


Figure 3.10: Single-homed with packet loss. First packet is dropped at around 1.5 seconds. The retransmitted packet for the first lost packet is also lost. As a result, the transmission halts for more than 3 seconds before recovery. Source: Brennan et al., 2001 [14], by permission.

In multi-homed system, the sender detects the link failure and uses an alternative path for transmission. As a result, the transmission is interrupted for only about 0.5 seconds, as shown in Figure 3.11.

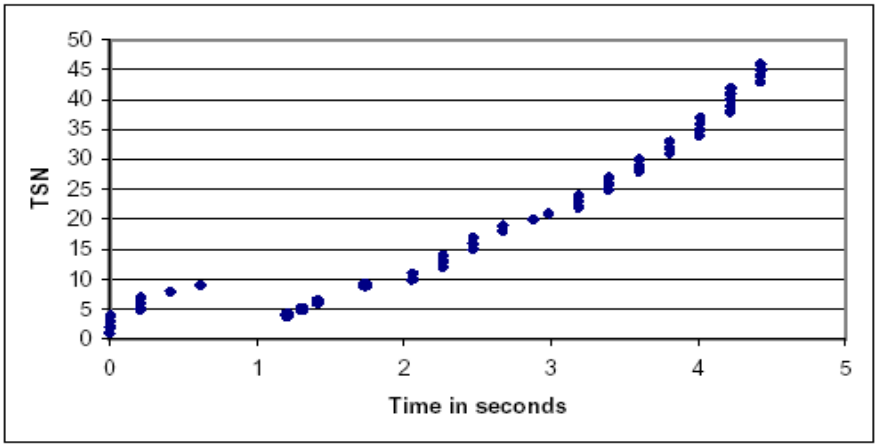


Figure 3.11: Multi-homed with packet loss. It takes about 0.5 seconds to recover. Source: Brennan et al., 2001 [14], by permission.

3.6 SCTP Implementation

SCTP implementations available for public include Siemens SCTP library (SCTPLIB) and Randall's SCTP Kernel (LKSCTP). They are both designed with the initial effort towards RFC conformance and Application Programming Interface (API) development. SCTPLIB and LKSCTP developers will address the performance issues in the future. Both implementations are intended for the Linux operating system. In this project, we use Redhat Linux 7.2 or 7.3 (the corresponding kernel is 2.4.18). At the time when I started working on this project, the latest SCTP implementations for ***Linux kernel 2.4.18*** are ***SCTPLIB-1.0.0-pre18*** and ***LKSCTP-2.4.18_0_4_6***. Therefore, the discussions presented in this report are based on these two releases.

In the Linux operating system, applications can run in two programming spaces: user space or kernel space. The purpose of these two distinct programming spaces is to protect the operating system (OS) from performing certain dangerous operations without being checked by the OS. An example is accessing memory segments that belong to other processes. SCTPLIB is designed to run in user space, while LKSCTP is designed to run in kernel space. In terms of RFC conformance, they are complete prototype implementations of the SCTP protocol as described in [5]. This section describes the high level architecture of these two SCTP implementations and analyzes the pros and cons of running SCTP in different programming spaces. This section also describes the architectural details of each implementation.

3.6.1 Siemens SCTP Library (SCTPLIB)

SCTPLIB is the result of the collaborative effort between Siemens AG and the Computer Networking Technology Group of the University of Essen, Germany. Figure 3.12 describes the high level architecture of SCTPLIB. To make use of the functions provided by the SCTPLIB, the application (SIP application in our case) is required to statically link with SCTPLIB. SIP application can access functions in the SCTPLIB using either Upper Layer Interface (ULP) or Socket Application Programming Interface (API). In SCTPLIB implementation, the socket API is simply a wrapper of ULP. The SCTPLIB opens a raw socket to catch all incoming packets with the IP protocol id byte set to 132 (hex 0x84), indicating SCTP messages. A memory copy takes place every time a packet cross the user-kernel space boundary. For example, SIP application in Machine 1 wants to send a packet to that in Machine 2 using SCTP, it passes the packet to be transmitted to SCTPLIB. SCTPLIB pushes the packet via a raw socket to the IP stack in the kernel. This involves a memory copy from user space to the kernel space. At the receiver side, SCTPLIB collects the packet via a raw socket from the IP stack in the kernel. This involves a memory copy from kernel space to user space. SCTPLIB then passes the packet to the application.

Since all the SCTP messages are processed at the user space, many memory copies are to be expected between the user space and kernel space. This impacts the performance of an application using SCTP as transport. Furthermore, the application is mostly interested in SCTP data messages. It has no interest in control messages such as HEARTBEAT, COOKIE_WAIT,

COOKIE_ECHO, and acknowledgment messages. Therefore, it is inefficient to process SCTP messages in the user space.

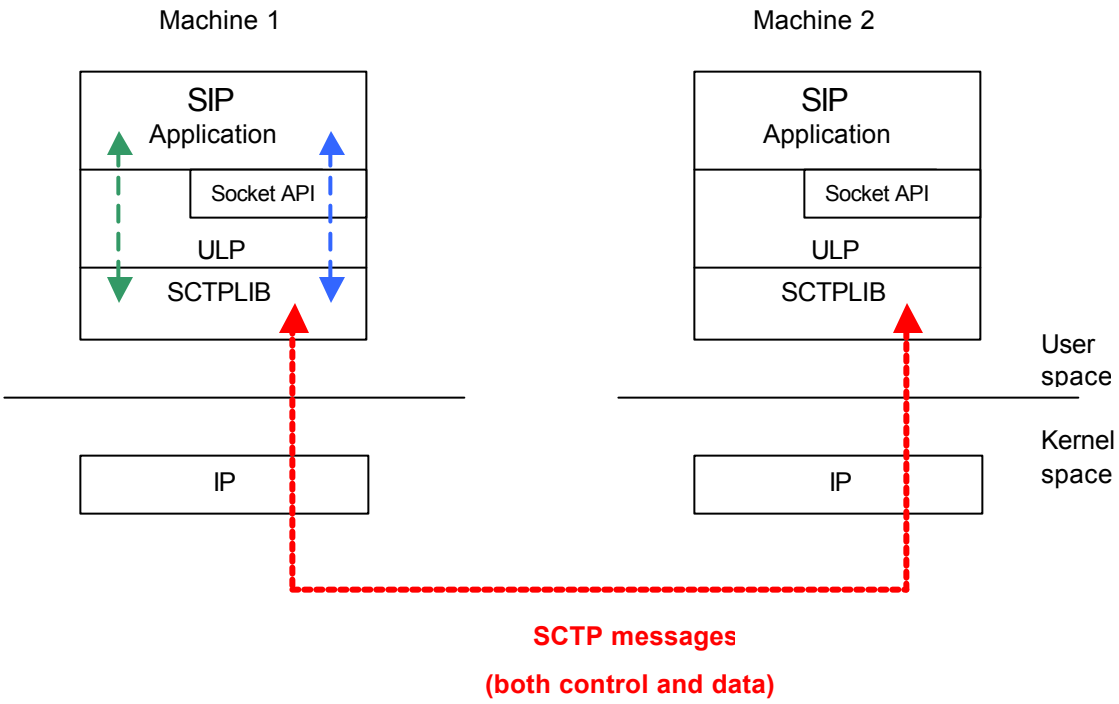


Figure 3.12: SCTPLIB architecture. This architecture allows only one instance of the SCTP library running.

Another drawback of this architecture is that there cannot be more than one instance of the SCTP library running on a host machine. Hence, an intermediary, called SCTP daemon (SCTPD), is necessary to allow multiple processes to use the services provided by the SCTP library, as shown in Figure 3.13.

In the above architecture, the SCTPD and the SIP applications run as separate processes and they communicate through UDP messages. The SIP application needs to register itself with the daemon and provides callback

handlers for SCTP event notifications from the daemon. The SIP application will bind three UDP ports with SCTPD [15]: sctpd_port used by SIP application to send administrative messages (registration messages) and function call messages, notif_port used by SCTPD to send notifications of SCTP event messages, and resp_port used by SCTPD to send a return value (responses) of function calls (requests) sent by the SIP application to the daemon.

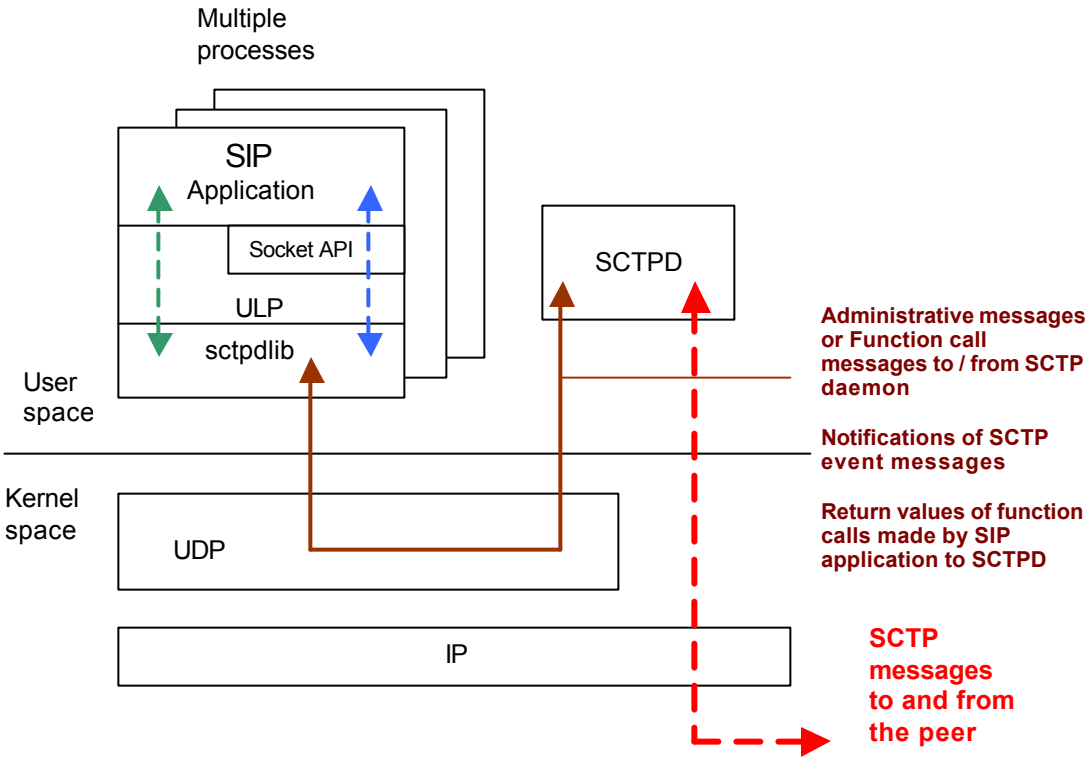


Figure 3.13: SCTP daemon. This allows multiple SIP applications running on the same machine. However, it involves three memory copies between user space and kernel space for each SCTP message to or from the peer.

As with the architecture in Figure 3.12, many memory copies are expected between user space and kernel space for the SCTP messages from and to the IP stack in the kernel. Even worse, the UDP messages between SIP applications and SCTPD create additional memory copies, adding significant processing

overhead, and, hence, adversely affecting the overall performance of the system. For example, when an application process wants to send a SCTP message to the network, this architecture requires three user-kernel memory copies while it requires only one user-kernel memory copy in the architecture depicted in Figure 3.12. SCTPD is no longer supported in sctplib-1.0.0.pre19 or later version because of a significant number of problems reported by the SCTPD users.

3.6.2 Randall's SCTP Kernel (LKSCTP)

As seen in Section 3.5.1, the user level implementation of SCTP has poor performance and lacks scalability. Researchers in the industry, including Motorola, Cisco and Nortel Networks, therefore developed a kernel level implementation of SCTP. Its design is parallel to the existing TCP and UDP stack in the Linux kernel as shown in Figure 3.14.

SIP applications access LKSCTP using either ULP or Socket API. Not all the SCTP messages received by LKSCTP from the network are passed to the application. Since the applications is mostly interested in SCTP data messages, SCTP control messages such as INIT, INIT_ACK, HEARTBEAT, COOKIE_WAIT, and so on will not be passed to the application, unless they are explicitly requested by the application. This can reduce unnecessary memory copies between the user space and the kernel space. Also, this architecture is more flexible than SCTPLIB as it allows multiple processes accessing the SCTP stack without going through an intermediary, like the SCTP daemon. Therefore, applications do not need to call special functions or perform a special setup before communicating with the daemon. With the support of socket-like API

specified in [16], developing applications for SCTP becomes similar to writing applications for TCP or UDP. This allows quick adoption of SCTP by the users who have TCP/UDP socket programming experience. The only disadvantage of LKSCTP is the high maintenance cost because debugging the kernel is more difficult than debugging programs running in the user space.

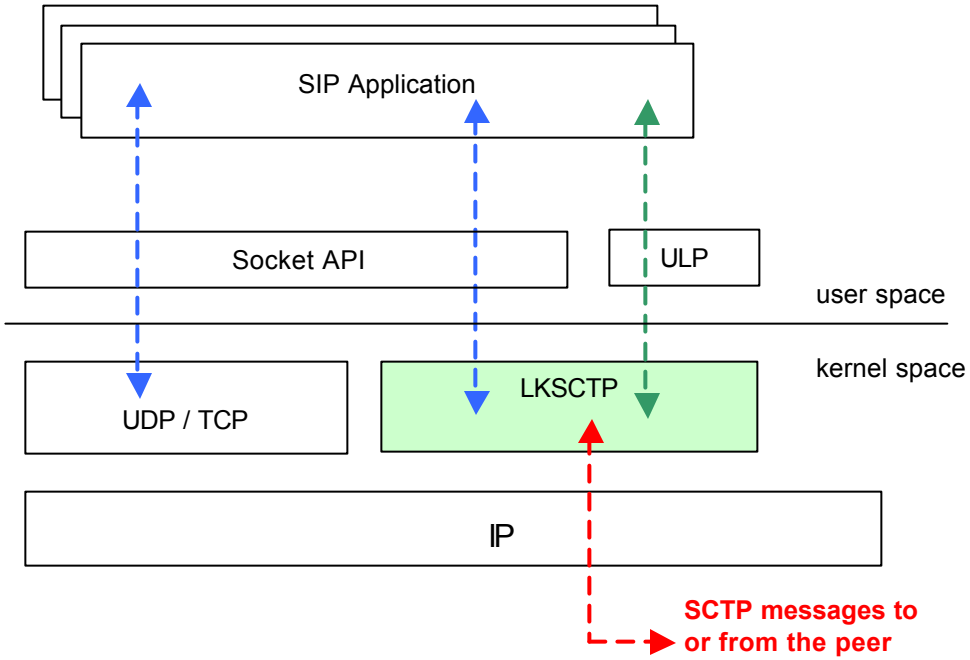


Figure 3.14: LKSCTP Architecture. This architecture allows multiple SCTP instances. Furthermore, only data messages or control messages explicitly requested are passed to the SIP application.

Based on the advantages presented in this section, Randall's SCTP Kernel (LKSCTP) has been chosen for this project. Therefore, the modified SIP proxy server (Partysip) as well as the user agent (Linphone) in Chapter 5 will be tested on LKSCTP.

4 High Level Design of SCTP Plugin

Partysip is a SIP proxy server developed by WellX Telecom, a French company [9]. The company's primary product is the PC based PBX that provides flexible, affordable communications to small and medium sized companies. Readers who are interested in knowing more about the company and its products can visit their website http://www.wellx.com/uk/index.htm. Partysip is written in C and depends on GNU oSIP library [17], an implementation of SIP. Partysip is free software under GNU Public License (GPL) with the following basic functionalities provided through plugins: UDP support, filtering (partially implemented), local registration, local location search, stateful mode, and authentication support. Features such as TCP and SCTP support, remote registration, and instant messaging are not supported yet. The plugins are loadable modules that allow the user to disable functionalities and implement new ones with little independent development. This section presents the high level architecture of Partysip and the design of the SCTP plugin.

4.1 High Level Architecture

Partysip is a multi-threaded application, consisting of a main program and seven threads listed in Table 4.1. When the program starts, the main thread reads the configuration file (partysip.conf) that specifies the plugins to be loaded when the program starts and defines the users configurations (for example, a list of known users with login and password). The plugins are dynamic libraries, e.g., libpsp_udp.so for UDP support. If Partysip is properly built and installed, the

plugins should be located at /usr/local/lib/Partysip. The program flow diagram of the main thread is shown in Figure 4.1. The main thread is also responsible for initializing the oSIP library and invoking the threads defined in Table 4.1.

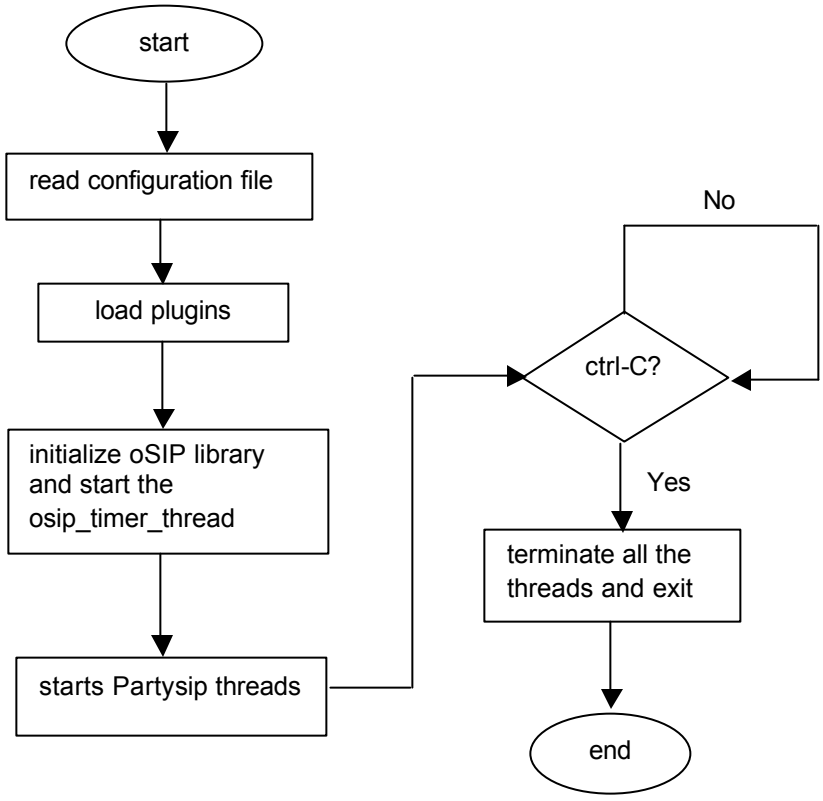


Figure 4.1: Program flow diagram of the main thread in Partysip.

Table 4.1: Threads in Partysip.

Thread	Functions
osip_timer_thread	Governs the retransmission timeout of SIP requests and responses
tlp	Listens for the SIP messages
imp	Processes incoming SIP messages
uap	Provides redirect server support (not used in this project)
slp	Provides stateless proxy server support (not used in this project)
sfp	Provides stateful proxy server support
Resolv	Address resolution: SIP URL interpretation, location lookup, domain mapping

The threads communicate via Linux communication mechanisms, pipes and shared memories (e.g., FIFO queue). Figure 5.2 shows the interactions between the TLP (transport layer) thread and the IMP (incoming message processing) threads. The IMP thread has two FIFO queues for message passing with the TLP thread: one for SIP new request and another for SIP response. When TLP receives a SIP message, it will determine if the message is a new SIP request or an acknowledgment to the previous request. TLP places the request message to SIP_TRAFFIC_FIFO and acknowledgement message to SIP_ACK_FIFO. TLP then signals IMP of the incoming message(s) using *pipe*. IMP retrieves the message from the FIFO queue and passes it to other thread(s) for further processing. Depending on the configuration file, the Partysip may be configured as a redirect server, stateless or stateful proxy server, and IMP will pass the message to UAP, SLP or SFP, respectively. The communication

mechanism between IMP and other threads is identical to that between IMP and TLP.

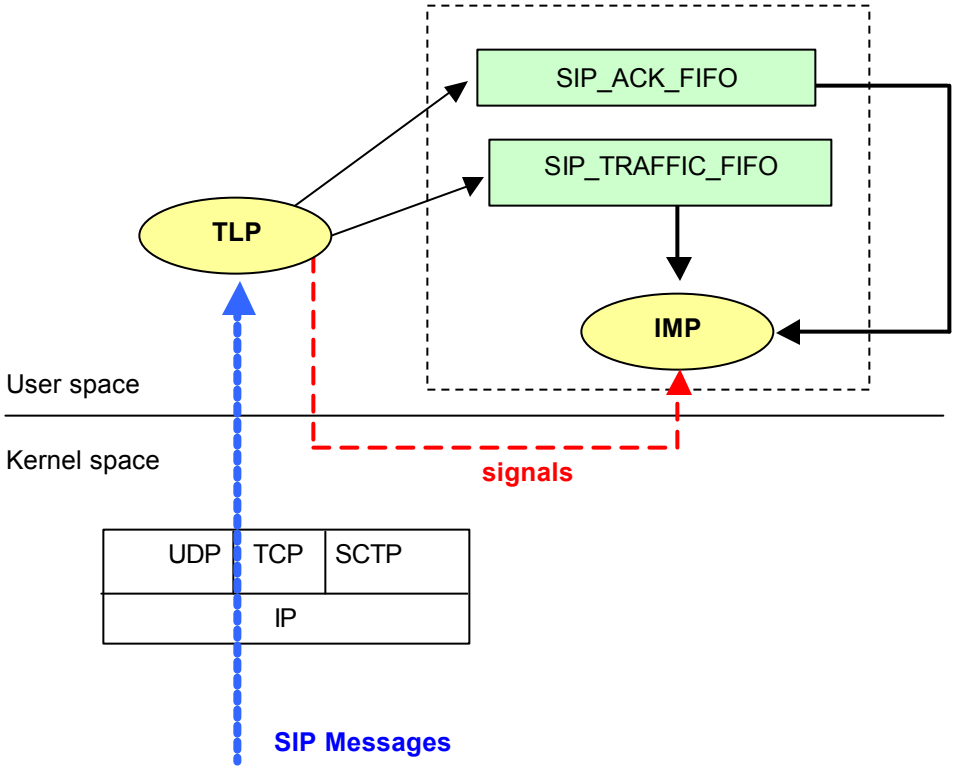


Figure 4.2: Interactions between TLP and IMP. SIP requests and responses received by TLP are respectively placed in SIP_TRAFFIC_FIFO and SIP_ACK_FIFO queues. TLP then signals IMP about the incoming SIP messages for processing.

4.2 SCTP Plugin

As mentioned in Section 3.6.2, LKSCTP comes with two interfaces, ULP and a socket-like API, for applications to communicate with the SCTP in the kernel. With the socket-like API, I reused the UDP plugin as a design reference for the SCTP plugin. Like UDP plugin, the SCTP plugin should provide necessary library functions to the TLP and IMP threads. The functions required by the SCTP plugin are listed in Table 4.2.

Table 4.2: SCTP functional requirements.

Function	Purposes
Common Routines required by plugin	
plugin_init	Read the configuration file. Load the SCTP plugin library. Prepare SCTP endpoints. Install all the call back functions.
plugin_start	Do nothing.
plugin_release	Clean up SCTP endpoints.
Plugin specific	
local_ctx_init	Create socket(s) for SCTP endpoint(s). Specify which local address the SCTP endpoint should associate itself with. Mark the created socket(s) as being able to accept new associations. This function is invoked by plugin_init.
local_ctx_free	Close the socket(s) created by local_ctx_init. This will close the established SCTP association(s) by sending SHUTDOWN message(s) to the peer(s). This function is invoked by plugin_release.
cb_rcv_sctp_message	Receive SCTP message.

Function	Purposes
	Distinguish SCTP DATA message from CONTROL message. Process SCTP DATA message by calling sctp_process_message. This callback function is installed by local_ctx_init and used by TLP thread.
cb_snd_sctp_message	Send SCTP DATA message to the specified peer. This callback function is installed by local_ctx_init and used by SFP thread.
sctp_process_message	Parse SIP message in the SCTP DATA chunk by calling osip_parse (in oSIP library). Log the event returned oSIP by calling sctp_log_event. Add the message to IMP thread's FIFO queue.
sctp_log_event	Log SIP events to the console if the program is running in the debug mode. (It currently does nothing!)

To support SCTP plugin, a new field called ***transport*** is added to the configuration file and the TLP thread is updated accordingly to interpret this new field. To use SCTP, user has to specify the following line in the configuration: ***transport = "sctp"***. If the transport field is not specified, it will be defaulted to UDP. Modifications made to Partysip are given in Appendix C.

4.2.1 Data Structures

The only data structure newly defined is local_ctx_t. It defines the input and output port of the proxy server.

```
typedef struct local_ctx_t {
        int in_port;        /* Incoming port number */
        int in_socket;      /* Socket file descriptor of the incoming port */
        int out_port;       /* Outgoing port number */
        int out_socket;     /* Socket file descriptor of the outgoing port */
```

```
} local_ctx_t;
```

Two oSIP data structures named transaction_t and sipevent_t (defined in /usr/local/include/osip/osip.h) are used by the SCTP plugin.

Structure name: transaction_t
Purpose: This structure stores the SIP transaction.

```
struct transaction_t
{
        void *your_instance;          /* add whatever you want here.      */
        int transactionid;            /* simple id used to identify the tr.  */
        fifo_t *transactionff;        /* events must be added in this fifo */

        via_t *topvia;                /* CALL-LEG definition */
        from_t *from;                 /* CALL-LEG definition */
        to_t *to;
        call_id_t *callid;
        cseq_t *cseq;

        sip_t *orig_request;          /* last request sent */
        sip_t *last_response;         /* last response received */
        sip_t *ack;                   /* ack request sent */

        state_t state;                /* state of transaction */

        time_t birth_time;            /* birth_date of transaction */
        ime_t completed_time;         /* end date of transaction */

        /* RESPONSE are received on this socket */
        int in_socket;

        /* REQUESTS are sent on this socket */
        int out_socket;

        void *config;                 /* transaction is managed by config  */

        context_type_t ctx_type;
        ict_t *ict_context;
        ist_t *ist_context;
        nict_t *nict_context;
        nist_t *nist_context;
};
```

Structure name: sipevent_t
Purpose: This structure stores the SIP event type of a transaction.

```
struct sipevent_t
{
        type_t type;            /* SIP event type, e.g., INVITE request,  */
                                /* incoming 2XX response, etc.            */
        int transactionid;      /* SIP transaction identifier */
        sip_t *sip;             /* SIP request or response message */
};
```

4.2.2 Function Prototypes

This section describes all the functions defined in the SCTP plugin module.

Function	int plugin_init(void)
Purpose	Function invoked by the Partysip core program to initialize and configure the SCTP plugin.
Input parameters	None
Output parameters	None
Return	0 The SCTP plugin is successfully initialized -1 Error

Function	int plugin_start()
Purpose	Function invoked by the Partysip core program to start the SCTP plugin.
Input parameters	None
Output parameters	None
Return	0 The SCTP plugin is started. -1 Failed to start the SCTP plugin.

Function	int plugin_release (void)
Purpose	Function invoked by the Partysip core function to unload the

	SCTP plugin module
Input parameters	None
Output parameters	None
Return	0 The SCTP plugin is successfully removed -1 Error

Function	int local_ctx_init (int in_port, int out_put)
Purpose	Create and configure socket for SIP message passing.
Input parameters	in_port - incoming port number out-port - outgoing port number
Output parameters	None
Return	0 socket successfully created and configured -1 fail to create the socket

Function	void local_ctx_free (void)
Purpose	Close the socket(s) created by local_ctx_init.
Input parameters	None
Output parameters	None
Return	None

Function	int cb_rcv_sctp_message (int max)
Purpose	Non-blocking function used by the TLP thread to receive SCTP messages.
Input parameters	max - maximum number of SCTP messages that the function can parse without returning
Output parameters	None
Return	0 no message available 1 maximum number of SCTP messages being analyzed -1 reached Error

Function	int snd_sctp_message (const transaction_t *transaction,

	sip_t *message, char *host, int port, int socket)
Purpose	Non-blocking function used by the TLP thread to build and send the SCTP messages
Input parameters	transaction SIP transaction information message SIP message to be sent host Host name of the destination port Destination port socket Socket of the outgoing port
Output parameters	None
Return	0 Message successfully sent -1 Fail to send

Function	int sctp_process_message (char *buf)
Purpose	Callback function for the IMP thread to process the incoming message
Input parameters	buf Received message to be processed by the SIP parser
Output parameters	None
Return	0 Success -1 Error

Function	int sctp_log_event (sip_event_t *evt)
Purpose	Log SIP events to the console if the program is running in the debug mode
Input parameters	evt SIP event to be logged
Output parameters	None
Return	0 Success -1 Error

5 Experimental Setup and Results

This Chapter describes six experimental scenarios that were investigated in this project. The underlying transport mechanism can affect the performance of the SIP proxy server. We, therefore, first compare the throughput performance among different transport protocols including TCP, UDP, SCTPLIB, and LKSCTP, under no loss condition. Next, we investigate the performance of the SIP proxy server under single session environment by measuring the transaction response times for SIP registration, SIP call setup, and SIP call termination. We then measure the performance of the modified SIP proxy server under multi-session environment. Finally, we test the multi-homing feature.

5.1 Software configuration

This section describes the software configuration required for our experiments, the installation procedures of each software component, and all special operation procedures.

5.1.1 SCTP Kernel

The SCTP implementation that we use for this project is Randall Stewart's SCTP kernel, version 2.4.18-0.4.6 [18]. The kernel is delivered in a compressed ("g-zipped") archive (i.e., "tar" file). The following commands can be used to retrieve the software from the compressed archive, change directory, and view the file named README for the compilation and installation procedures:

```
tar -zxvf lksctp-2_4_18-0_4_6.tgz
cd lksctp-2_4_18-0_4_6
vi README
```

After the successful build, the kernel image *bzImage* should be located under lksctp-2_4_18-0_4_6/linux_sctp/arch/i386/boot. To avoid confusion with the original Linux kernel, the image should be renamed bzImage.sctp and copied to /boot. Linux allows the user to select kernel to load during the system start-up. This is achieved by modifying the boot loader script */etc/lilo.conf*. The sample boot loader script, which allows the user to select either the original Linux kernel or SCTP kernel, named *lilo.conf* is*:*

```
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
message=/boot/message
compact
root=/dev/hda5
read-only
image=/boot/vmlinuz-2.4.7-10
      label=linux
image=/boot/bzImage.sctp
        label=linux(SCTP)
```

The following command should be used so that the new boot loader script becomes effective when the system boots up next time:

```
/sbin/lilo -v
```

After the system successfully starts up, the command “uname -a” should be used to verify that SCTP kernel is properly loaded. User should see the kernel version of *2.4.18lksctp*.

5.1.2 SIP Proxy Server

As mentioned in Chapter 4, the SIP proxy server that we use in this project is called *Partysip*, developed by WellX Telecom. *Partysip* can be downloaded from [11] and is usually in the format of compressed archive. The following commands can be used to retrieve the software from the compressed archive, change directory, and view the file named INSTALL for building and installation procedures:

```
tar -zxvf Partysip-0.5.0.tar.gz
cd Partysip-0.5.0
vi INSTALL
```

The source code and the installation procedure of the SCTP plugin are given in Appendix C. After installing both *Partysip* and the *sctp plugin*, the user can run the *Partysip* program using the default configuration file, i.e., UDP transport will be used. To use SCTP, please modify the following line in the configuration file (i.e., ~/Partysip-0.5.0/conf/Partysip.conf):

```
transport="sctp"
```

In order to access the SCTP kernel, *Partysip* must be running in the privileged user mode:

```
su
passwd:  <root passwd>
Partysip –f <Partysip configuration file> -d 6 –i
```

The program is terminated by issuing command ‘q’ or ‘ctrl-c’.

5.1.3 SIP User Agent

The SIP user agent program that we use for testing our modified SIP proxy server is called *linphone* [7]. I have modified *linphone* version 0.9.1 to support SCTP. The changes are documented in Appendix D. In order to access the SCTP kernel, *linphone* must be running in the privileged user mode:

```
su
passwd:  <root passwd>
linphone &
```

The '&' sign will put the program into background and, therefore, the shell where the user invokes the program will not be frozen. User should now see the linphone graphical user interface. Go to the sub-menu connection -> parameters -> SIP and fill out the following information:

- SIP user agent port: 5060
- Assign SIP address, e.g., tkpang@10.241.244.202
- Turn on "Use SIP registrar'
- Enter the IP address of the redirect server to be connected to
- Select Redirect server

Now, the SIP user agent program can communicate with the peer via the designated SIP proxy server.

5.1.4 Ethereal

Ethereal is a free network protocol analyzer for examining data from a live network [19]. In this project, *Ethereal* (version 0.9.3) was installed. It ran on the host machine of the SIP proxy server to monitor all SCTP messages received or

transmitted by the SIP proxy server. To filter out SCTP messages from the network traffic, I set the filter field in the capture options menu of the *ethereal* network analyzer to “proto 132”. IP protocol numbers can be found in */usr/include/linux/in.h* and the protocol number for SCTP is 132.

5.1.5 NIST Net

NIST Net is a network emulation package developed by National Institute of Standards and Technology (NIST) [20]. It allows a single Linux box set up as a router to emulate a wide variety of network conditions such as packet loss, duplication, delay and jitter, bandwidth limitations, and network congestion. In this project, we use NIST Net to emulate the packet loss condition for our multi-streaming performance testing in Section 5.4.

NIST Net comes with a command-line interface called *cnistnet* and graphical user interface called *xnistnet*. Using xnistnet is quite self-explanatory and therefore will not be explained here. Table 5.1 lists most commonly used cnistnet commands in this project.

Table 5.1: Commonly used cnistnet commands.

Command	Purpose
cnistnet –u	Turn on the emulation
cnistnet –d	Turn off the emulation
cnistnet –a <src> <dest> --drop <percentage>	Add the emulation entry Example: cnistnet –a 10.241.245.58 10.241.244.202 --drop 10 This emulates a drop rate of 10 percentage for IP packets originated from 10.241.245.58 and destined to 10.241.244.202.
cnistnet –d <src> <dest>	Remove the emulation entry Example: cnistnet –d 10.241.245.58 10.241.244.202

5.1.6 TTCP

Test TCP (TTCP) is a command-line sockets-based benchmarking tool for measuring TCP and UDP performance between two systems [21]. A version called WSTTCP for Windows operating system is also available.

5.2 Comparison of Transport Protocol Throughput

In this test, the throughput performance of different transport protocols on different packet size under no loss condition is measured. TTCP is used for measuring TCP and UDP performance between two Linux systems. To measure the SCTP performance, I developed a similar test program. The source code can be found in Appendix F. Figure 5.1 depicts the experimental setup for this test.

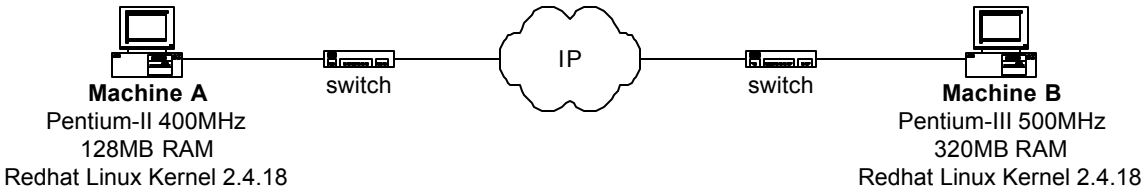


Figure 5.1: Experimental setup for throughput test.

Machine A is running as a client, while Machine B is running as a server. Table 5.2 describes the procedures of running TTCP and the SCTP test program on each machine.

Table 5.2: Test program execution procedures.

	Machine A	Machine B
TCP Test	*ttcp -t -s -n <num> -D -fm <peer ip> -l <pktsz>* *options:* *-t transmitter* *-s sink* *-n number of packets* *-D TCP_NODELAY* *-fm peer IP address* *-l packet size* For example: *ttcp –t –s –n 10000 –D –fm 10.241.244.202 –l 1024* Machine A will send 10000 TCP packets of 1024 bytes each, to 10.241.244.202.	*ttcp -r -s -l <pktsz>* *options:* *-r receiver* *-s sink* *-l packet size* For example: *ttcp –r –s –l 1024* Machine B is expecting to receive TCP packets, with each packet of 1024 bytes.
UDP Test	*ttcp -t -s -n <num> -fm <peer ip> -l <pktsz> -u* For example: *ttcp –t –s –n 20000 –fm*	*ttcp -r -s -u -l <pktsz>* For example: *ttcp –r –s –u –l 512*

	Machine A	Machine B
	10.241.244.202 –l 512 –u The additional option "-u" specify that it is UDP test. Machine A will send 20000 UDP packets of 512 bytes each to 10.241.244.202.	The additional option "-u" specify that it is UDP test. Machine B is expecting UDP packets of 512 bytes each.
SCTP Test	*client <local port> <remote port> <peer ip> <num> <pktsz>* For example: *client* 9000 9001 10.241.244.202 25000 256 Machine A will send 25000 SCTP data messages with payload size of 256 bytes each from the local port 9000 to the remote port 9001 at 10.241.244.202.	*server <local port> <num> <pktsz>* For example: *server* 9001 25000 256 Machine A is expecting 25000 SCTP data messages with payload size of 256 bytes each.

<num> is number of packets to be transmitted or received, <pktsz> is the packet size in bytes, <peer ip> is the IP address of the remote machine (Machine B), <local port> is the local SCTP endpoint, and <remote port> is the remote SCTP endpoint.

The performance results shown in Figure 5.2 indicate that UDP has the best throughput performance because it has the least framing overhead. Unlike TCP and SCTP, UDP is connection-less and does not have any mechanism to detect if a packet is successfully transmitted or not. UDP counts on the application to provide the detection and recovery mechanism for the lost packets, duplicate packets, and packets out of sequence. UDP does not provide any

congestion and flow control mechanisms. Therefore, UDP has much simpler framing structure and, hence, less framing overhead than TCP and SCTP. Figure 5.3 depicts the framing structures of UDP, TCP and SCTP. The detailed description of each field in the frames is given in RFC 768 [22], RFC 793 [23], and RFC 2960 [5].

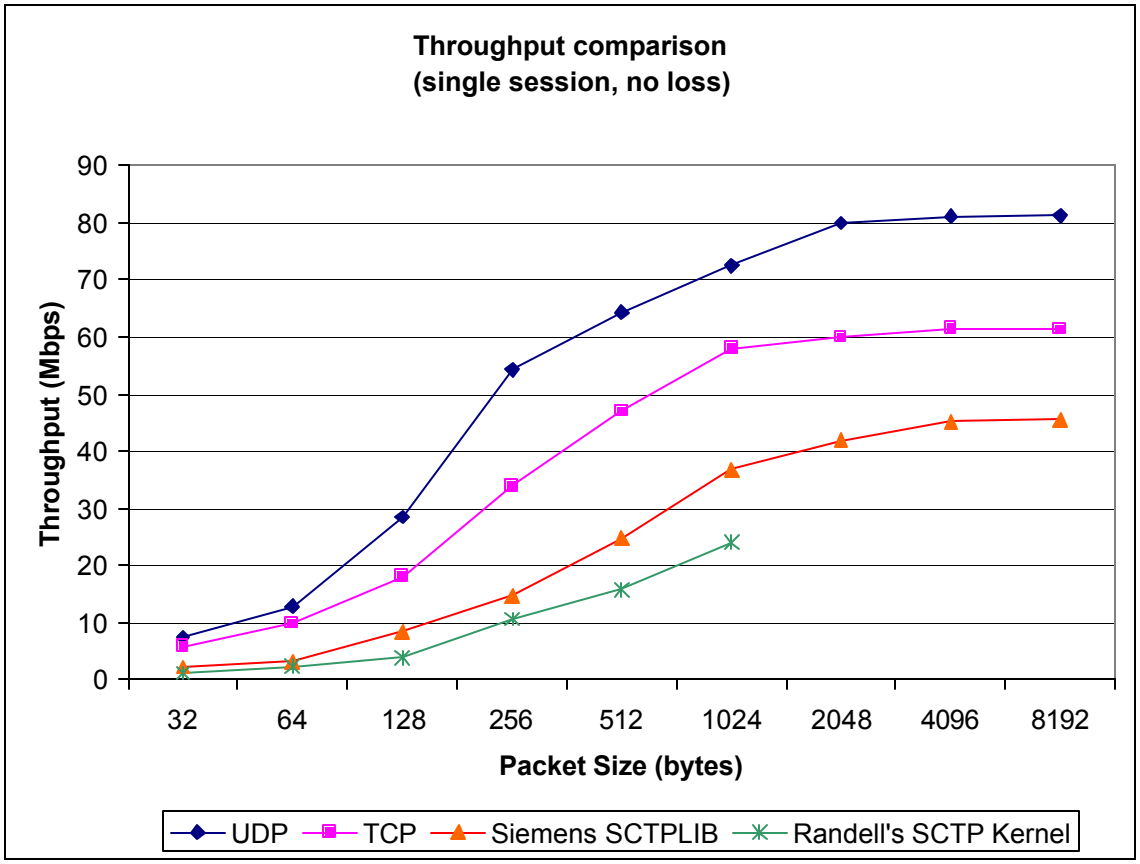
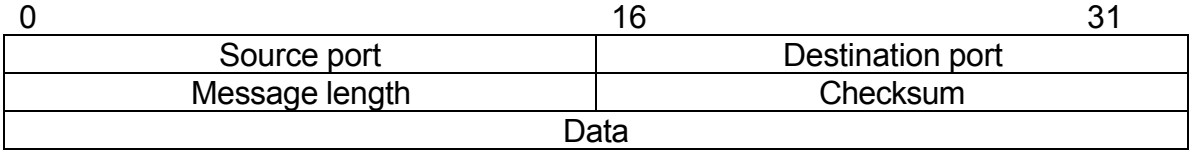
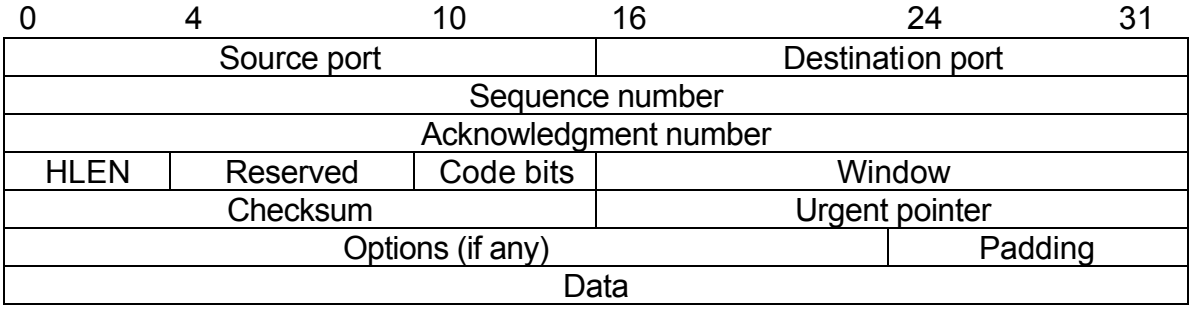


Figure 5.2: Throughput comparison among different transport protocols. The throughput performance is measured in terms of Mbps against various packet size in bytes. The results show that UDP has the best throughput performance followed by TCP, Siemens SCTP, and Randall's SCTP kernel.

UDP Frame



TCP Frame



SCTP Frame

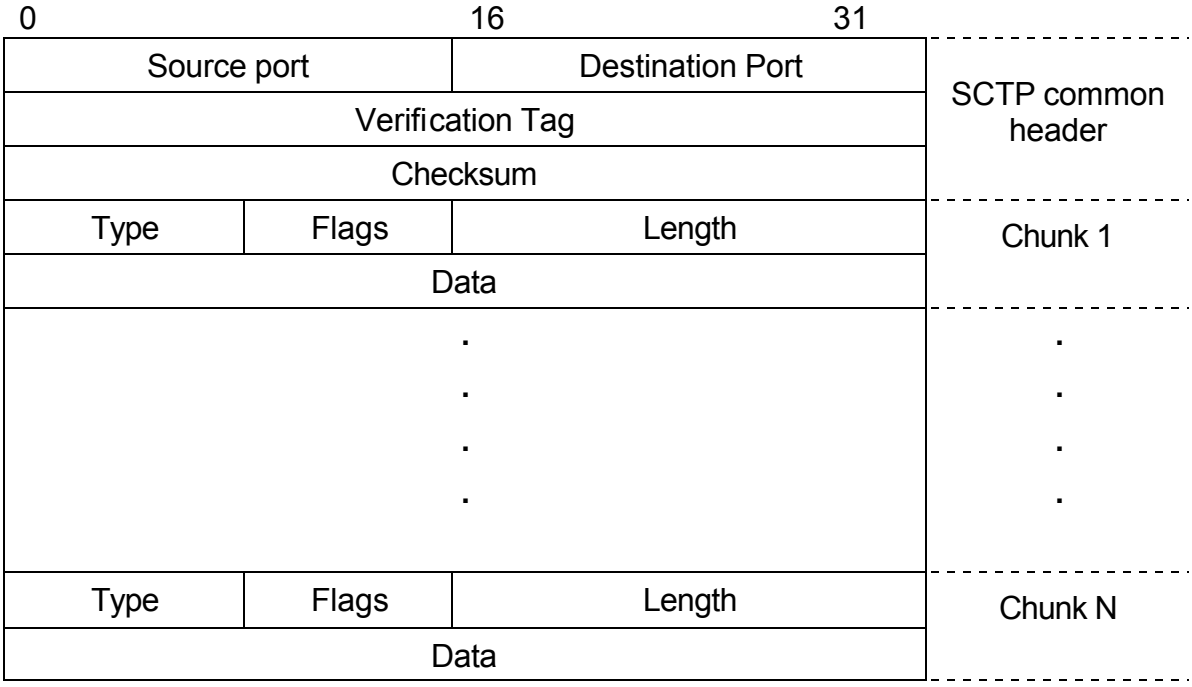


Figure 5.3: Framing structures of UDP, TCP, and SCTP.

Other observations in Figure 5.2 worth mentioned are:

- As discussed in Section 3.4, SCTP's performance should be slightly worse than TCP in single session mode. However, in our test the performance gap between TCP and SCTP is quite large. The performance discrepancy is caused by the non-optimized SCTP implementations, where the initial efforts are towards RFC conformance and API development, rather than performance.

- Fragmentation is not working in the SCTP kernel version that I used. As a result, no performance data are collected for packet size greater than 1024 bytes. Since the average size of SIP messages is about 500 bytes, this fragmentation problem will not affect our test.

- SCTPLIB has better performance than LKSCTP. As discussed in Section 3.6, SCTP stack running in kernel space should have better performance than the stack running in user space. The performance discrepancy, again, is caused by the non-optimized SCTP kernel in Randall's implementation.

5.3 Single-session Performance Test

Figure 5.4 depicts the experimental SIP network for three tests: SIP registration, basic call setup, and termination. All machines have SCTP kernel *lksctp-2.4.18_0_4_6* and GNU SIP library *osip-0.9.0* installed. User agents modified from *linphone-0.9.1* for using SCTP are installed in machines UA1 and

UA2. SIP proxy server modified from *Partysip-0.5.0* for using SCTP is installed in the laptop, with IP address 10.241.244.40. UA1 and UA2 are configured to use the default port 5060 for inbound and outbound traffic, while the Proxy Server is configured to use the default setting that is port 5060 for inbound and 5061 for outbound. To monitor the SCTP and SIP messages in the network, *Ethereal* is running on the host of the Proxy Server.

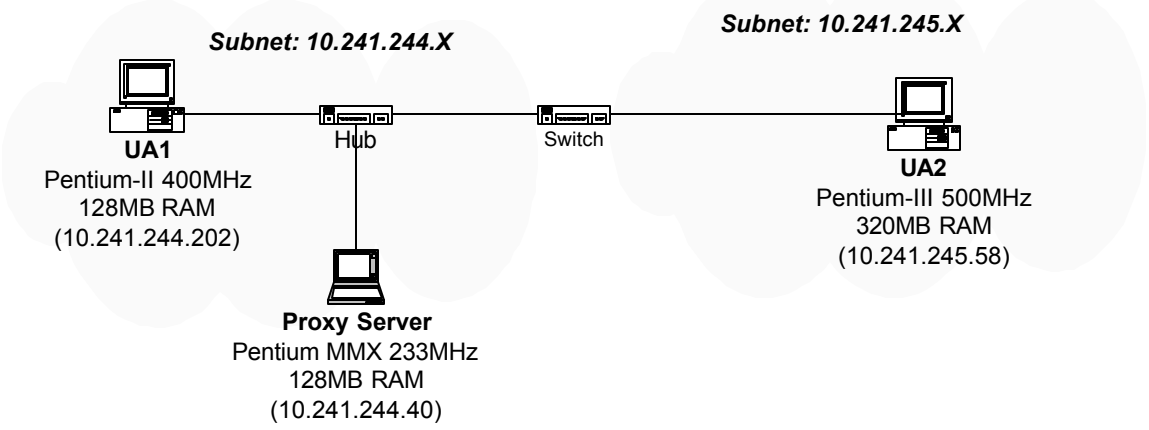


Figure 5.4: Experimental SIP network. UA1 and UA2 are located in different subnets and are connected via a SIP proxy server.

The test results of the SIP registration, basic call setup, and call termination will be presented and discussed in Section 6.3.1, Section 6.3.2, and Section 6.3.3 respectively.

5.3.1 SIP Registration

This test verifies that: (1) SCTP associations are successfully established between UA1 and the Proxy Server, and between UA2 and the Proxy Server, and (2) both UA1 and UA2 are successfully registered to the Proxy Server. The transaction response time was also measured. The registration procedures

captured by Ethereal is shown in Figure 5.5. The transaction response time is about 7 milliseconds in the case of UDP and 14 milliseconds in the case of SCTP.

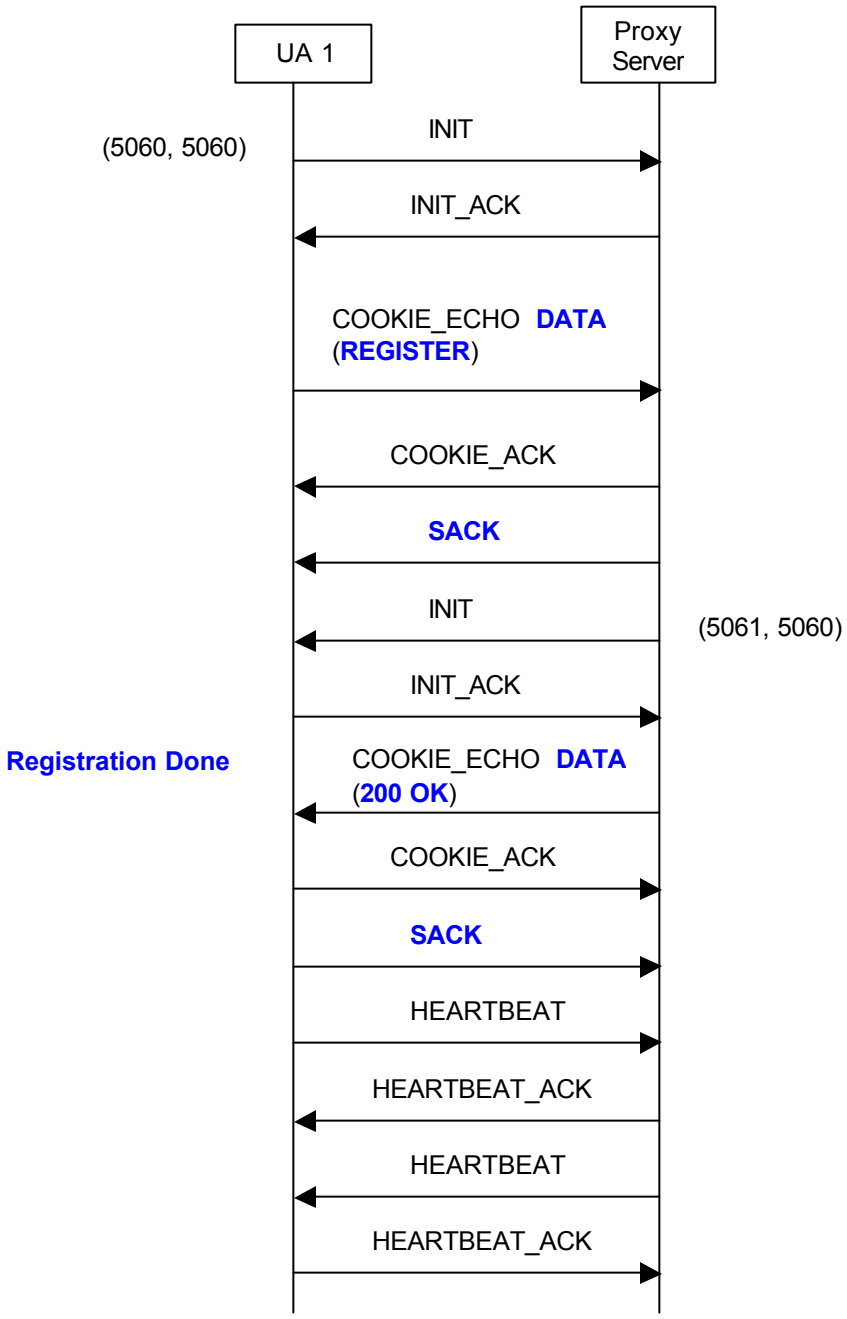


Figure 5.5: SIP registration test result. This is the message flow captured by Ethereal. Please note the additional overhead caused by SCTP handshaking. The message of the registration from UA2 should be the same.

In the test, two SCTP associations are established. The first one is initiated by UA1 between the local port 5060 and the remote port 5060 at the Proxy Server. The second association is initiated by the Proxy Server between the local port 5061 and the remote port 5060 at the UA1. Once the association is established, each endpoint will monitor the reachability of its peer by sending a HEARTBEAT chunk periodically to the peer. The SIP "REGISTER" request is sent from UA1 to Proxy Server as a data chunk in the COOKIE_ECHO message. The Proxy Server acknowledges the REGISTER request by sending "200 OK" as a data chunk in the COOKIE_ECHO message. Now, UA1 is successfully registered to the Proxy Server.

Comparing Figure 5.5 with the call flow for UDP in Figure 2.4, it is obvious that more messages are involved in the registration process in SCTP, especially when the SCTP associations required for the communication between the User Agent and the Proxy Server have not been established prior to the registration. If the required associations have been established, the number of messages will be reduced to four. That is, for each SCTP DATA message received, the recipient will return an acknowledgment SACK to the sender. Additional SCTP messages are also required for setting up a SCTP association if the inbound and outbound ports in the Proxy Server are not the same. In our example, the inbound port is 5060 and the outbound port is 5061. Therefore, the transaction response is expected to be longer for the Proxy Server using SCTP than UDP.

5.3.2 SIP Call Setup

This test verifies that a SIP session is successfully initiated measures the transaction response time of the SIP call setup. The call setup procedures captured by the Ethereal are shown in Figure 5.6. The transaction response time is about 6 seconds in the case of using SCTP and 4.6 seconds in the case of using UDP.

The SIP messages are transported using SCTP DATA messages. For each SCTP DATA message received, the recipient will return an acknowledgment SACK. Therefore, the number of messages involved in the basic call setup is at least twice as many as when using UDP as the transport. Additionally, HEARTBEAT and HEARTBEAT_ACK may be passing between the User Agent and the Proxy Server to monitor the integrity of the established associations. The transaction response time of the call setup also depends on when UA2 answers the call. Since *Linphone* does not have automatic answer feature, the call is manually answered, i.e., value of T1 in Figure 6.6 may be varied from test to test. To make a fair comparison, T1 is subtracted from the measured transaction response time.

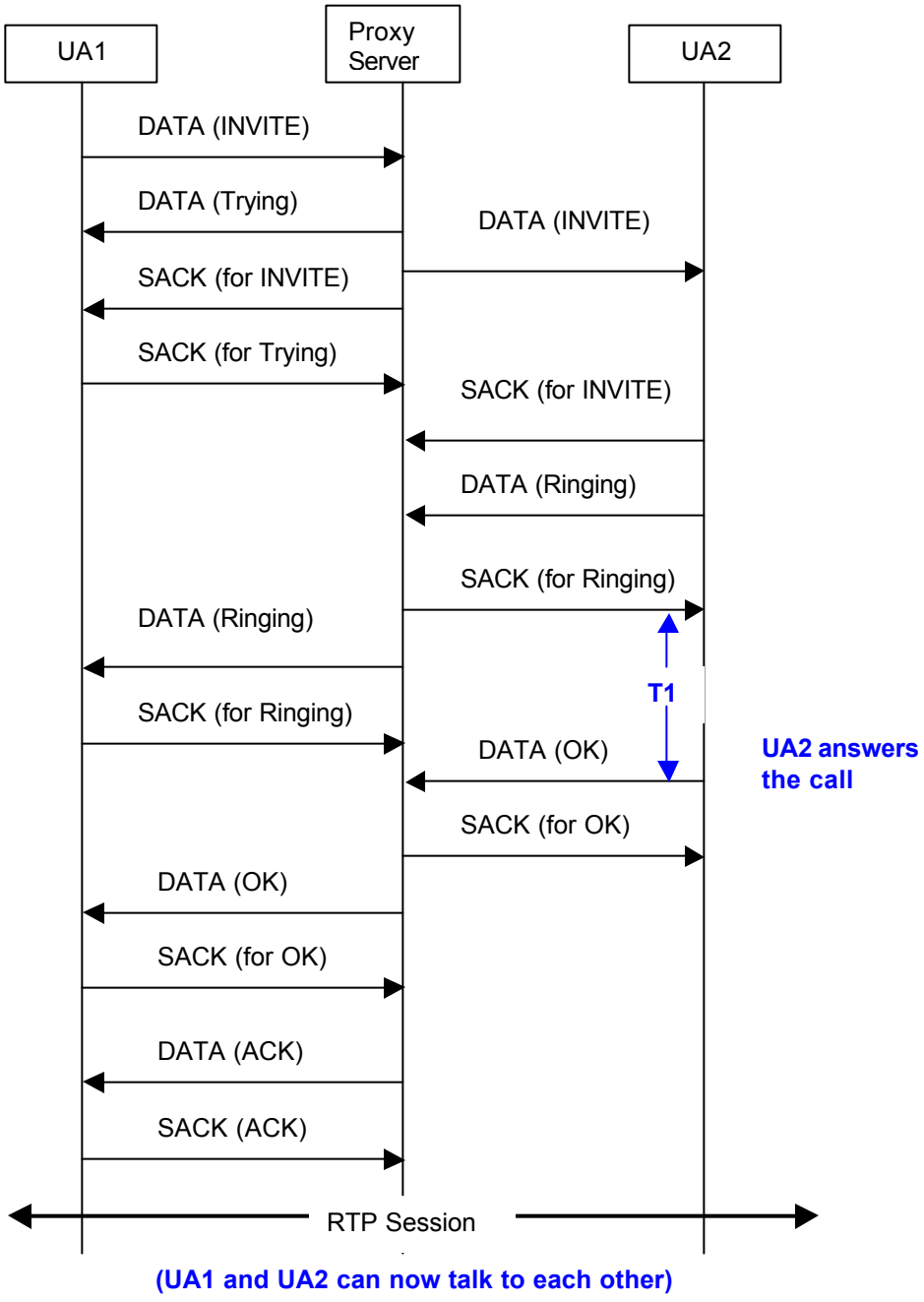


Figure 5.6: Call setup test result. This message flow is captured by *Ethereal*. T1 is the time taken for UA2 to answer the call after the phone rings.

5.3.3 SIP Call Termination

This test verifies that SIP call is successfully terminated and measures the corresponding transaction response time. The call termination procedures captured by the Ethereal are shown in Figure 5.7. The transaction response is about 1.7 seconds and 2.4 seconds in the case of using SCTP and UDP respectively.

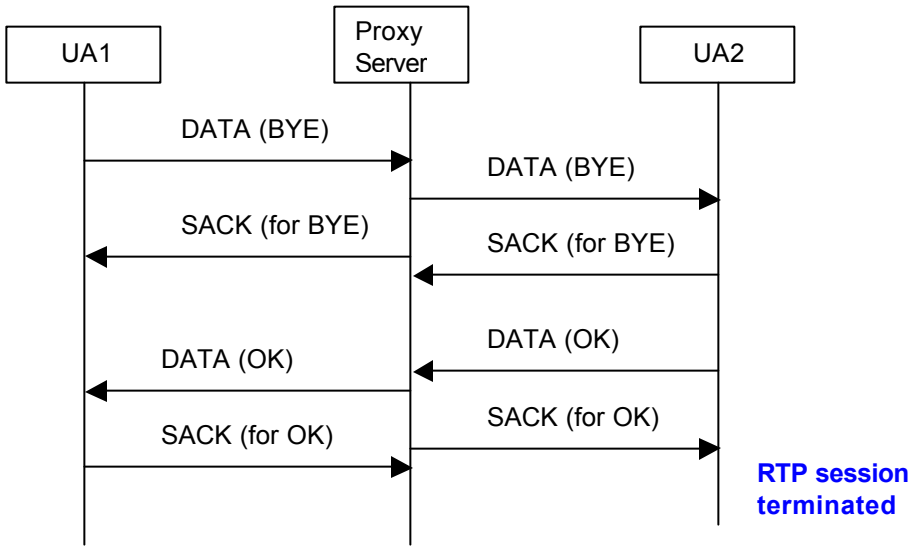


Figure 5.7: SIP call termination test result captured by *Ethereal.* SACK messages add overhead to the call termination duration.

As in the case of call setup procedures in the previous section, the number of messages involved in the case of using SCTP is expected to be at least twice as many as in the case of using UDP. The SCTP associations between the User Agent and the Proxy Server remain active until the programs (Partysip or Linphone) are terminated. Therefore, HEARTBEAT and HEARTBEAT_ACK messages are expected between the User Agent and the Proxy Server.

5.4 Multi-session Performance Test

Figure 5.8 illustrates the system setup for testing the multi-streaming feature of SCTP and comparing the performance of the proxy server using SCTP with that using UDP under the lossy conditions. To achieve this, eight identical SIP user agent programs on each machine A and B are configured to run simultaneously within the same association, i.e., eight streams within the same association. When more than eight SIP user agent programs are running within the machine, the machine becomes very slow because much CPU time is wasted on context switching among processes.

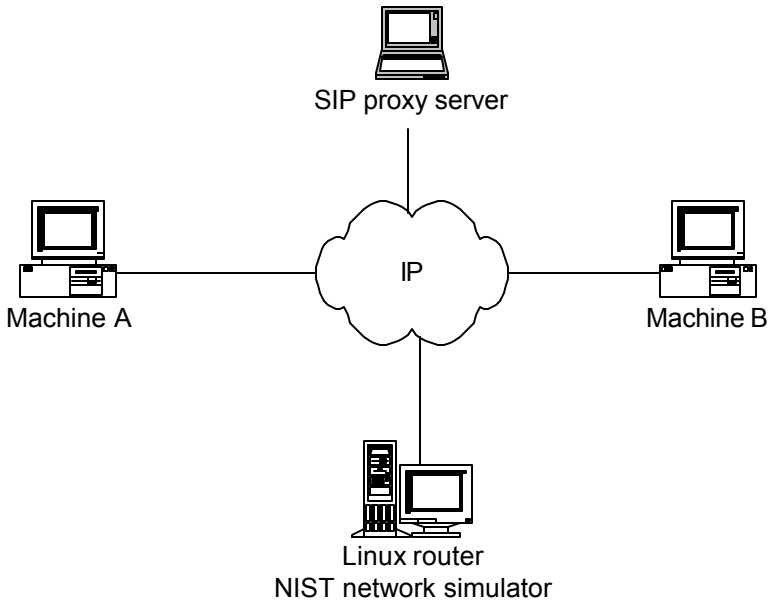


Figure 5.8: Multi-streaming test setup. The Linux machine installed with NIST network emulator is the default gateway of machines A, B and the SIP proxy server. Eight identical SIP user agent programs are running simultaneously on machine A and B.

The default gateways of the SIP proxy server, machine A and machine B are set to the Linux router. Therefore, all the traffic has to go through the Linux router. Assuming the IP address of the Linux router is 10.241.245.19, to

configure the default gateway in machine A, the following commands should be executed:

```
(add a routing entry for the default gateway)
/sbin/route add default gw 10.241.245.19

(list the routing table)
/bin/netstat -nr

Destination Gateway         Genmask         Iface
0.0.0.0     10.241.245.19   255.255.255.0   eth0
```

The Linux router is installed with the NISTnet as described in Section 5.1.5. The NISTnet is used to emulate packet loss in the network. The number of SIP call transactions handled by the proxy server under various packet loss conditions is measured. The results, shown in Table 5.3, indicate that the proxy server using SCTP can handle higher number of call transactions than the one using UDP when the packet loss in the network is 10% or higher. This is because UDP does not provide any recovery mechanism such as fast retransmit and recovery (FRR) to quickly recover the lost packets. Without FFR, the SIP user agent using UDP has to wait for retransmission timeout before retransmitting any lost packets. With FRR, if a receiver receives a data segment that is out of order, it immediately sends a duplicate acknowledgement to the sender. If the sender receives three duplicate acknowledgements, it assumes the data segment indicated by the acknowledgements is lost and immediately retransmits the lost segment.

With high packet loss, user agent has to slow down traffic injecting in the network in order to alleviate the network congestion. Proxy server using SCTP counts on the congestion control, such as windowing mechanism provided by the

transport layer (SCTP), to reduce the network congestion. Therefore, under lossy condition, the proxy server using SCTP is expected to perform better than the server using UDP. Since our testbed is much simpler than the real network and the performance difference between SCTP and UDP is small, we currently cannot conclude that proxy server using SCTP performs better than the one using UDP.

Table 5.3: Multi-streaming performance (number of SIP call transactions).

Protocol \ Packet loss	5%	10%	20%	30%
UDP	70	40	32	20
SCTP	60	43	37	26

Table 5.4 shows the monthly median ping packet loss measured by Stanford Linear Accelerator Center (SLAC) between different geographical locations for the month of April 1999 [24]. This kind of packet loss information can be used to determine whether we should run the proxy server using UDP or SCTP. For example, assuming that our results were valid for a real network and the packet loss information was up-to-date, a proxy server using UDP is expected to have better performance than one using SCTP when the packet loss in the network is less than 10%. However, SCTP is a better candidate as transport layer for SIP applications for networks where packet loss is above 10%, such as

networking between United Kingdom and Canada, and between United Kingdom and United Sates.

Table 5.4: Packet loss between different geographical locations. Source: Stanford Linear Accelerator Center (SLAC), 2003 [24], by permission.

World	Canada	Hungary	United States	Italy	Switzer-land	Germany	United Kingdom
Canada	3.50	4.82	0.62	3.98	2.51	6.31	12.75
Hungary	7.69	2.75	5.39	3.43	4.11	3.80	3.95
United States	3.43	3.40	0.27	2.20	0.93	6.41	11.00
Italy	5.01	2.06	0.50	0.40	0.41	1.31	0.35
Switzer-land	3.64	2.50	0.88	0.40	0.41	1.31	0.35
Germany	8.60	2.42	3.41	1.70	0.42	0.33	0.59
United Kingdom	13.88	2.37	10.15	0.53	0.40	0.64	0.13

5.5 Multi-homing

Multi-homing is one of the key features of SCTP. To test multi-homing in LKSCTP, an experimental network shown in Figure 5.9 was used. Machine A with two network interfaces was connected to Machine B via a 100Mbps Hub, i.e., both machines were in the same subnet (192.168.10.0). An SCTP association was established between Machine A and Machine B, with two IP addresses 192.168.10.1 (primary) and 192.168.10.2 (secondary) bound to the endpoint A, and 192.168.10.3 to endpoint B. The SCTP test program described

in Section 5.2 was used to pass the SCTP data messages passing from Machine A to Machine B through the primary path. *Ethereal* is used to monitor the traffic in the network. During the data transfer, the Ethernet cable attached to network interface 192.168.10.1 was unplugged from the network. The multi-homing feature should ensure that the SCTP data messages switch to the secondary path. However, Machine A kept retransmitting the data packets using the primary network interface and paused after the maximum number of re-transmissions.

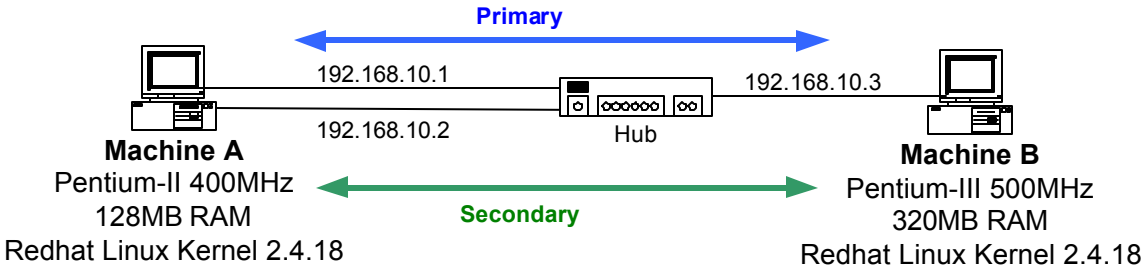


Figure 5.9: Experimental setup for multi-homing.

This multi-homing problem has been reported to the LKSCTP developers. They confirmed that the multi-homing feature was not working in lksctp-2.4.18_0_4_6 and suggested that the latest version should be used. However, the latest LKSCTP is intended for Linux kernel 2.5.x that has not been confirmed stable yet. As a result, the multi-homing feature is not demonstrated in this project.

5.6 Test Result Summary

Table 5.5 summarizes the results of the tests conducted in this Chapter.

Table 5.5: Test result summary.

<table>
<tr><th>Test case</th><th colspan="3">Results</th></tr>
<tr><td>Transport layer throughput test</td><td colspan="3">UDP has best throughput performance, and LKSCTP has the worst.</td></tr>
<tr><td>Single-session performance test</td><td colspan="3">Transaction response time:
<table><tr><td></td><td>UDP</td><td>SCTP</td></tr><tr><td>SIP registration</td><td>7 milliseconds</td><td>14 milliseconds</td></tr><tr><td>SIP call setup</td><td>4.6 seconds</td><td>6 seconds</td></tr><tr><td>SIP call termination</td><td>1.7 seconds</td><td>2.4 seconds</td></tr></table></td></tr>
<tr><td>Multi-session performance test</td><td colspan="3">With packet loss in the network greater than 10%, the proxy server using SCTP can handle slightly higher number of SIP calls than the one using UDP.</td></tr>
<tr><td>Multi-homing Feature</td><td colspan="3">Not supported in lksctp-2.4.18_0_4_6. Problem has been reported to LKSCTP developers.</td></tr>
</table>

6 Areas for Improvement

The multi-homing feature was not demonstrated in this project. The SCTP kernel of the SIP system should be first upgraded to the latest version and the multi-homing test should be repeated. Under such faulty condition as a broken data link, multi-homed proxy server should perform better than the single-homed proxy server, as traffic will be detoured from the faulty link to the standby link. We may also want to investigate the feasibility of using multi-homing feature for load balancing.

Due to limited number of machines available for this project, the multi-streaming test was only conducted between the user agent and the proxy server. A more realistic scenario would be to have a proxy server connecting to another proxy server, i.e., two user agent programs are communicating with each other via two proxy servers. Multiple streams are set up between two proxy servers. Multiple user agents are connected to the each proxy server using different streams, i.e., single stream between the user agent and the proxy server.

In this project, significant effort is devoted to developing a SIP-based Internet telephony using SCTP. A more complete benchmarking such as the one proposed by Columbia University and Ubiquity Incorporation should be conducted [25]. We compared the performance of the proxy server using SCTP with that using UDP. We should also develop a TCP plugin for the Partysip, investigate how SCTP multi-streaming feature resolves the head-of-line blocking problem that TCP experiences with, and compare their performance.

7 Conclusions

In this project, I successfully modified a publicly available SIP proxy server called *Partysip* to be used with SCTP. To functionally test the modifications, a publicly available SIP user agent called *linphone* was also modified to operate over SCTP. The SCTP implementation chosen for this project is Randall's SCTP (LKSCTP) kernel. The transaction response times of SIP registration, basic call setup, and call termination under no loss and single session conditions were also measured. In general, SIP applications using SCTP yield longer transaction response times than those using UDP because of the additional SCTP control messages required for handshaking. However, in multi-session environments and lossy conditions, where the packet loss rate is 10% or higher, the SIP proxy server using SCTP can handle more SIP call transactions than the one using UDP. The multi-homing feature could not be demonstrated in this project because it was not working in LKSCTP.

Under GNU Public License, the modifications to *Partysip* have been released to the public. This is to encourage future benchmarking activities, academic work, and research on SCTP support for SIP. Due to the simplicity of our test network and insignificant performance differences between UDP and SCTP, we cannot conclude that a SIP proxy server using SCTP performs better than the one using UDP. All the work presented in this report, however, provides a framework and invaluable design techniques for future development of a SIP-based Internet telephony system.

List of References

[1] J. Rosenberg, H. Schulzrinne, and G. Camarillo, “The Stream Control Transmission Protocol as a transport for the Session Initiation Protocol,” IETF Internet-Draft, Work in Progress, draft-ietf-sip-sctp-03.txt, June 2002.

[2] G. Camarillo, H. Schulzrinne, and R. Kantola, “A transport protocol for SIP,” Ericsson, Finland, November 2001. http://standards.ericsson.net/gonzalo/papers/SIP_SCTP.pdf (March 28, 2003).

[3] Howstuffworks “How IP telephony works”: http://electronics.howstuffworks.com/ip-telephony.htm (March 14, 2003).

[4] Softswitch consortium: http://www.softswitch.org (March 14, 2003).

[5] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, “Stream Control Transmission Protocol,” IETF RFC 2960, October 2000.

[6] Wirlab Kphone: http://www.wirlab.net/kphone (March 14, 2003).

[7] Linphone: http://www.linphone.org (March 14, 2003).

[8] Vovida.org – Your source for open source communication: http://www.vovida.org (March 14, 2003).

[9] The Partysip SIP proxy server: http://www.Partysip.org (March 14, 2003).

[10] Stream Control Transmission Protocol: http://www.sctp.de (March 14, 2003).

[11] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol,” IETF RFC 3261, June 2002.

[12] R. Alamgir, M. Atiquzzaman, and W. Ivancic, “Effect of congestion control on the performance of TCP and SCTP over satellite networks,” ESTC-2002, June 2002.

[13] R. Rajamani, S. Kumar, and N. Gupta. “SCTP versus TCP: Comparing the performance of transport protocols for web traffic,” University of Wisconsin-Madison, May 2002.

http://www.cs.wisc.edu/~raj/sctp/report.pdf (March 28, 2003).

[14] R. Brennan, T.Ravier, and T. Curran, “Experimental studies of SCTP multi-homing,” Teltec DCU, Dublin 9, Ireland, 2001. http://telecoms.eeng.dcu.ie/symposium/papers/B4.pdf (March 28, 2003).

[15] S. W. Chiat, “SCTP daemons’ User application guidelines: using the sctpdlib,” Siemens AG, Germany, November 2001. (Part of sctplib-1.0.0-pre18 release, http://www.sctp.de/download/sctplib-1.0.0-pre18.tar.gz.)

[16] R. Steward, Q. Xie, L. H. P. Yarroll, J. Wood, K. Poon, and K. Fujita, “Sockets API extensions for Stream Control Transmission Protocol,” IETF Internet-Draft, Work in Progress, draft-ietf-tsvwg-socketapi-04.txt, April 2002.

[17] The GNU oSIP library: http://www.gnu.org/software/osip (March 14, 2003).

[18] SourceForge.net, Linux Kernel SCTP: http://sourceforge.net/projects/lksctp (March 14, 2003).

[19] Ethereal: http://www.ethereal.com (March 14, 2003).

[20] NIST Net home page: http://snad.ncsl.nist.gov/itg/nistnet (March 14, 2003).

[21] Test TCP (TTCP) benchmarking tool for measuring TCP and UDP performance: http://www.pcausa.com/Utilities/pcattcp.htm (March 14, 2003).

[22] J. Postel, “User Datagram Protocol,” IETF RFC 768, August 1980.

[23] J. Postel, “Transmission Control Protocol,” IETF RFC 793, September 1981.

[24] SLAC-RAL Internet packet loss performance: http://www.slac.stanford.edu/comp/net/wan-mon/ral.html (March 14, 2003).

[25] H. Schulzrinne, S. Narayanan, M. Doyle, and J. Lennox, “SIPstone – benchmarking SIP server performance,” Columbia University and Ubiquity Inc., April 12, 2002.

Appendix A – Disclaimer of Partysip

Aymeric MOIZARD
146 Quai louis Bleriot
75016 PARIS
FRANCE

Dec 30, 2002

Disclaimer of Partysip

Under GNU General Public License (version 2, June 1991), I, Thomas Pang, hereby disclaim all copyright interest for the contributions made in 'Partysip' and its plugins written by Aymeric MOIZARD and WellX Telecom. I, Thomas Pang, also disclaim all future contributions that I will do with Partysip.

On December 19, 2002, I, Thomas Pang, released the SCTP plugin to Mr. Aymeric Moizard. The plugin allows the user of Partysip to run Partysip on SCTP, as opposed to UDP. The plugin has been tested with linphone-0.9.1 (modified for running on SCTP) on SCTP kernel (lksctp-2.4.18-0_4_6). However, there is no warranty for the changes made to Partysip and I, Thomas Pang, will not be responsible for any damages caused by using this software.

Should you have any question about this disclaimer, please feel free to contact me at the above address or email me thomaskc@yahoo.com.

_____
Thomas Pang

Appendix B - SCTP Plugin Release Notice

From: Aymeric Moizard [mailto:jack@atosc.org]
Sent: Thursday, January 30, 2003 8:11 AM
To: Pang, Thomas
Cc: Partysip-dev@nongnu.org
Subject: SCTP plugin for Partysip

Thanks to Thomas Pang, Partysip has now a SCTP plugin.

You can download it at:
http://osip.atosc.org/download/Partysip/

You'll need a sctp kernel to compile it.

Partysip must be compiled with a specific option
and I'm not sure that you can use both UDP and
SCTP together.

I think there are still bugs as I did not change
the Via headers for SCTP support and other issue
might also prevent it from working. But anyway,
this is a good start.

I'd like to get feedback from users that have
implemented reliable protocols in their applications.
Did you find any issue?

Aymeric

Appendix C – Modifications to Partysip

A new plugin module called *sctp* is created and is located at Partysip-X.X.X/plugin/sctp. It consists of the following files:

- Makefile.am: Define build environment variables and is used to generate Makefile.in
- Makefile.in: It is generated by *automake* from Makefile.am and will not be listed here. This file will be used to generate the final Makefile for the sctp plugin module.
- sctp.h: Header file of sctp plugin module
- sctp_core.c: It contains module initialization and release functions
- sctp.c: It contains functions for receiving, sending, and processing SCTP messages.

As mentioned in Appendix B, this module can be download from http://osip.atosc.org/download/Partysip/

Please read and follow the README and the INSTALL files to build and install the sctp plugin.

Makefile.am:

```
EXTRA_DIST = sctp.h

libdir = $(prefix)/lib/Partysip/
plugindir = $(libdir)

lib_LTLIBRARIES = libpsp_sctp.la

libpsp_sctp_la_SOURCES = sctp.c sctp_core.c

libpsp_sctp_la_LDFLAGS = -export-dynamic -release $(PLUGIN_RELEASE) \
-L$(prefix)/lib -losip -lfsmtl @PLUGIN_LIB@

INCLUDES = -I$(top_srcdir) -I$(prefix)/ppl/unix -I$(prefix)/include

AM_CFLAGS = -Wall @PLUGIN_FLAGS@ `Partysip-config --cflags`

noinst_HEADERS = sctp.h
```

sctp.h:

```
#ifndef _SCTP_H_
#define _SCTP_H_

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#ifdef _SCTPLIB
#include <ext_socket.h>
#else
#include <netinet/sctp.h>
// #include <sctp_lib.h>    /* for bindx */
#endif

typedef struct local_ctx_t
{
  int in_port;
  int in_socket;
  int out_port;
  int out_socket;
}
local_ctx_t;

int local_ctx_init (int in_port, int out_port);
void local_ctx_free ();

/* callback for tlp plugin */
int cb_rcv_sctp_message (int max);

int cb_snd_sctp_message (const transaction_t * transaction,
                         sip_t * message,        /* message to send   */
                         char *host,     /* proposed destination host */
                         int port,       /* proposed destination port */
                         int socket);    /* proposed socket (if any)  */

/* miscelaneous methods */
int sctp_log_event (sipevent_t * evt);
int sctp_process_message (char *buf);


#endif
```

sctp_core.c:

```
#include <Partysip/Partysip.h>

#include "sctp.h"

/* this structure is retreived by the core application
   with dlsym */
psp_plugin_t PSP_PLUGIN_DECLARE_DATA sctp_plugin;

/* element shared with the core application layer */
/* all xxx_plugin elements MUST be DYNAMICLY ALLOCATED
   as the core application will assume they are */
tlp_plugin_t *sctp_plug;

extern local_ctx_t *ctx;

/* this is called by the core application when the module
   is loaded (the address of this method is located in
   the structure upd_plugin->plugin_init() */
PSP_PLUGIN_DECLARE (int)
plugin_init ()
{
  tlp_rcv_func_t *fn_rcv;
  tlp_snd_func_t *fn_snd;
  int flag = TLP_SCTP;           /* really unused by now */
  int i;
  char *in_sctp_port;

  /* plugin MUST create their own structure and give them
     back to the core by calling:
```

```
     psp_core_load_xxx_plugin();
     where xxx is the module name related to the plugin.
     psp_plugins can have more than one xxx_plugins attached
   */
  OSIP_TRACE (osip_trace (__FILE__, __LINE__, OSIP_INFO3, NULL,
                          "sctp plugin: plugin_init()!\n"));

  in_sctp_port = psp_config_get_element ("serverport_sctp");
  if (in_sctp_port == NULL)
    i = local_ctx_init (5060, 5060); /* same port for both req and anwsers */
  else
    {
      int p = atoi (in_sctp_port);

      i = local_ctx_init (p, p); /* same port for both req and anwsers */
    }
  if (i != 0)
    return -1;

  psp_plugin_take_ownership (&sctp_plugin);
  i = psp_core_load_tlp_plugin (&sctp_plug, &sctp_plugin, flag);
  if (i != 0)
    goto pi_error1;             /* so the core application known initialization failed! */

  tlp_plugin_set_input_socket (sctp_plug, ctx->in_socket);
  tlp_plugin_set_output_socket (sctp_plug, ctx->output_socket);
  tlp_plugin_set_multicast_socket (sctp_plug, -1);

  /* add hook */
  i = tlp_rcv_func_init (&fn_rcv, &cb_rcv_sctp_message, sctp_plugin.plug_id);
  if (i != 0)
    goto pi_error2;             /* so the core application known initialization failed! */
  i = tlp_snd_func_init (&fn_snd, &cb_snd_sctp_message, sctp_plugin.plug_id);
  if (i != 0)
    goto pi_error3;              /* so the core application known initialization failed! */

  i = tlp_plugin_set_rcv_hook (sctp_plug, fn_rcv);
  if (i != 0)
    goto pi_error4;             /* so the core application known initialization failed! */
  i = tlp_plugin_set_snd_hook (sctp_plug, fn_snd);
  if (i != 0)
    goto pi_error4;             /* so the core application known initialization failed! */

  /* from this point, Partysip can hook from this plugin */
  return 0;

  /* TODO */
pi_error4:
pi_error3:
pi_error2:
pi_error1:
  local_ctx_free (ctx);
  return -1;
}

PSP_PLUGIN_DECLARE (int)
plugin_start ()
{
  /* OSIP_TRACE(osip_trace(__FILE__,__LINE__,OSIP_INFO3,NULL,
     "sctp plugin: plugin_start()!\n")); */

  return -1;
}

PSP_PLUGIN_DECLARE (int)
plugin_release ()
{
  OSIP_TRACE (osip_trace (__FILE__, __LINE__, OSIP_INFO3, NULL,
                          "sctp plugin: plugin_release()!\n"));
  local_ctx_free (ctx);
  ctx = NULL;
```

```
  return 0;
}

psp_plugin_t PSP_PLUGIN_DECLARE_DATA sctp_plugin = {
  0,                            /* uninitialized */
  "Sctp plugin",
  "0.5.4",
  "plugin for receiving and sending SCTP message",
  PLUGIN_TLP,
  0,                            /* number of owners is always 0 at the begining */
  NULL,                         /* future place for the dso_handle */
  &plugin_init,
  &plugin_start,
  &plugin_release
};
```

sctp.c:

```
#include <Partysip/Partysip.h>
#include <ppl/ppl_dns.h>
#include <ppl/ppl_socket.h>
#include "sctp.h"

#include <fcntl.h>

local_ctx_t *ctx = NULL;

// create and configure socket for sip message passing
int local_ctx_init (int in_port, int out_port)
{
  int i;
  struct sockaddr_in raddr;

  ctx = (local_ctx_t *) smalloc (sizeof (local_ctx_t));

  if (ctx == NULL) return -1;

  ctx->in_port = in_port;

  /* not used by now */
  ctx->out_port = out_port;

#ifdef _SCTPLIB
  ctx->in_socket = ext_socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
#else
  ctx->in_socket = socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
#endif

  if (ctx->in_socket == -1)
  {
      OSIP_TRACE (osip_trace (__FILE__, __LINE__, OSIP_ERROR, NULL,
          "sctp plugin: cannot create descriptor for port %i!\n",
          ctx->in_port));
      goto lci_error1;
  }

  if (out_port == in_port)
    ctx->out_socket = ctx->in_socket;
  else
  {
      int try;

#ifdef _SCTPLIB
      ctx->out_socket = ext_socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
#else
      ctx->out_socket = socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
#endif

      if (ctx->out_socket == -1)
```

```
      {
          OSIP_TRACE (osip_trace (__FILE__, __LINE__, OSIP_ERROR, NULL,
              "sctp plugin: cannot create descriptor for port %i!\n",
              ctx->out_port));
          goto lci_error2;
      }
      i = (-1);
      try = 0;
      bzero(&raddr, sizeof(raddr));
      while (i < 0 && try < 40)
      {
          try++;
          raddr.sin_addr.s_addr = INADDR_ANY;
          raddr.sin_port = htons(ctx->out_port);
          raddr.sin_family = AF_INET;

#ifdef _SCTPLIB
          i = ext_bind(ctx->out_socket,
              (struct sockaddr *)&raddr, sizeof(raddr));
#else
          i = bind(ctx->out_socket,
              (struct sockaddr *)&raddr, sizeof(raddr));
#endif

          if (i < 0)
          {
            OSIP_TRACE (osip_trace
                (__FILE__, __LINE__, OSIP_ERROR, NULL,
                "sctp plugin: cannot bind on port %i (%s)!\n", ctx->out_port,
                strerror (errno)));
            ctx->out_port++;
          }
      }
      if (i != 0) goto lci_error3;
  }

  bzero(&raddr, sizeof(raddr));
  raddr.sin_addr.s_addr = INADDR_ANY;
  raddr.sin_port = htons(ctx->in_port);
  raddr.sin_family = AF_INET;

#ifdef _SCTPLIB
  if (ext_bind(ctx->in_socket,
      (struct sockaddr *)&raddr, sizeof(raddr)) < 0) {
#else
  if (bind(ctx->in_socket,
      (struct sockaddr *)&raddr, sizeof(raddr)) < 0) {
#endif

      OSIP_TRACE (osip_trace
          (__FILE__, __LINE__, OSIP_ERROR, NULL,
          "sctp plugin: cannot bind on port %i (%s)!\n", ctx->in_port,
          strerror (errno)));
      goto lci_error4;
  }

#ifdef _SCTPLIB
  if (ext_listen(ctx->in_socket,0) < 0)
#else
  if (listen(ctx->in_socket,0) < 0)
#endif
      OSIP_TRACE (osip_trace
          (__FILE__, __LINE__, OSIP_ERROR, NULL,
          "sctp plugin: failed to listen on port %i (%s)!\n", ctx->in_port,
          strerror (errno)));

  return 0;

lci_error5:
lci_error4:
lci_error3:
```

```
  if (out_port != in_port)
    close (ctx->out_socket);
  ctx->out_socket = -1;
lci_error2:
  close (ctx->in_socket);
lci_error1:
  sfree (ctx);
  ctx = NULL;
  return -1;
}

void
local_ctx_free ()
{
  if (ctx == NULL)
    return;
  if (ctx->in_socket != -1)
    close (ctx->in_socket);
  if (ctx->in_port == ctx->out_port)
    ctx->out_socket = (-1);
  else if (ctx->out_socket != -1)
    close (ctx->out_socket);
  sfree (ctx);
  ctx = NULL;
}

/* max_analysed = maximum number of message when this method can parse
   messages whithout returning.

   This method returns:
   -1 on error
   0  on no message available
   1  on max_analysed reached
*/
int
cb_rcv_sctp_message (int max)
{
  char *buf;
  int i;
  int proceed;

#ifndef _SCTPLIB
  struct msghdr msg;
  struct iovec iov;
  struct SCTP_cmsghdr cmsghdr;
  char *data;
  union sctp_notification *sn;
  struct sockaddr_in remote_addr;
#endif

  if (ctx == NULL)
  {
       return -1;
  }

  /* create a set of file descritor to control the stack */

  for (; max != 0; max--)
  {
      buf = (char *) smalloc (SIP_MESSAGE_MAX_LENGTH * sizeof (char) + 1);
      OSIP_TRACE (osip_trace
          (__FILE__, __LINE__, OSIP_INFO2, NULL,
          "WAITING FOR SCTP MESSAGE\n"));

      proceed = 1;

#ifdef _SCTPLIB
      i = ext_recv(ctx->in_socket,buf,SIP_MESSAGE_MAX_LENGTH,0);
#else
      iov.iov_base = buf;
      iov.iov_len = SIP_MESSAGE_MAX_LENGTH*sizeof(char)+1;
```

```
      msg.msg_iov = &iov;
      msg.msg_iovlen = 1;
      msg.msg_control = &cmsghdr;
      msg.msg_controllen = sizeof(cmsghdr);
      msg.msg_name = &remote_addr;
      msg.msg_namelen = sizeof(remote_addr);

      i = recvmsg(ctx->in_socket,&msg,MSG_WAITALL);
#endif
      if (i > 0)
      {
#ifndef _SCTPLIB
          if (msg.msg_flags & MSG_NOTIFICATION)
         {
            proceed = 0;
         }
#endif
          if (proceed)
         {
              /* Message might not end with a "\0" but we know
                 the number of char received! */
              sstrncpy (buf + i, "\0", 1);
#ifndef HIDE_MESSAGE
              OSIP_TRACE (osip_trace
                  (__FILE__, __LINE__, OSIP_INFO1, NULL, "\n%s\n", buf));
#endif
              sctp_process_message (buf);
          }
      }
      else if (i == -1)
      {
          if (errno == EAGAIN)
          {
              sfree (buf);
              return 0;
          }
          OSIP_TRACE (osip_trace
              (__FILE__, __LINE__, OSIP_ERROR, NULL,
              "sctp plugin: error while receiving data!\n"));
          sfree (buf);
          return -1;
      }
    }
  /* max is reached */
  return 1;
}

int
sctp_process_message (char *buf)
{
  sipevent_t *evt;

  if (buf == NULL
      || *buf == '\0'
      || buf[1] == '\0'
      || buf[2] == '\0' || buf[3] == '\0' || buf[4] == '\0' || buf[5] == '\0')
  {
      sfree (buf);
      return -1;
  }

  evt = osip_parse (buf);
  sfree (buf);
  if (evt == NULL)
  /* discard... */
  {
      OSIP_TRACE (osip_trace
          (__FILE__, __LINE__, OSIP_ERROR, NULL,
          "sctp module: Could not parse response!\n"));
      return -1;
  }
```

```
  if (evt->sip == NULL)
  {
      OSIP_TRACE (osip_trace
          (__FILE__, __LINE__, OSIP_ERROR, NULL,
          "sctp module: Could not parse response!\n"));
      msg_free (evt->sip);
      sfree (evt->sip);
      sfree (evt);
      return -1;
  }
  sctp_log_event (evt);

  /* modify the request so it's compliant with the latest draft */
  psp_core_fix_strict_router_issue (evt);

  psp_core_event_add_sip_message (evt);
  return 0;
}


int
sctp_log_event (sipevent_t * evt)
{
#ifdef SHOW_LIMITED_MESSAGE
  via_t *via;
  generic_param_t *b;

  via = list_get (evt->sip->vias, 0);
  via_param_getbyname (via, "branch", &b);
  if (b == NULL)
  {
      if (MSG_IS_REQUEST (evt->sip))
      {
      } else
        {
        }
      return -1;
    }
  if (MSG_IS_REQUEST (evt->sip))
    {
  } else
    {
    }
#endif
  return 0;
}

/* return
   -1 on error
   0  on success
*/
int
cb_snd_sctp_message (const transaction_t * transaction,  /* read only element */
                     sip_t * message,    /* message to send           */
                     char *host, /* proposed destination host */
                     int port,   /* proposed destination port */
                     int socket) /* proposed socket (if any)  */
{
  int i;
  char *buf;
  struct hostent *hp;
  struct hostent **hp_ptr;
  struct sockaddr_in addr;
  unsigned long int one_inet_addr;
  int sock;

#ifndef _SCTPLIB
  struct msghdr msg;
  struct iovec iov;
#endif
```

```
  if (ctx == NULL)
    return -1;

  hp = NULL;
  hp_ptr = NULL;

  i = msg_2char (message, &buf);

  if (i != 0)
    return -1;
#ifndef HIDE_MESSAGE
  OSIP_TRACE (osip_trace (__FILE__, __LINE__, OSIP_INFO1, NULL, "\n%s\n", buf));
#endif
  /* For RESPONSE, oSIP ALWAYS provide host and port from the top via */
  /* For REQUEST,  oSIP SOMETIMES provide host and port to use which
     may be different from the request uri */

  if (host == NULL)
    { /* when host is NULL, we use the request uri value */
      host = message->strtline->rquri->host;
      if (message->strtline->rquri->port != NULL)
        port = satoi (message->strtline->rquri->port);
      else
        port = 5060;
    }

  if ((int) (one_inet_addr = inet_addr (host)) == -1)
    {
      i = ppl_dns_gethostbyname (&addr, host, port);
      if (i != PPL_SUCCESS)
        {
          sfree (buf);
          return -1;
        }
  } else
    {
      addr.sin_port = htons ((short) port);
      addr.sin_family = AF_INET;
      addr.sin_addr.s_addr = one_inet_addr;
    }

  sock = ctx->out_socket;

#ifdef _SCTPLIB
  if (0  > ext_sendto (sock, (const void*) buf, strlen (buf), 0,
                (struct sockaddr *) &addr,sizeof(addr)))
#else
  iov.iov_base = buf;
  iov.iov_len = strlen(buf);
  msg.msg_iov = &iov;
  msg.msg_iovlen = 1;
  msg.msg_control = NULL;
  msg.msg_controllen = 0;
  msg.msg_name = (void *) &addr;
  msg.msg_namelen = sizeof(addr);
  if (0 > sendmsg(sock, &msg, 0))
#endif
    {
      sfree (buf);
      if (ECONNREFUSED == errno)
        {
          OSIP_TRACE (osip_trace (__FILE__, __LINE__, OSIP_ERROR, NULL,
                                  "SIP_ECONNREFUSED - No remote server.\n"));
          return 1;             /* I prefer to answer 1 by now..
                                   we'll see later what's better */
        }
      OSIP_TRACE (osip_trace (__FILE__, __LINE__, OSIP_ERROR, NULL,
                              "SIP_NETWORK_ERROR - Network error %i (%s) sending message
to %s on port %i.\n",
                              i, strerror (i), host, port));
      /* SIP_NETWORK_ERROR; */
```

```
      return -1;
    }

  OSIP_TRACE (osip_trace
              (__FILE__, __LINE__, OSIP_INFO1, NULL,
               "sctp_plugin: message sent to %s on port %i\n", host, port));

  sfree (buf);
  return 0;
}
```

Appendix D – Modifications to Linphone

The user agent program that I am using for this project is linphone-0.9.1 which can be downloaded http://simon.morlat.free.fr/download (last access March 16, 2003). The following files under linphone-0.9.1/osipua/src are modified:

- Makefile: Script for compiling the program
- osipmanager.c: User agent configuration, initialization and release.
- osipua.h: Header file of the oSIP user agent
- udp.c: It contains functions for handling the UDP transport. Here I modify it to handle SCTP transport instead.
- udp.h: Header file of udp.c

To list the changes I made, I rename the original *src* to *src.org* and the modified *src* to *src.sctp*, and use the following command to capture the differences to a file called *diff.log* as listed below: *diff -r src.org src.sctp > diff.log*

diff.log:

```
diff -r src.org/Makefile src.sctp/Makefile
63c63
< host_triplet = i686-pc-linux-gnu
---
> host_triplet = i586-pc-linux-gnu
129,130c129,130
<   -lpthread -lnsl
<
---
> -lpthread -lnsl
> #-lpthread -lnsl -lsctpsocket -lsctp -lglib -lstdc++
134c134,140
< INCLUDES = -I$(top_srcdir) -I$(osip_prefix)/include
---
> SCTP=/home/tkpang/lksctp-2_4_18-0_4_6
> SCTP_INCL = -I$(SCTP)/test/libc/include -I$(SCTP)/test -D_LKSCTP
>
> #INCLUDES = -I$(top_srcdir) -I$(osip_prefix)/include
> #DEFS = -DHAVE_CONFIG_H   -DENABLE_DEBUG -g -D_SCTPLIB
> INCLUDES = -I$(top_srcdir) -I$(osip_prefix)/include $(SCTP_INCL)
> DEFS = -DHAVE_CONFIG_H   -DENABLE_DEBUG -g -D_DEBUG
136d141
< DEFS = -DHAVE_CONFIG_H   -DENABLE_DEBUG -g
diff -r src.org/osipmanager.c src.sctp/osipmanager.c
132a133,138
> #ifdef _DEBUG
>      printf("[Thomas] osip_manager_add_udpport port: %d\n", port);
>      printf("[Thomas] osip_manager_add_udpport send_port: %d\n",
>                     manager->send_port);
> #endif
>
144,147c150,159
<             newfd = socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP);
<
<             laddr.sin_addr.s_addr = htons (INADDR_ANY);
<             laddr.sin_port = htons ((short) port);
---
> #ifdef _SCTPLIB
>             newfd = ext_socket (PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
> #else
```

```
>             newfd = socket (PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
> #endif
>
> //          laddr.sin_addr.s_addr = htons (INADDR_ANY);
> //          laddr.sin_port = htons ((short) port);
>             laddr.sin_addr.s_addr = INADDR_ANY;
>             laddr.sin_port = htons (port);
149,150c161,165
<             if (bind (newfd, (struct sockaddr *) &laddr, sizeof (laddr)) <
<                 0)
---
> #ifdef _SCTPLIB
>             if (ext_bind (newfd, (struct sockaddr *) &laddr, sizeof (laddr)) < 0)
> #else
>             if (bind (newfd, (struct sockaddr *) &laddr, sizeof (laddr)) < 0)
> #endif
156a172,189
>
>             // Thomas
> #ifdef _SCTPLIB
>             if (ext_listen(newfd,0)<0)
> #else
>             if (listen(newfd,0)<0)
> #endif
>             {
>                    osip_trace (OSIP_ERROR,
>                                ("Failed to listen socket !\n"));
>                    close (newfd);
>                    return -errno;
>             }
>
> #ifdef _SCTPLIB
> //          err = ext_setsockopt (newfd, SOL_SOCKET, SO_REUSEADDR,
> //                          (void *) &option, sizeof (option));
> #else
158a192
> #endif
360a395,399
> #ifdef _DEBUG
>      printf("[Thomas] manager->send_port: %d\n", manager->send_port);
>      printf("[Thomas] port: %d\n", port);
> #endif
>
367,370c406,419
<      newfd = socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP);
<
<      laddr.sin_addr.s_addr = htons (INADDR_ANY);
<      laddr.sin_port = htons ((short) port);
---
> #ifdef _SCTPLIB
>      newfd = ext_socket (PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
> #else
>      newfd = socket (PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
> #endif
>
> #ifdef _DEBUG
>      printf("[Thomas] newfd: %d\n", newfd);
> #endif
>
> //   laddr.sin_addr.s_addr = htons (INADDR_ANY);
> //   laddr.sin_port = htons ((short) port);
>      laddr.sin_addr.s_addr = INADDR_ANY;
>      laddr.sin_port = htons (port);
371a421,423
> #ifdef _SCTPLIB
>      if (ext_bind (newfd, (struct sockaddr *) &laddr, sizeof (laddr)) < 0)
> #else
372a425
> #endif
378a432,435
> #ifdef _SCTPLIB
```

```
> //   err = ext_setsockopt (newfd, SOL_SOCKET, SO_REUSEADDR, (void *) &option,
> //                   sizeof (option));
> #else
380a438
> #endif
383a442,453
>      // Thomas
> #ifdef _SCTPLIB
>      if (ext_listen(newfd,0)<0)
> #else
>      if (listen(newfd,0)<0)
> #endif
>      {
>             osip_trace (OSIP_ERROR, ("Failed to listen socket !\n"));
>             close (newfd);
>             return -errno;
>      }
>
diff -r src.org/osipua.h src.sctp/osipua.h
39d38
<
Only in src.sctp: tags
diff -r src.org/udp.c src.sctp/udp.c
40a41,45
>
> #ifdef _DEBUG
>      printf("[Thomas] ev->type = %d\n", ev->type);
>      printf("[Thomas] return value of osip_find_transaction_and_add_event: %d\n", err);
> #endif
118a124
>
130a137,147
>      struct sockaddr_in remote_addr;
>      int length;
>
> #ifdef _LKSCTP
>      struct msghdr msg;
>      struct iovec iov;
>      struct SCTP_cmsghdr cmsghdr;
>      char *data;
>      union sctp_notification *sn;
> #endif
>
133a151,154
> #ifdef _SCTPLIB
>      nb = ext_select (manager->max_udpfd + 1, &osipfdset, NULL, NULL,
>                  &timeout);
> #else
135a157
> #endif
150a173,190
>
>                           length = sizeof(remote_addr);
> #ifdef _SCTPLIB
>                           i = ext_recvfrom(manager->udpfds[j], manager->udp_buf,
>                                  SIP_MESSAGE_MAX_LENGTH, 0,
>                                  (struct sockaddr *)&remote_addr, &length);
> #else
>                           iov.iov_base = manager->udp_buf;
>                           iov.iov_len = SIP_MESSAGE_MAX_LENGTH+1;
>                           msg.msg_iov = &iov;
>                           msg.msg_iovlen = 1;
>                           msg.msg_control = &cmsghdr;
>                           msg.msg_controllen = sizeof(cmsghdr);
>                           msg.msg_name = (void *) &remote_addr;
>                           msg.msg_namelen = length;
>                           i = recvmsg(manager->udpfds[j], &msg, MSG_WAITALL);
> #endif
>                           /*
153a194
>                                   */
```

```
159a201,205
> #ifdef _DEBUG & _LKSCTP
>                     printf("[Thomas] msg.msg_flags: 0x%x\n", msg.msg_flags);
>                     printf("[Thomas] length of recv msg: %d\n", i);
> #endif
>
162a209,213
>
> #ifdef _LKSCTP
>                                 if (!(msg.msg_flags & MSG_NOTIFICATION))
>                                 {
> #endif
164a216,220
>
> #ifdef _DEBUG
>                     printf("[Thomas] sipevent: %d\n", sipevent);
>                     printf("[Thomas] sipevent type: %d\n", sipevent->type);
> #endif
166a223,226
>                                 else
> #ifdef _DEBUG
>                     printf("[Thomas] osip_parse error\n");
> #endif
168c228,232
<
---
> #ifdef _LKSCTP
>                                 }
>                                 else
>                                     printf("info: receiving SCTP ctrl msg\n");
> #endif
188a253,259
>
> #ifdef _LKSCTP
>      struct msghdr msg;
>      struct iovec iov;
>      struct SCTP_cmsghdr cmsghdr;
> #endif
>
212c283,284
<      if (0 > sendto (sock, (const void *) message, strlen (message), 0,
---
> #ifdef _SCTPLIB
>      if (0 > ext_sendto (sock, (const void *) message, strlen (message), 0,
213a286,296
> #else
>      iov.iov_base = message;
>      iov.iov_len = strlen(message)+1;
>      msg.msg_iov = &iov;
>      msg.msg_iovlen = 1;
>      msg.msg_control = NULL;
>      msg.msg_controllen = 0;
>      msg.msg_name = &addr;
>      msg.msg_namelen = sizeof(addr);
>      if (sendmsg(sock, &msg, 0) < 0)
> #endif
diff -r src.org/udp.h src.sctp/udp.h
20a21,29
> #ifdef _SCTPLIB
> #include "/usr/local/include/ext_socket.h"
> #endif
>
> #ifdef _LKSCTP
> #include <netinet/sctp.h>
> #include <sctp_lib.h>
> #endif
>
```

Appendix F – Test programs

These test programs are used to measure the throughput performance of SCTP kernel and SCTPLIB for various packet sizes. The client program sends SCTP data messages to the server program at the peer machine, which will measure the throughput.

Makefile - to build the client and server program

```
CC=gcc

# For SCTPLIB
#CFLAGS = -D_SCTPLIB -g
#LDFLAGS= -lsctpsocket -lsctp -lglib -lstdc++ -lpthread

# For LKSCTP
ROOT=/root
SCTP=${ROOT}/lksctp-2_4_18-0_4_6
CFLAGS= -g -I${SCTP}/test/libc/include -I${SCTP}/test -D_LKSCTP
LDFLAGS=

all: server client

server: sctp_server.c
	$(CC) $(CFLAGS) -o server sctp_server.c $(LDFLAGS)

client: sctp_client.c
	$(CC) $(CFLAGS) -o client sctp_client.c $(LDFLAGS)

clean:
	rm server client
```

sctp_client.c – client program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>

#ifdef _SCTPLIB
#include "ext_socket.h"
#endif

#ifdef _LKSCTP
#include <netinet/sctp.h>
#include <sctp_lib.h>
#endif

#include <string.h>


int main (int argc, const char * argv[])
{
  int sfd;
  int local_port;
  int remote_port;
  int one = 1;
  struct sockaddr_in local_addr, remote_addr;
```

```
  char *buffer;
  fd_set rset;
  int ret;
  int i, num, pktsz;

  struct timezone tz;
  struct timeval p, q;
  double timediff;

#ifdef _LKSCTP
  struct msghdr msg;
  struct iovec iov;
#endif

  if (argc < 6)
  {
     printf("Usage: %s <local port> <remote port> <peer ip> <num> <pktsz>\n",
     argv[0]);
     exit (-1);
  }

  local_port = (int) strtol(argv[1], NULL, 10);
  remote_port = (int) strtol(argv[2], NULL, 10);
  num = (int) strtol(argv[4],NULL,10);
  pktsz = (int) strtol(argv[5],NULL,10);


  /* get sockets */
#ifdef _SCTPLIB
  if ((sfd = ext_socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) < 0)
#else
  if ((sfd = socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) < 0)
#endif
    perror("socket call failed");

  /* bind the sockets to INADDRANY */
  bzero(&local_addr, sizeof(local_addr));
  local_addr.sin_family = AF_INET;
  local_addr.sin_addr.s_addr = INADDR_ANY;
  local_addr.sin_port   = htons(local_port);

#ifdef _SCTPLIB
//  ext_setsockopt(sfd, IPPROTO_SCTP, SCTP_NODELAY, &one, sizeof(one));
#endif

#ifdef _SCTPLIB
  if(ext_bind(sfd, (struct sockaddr *)&local_addr, sizeof(local_addr)) < 0)
#else
  if(bind(sfd, (struct sockaddr *)&local_addr, sizeof(local_addr)) < 0)
#endif
  {
       perror("bind call failed");
      exit (-1);
  }

  bzero(&remote_addr, sizeof(remote_addr));
  remote_addr.sin_family = AF_INET;
  remote_addr.sin_addr.s_addr = inet_addr(argv[3]);
  remote_addr.sin_port   = htons(remote_port);

  // initiate an association without sending data
#ifdef _SCTPLIB
  ret = ext_connect(sfd, (struct sockaddr *)&remote_addr, sizeof(remote_addr));
#else
  ret = connect(sfd, (struct sockaddr *)&remote_addr, sizeof(remote_addr));
#endif

  buffer = malloc(pktsz);
  memset(buffer, 44, pktsz);
  gettimeofday(&p, &tz);
```

```
  printf("num: %d\n", num);

  for (i=0; i<num; i++)
  /* Handle incoming requests */
  {

#ifdef _SCTPLIB
      if (ext_sendto(sfd, buffer, pktsz, 0,
          (struct sockaddr *)&remote_addr, sizeof(remote_addr)) < 0)
#else
      iov.iov_base = buffer;
      iov.iov_len = strlen(buffer) + 1;
      msg.msg_iov = &iov;
      msg.msg_iovlen = 1;
      msg.msg_control = NULL;
      msg.msg_controllen = 0;
      msg.msg_name = (void *) &remote_addr;
      msg.msg_namelen = sizeof(remote_addr);
      if (sendmsg(sfd, &msg, 0)< 0)
#endif
          perror("sendto call failed.\n");
  }

  gettimeofday(&q, &tz);

  timediff = (q.tv_sec - p.tv_sec) * 1000000.0 + q.tv_usec - p.tv_usec;
  printf("Throughput (Mbps):  %9g\n", pktsz * 8 * num / timediff);

  free(buffer);

  // Wait 10 seconds for the peer to receive all the packets before
  // shutting down the association.
  sleep (10);

#ifdef _SCTPLIB
  ext_close(sfd);
#else
  close(sfd);
#endif

  return 0;
}
```

sctp_server.c – server program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifdef _SCTPLIB
#include "ext_socket.h"
#endif

#ifdef _LKSCTP
#include <netinet/sctp.h>
#include <sctp_lib.h>
#endif

#include <sys/time.h>
#include <string.h>

#define BUFFER_SIZE 10000

int main (int argc, const char * argv[])
{
  int sfd, idle_time, backlog, remote_addr_size, length, nfds;
  int port;
  struct sockaddr_in local_addr, remote_addr;
```

```
  char buffer[BUFFER_SIZE];
  fd_set rset;
  int num, pktsz;
  unsigned int expected;
  unsigned int sofar;
  int first = 1;
  struct timeval p, q;
  struct timezone tz;
  double timediff;

#ifdef _LKSCTP
  struct msghdr msg;
  struct iovec iov;
  struct SCTP_cmsghdr cmsghdr;
  char *data;
  union sctp_notification *sn;
#endif

  int one = 1;

  if (argc < 4)
  {
     printf("Usage: %s <sctp port> <num> <pktsz>\n", argv[0]);
     exit (-1);
  }

  port = (int) strtol(argv[1], NULL, 10);
  num = (int) strtol(argv[2], NULL, 10);
  pktsz = (int) strtol(argv[3], NULL, 10);

  expected = num * pktsz;


  /* get sockets */
#ifdef _SCTPLIB
  if ((sfd = ext_socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) < 0)
#else
  if ((sfd = socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) < 0)
#endif

    perror("socket call failed");

#ifdef _SCTPLIB
//    ext_setsockopt(sfd, IPPROTO_SCTP, SCTP_NODELAY, &one, sizeof(one));
#endif

  /* bind the sockets to INADDRANY */
  bzero(&local_addr, sizeof(local_addr));
  local_addr.sin_family = AF_INET;
  local_addr.sin_addr.s_addr = INADDR_ANY;
  local_addr.sin_port   = htons(port);

#ifdef _SCTPLIB
  if(ext_bind(sfd, (struct sockaddr *)&local_addr, sizeof(local_addr)) < 0)
#else
  if(bind(sfd, (struct sockaddr *)&local_addr, sizeof(local_addr)) < 0)
#endif
  {
        perror("bind call failed");
       exit (-1);
  }

  /* make the sockets 'active' */
  backlog = 0; /* it is ignored */
#ifdef _SCTPLIB
  if (ext_listen(sfd, backlog) < 0)
#else
  if (listen(sfd, backlog) < 0)
#endif
    perror("listen call failed");
```

```
  /* set autoclose feature: close idle assocs after 5 seconds */
#ifdef _SCTPLIB
  idle_time = 5;
  if (ext_setsockopt(sfd, IPPROTO_SCTP, SCTP_AUTOCLOSE,
      &idle_time, sizeof(idle_time)) < 0)
  perror("setsockopt SCTP_AUTOCLOSE call failed.");
#endif

  nfds = FD_SETSIZE;
  FD_ZERO(&rset);

  sofar = 0;
  /* Handle incoming requests */

  printf("Expected (bytes): %d\n", expected);

#ifdef _LKSCTP
      iov.iov_base = buffer;
      iov.iov_len = BUFFER_SIZE;
      msg.msg_iov = &iov;
      msg.msg_iovlen = 1;
      msg.msg_control = &cmsghdr;
      msg.msg_controllen = sizeof(cmsghdr);
#endif

//  while (sofar < expected)
  while (1)
  {
    FD_SET(sfd, &rset);

#ifdef _SCTPLIB
    ext_select(nfds, &rset, NULL, NULL, NULL);
#else
    select(nfds, &rset, NULL, NULL, NULL);
#endif

    if (FD_ISSET(sfd, &rset))
    {
      if (first)
      {
         gettimeofday(&p,&tz);
         first = 0;
      }

      remote_addr_size = sizeof(remote_addr);
      bzero(&remote_addr, sizeof(remote_addr));

#ifdef _SCTPLIB
      if ((length =ext_recvfrom(sfd, buffer, pktsz+1,
          0, (struct sockaddr *) &remote_addr, &remote_addr_size)) < 0)
#else
      iov.iov_len = BUFFER_SIZE;
      msg.msg_control = &cmsghdr;
      msg.msg_controllen = sizeof(cmsghdr);

//      msg.msg_name = (void *) &remote_addr;
//      msg.msg_namelen = remote_addr_size;
      //
      if ((length = recvmsg(sfd, &msg, MSG_WAITALL)) < 0)
#endif
          perror("recvfrom call failed");
      else
#ifdef _SCTPLIB
        sofar += length;
#else
      {
          printf("pkt received: (msg_flag %d)\n", msg.msg_flags);

          if (msg.msg_flags & MSG_NOTIFICATION)
         {
               data = (char *) msg.msg_iov[0].iov_base;
```

```
                sn = (union sctp_notification *) data;

                if (sn->h.sn_type == SCTP_ASSOC_CHANGE)
                {
                    if (sn->sn_assoc_change.sac_state == SHUTDOWN_COMPLETE)
                    {
  timediff = (q.tv_sec - p.tv_sec) * 1000000.0 + q.tv_usec - p.tv_usec;
  printf("Throughput (Mbps): %9g\n", expected * 8 / timediff);

#ifdef _SCTPLIB
                       ext_close(sfd);
#else
                        close (sfd);
#endif
                       exit (0);
                    }

                }
            }
            else
            {
                gettimeofday(&q, &tz);
                 sofar += length;
            }
        }
#endif
//      printf("sofar (bytes): %d\n", sofar);
    }
  }

  gettimeofday(&q, &tz);
  timediff = (q.tv_sec - p.tv_sec) * 1000000.0 + q.tv_usec - p.tv_usec;
  printf("Throughput (Mbps): %9g\n", expected * 8 / timediff);

#ifdef _SCTPLIB
  ext_close(sfd);
#else
  close(sfd);
#endif

  return 0;
}
```