NTNU
Norwegian University of
Science and Technology

Performance Evaluation Framework for a SIP-based Telecommunication Call Handling System

Nattanond Sangvanphant

Master in Security and Mobile Computing

Submission date:	June 2009
Supervisor:	Finn Arve Aagesen, ITEM
Co-supervisor:	Mazen Malek Shiaa, ITEM
	Markus Hidell (Asst. Professor), KTH, Sweden

Norwegian University of Science and Technology
Department of Telematics

Problem Description

The research work comprises the study and development of a performance evaluation framework for a telecommunication call handling system that utilizes SIP platform. This call handling system is part of an overall solution provided by Gintel for Virtual PBX. The performance evaluation framework will focus on two main performance characteristics: system throughput (number of calls) and call set-up delay.

1. Make a requirement specification for a performance evaluation framework.
2. Study the possibilities for using existing tools.
3. Study the telecommunication call handling system that will be used.
4. Develop a performance evaluation framework which facilitates throughput and call set-up delay measurements.
5. Work out a selection of traffic load patterns, and make experiments for various call handling system architectures.

Assignment given: 15. January 2009
Supervisor: Finn Arve Aagesen, ITEM

Abstract

Session Initiation Protocol (SIP) has been used for signaling in many Voice over IP (VoIP) applications. Being more cost-effective than conventional circuit-switched systems, IP-based telecommunication systems are extensively employed by many service providers. As these systems gain more popularity, the need for dimensioning of such systems grows correspondingly. Moreover, accurate information about system capacity is necessary for future improvements of the system, as well as service provision and implementation planning. For these reasons, a solution supporting system performance evaluation is useful and beneficial in several ways.

The goal of this research was to develop a performance evaluation framework for a SIP-based telecommunication system. The developed framework facilitates measurements of the maximum number of requests which can be processed by a system, and the amount of time required for call session establishment. With a user-friendly interface, the framework enables system testers to perform experiments using simulated SIP traffics, as well as to deal with results interpretation easily.

In order to achieve the objective, studies of related technologies and available tools for SIP traffic generation have been carried out. Afterwards, the performance evaluation framework is designed and implemented. Lastly, the developed framework is used for evaluating the performance of EasyVPaBX, a SIP-based call handling system, in various system configurations.

Keywords: SIP, Performance, Evaluation, Dimensioning, Measurement

Acknowledgement

This thesis is a part of the Master of Science in Security and Mobile Computing degree in the NordSecMob program. It is done at the Department of Telematics (ITEM), Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisors, Mazen Malek Shiaa, and Professor Finn Arve Aagesen for all their guidance, assistance, and valuable feedback throughout the period of this thesis. Additionally, I would like to thank Asst.Prof. Markus Hidell at the Royal Institute of Technology (KTH) in Stockholm, Sweden for his supervision. Special thanks to Kirati Suttikulpanich for his beneficial ideas and advice.

Furthermore, I am indebted to the NordSecMob consortium and the European Commission for giving me the opportunity to participate in this program. I would also like to thank program coordinators, Eija Kujanpaa, Assoc.Prof. Peter Sjodin, May-Britt Eklund-Larsson, and Mona Nordaune for their helpful assistance. Special thanks to NordSecMob friends and classmates for their encouragement.

Lastly, I would like to thank my family and friends in Thailand for their love and support throughout the time that I have to be away from home.

Trondheim, June 2009

Nattanond Sangvanphant

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Background

Session Initiation Protocol (SIP [1]) has become an industry standard for Voice over IP (VoIP) applications. IP-based telecommunication systems are widely used because of their cost-effectiveness compared to conventional circuit-switched systems. As a basis for cost-effective implementations, knowledge about the performance of the SIP-system is needed.

Many aspects of performance have been studied and certain evaluation approaches are suggested. According to [2], the most important metric of SIP performance is the number of requests that the server can process successfully (throughput). Another significant aspect is the latency needed for setting up the session since SIP has sub-second timing requirements that affect the overall performance of the system [3].

A common method used for SIP system performance evaluation is to simulate activities of users generating SIP calls to the target system. There are some open source projects implementing SIP traffic generator for evaluation purpose such as SIPp [4] which is used in many research works (e.g. [5], [6], [7], and [8]). Nevertheless, using such tools demands deep knowledge and understandings. Moreover, the text-based interface provided requires substantial efforts for controlling the tool and the interpretation of the results.

For these reasons, an easy-to-use framework for performance evaluation of a SIP-based system was needed. A user-friendly interface was needed for efficient specification and performance of experiments as well as easy results interpretation.

1.2 Objective and Scope

The goal of this thesis is to design and implement a user-friendly performance evaluation framework for a SIP-based telecommunication system. The framework aims to be used in the dimensioning of the system.

An overview of the performance evaluation framework is illustrated in Figure 1-1. User configures and controls an evaluation test through a standalone application. The traffic generator creates SIP requests destined to the target system. An intermediate node acts as a SIP proxy is responsible for forwarding SIP traffic. The evaluation results are presented to the user in graphical formats.

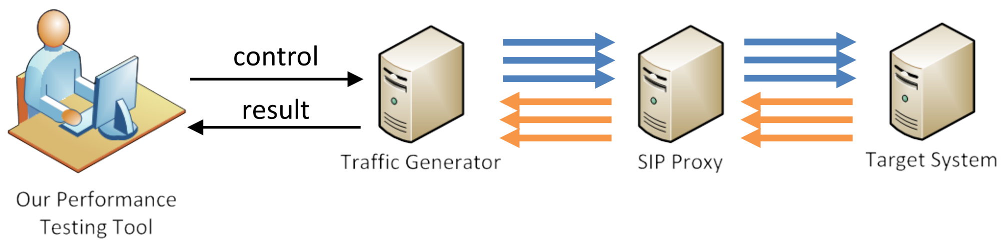


Figure 1-1: Overview of the Performance Evaluation Framework

The thesis focuses on two metrics, 1) system capacity and 2) the time needed for call establishment. These two aspects of performance are measured using simulated SIP traffics towards a target system. Considering the SIP traffic generation, there are two issues that must be noted. First, SIP call sessions produced by the framework do not contain any media data. For simplicity, the thesis focuses only on the control plane of call sessions. Therefore, performance evaluation performed by the framework is based on SIP signaling alone. Call sessions with media streams may have certain degrees of influence on the performance, but this is not taken into consideration in this thesis work. Second, implementation of the traffic generator and the SIP proxy used in the framework are not parts of this thesis. The two components are chosen from existing tools available as open source projects.

In addition to developing an evaluation framework, the thesis includes a study of the EasyVPaBX system and its performance. Various computers with different computational power are used for the experiments. Target systems are set up from a number of system configurations. EasyVPaBX as well as other necessary software components are installed in the target systems. We apply the developed framework to evaluate the performance of these target systems.

1.3 Research Work Activities

The research work performed in this thesis is divided into three steps, *study*, *develop*, *experiment*, and *evaluate*. Details of each step are given as follow.

Study

The first step includes a study of related technologies, available tools to be used as SIP traffic generator and SIP proxy, as well as the target system. As for the SIP protocol, we focused on SIP messages, transactions, and roles of SIP participants. Two tools, SIPp and OpenSIPS were studied. We configured a simple SIP system based on these tools and some tests were performed.

The target system, EasyVPaBX, is a SIP-based system providing PBX functionalities. We studied its architecture, required working environment, and deployment and operation of the system.

Develop

With the available tools studied in the previous step, we designed and implemented the performance evaluation framework. We analyzed the requirement specifications of the framework. After that, a detailed design of the application was made. We continued with the implementation step which includes defining SIP scenarios, managing the interactions with a SIP traffic generator, processing captured results, and drawing graphical diagrams.

Experiment

From the knowledge of the EasyVPaBX obtained in the first step, we identified certain system configurations for *target systems*. The target systems were prepared with necessary components such as an application server and a database. The EasyVPaBX was deployed on the target systems. In addition to the target systems, we configured another system with the performance evaluation framework as well as other elements (i.e. a SIP proxy and a call recipient) to act as a *tester*.

Repetitive experiments on performance evaluation of the target systems (EasyVPaBX) using the framework were carried out. We then compared the result data gathered with the framework.

Evaluate

Following the development process, we evaluation the performance evaluation framework. We verified the framework we have developed against the requirement specification. We compared the framework and the existing tool, SIPp, for the usability and user-friendliness. In addition, we discussed some issues regarding the developed framework.

1.4 Structure of the Report

Chapter 1: Introduction introduces a background motivation of this thesis. A scope of the work, an overview of the performance evaluation framework developed in this thesis, and the method used to accomplish the work are also presented.

Chapter 2: Background gives explanations of related technologies and tools used in this thesis work including SIP, SIPp, OpenSIPS, and EasyVPaBX.

Chapter 3: Experiments on SIPp and OpenSIPS presents numbers of experiments with the tools SIPp and OpenSIPS. Certain scenarios are setup and evaluation tests are carried out.

Chapter 4: SIP Performance Evaluation Framework explains the requirements of the framework and gives details of the design and implementation of the performance evaluation tool.

Chapter 5: EasyVPaBX Performance Evaluation describes the details of system configurations to be evaluated. The performance results from multiple experiments are presented. Comparison of the results is given at the end of the chapter.

Chapter 6: Framework Evaluation and Discussion presents the evaluation of the framework developed. It also discusses some issues of the framework.

Chapter 7: Conclusions summarizes the work in this thesis. In addition, interesting ideas are pointed out for potential future works.

2 Background

This chapter explains related technologies and tools used in this project. We start with section 2.1 explaining Sesstion Initiation Protocol (SIP), a signaling protocol providing communication session setting up and tearing down. We continue with the SIPp tool used for SIP performance testing in section 2.2. Next, section 2.3 gives details about the OpenSIPS project which implements SIP servers. Lastly, we describe the EasyVPaBX, product of Gintel AS, which is the target system to be evaluated in section 2.4.

2.1 Session Initiation Protocol (SIP)

Session Initiation Protocol (SIP) [1] is a signaling protocol for session management. Typically, the main functions of SIP include locating end points, contacting and exchanging information between end points for session establishment, modifying, and terminating multimedia sessions with one or multiple participants [9]. SIP works in an application layer and is independent of underlying transport layer. Therefore, it can run on top of different transport protocols e.g. TCP, UDP.

SIP can be easily incorporated with various types of Internet applications. Since SIP only deals with managing session, it is not an entire communication system by itself. SIP is independent of the type of multimedia session handled and of the mechanism used to describe the session [10]. Hence, SIP should be used with other protocols to provide a complete service. These protocols include the Real-time Transport Protocol (RTP) [11] for transporting real-time data such as audio and video, the Real-Time Streaming Protocol (RTSP) [12] for media streaming control, and the Session Description Protocol (SDP) [13] which is used to describe multimedia communication sessions for session negotiation purpose. Nevertheless, SIP is not dependent on any of them. Introducing new SIP applications does not require changes to the network infrastructure. For these reasons, SIP has been widely acknowledged as the industry standard for voice over IP (VoIP) protocols.

2.1.1 SIP User Identifier

SIP allows SIP user agents (see section 2.1.2.1) on the Internet to locate each other. SIP provides name mapping and redirection service which allows user mobility. The namespace maps between user identifier and the current location of the user. Therefore, users can use a single identifier regardless their network location. A SIP participant is identified by a Uniform Resource Identifier (URI). This URI acts as a contact number for the participant. The syntax of SIP URIs is *sip:username@host:port*, where *host* is the domain name of the SIP service provider, and *port* is the port number which listens to SIP requests. For secure transmission mode, *sips:* is used instead of *sip:* to represent the transportation over the Transport Layer Security (TLS) protocol.

2.1.2 SIP Components

According to RFC 3261 [1], the SIP architecture consists of different SIP components. These components interact with each other in a SIP scenario. There are two main components in a SIP system: *user agents* and *SIP servers*.

2.1.2.1 User Agents

A User agent (UA) is an end-point entity which creates or receives SIP messages. End users interact with a user agent through an interface provided by the agent. By means of UA, end users can initiate, hold/unhold, transfer, and answer/reject a SIP call.

A SIP UA can be divided into two roles, a *User Agent Client (UAC)*, and a *User Agent Server (UAS)*. Within a SIP transaction, UAC generates SIP requests, while UAS receives the requests, produces SIP response, and sends it back to UAC. A single UA can function as both UAC and UAS.

SIP UA is available in the form of hardware as well as software. Hardware-based user agent includes SIP-enabled mobile phone and SIP phones from Cisco, Linksys, Aastra, etc. On the other hand, software-based agent is a program which allow user to make or receive calls from a computer. Examples of SIP soft-phones are SJPhone (www.sjlabs.com), Ekiga (www.gnomemeeting.org), and X-Lite (www.counterpath.com).

2.1.2.2 SIP Servers

Even though two SIP user agents can communicate with each other directly, it is not practical for a real-world scenario. Therefore, SIP servers are needed as intermediary network elements to facilitate end points discovery as well as process and forward SIP requests. Four types of SIP servers are

- **Proxy Server :** SIP proxy server acts on behalf of clients to forward SIP requests. It looks up a location of the destination and routes the traffic to the corresponding recipient. Two types of proxy server are *stateful proxy* and *stateless proxy*. The stateful proxy maintains the state of the transaction while the stateless proxy relays SIP requests without keeping any state information
- **Redirect Server :** SIP redirect server generates redirection answers to indicate the client to try a different route. This happens when a recipient has moved from its original location.
  Redirect server does not forward the requests to the destination itself. Instead, it guides the caller to the next destination by replying with a location of the target URI. The client uses the received route information to send the request to the destination.
- **Registrar Server :** A registrar processes REGISTER requests from users to update their locations. A SIP URI is bound to its current location temporarily. Therefore, every time a user logins, the UA sends a registration message containing the location of the user to the registrar
- **Location Server :** A location server provides current location information of SIP users. It cooperates with a registrar to maintain the mapping between URIs and corresponding locations. The addresses registered at a registrar are stored in a location server.

2.1.3 SIP Messages

SIP is a text-based protocol based on the UTF-8 character set. SIP messages consist of a *start-line*, header fields, one empty line indicating the end of the header fields, and an optional message body. The start-line can be either a *Request-line* or a *Status-line* depends on the type of the message. The syntax of SIP messages and header fields are similar to messages in the Hypertext Transfer Protocol (HTTP). SIP messages can be categorized into two types, *SIP requests* and *SIP responses*.

2.1.3.1 SIP Requests

A SIP request is a SIP message which contains a Request-line as a first line. A Request-Line consists of a method name, a Request-URI, and the protocol version. A single space is needed between a method name and a Request-URI, as well as between the URI and the version number. This Request-Line ends with a CRLF (Carriage-Return, Line-Feed) character.

Only six methods are defined in RFC 3261 [1] , they are INVITE, ACK, BYE, CANCEL, REGISTER, and OPTIONS. Additional methods have been defined later to support more complicated services. SIP methods and their descriptions can be found in Table D-1, Appendix D.

An example of a SIP request is shown in Figure 2-1 [1]. The INVITE message is created by *Alice (alice@atlanta.com)* to invite *Bob (bob@biloxi.com)* to join a new call session. The Request-Line contains the method name *INVITE*, the request URI, and *SIP/2.0* as the SIP version. The rest of the lines are message headers which are explained later. Note that the content (SDP message) is not shown in the figure.

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com>
Content-Type: application/sdp
Content-Length: 142
```

Figure 2-1: Example of a SIP INVITE request (adapted from [1])

2.1.3.2 SIP Responses

A SIP response has a *Status-Line* as its start-line. A *Status-Line* consists of the protocol version, a numeric *Status-Code*, and a *Reason-Phrase*. In the same fashion as the *Request-Line* for a SIP request, each element in a *Status-Line* is separated by a single space, and a CRLF character at the end of the line.

Figure 2-2 below shows an example of a SIP response. The response message is the result associated to an *INVITE* message initiated by *Alice (alice@atlanta.com)* to *Bob (bob@biloxi.com)*. This response contains *200* as a *Status-Code*, and *OK* as a *Reason-Phrase*.

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP server10.biloxi.com
;branch=z9hG4bKnashds8;received=192.0.2.3
Via: SIP/2.0/UDP bigbox3.site3.atlanta.com
;branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2
Via: SIP/2.0/UDP pc33.atlanta.com
;branch=z9hG4bK776asdhds ;received=192.0.2.1
To: Bob <sip:bob@biloxi.com>;tag=a6c85cf
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:bob@192.0.2.4>
```

Figure 2-2: Example of a SIP response (adapted from [1])

The *Status-Code* is an integer with 3 digits. This code indicates the result of an attempt of a request. The *Reason-Phrase*, which is intended for human user, gives short description explaining the *Status-Code*.

SIP defines the classes of response using the first digit of the *Status-Code*. The other two digits vary for different results. For instance, a code between 100 and 199 is in the class *1xx response*. Six classes of response and their descriptions can be found in Table D-2, Appendix D.

Some examples of SIP responses are

- ***100 Trying*** indicates that the request is successfully received and being processed by the server.
- ***180 Ringing*** indicates that the UA is trying to alert the user.
- ***200 OK*** means that the request has been successful.
- ***400 Bad Request*** indicates a syntax error in the request
- ***404 Not Found*** informs that the requested user does not exist.
- ***500 Server Internal Error*** indicates the unexpected problem at the server side. The client may retry after some time.

2.1.4 SIP Headers

Apart from a *Status-line* of a *Request-line* in the first line of a SIP message, there is a set of SIP headers in the following lines. Each header field consists of a field-name, a colon, and a field-value. Whitespaces are allowed on either side of the colon. However, it is suggested in RFC 3261 [1] that the implementations use only one single space between the colon and the field-value. Examples can be found in SIP messages in Figure 2-1 and Figure 2-2. (See Table D-3 for lists of SIP headers)

2.1.5 SIP Conversation

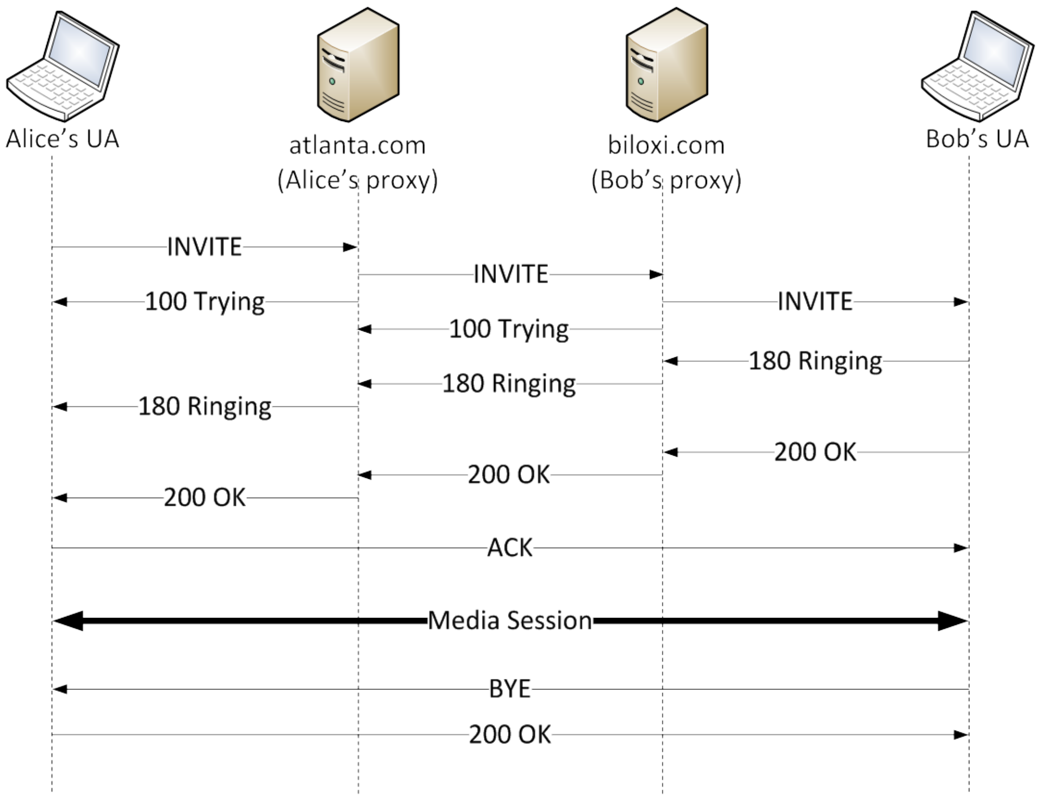


Figure 2-3: Example of SIP transactions (adapted from [1])

Figure 2-3 shows an example of SIP conversation between multiple participants. The SIP session is established between *Alice* and *Bob*. First, Alice initiates a call by sending an *INVITE* message to Bob through a proxy server in her domain. Alice's proxy server then finds out the address of Bob's proxy server, and contacts that proxy server to forward the invitation to Bob. Alice's proxy also sends a *100 Trying* message back to Alice's UA to inform that it is performing the request.

Once the *INVITE* message is received by Bob's UA, it replies with a *180 Ringing* message. When Bob picks up the call, a *200 OK* message is generated and sent back to Alice's UA via the two proxies. After Alice's UA gets the *200 OK*, it acknowledges Bob's UA by sending an *ACK* message. From this point on, the session has been established. Further media sessions can begin right after this.

When Bob hangs up the call, his UA generates a *BYE* message and sends it to Alice's UA to terminate the session. Alice's UA replies with a *200 OK* message, and the session is terminated at both sides.

2.2 SIPp - SIP Performance Testing Tool

SIPp[4] is a free performance testing tool for the SIP protocol. The SIPp project has been initiated by Richard Gayraud, and Olivier Jacques from Hewlett Packard Company. It is now an open source project being developed by many contributors. SIPp can be used to test SIP equipments by generating SIP traffic and recording performance measurements. The measurements provided include response time, call time, number of successful calls and failed calls, number of ongoing calls, etc.

SIPp can generate multiple SIP calls to a remote system. In addition, many instances of SIPp can be run simultaneously as long as they are bound to different socket on the same machine. SIPp tool is based on the command line interface. The interactive text-based results are displayed in several screens. For the performance testing purpose, SIPp provides the ability to periodically save the statistics as well as other logs to output files. Those files are of the comma-separated-values or CSV format.

SIPp tool is available on almost all UNIX platforms including Linux. It can be also run on the Windows platform. However, the Windows version of SIPp may not be suitable for testing purpose since it cannot handle high performances. To reach high performances, the UNIX systems are recommended.

The following parts describe in details the important features of SIPp.

2.2.1 Running SIPp

SIPp can be started from the command line. Several startup parameters are passed to the SIPp process to configure it. The format of the command is

```
sipp remote_host[:remote_port] [options]
```

The *remote_host* and *remote_port* is the target of the generated traffic. After the execution, SIPp starts generating and sending sequences of SIP messages to the target. The sequence of SIP messages are defined by the embedded scenarios or the external XML scenario file. Important parameters used when starting SIPp are listed in Table E-1, Appendix E.

After an instance of SIPp tool has been created, the display screen shows the diagram of SIP messages as well as message counters. Further interactions with the tool are addressed in the next section.

2.2.2 Controlling SIPp

Once a SIPp instance has been created and running, it can be controlled using two sets of commands, the hot keys and the interactive commands. Hot keys refer to single key commands which can be entered at any time. See Table E-3 in Appendix E, for the hot keys supported by SIPp.

Interactive commands offer more flexibility than the hot keys. They are single line commands which require the command mode. The command mode can be entered by the hot key “c”. The list of interactive commands is shown in Table E-2.

To control the SIPp tool, there are two approaches provided. The first way is using the interactive command line. The SIPp process can be controlled by passing the commands using keyboard. Hot keys can be executed directly in any screens, while other commands must be used once the process is in the command mode.

The other way to execute the commands is through the controlling UDP socket. Each SIPp instance listens to a specific UDP socket. This allows SIPp to be remote-controlled. The commands can be sent to this controlling socket in order to control the traffic generation. The controlling socket can accept both hot keys and interactive command described above. The commands must be prefaced with the letter *c*, for instance, *cset rate 5.*

2.2.3 Call Rate

The number of calls generated by SIPp in a specific amount of time can be adjusted. This is called the *call rate*. The startup parameters specify the initial call rate of the traffic. Once a SIPp instance is running, the call rate can be controlled through the hot keys, and the interactive commands.

The unit of rate which is controlled by those commands is number of calls per one period of time. The default value of the period if not specified otherwise is 1000 ms or 1 second. However, this period can be adjusted by the startup parameters.

2.2.4 XML Scenario File

SIPp can take an external XML scenario file as a startup parameter. The scenario file specifies the sequence of SIP messages to be sent and received. In addition, it also defines starting and stopping points for timers, actions, logging events, etc. The scenario file are passed through -sf option when starting SIPp.

The XML scenario file contains *commands* which are used to control SIPp behavior. Within each command tag, there are numbers of *attributes* for further adjustment. SIPp also provides *keywords* for constructing SIP messages in the scenario file. The following subsections address more details about these elements.

2.2.4.1 Commands

Commands are used to instruct SIPp to perform certain tasks e.g. sending and receiving SIP messages. The commands are in XML tags format. Some basic commands are *<send>, <recv>,* and *<pause>*.

- **<send>** This command is used to send SIP messages to the target system. Inside the *<send>* tag, a SIP message must be enclosed *<![CDATA[* and *]]>.*
- **<recv>** makes SIPp wait for the specified SIP message.
- **<pause>** tells SIPp to pause the scenario for some amount of time. During the pause period, SIPp will not send any messages. This command can be used to emulate the call length. When there is no attribute used with the *<pause>* command, SIPp will use the value specified by the startup parameter *-d* as the delay.

Apart from the basic commands, there are commands regarding distribution of timers. *<ResponseTimeRepartition>* and *<CallLengthRepartition>* are used to define the intervals of distribution counters. The distribution counters will be shown in the display window. They can also be written to an output file if *-trace_stat* parameter is used.

2.2.4.2 Attributes

Each command has its own attributes for fine-tuning the command as shown in Table E-4, Appendix E. Besides the unique attributes for certain commands, there are numbers of attributes which are applicable for every command. The common attributes are listed in Table E-5, Appendix E.

2.2.4.3 Keywords

There are some keywords available for constructing SIP messages. Keywords are always in square brackets, *[ ]*. These keywords will be replaced by some values before the messages are sent. See Table E-6 in Appendix E for available keywords.

2.2.4.4 Injection File

Apart from the keywords, the values can be injected using an external CSV file. The file name has to be specified when executing SIPp with the parameter *-inf*. To refer to the values, the keyword *[field]* is used in the XML scenario file. The order of usage can be defined in the first line of the injection file as *SEQUENTIAL* or *RANDOM*.

2.2.5 Statistics

SIPp offers timers and many counters to keep track of the calls generated. These statistics values can be saved to output files for further use. The available timers and counters are as follows.

2.2.5.1 Response Time Timers

SIPp provides five timers to measure the time between two SIPp commands. The commands are *send, recv*, and *nop* which are specified in the XML scenario file. In each command, the timers can be started or stopped using *start_rtd* and *rtd* attributes of the command.

The measured response times are shown in display screens. In addition, these timers can also be saved in a CSV file by specifying startup parameters, i.e., *-trace_rtt* and *-trace_stat*.

2.2.5.2 Counters

There are number of counters provided by SIPp. Two types of counters are *periodic (P)* and *cumulated (C)* counters. The periodic counters are reset every time after the statistic row is updated. The counters are dumped in an output file of CSV format using *-trace_stat* parameter at startup. To adjust the frequency of dumping these values to the file, *-fd* parameter is used. Some important counters are shown in Table E-7, Appendix E.

2.2.6 Output Files

SIPp provides various types of output files. To enable the output files, the startup parameters need to be set. Some of the output files are stored in CSV format. CSV format or comma-separated-values file is used for storing data in a table form. Each line of a CSV file represents one row in the table. Within a line, data fields are separated by commas. However, different separators can also be used instead of a comma, for instance, a semicolon. The output file can be imported to a spreadsheet application for further analysis or presentation. List of output files are shown in Table E-8. Among all output files, two files which store the counters and timers values are the statistics file, and the response time file.

2.2.6.1 Statistics File

The statistics file consists of all counters values. The first line contains a header line with all counter names. All other lines are the counter values at the time of the report. The file is dumped periodically. The frequency of the report can be set by *-fd* parameter.

2.2.6.2 Response Time File

This file contains the timer values measured by SIPp. A measure is triggered by a message reception defined in an XML scenario file. Each line shows the recorded response time together with the timer number (1-5).

As for the performance testing purpose, especially with high call rate, heavy traces may affect the performance result. For example, usage of *-trace_msg* and *-trace_logs* are suggested to be limited to debugging purpose only.

2.3 OpenSIPS - Open Source SIP Server

OpenSIPS (Open SIP Server) [14] is an open source implementation of a SIP server. The project has been started as the older name, OpenSER, by the *Voice System* [1] team in 2005. OpenSIPS offers many features such as SIP registrar server, SIP proxy/router, SIP redirect server, SIP load-balancer/dispatcher, etc. In addition to its functionalities, OpenSIPS is flexible and customizable for various solutions. According to the documentation provided in the website, OpenSIPS is a reliable and high-performance SIP server. It is one of the fastest SIP servers, with performance up to hundreds of call setups per second. And it can run on embedded systems with limited resources.

OpenSIPS provides a powerful scripting language for routing logic. OpenSIPS can be configured by editing the configuration script which is loaded at execution time. Certain important keywords, parameters, and functions are given in Appendix F. The complete documentation of the scripting can be found in the website.

Keywords

The keywords are specific to SIP messages received by the server. These keywords can be used in *if* statements for conditional decision. Significant keywords are listed in Table F-1, Appendix F.

Parameters

The parameters are used for configuring the SIP server such as enabling or disabling certain functionalities. Table F-2 in Appendix F contains key parameters of OpenSIPS.

Functions

Functions can be used within the *route* blocks to manipulate SIP messages. The route blocks are the core routing logic executed when a SIP message is received. The functions used for configuring a simple SIP proxy can be found in Table F-3.

[1] http://www.voice-system.ro/

2.4 EasyVPaBX

EasyVPaBX is a software framework for operators and service providers to deliver advanced virtual PaBX solutions for businesses and enterprises. The service controls the call behaviors including internal, outgoing, and incoming calls to employees, as well as company common numbers. The engine of EasyVPaBX is based on the session initiation protocol (SIP). EasyVPaBX is a product of Gintel AS [15], a developer of advanced application software for telecommunications operators and service providers.

EasyVPaBX can be considered as a back-to-back user agent (B2BUA). B2BUA is a SIP element which resides between the two end points of a call session. The session is divided into two call legs. B2BUA acts as a user agent server (UAS) for the session initiator and as a user agent client (UAC) to the destination recipient.

EasyVPaBX offers various features such as private numbering plans, screening/barring schemes, group numbers, call forwarding, call transfer, conference, and many more. The logic of PaBX services can be custom-tailored for each customer and instantly deployed on the system. User-friendly tools are available for easy configurations including switchboard operation, service provisioning, and user administration. Beside several features, EasyVPaBX can be composed by adding service packages from feature modules, thus it is a flexible solution for both the service providers and the customers. The entire framework is hosted by the service provider, requiring no hardware on customer premises [16].

2.4.1 PaBX and Virtual PaBX

Private Automatic Branch Exchange (PaBX) is a telephone exchange device which serves as a point of entry into the public switched telephone network (PSTN). It can handle calls between users in the same organization (internal calls) as well as calls to/from the outside through the PSTN lines. PaBX is less expensive than connecting an external telephone line to every telephone in the organization. Therefore, it is a popular choice for companies and businesses. [17]

Virtual PaBX or hosted PaBX system, in contrast, provides similar services as of PaBX systems without requiring customers to buy and install on-site equipment. A telephone company with switching equipment can be a PaBX service provider. Hence, customer organizations do not need to invest in their own PaBX devices, but instead purchase services from the service provider. In addition, the provider can use the same equipment for multiple PaBX accounts. Furthermore, the hosted PaBX service promotes user mobility. A company can be geographically distributed while using one single number for the entire company. [18]

2.4.2 Key Features

EasyVPaBX implements classical PaBX features as well as many others. Certain features of the EasyVPaBX are listed below [16].

- Automated call handling
- Call barring/screening
- Call diversion to a switchboard
- Corporate main numbers
- Call forwarding, transfer, conference
- Self-service configuration for service provider, company, switchboard operator and end-user.

2.4.3 System Architecture

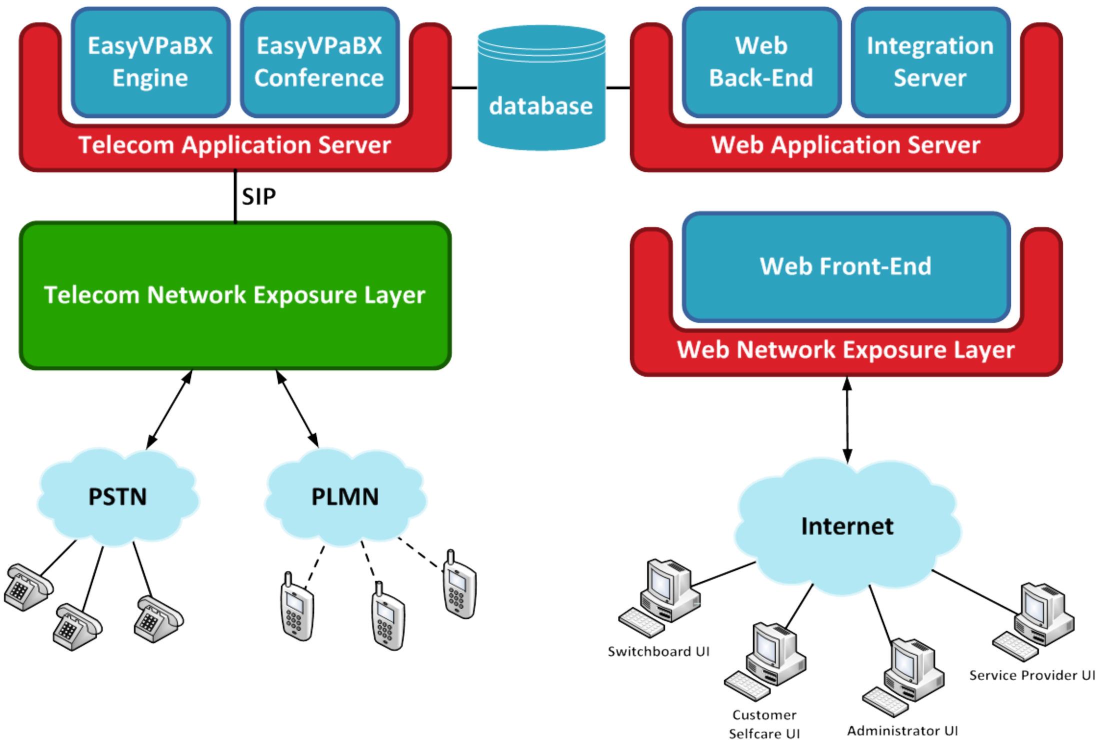


Figure 2-4: EasyVPaBX Architecture (adapted from [19])

The overall architecture of EasyVPaBX is illustrated in Figure 2-4. The components of EasyVPaBX are denoted by blue boxes. The architecture can be divided into two parts, the telecom part on the left-hand side, and the web part on the right-hand side. The telecom part carries out the call handling features, while the web part provides user interfaces for system configurations. The two parts are based on underlying layers which are independent of the EasyVPaBX components.

The database in the middle is connected to both the telecom and the web parts. It is used to store customer as well as user data including user numbers, application logic strategies, profiles, parameters, statistics counters, etc.

The network exposure layer allows interactions between applications in the upper layer, and network resources in the lower layer. This layer communicates with the telecom application server using SIP protocol. There are the *telecom network exposure layer* in the telecom part, and the *web exposure layer* in the web part.

2.4.3.1 Telecom Network Exposure Layer

The telecom network exposure layer contains network gateway equipments. This layer enables the telecom application to access the networks below e.g. public switched telephone network (PSTN), and public land mobile network (PLMN). The telephone and mobile network are connected to the telecom server through the devices in this layer.

Since the EasyVPaBX components running in the telecom application server are based on SIP protocol, the gateway in this layer is responsible for conversion between SIP traffics and telephone traffics.

2.4.3.2 Web Network Exposure Layer

The web network exposure layer contains a web server, and a web front-end. The web server handles HTTP requests from the Internet. This web server could be a Tomcat [20] server or similar servers. The web front-end provides user interfaces for configurations and monitoring. The user interfaces include the Service Provider UI, the Administrator UI, the Switchboard UI, and the Customer UI.

The application server layer provides execution environment for service applications. In this layer, there are the *telecom application server* in the telecom part, and the *web application server* in the web part.

2.4.3.3 Telecom Application Server

Call handling and network traffic handling functionality is performed in this telecom application server. On top of the application server, the EasyVPaBX components are deployed and executed.The EasyVPaBX telecom application components are the *EasyVPaBX Engine* which processes incoming and outgoing calls, and the *EasyVPaBX Conference* which is responsible for conference call services. These components are SIP-based applications. Hence, the application server should be SIP- enhanced to support the connection between the applications and the telecom network exposure layer. One example of the telecom application server is the BEA WebLogic SIP Server [21].

2.4.3.4 Web Application Server

The web application server hosts the *web back-end* and the *integration server* applications. This server could be a Tomcat server or any similar J2EE application server. The integration server provides interfaces to interact with external systems. The web back-end allows the web front-end to access the internal database and communicate with the telecom applications.

3
Experiment on SIPp and OpenSIPS

In the last chapter, we had a glance on the capabilities of the SIPp and OpenSIPS tools as well as how to use them. In this chapter, we continue on with some experiments with them. In order to get familiar with SIPp and OpenSIPS tools, we install and run them on a personal computer. Section 3.1 describes the software and environment used in the experiment. In section 3.2, we set up two scenarios using the tools. Numbers of performance measurement are performed under these scenarios. Various results from different configurations are presented in section 3.3. We end this chapter with some discussions regarding the tools in section 3.4.

3.1 Installation

3.1.1 Environment

The experiments are done on a computer with following specifications.

- Intel® Core™2 Duo Mobile Processor T5300 1.73 GHz
- 1.5 GB DDR2 RAM
- Ubuntu 8.10 Operating System

3.1.2 SIPp

We obtain the SIPp tool 3.1 (*sip.3.1.src.tar.gz*) from http://sipp.sourceforge.net. The SIPp used in this thesis is compiled without SIP authentication and TLS (Transport Layer Security) support. The source file includes a make file for easy compilation. Once the installation is completed, the executable binary file is automatically copied to user binary directory (*/usr/local/bin/*). Therefore, SIPp can be started by calling *sipp* in the command line.

3.1.3 OpenSIPS

We compiled and installed OpenSIPS version 1.4.4 using the source file (*openSIPS-1.4.4-notls-src.tar.gz*) downloaded from http://opensips.org. We specified */usr/local/* as the prefix for compilation and installation. The default configuration file is put into */usr/local/etc/opensips/opensips.cfg*. And the executable binary file is at */usr/local/sbin/opensips*.

3.2 Scenarios

After we have installed the tools, we set up two communication scenarios. For the first scenario, we use only the SIPp tool in the scenario. We connect two instances of SIPp together, so the SIP participants talk to each other directly. With this set up, we can see how the SIPp tool performs.

As for the second one, we insert the OpenSIPS as a SIP proxy between the two participants. This scenario is more similar to the practical SIP-based communication system.

3.2.1 UAC and UAS

We start with the scenario which makes use only the SIPp tool. In this scenario, the caller (UAC) knows where the callee (UAS) is located. The caller initiates a SIP session directly to the caller. Therefore, no intermediate proxy servers are needed. To simulate the caller and the callee, two instances of SIPp are executed on the same system. Therefore, they have to be bound to different addresses (i.e. IP address and/or port number). Figure 3-1 shows SIP messages exchanged between the two parties.

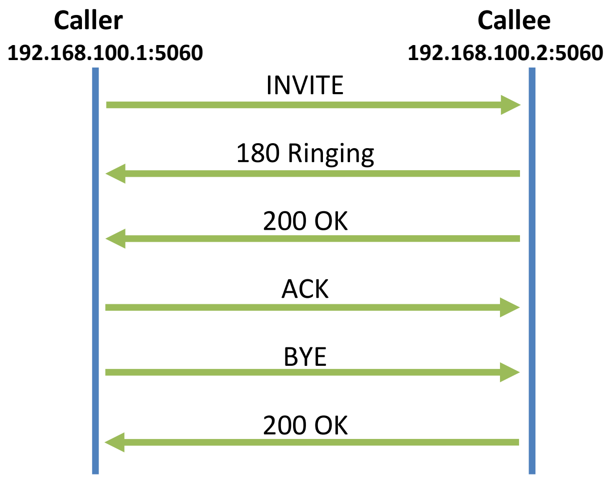
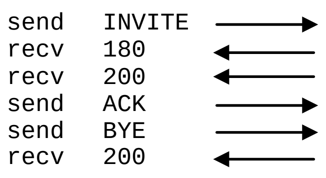


Figure 3-1: SIP messages between UAC and UAS

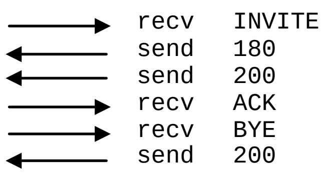
From Figure 3-1, the caller first sends out an INVITE message. Then the callee replies with a *180 Ringing* response and a *200 OK* response respectively. After that, the caller sends an ACK back to the callee. At this point, the SIP session has been established. In the real call, the media streams begin from this point on. To emulate the call session, the caller waits for some time and sends a *BYE* message to the callee to end the session. The callee replies with a *200 OK* message, and the call is terminated.

We created two XML scenario files (*caller.xml*, and *callee.xml*) for SIP message sequences at both parties. The diagram shown in Figure 3-2 illustrates a sequence of commands in the scenario file for the caller. In addition, Figure 3-3 shows commands for the callee.



```
send   INVITE
recv   180
recv   200
send   ACK
send   BYE
recv   200
```

Figure 3-2: Sequence of commands (simple UAC)



```
recv   INVITE
send   180
send   200
recv   ACK
recv   BYE
send   200
```

Figure 3-3: Sequence of commands (simple UAS)

Using two virtual interfaces on the same machine, we bind the caller and the callee to 192.168.100.1 and 192.168.100.2 at port 5060 respectively. As the calls are generated by the caller, we control the call rate at the caller side, and also use the statistic file produced by the caller. The target address of the caller is 192.168.100.2 at port 5060 (callee). The command for starting the caller is

sipp -sf caller.xml 192.168.100.2:5060 -i 192.168.100.1 -p 5060 -r 100 -trace_stat

Likewise, the target address of the callee instance is 192.168.100.1 at port 5060 (caller). Therefore, the command for starting the callee is

sipp -sf callee.xml 192.168.100.1:5060 -i 192.168.100.2 -p 5060

We set the caller to generate SIP calls at certain rates using *-r* parameter. Call rates are of calls per second (cps). With the *-trace_stat* parameter, The caller periodically saves the counters and timers into the statistics file. The interval between each update can be configured using *-fd* parameter.

The statistics file produced by the SIPp process is in the CSV format. The file is in tabular form separated by a semicolon. Diagrams showing the number of successful call attempts and the call response times are presented in section 3.3.

3.2.2 UAC and UAS with a SIP proxy

After we have set up the UAC and UAS using SIPp in the previous scenario, we add another component to the scenario. In regular SIP systems, the caller has no information about the exact location (i.e. IP address and port number) of the callee. In order to make it more realistic, we introduce an intermediate party, a *SIP proxy server*. A proxy server is responsible for forwarding messages to corresponding parties. Hence, the caller and callee only need to know their SIP proxy server.

The three parties, the caller, the callee, and the proxy exchange SIP messages as illustrated in Figure 3-4. At the caller and the callee points of view, the sequence of SIP messages remains the same as of the first scenario. Therefore, we can use the same scenario files. However, the *100 Trying* message generated by the proxy server is sent to the caller after the *INVITE* message. Hence, we make a minor change in the caller scenario file in order to support this message.

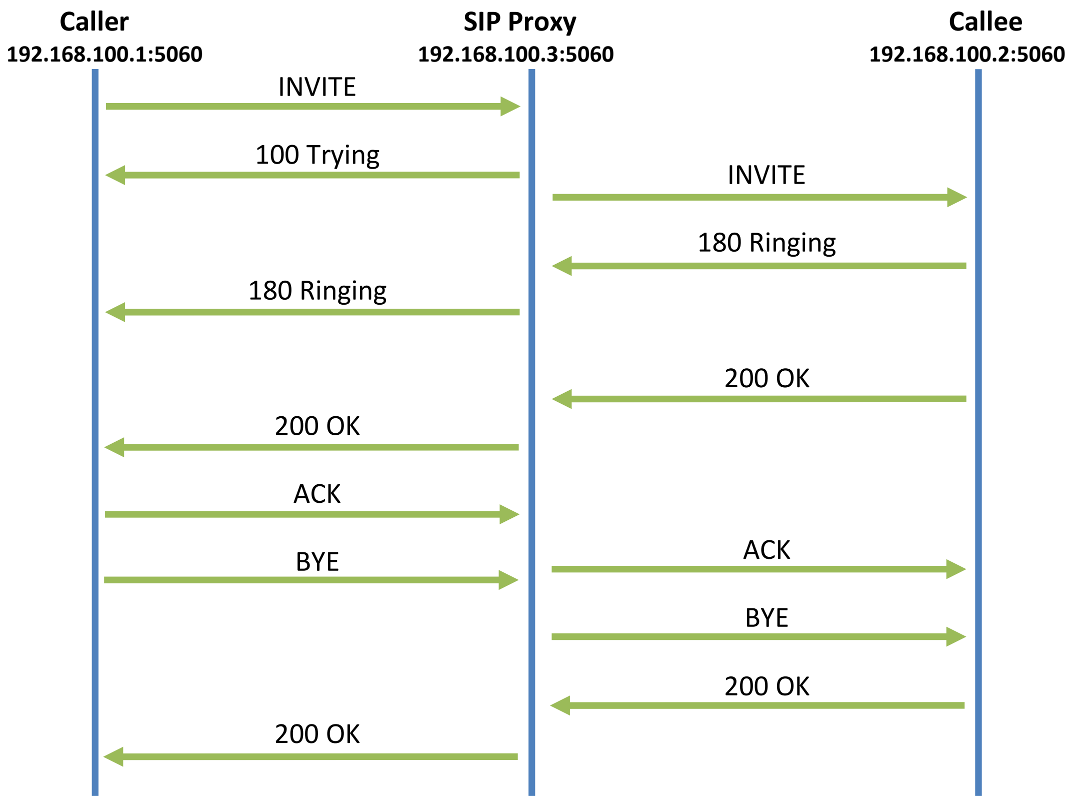


Figure 3-4: SIP messages between UAC, SIP proxy, and UAS

We configure the OpenSIPS tool as a SIP proxy. The SIP proxy listens to 192.168.100.3 at port 5060. Figure 3-5 shows some part of the OpenSIPS configuration file (*opensips.cfg*) used. For the routing logic, the OpenSIPS accounts the INVITE message received and forward it to the callee (192.168.100.2 port 5060) by replacing the address in the request URI.

```
…
listen=udp:192.168.100.3:5060
…
route{
…
if (is_method(“INVITE”)) {
        setflag(1);
        rewritehostport(192.168.100.2:5060);
}
…
}
```

Figure 3-5: OpenSIPS configuration file (partial) for simple UAC and UAS

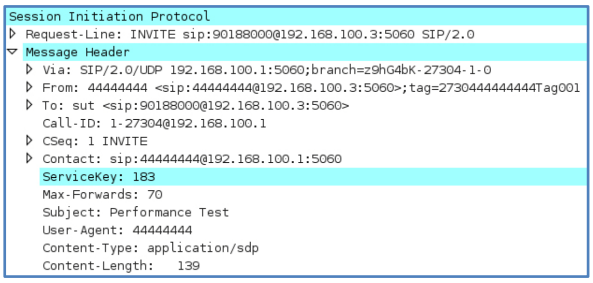
After the OpenSIPS proxy has been started, we start the caller and callee the same way as in previous scenario. However, the target address must be changed to the proxy (192.168.100.3 port 5060). The command for starting the caller is

sipp -sf caller.xml 192.168.100.3:5060 -i 192.168.100.1 -p 5060 -r 100 -trace_stat

The command for starting the callee is

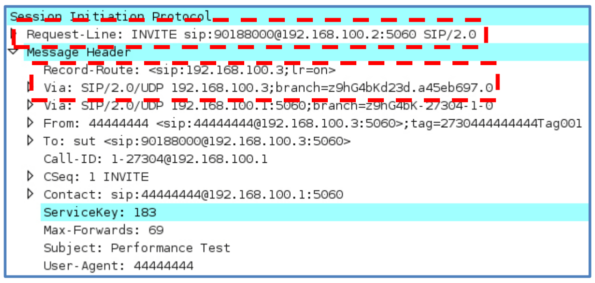
sipp -sf callee.xml 192.168.100.3:5060 -i 192.168.100.2 -p 5060

The detail of the INVITE messages received and forwarded by OpenSIPS proxy server is shown in Figure 3-7 and Figure 3-6 respectively. We can see that the address in *Request-Line* is changed to the final destination address. In addition, the *Via* and the *Record-Route* fields are added to the message header. The two figures are taken from the network protocol analyzer application, *Wireshark* [22].



```
Session Initiation Protocol
▷ Request-Line: INVITE sip:90188000@192.168.100.3:5060 SIP/2.0
▽ Message Header
  ▷ Via: SIP/2.0/UDP 192.168.100.1:5060;branch=z9hG4bK-27304-1-0
  ▷ From: 44444444 <sip:44444444@192.168.100.3:5060>;tag=2730444444444Tag001
  ▷ To: sut <sip:90188000@192.168.100.3:5060>
    Call-ID: 1-27304@192.168.100.1
  ▷ CSeq: 1 INVITE
  ▷ Contact: sip:44444444@192.168.100.1:5060
    ServiceKey: 183
    Max-Forwards: 70
    Subject: Performance Test
    User-Agent: 44444444
    Content-Type: application/sdp
    Content-Length:   139
```

Figure 3-7: INVITE message received by the proxy



```
Session Initiation Protocol
▷ Request-Line: INVITE sip:90188000@192.168.100.2:5060 SIP/2.0
▽ Message Header
    Record-Route: <sip:192.168.100.3;lr=on>
  ▷ Via: SIP/2.0/UDP 192.168.100.3;branch=z9hG4bKd23d.a45eb697.0
  ▷ Via: SIP/2.0/UDP 192.168.100.1:5060;branch=z9hG4bK-27304-1-0
  ▷ From: 44444444 <sip:44444444@192.168.100.3:5060>;tag=2730444444444Tag001
  ▷ To: sut <sip:90188000@192.168.100.3:5060>
    Call-ID: 1-27304@192.168.100.1
  ▷ CSeq: 1 INVITE
  ▷ Contact: sip:44444444@192.168.100.1:5060
    ServiceKey: 183
    Max-Forwards: 69
    Subject: Performance Test
    User-Agent: 44444444
```

Figure 3-6: INVITE message forwarded by the proxy

3.3 Results for Various Settings

3.3.1 Traffic Generation Rates

In this section, we perform the tests at different call generating rates ranging from 100 cps to 1000 cps. The averaged results from 10 repetitive experiments are compared in the two diagrams below. Figure 3-8 shows the successful call attempts per second, while the average response times are illustrated in Figure 3-9.

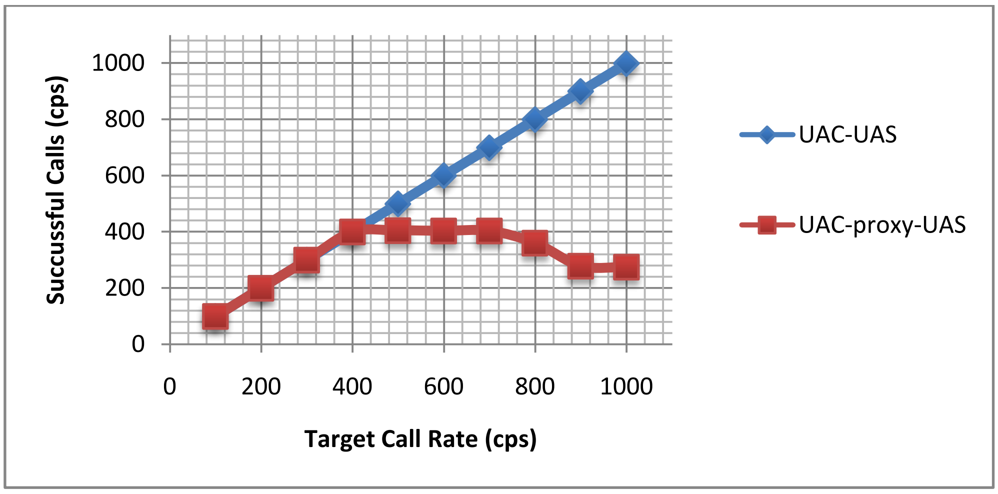


Figure 3-8: Successful calls at different target rates

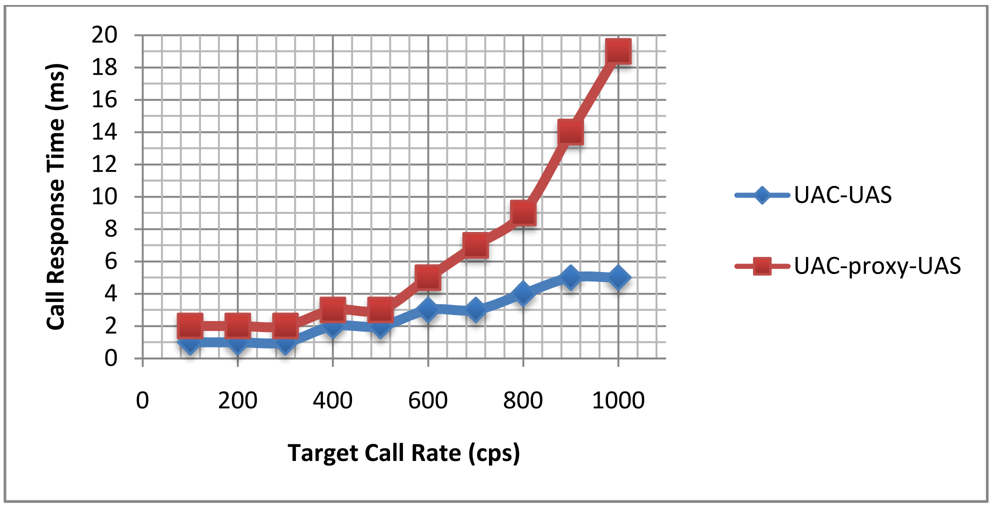


Figure 3-9: Call response times at different target rates

For the case of UAC and UAS, most of the call attempts are successful up to 1000 cps. On the other hand, when an OpenSIPS proxy is included, the successful call starts dropping at around 400 cps. Regarding the response time, we can see that adding an intermediate proxy increases the call latency by small degrees. However, significant differences take place after 500 cps.

3.3.2 OpenSIPS Memory Configuration

From the result in the last section, we can see that the scenario which involves an OpenSIPS proxy failed to process all requests at target rates higher than 400 cps. This limitation must be related to the OpenSIPS process in some ways. Therefore, we investigate deeper into the configurations of OpenSIPS. We find out that the memory allocated to the OpenSIPS process at startup influences the number of call requests it can process successfully.

Allocated memory can be configured using *-m* parameter. We perform more experiments with different amount of allocated memory (16-128 Megabytes). For each target rate, the test duration is 60 seconds. The results from 10 repetitive tests are averaged and shown in Figure 3-10: Successful calls for different memory allocationFigure 3-10.

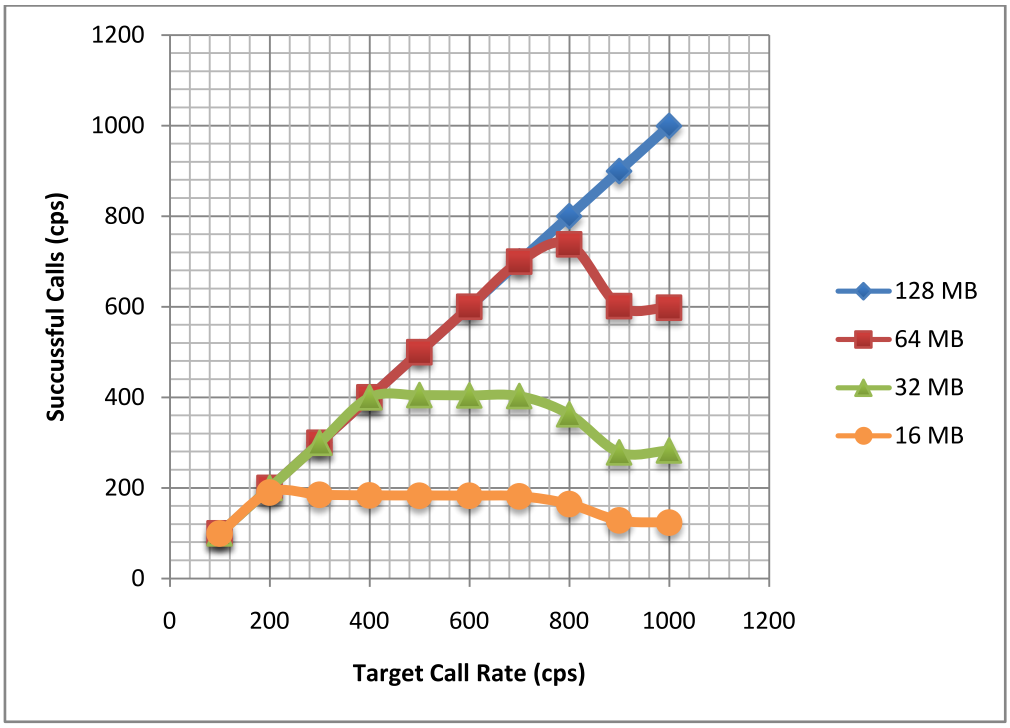


Figure 3-10: Successful calls for different memory allocation

As we can see that OpenSIPS has some capacity limitations, using it in any performance evaluation should be done with this in mind. Otherwise, the results may be affected by this boundary.

3.4 Discussion about SIPp and OpenSIPS

From the experiments, we see that SIPp can produce SIP messages according to the XML scenario file. It also provides many timers as well as statistic counters captured from the calls generated. SIPp is really flexible and can be configured in many ways to suit different testing purposes. For these reasons, SIPp is appropriate for evaluating various SIP equipments. However, using and controlling SIPp requires deep knowledge about the tool. In addition, the result statistic file produced by SIPp may not be convenient to interpret directly. Further processing in a spreadsheet application can help promote the understandability. Nevertheless, it introduces new complexity in evaluating a system.

Regarding the performance aspect, SIPp can generate SIP traffic and capture the statistics of the traffic at very high rate (thousands of calls per second). On the other hand, OpenSIPS which acts as a proxy server has some limitations. Even though it is claimed to handle hundreds of calls per second, the maximum number of calls is most likely determined by the amount of allocated memory as experimented in section 3.3. Therefore, users must ensure that performance evaluation tools are capable of handling more intense traffic than the target system before using them. This is to avoid a mistaken result caused by the constraints of the evaluation tool itself.

4 SIP Performance Evaluation Framework

In the previous chapter, we studied the existing tools, SIPp and OpenSIPS. This chapter handles the design and development of a performance evaluation framework. We start with the overview of the framework in section 4.1. Then the requirements of the framework are addressed in section 4.2. Following the requirements, the detailed design of the tool is explained in section 4.3. The last part, section 4.4, describes the implementation of the framework. The user manual of the SIP performance evaluation framework can be found in Appendix A.

4.1 Framework Overview

The performance evaluation framework aims to provide better usability for system testers. Predefined scenarios embedded in the framework can help cutting down the time needed for editing XML scenario files. The graphic user interface reduces the complexity of test configurations. The framework also displays the test results as graphical diagrams for easier result interpretation. In addition, the diagrams are real-time updated as the test is progressing.

The overall view of the framework is shown in Figure 4-1. The performance testing tool works with an instance of SIPp, while users interact only with the tool using graphic user interface (GUI). The tool creates a shell script with SIPp startup

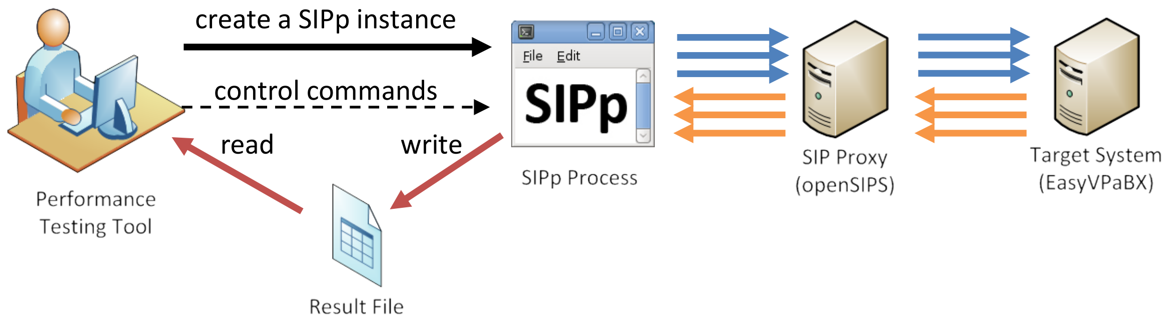


Figure 4-1: The performance evaluation framework

parameters according to the configurations specified by the user. Once the shell script is executed, it starts up a new instance of SIPp. The SIPp process generates SIP requests towards the target system. All SIP messages between the SIPp process and the target system are transmitted through a SIP proxy implemented using OpenSIPS.

Once the traffic generator is started, the application controls the traffic during the evaluation by sending control commands to the traffic generator. The result file from the traffic generator is periodically read by the application. Graphical diagrams are then created and displayed to the user.

4.2 Requirement Specification

This thesis aims to develop a user-friendly performance evaluation framework. The framework should be able to dimension a SIP-based telecommunication system. This section describes the requirements of the framework. We define the performance metrics which will be dimensioned in section 4.2.1. The call scenarios which should be supported are explained in section 4.2.2. Lastly, section 4.2.3 presents the user interface of the framework.

4.2.1 Performance Measurements

There are two performance metrics we are focusing in this thesis, the *system capacity*, and the *call setup time*. The capacity of the target system is the number of requests which can be processed properly within a period of time. A properly serviced request refers to a successful call served by the system.

To measure the capacity of a system, calls are generated and targeted to the system at various *call rates*. The maximum call rates of which the system can still serve the requests can be considered as the capacity of that system.

The other aspect of performance that should be supported by the framework is the call setup time. The response time of a call is the time it takes to get an answer back from the target system. In this thesis, we refer the time between sending an *INVITE* message out and receiving the corresponding *200 OK* reply as the response time.

In summary, to measure the system capacity and call setup time, the framework must be able to capture the response time of SIP calls as well as the number of successful SIP calls. Moreover, it must be configurable to generate SIP calls at different call rates to facilitate the measurement of the system capacity.

4.2.2 Scenarios

The SIP performance evaluation framework should support two scenarios. The traffic generator generates a sequence of SIP messages according to the scenario selected by users. The users do not need to study the SIP sequences or construct the XML files themselves. In this way, the predefined XML scenario files can shorten the time for configuring the test.

In this thesis, we take the EasyVPaBX as the target system. The EasyVPaBX receives a call initiated from a *caller*, and forwards that call to the corresponding *callee*. There are two scenarios which we cover, the *simple call*, and the *queue*.

To study the sequence of SIP messages in the two scenarios, we use a SIP soft phone application to generate a call to the destination server. During the call, we captured SIP messages transmitted between the soft phone and the server using a network traffic analyzer. The tools we used for this purpose are *SJphone* [23] and *Wireshark* [22]. From the captured sequence of messages, we draw a diagram consisting of different participants of the call session. The following subsections explain in details about the two scenarios.

4.2.2.1 Simple Call

For the simple call scenario, the actual destination of a call is always available. Once the INVITE message from the caller is received by the server (EasyVPaBX), the server immediately initiates a new call session to the callee. After the SIP session with the callee has been established, the caller and the callee can start the conversation. SIP messages exchanged in this scenario is illustrated in Figure 4-2. The red dotted arrow in the diagram denotes the media conversation.

The server maintains two SIP sessions when the call is active. One session is between the caller and the server denoted by the green arrows, while the other session is between the server and the callee denoted by the orange arrows. The server terminates the call session when either side hangs up.

The caller sends SIP messages to the server through a SIP proxy. In addition, the call created by the server targeted at the callee is also via a SIP proxy. Normally, they can be two different proxy servers. However, the EasyVPaBX server tested in this thesis sends SIP messages to a callee through the same SIP proxy as the one it receives a call from the original caller. Therefore, the SIP proxy used in the test must have address information of the server (EasyVPaBX) as well as the callee.

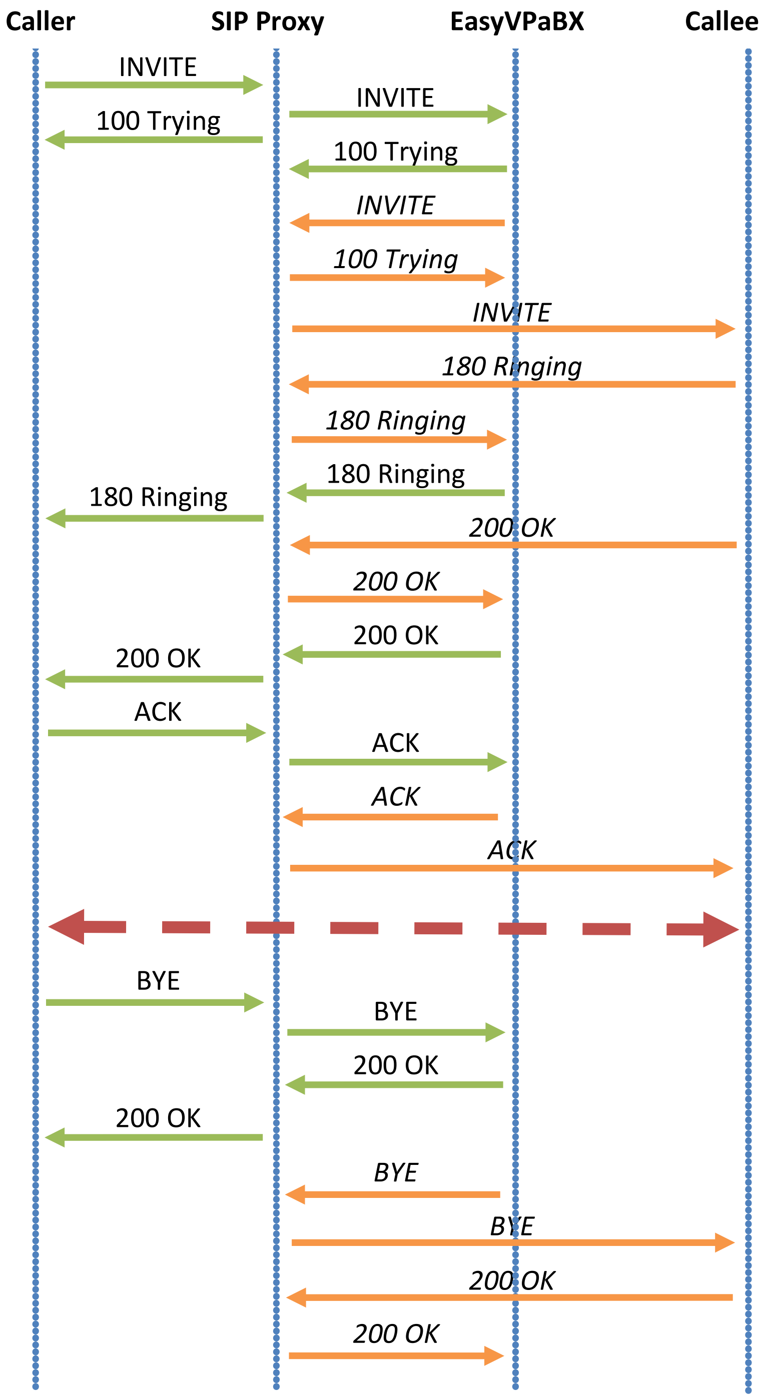


Figure 4-2: SIP messages in the simple call scenario

4.2.2.2 Queue

The queue scenario is slightly different from the simple call. When the server receives a new call from the caller, it tries to find the corresponding callee. If the callee is available, the server creates a new call session to the callee. Otherwise, the call is placed in a waiting queue. During its time in the waiting queue, the caller receives some audio messages. The audio playback is originated from a *media server*. The SIP server (EasyVPaBX) contacts the media server every time it has a new call waiting in the queue.

Once the destination is accessible, the server then initiates a call to the callee, removes the call from the waiting queue, and the session between the caller and the callee is then created.

One example of this scenario is a customer service center. The call center has one public phone number where customers can call to get the service. When at least one officer (callee) is available, the call is served immediately. If there is no available officer, the customer is told to hold the line. Afterwards, the customer will get connected with a vacant officer.

Figure 4-3 shows a sequence of SIP messages exchanged among different participants in case that a call is placed in a waiting queue. The server (EasyVPaBX) sends an INVITE to the media server to create a session between the caller and the media server (denoted by blue arrows). After the session is established, the caller gets a stream of audio packets from the server. Until the callee is accessible, the server starts another session with the callee (denoted by orange arrows). When the session with the callee is created, the server terminates the session with the media server by a BYE message. After that, the server sends another INVITE message back to the caller to activate the call between the caller and the callee. The media conversation begins after the ACK message from the server is sent to the callee which is denoted by the dotted arrow in the diagram. The two sessions go on in the same way as the case of a simple call. The server terminates the sessions when it receives a BYE message from either the caller or the callee.

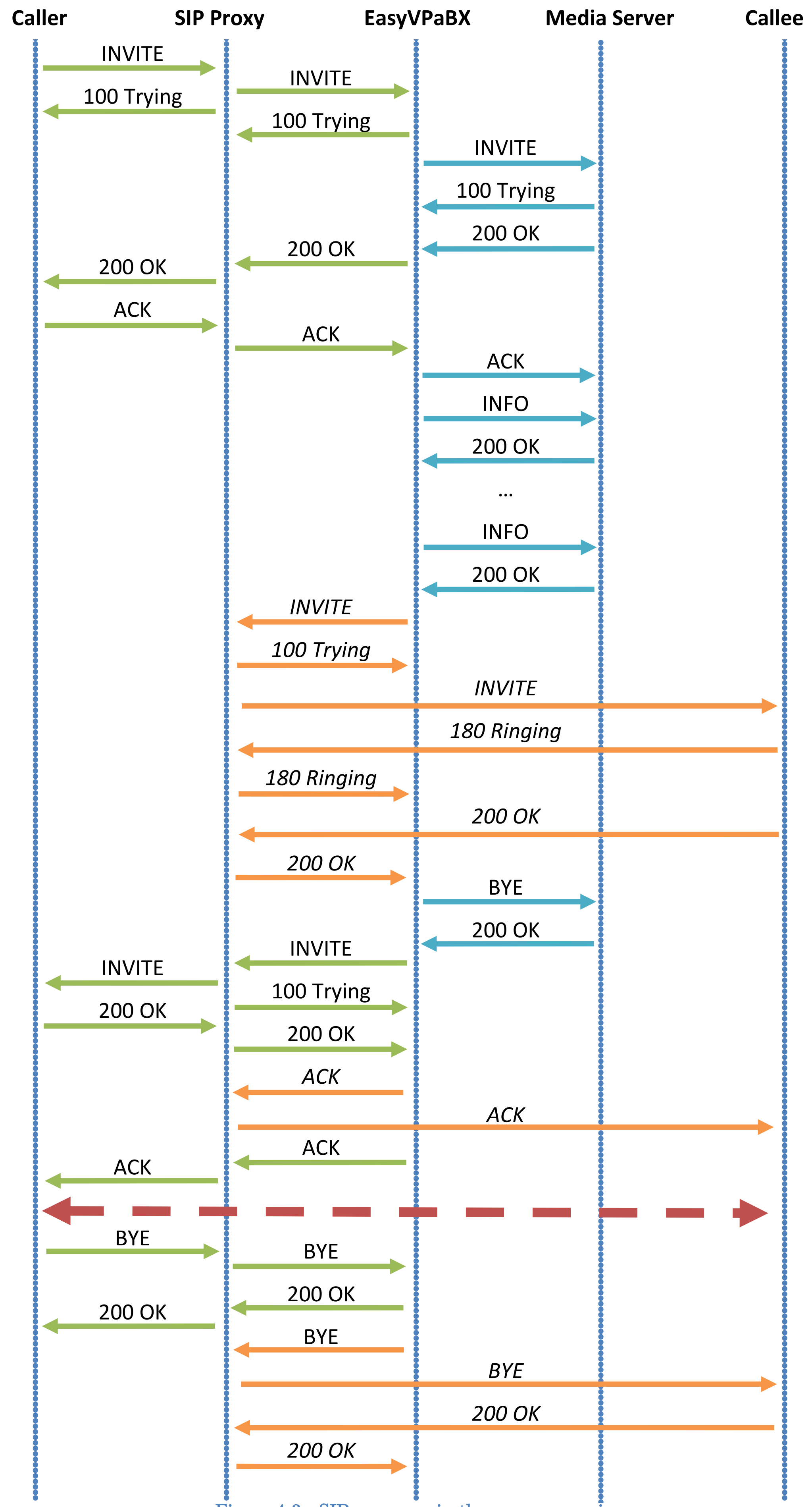


Figure 4-3: SIP messages in the queue scenario

4.2.3 User Interface

Graphical user interface (GUI) allows users to interact with computer programs in easier way compared to the command line interface (CLI) which has a steep learning curve. With CLI, users are required to enter commands through the command line. This could be complicated and time consuming since command words may not be easily memorized by users. On the other hand, GUI offers windows, icons, menus, and a pointing device for simpler control over applications.

Despite the fact that SIPp provides number of functionalities and configurations and can be used for testing various SIP systems, it is available only in CLI mode as discussed earlier in section 3.4. Starting, controlling, as well as monitoring the result of SIPp have to be done with the command line. System testers need to learn the commands and parameters in order to use the tool. In addition, the results display is limited to text-based presentation.

For user-friendliness, the performance evaluation framework should provide an easy-to-use graphical user interface. Users should be able to select testing scenarios, as well as specify the destination number, IP address and port number of the target system. Parameters such as the call rate and the duration of the test should also be configurable through the interface. Moreover, the real-time evaluation results should be displayed in graphical format for easy interpretation.

4.3 Design

Following the requirements discussed earlier, this section give details the design of the performance evaluation framework. Section 4.3.1 describes the components of the framework. Then the user interface design is explained in section 0.

4.3.1 Components

The performance evaluation framework is designed as illustrated in Figure 4-4. The framework comprises of three main tasks, creating startup script, controlling the traffic generator, and processing the result. These tasks are represented as circles in the figure. In addition, the framework contains a SIP traffic generator (SIPp), a SIP proxy (OpenSIPS), and a target system (EasyVPaBX).

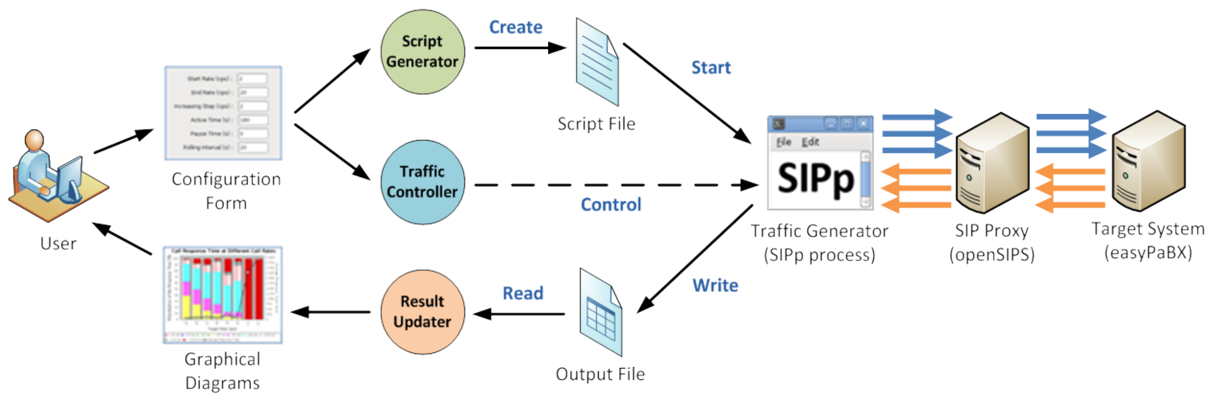


Figure 4-4: Performance evaluation framework (detailed design)

As discussed in section 3.4, SIPp is appropriate for evaluating SIP equipments. It can generate SIP requests at high speed and can be configured to suit different tests. In addition, experiments in section 3.3 show that OpenSIPS has reasonable performance as a SIP proxy. Moreover, both SIPp and OpenSIPS are open source software, so we can use them as a basis of the framework development. For these reasons, we choose SIPp and OpenSIPS as the traffic generator and the SIP proxy for the performance evaluation framework.

The framework provides a graphic form for evaluation specifications from a user. The configuration parameters specified by the user are used for generating a startup script as well as controlling the behavior of the traffic generator. Once the script file has been created and executed, an instance of SIPp will start generating SIP traffic towards the target system through the SIP proxy.

During an evaluation, the SIPp instance keeps listening to a particular UDP port called *control port* for run-time commands. In order to control the traffic generated by the SIPp process, the traffic controller opens a connection to this control port. Based on the configurations specified by the user, the traffic controller sends control commands such as increasing or decreasing the call rate and pausing the call generation to this port.

For performance measurements, the SIPp process captures statistic counters and timers, and writes the result to an output file. This file is retrieved by a result updater. Furthermore, statistic values in the file are extracted and processed. As a result, the user can monitor the real-time updated graphical diagrams created from the output file.

4.3.2 Graphic User Interface

The graphical user interface provided by the framework can be divided into two parts, the *parameters configuration*, and the *results display*. The following subsections give details about them.

4.3.2.1 Parameters Configuration

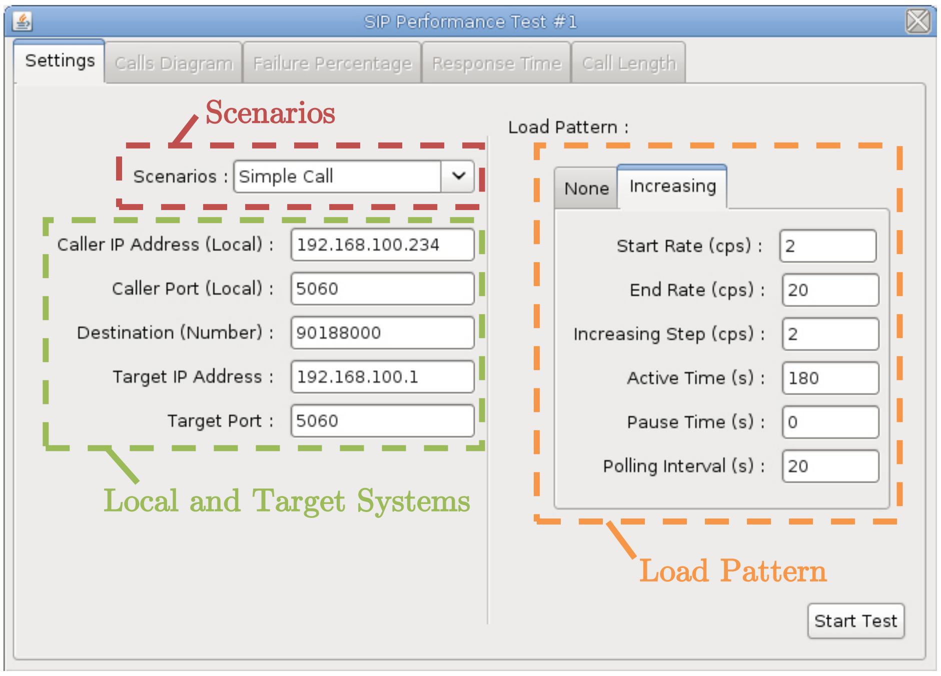
Users can configure the evaluation parameters through a graphical form. This form contains several text fields and drop down lists as shown in Figure 4-5 below. The parameters can be categorized into three groups, *scenarios*, *local and target systems*, and *load patterns*.

Scenarios

For the SIP scenarios, there are two options, the *simple call*, and the *queue*. The selected choice determines the XML scenario file to be used for generating SIP traffic to the target system.

Local and Target Systems

User can specify the IP address of the local system as well as the port number used to send out the SIP messages. The destination number, IP address and port number of the target system are also configurable.



Scenarios : Simple Call

Caller IP Address (Local) :	192.168.100.234
Caller Port (Local) :	5060
Destination (Number) :	90188000
Target IP Address :	192.168.100.1
Target Port :	5060

Start Rate (cps) :	2
End Rate (cps) :	20
Increasing Step (cps) :	2
Active Time (s) :	180
Pause Time (s) :	0
Polling Interval (s) :	20

Figure 4-5: Configurable Parameters

Load Pattern

There are two patterns of traffic available, the fixed rate, and the increasing rate. For the fixed call rate, user defines the rate of the traffic to be generated, and the duration of the test.

As for the increasing case, user gives parameters such as the *start call rate*, the *final call rate*, and the *increasing step* to define the traffic pattern. In addition, between different rates, the traffic can be paused for some amount of time which is configured by the *pause time*. The period of each call rate is identified by the *active time*. For both load patterns, the frequency of result updating can be specified in the *polling interval* field.

4.3.2.2 Results Display

Based on the statistics values retrieved from the SIPp output file, the framework creates graphical diagrams and updates them periodically. These diagrams let users interpret the result easily. Therefore, we choose to generate four diagrams, the successful/failed call numbers, the failure rate, the call response time, and the total call length. The examples of these diagrams are shown in Figure 4-6, Figure 4-7, Figure 4-8, and Figure 4-9.

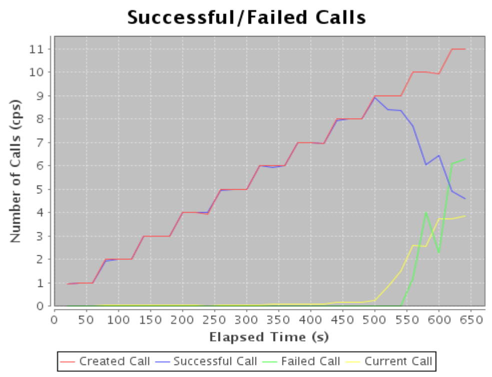


Figure 4-6: Successful/Failed calls diagram

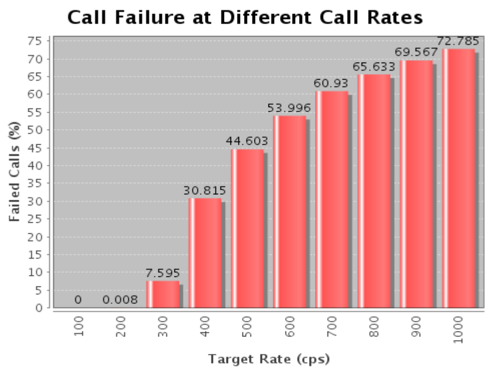


Figure 4-7: Call failure diagram

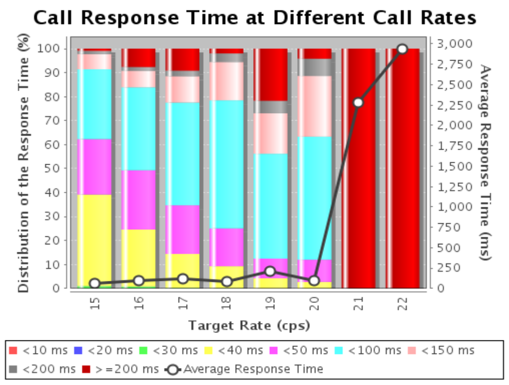


Figure 4-8: Response time diagram

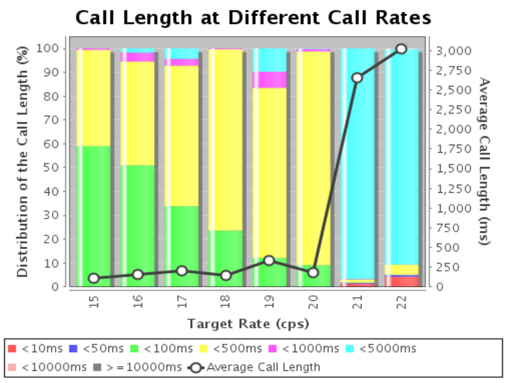


Figure 4-9: Call length diagram

4.4 Implementation

This section addresses the implementation of the SIP performance evaluation framework. We mention certain assumptions for the tool in section 4.4.1. We continue with composing the XML scenario files in section 4.4.2. The main logics of the tool are explained in section 4.4.3 where we divide the tasks into three threads. The details of starting and controlling a SIPp process, reading the output file, and generating graphs are described in sections 4.4.4, 4.4.5, 4.4.6, and 4.4.7.

4.4.1 Initial Requirements

Before implementing the SIP performance evaluation framework, we addressed some requirements and assumptions for running the tool as follows.

Java Environment

The application developed in this thesis is based on Java 2 Platform Standard Edition [24]. It is tested on systems with Java Runtime Environment JRE 6 .

SIPp

The application does not include the SIPp tool inside. Nevertheless, an instance of SIPp is created when the tool starts the test. Therefore, it is required that SIPp is properly installed in the system. The application in this thesis has been tested with SIPp version 3.1. Other versions of SIPp may also be functional with the tool as well, but this is not assured.

Proxy Server

The complete test system requires a proxy server as shown earlier in Figure 4-1. For the application to run properly, a proxy server which forwards SIP messages to the target system must be configured. In this thesis, we set up the OpenSIPS version 1.4.4 to perform this task.

Library Files

The graphical user interface of the framework is implemented using Swing Application Framework [25]. In addition, the diagrams showing test results are produced using JFreeChart library [26]. The framework comes with these libraries as Java archive (JAR) files in the *lib* directory.

4.4.2 Scenario Files

The SIP performance evaluation framework comes with certain scenarios. These SIP scenarios are defined by XML scenario files included in the application. The files are copied to the testing directory before the test is started. As stated earlier in the requirements section, the *simple call*, and the *queue* scenarios are embedded in the tool. The following scenario files are created according to the two SIP sequences discussed in section 4.2.2.

4.4.2.1 Simple Call Scenario File

The XML file describing the simple call scenario is shown in Appendix B. The caller starts by sending an *INVITE* message. Then it expects a *100 Trying* message, and a *180 Ringing* message, but these messages are optional. After that, it waits for a *200 OK* message which triggers the response-time timer to stop. The value of this timer is stored and written to the statistics file. Upon receiving the *200 OK* message, it sends out an *ACK* message and the call session is established. To terminate the session, the caller sends a BYE message, and expects a 200 OK message as a reply.

4.4.2.2 Queue Scenario File

For the queue scenario, we modify the simple call scenario file to support additional messages from the server. The XML file of the queue scenario is shown in Appendix C. At the beginning, we define a variable *q* for storing the status of the call session. If *q* is set, it means that the session is put in a waiting queue in the server. This variable is used to test a conditional branching.

To detect whether the session is placed in the waiting queue or not, we explore the first *200 OK* message received. We search the message for a keyword using a regular expression. Since, the EasyVPaBX system in this thesis uses the SnowShore IP media server [27], we take *snowshore* as a keyword which reflects the originator of the message. If the *200 OK* message contains this keyword, the session is established with the media server. This means that it is currently in the queue. Therefore, the caller has to wait until it gets a new *INVITE* message denoting that the ultimate destination can be connected and the session has been removed from the waiting queue. Otherwise, the caller jumps to another part of the scenario file to continue on the normal session establishment.

4.4.3 Threads

A thread of execution is a forked task of a program. Within a single computer program, multiple threads can run simultaneously. Multiple threads can share process state as well as resources. However, they are executed independently. Depends on systems, threads can be implemented by *multiplexing* or really running them concurrently using multiple processors. Multiplexing refers to context switching between threads by one processor. The switching happens fast enough as if the threads are running at the same time.

With the thread concept, programmers have better control over concurrent executions. For instance, an application which requires responsiveness to users' interaction may need multiple threaded programming. While one execution thread is busy with long running task or has to wait for an input/output event, another thread can handle the users' interaction. Therefore, the application will not freeze during the completion of the time consuming tasks.

As explained in the requirements section, the performance evaluation framework consists of several tasks. These tasks include generating graphical user interface, starting an instance of SIPp, controlling the SIPp process, and processing the result file.

Most of the tasks listed above need to be done concurrently. For easier management, we divided them into multiple threads. Considering Java as a programming language used in developing the framework, we can straightforwardly implement these threads using Java. Since the Java Virtual Machine allows an application to have multiple threads of execution running concurrently [28].

For the application, we implement three threads, the *main thread*, the *control thread*, and the *result update thread*. Because the latter two threads have certain blocking events, we split them from the main thread to prevent the tool to halt while waiting for those tasks.

The main thread deals with interfaces and interactions with user and also manages the other threads. The control thread is responsible for sending commands to the SIPp instance. The result update thread periodically reads the output file for statistic values. Additionally, it processes them and generates result diagrams.

4.4.3.1 Main Thread

The main thread is the main process started when user launch the application. It manages the appearance of the tool as well as the interaction with a user. The following steps are the logic of the main process.

1. The graphical user interfaces are prepared and displayed to the user. This step includes creating several windows, menus, and panels, initialize test configuration options and text fields.
2. When the user chooses to start a test, the main thread prepares a new test folder. It creates a test scenario file as well as other necessary files in the folder.
3. It then makes a new shell script consisting of the command to execute a SIPp process. In addition, the shell script also contains startup parameters which are derived from the configurations specified by the user.
4. Once everything is ready, the test can be started. The process executes the shell script to start up an instance of SIPp. From this point on, SIP traffic are generated and sent to the target system.
5. The main process starts the other two threads, the *control thread* for controlling the SIPp, and the *result update thread* for processing the result.
6. When the test has been completed, the main process is informed by the control thread. It manages the close down of the test. This includes terminating all other threads, and killing the SIPp process.

All steps preformed in the main process do not require much time compared to the tasks in the other two threads. In other words, all the tasks in the main thread are not blocking or having to wait for certain events. Hence, from user's point of view, the main process which is responsible for interaction and interface is always responsive.

4.4.3.2 Control Thread

The control thread is created after the main process has started the SIPp instance. This thread is responsible for sending control commands to the SIPp process to increase/decrease the traffic generation rate, and pause the generation. At the time of creation, this thread is initialized with parameters specified by the user for traffic generation pattern.

According to the configurations initialized at startup, the control thread keeps track of the time period and sends out traffic control commands to the SIPp process through a specific UDP port called *control port*.

The reason of separating this as another thread is because it has to periodically perform its task (sending command) at a specific time. During the period of time between each command, the thread does nothing but waiting. This can be considered as a blocking event.

4.4.3.3 Result Update Thread

The result update thread is also created after the SIPp instance has been started. This thread keeps reading the result file from the SIPp process. During the test, the SIPp process saves the counters and timers into this result file once every some period of time. Similar to the control thread, this result update thread needs to wait for certain events. Therefore, it is separated as an independent thread.

The frequency of reading the result file is specified by the user. It matches with the result writing event of the SIPp process. The thread waits for a specific amount of time and updates statistic values from the file. Furthermore, it processes these values and plots several graphs. These tasks go on iteratively until the test ends.

At the end of the test, the main process sends termination signal to this thread. The thread stops reading and processing the result. Additionally, before the thread is terminated, it exports the final diagrams to image files for further usage.

4.4.4 Starting SIPp

This step is a part of the *main thread* (section 4.4.3.1). When the user chooses to start the test, all configuration parameters are collected and stored in the application. Certain parameters are used as startup options passed to the SIPp process. Other parameters instruct the *control thread* which is created after the SIPp process has been started.

Working with SIPp requires interactions through a command line prompt. We created one Java method to support execution of shell commands from the application. This method takes a shell command together with a working directory as input parameters. It then sets the working directory and creates a new process. After the command is executed, the method returns with the result from the command line. Figure 4-10 shows the *execute* method which carries out the task.

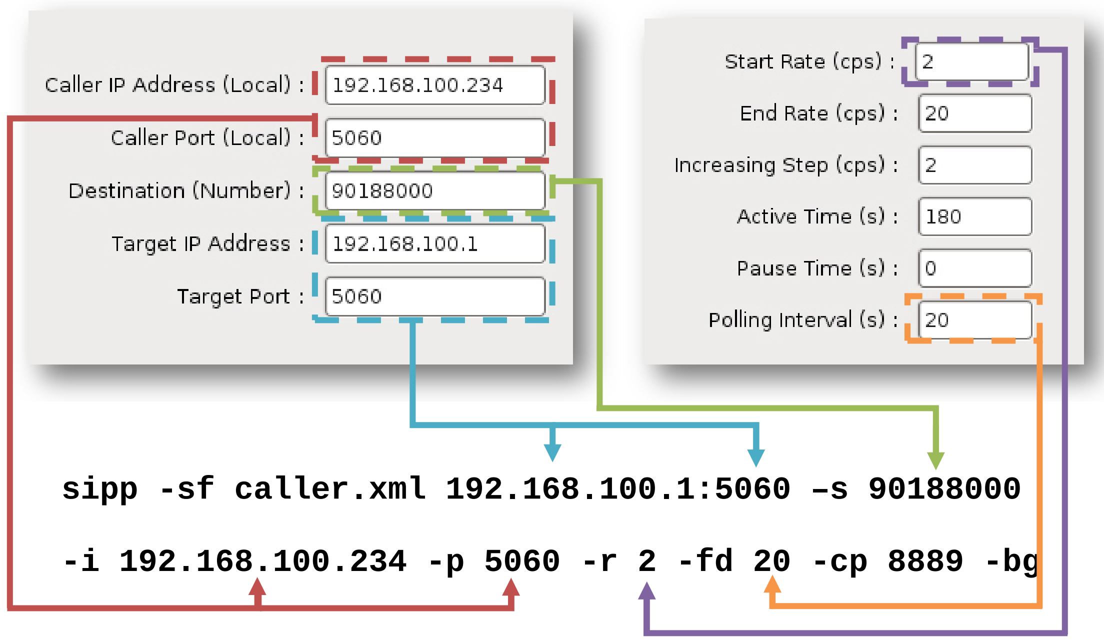
```
public static String execute(String folder, String command){
       String result = "";
       try{
              ProcessBuilder pb = new ProcessBuilder("bash", "-c", command);
              pb.directory(new File(folder));
              Process proc = pb.start();
              proc.waitFor();
              BufferedReader resultReader = new BufferedReader(
                     new InputStreamReader(proc.getInputStream()));
              String temp;
              while ((temp = resultReader.readLine()) != null){
                     result = result + temp + "\n";
              }
              resultReader.close();

       }catch(Exception e){e.printStackTrace();}
       return result.trim();
}
```

Figure 4-10: Java method for command line execution

The tool prepares for the new test by creating a new directory containing the scenario file and some other necessary files. The directory name consists of the current date and time for its uniqueness.

A new shell script is generated within the directory. This shell script contains the command to start a SIPp process as well as startup parameters. To make the SIPp process controllable from the application, a control port must be identified when starting the process. The application finds an available UDP port, and adds *-cp* parameter following by the UDP port number in the script file. Putting collected configurations into the shell script is illustrated in Figure 4-11.



Caller IP Address (Local) :	192.168.100.234
Caller Port (Local) :	5060
Destination (Number) :	90188000
Target IP Address :	192.168.100.1
Target Port :	5060

Start Rate (cps) :	2
End Rate (cps) :	20
Increasing Step (cps) :	2
Active Time (s) :	180
Pause Time (s) :	0
Polling Interval (s) :	20

```
sipp -sf caller.xml 192.168.100.1:5060 -s 90188000
-i 192.168.100.234 -p 5060 -r 2 -fd 20 -cp 8889 -bg
```

Figure 4-11: Generating a shell script from configuration parameters

After the shell script is written, the tool starts a configured SIPp process by running the shell script. Since the SIPp is started in background mode (denoted by the *-bg* parameter in the script), the result of executing the script is the process ID of the SIPp. This process ID is stored in the application for later use. Creation of the new directory, copying the files, generating, and executing the shell script file are achieved using the *execute* method. Once the test is started, the application creates two new threads, the *control thread*, and the *result update thread*.

4.4.5 Controlling SIPp

After starting the SIPp process, a new thread, *control thread* (section 4.4.3.2), is created. This thread sends control commands to the SIPp process. The commands are packetized and sent to the UDP port which is configured as the control port in the last step.

Load pattern parameters collected previously are passed to the control thread. The *final call rate*, the *increasing step*, the *active period*, and the *pause period* determine the commands and the specific time they are sent out. Figure 4-12 shows an example of simplified Java code which sends the command to the control port periodically.

```
do{
      try{
            sleep(activeTime);
            if(pauseTime>0){
                  //send the pause signal
                  UDPUtility.sendCommand("p", IP, port);

                  sleep(pauseTime);

                  //send the unpause signal
                  UDPUtility.sendCommand("p", IP, port);
            }

            //send the increase rate signal
            currentRate += increaseStep;
            UDPUtility.sendCommand("cset rate "+currentRate, IP, port);

      }catch(Exception e){}

}while(currentRate < endRate);
```

Figure 4-12: Simplified Java code in the control thread

4.4.6 Reading Statistics File

Reading the statistics file is a part of the *result update thread* (section 4.4.3.3). Statistics file contains statistics and timer values. The file is updated periodically by the SIPp process. The application keeps reading this file until the test completes. The frequency of reading the file is identified by the *polling interval* parameter given by the user.

The statistic file is in a tabular format. One line of the file is read at a time, and is divided into fields. The values which we are interested are stored for further processing. These values include elapsed time, current call rate, number of calls created, number of successful/failed calls, call response time, and total call length.

4.4.7 Displaying Graphs

Processing values from the statistics file, and plotting them as graphical diagrams are also done within the *result update thread* (section 4.4.3.3). Statistics values and timers are grouped by the call generation rate. For each call rate, the averages of the values are calculated. These processed data are put in proper datasets for graph plotting.

For drawing graphical diagrams, we consider JFreeChart [26]. JFreeChart is an open source Java chart library which provides APIs supporting a wide range of chart types. We consulted the API documentation provided in their website [29] for generating the graphs. In total, the tool generates four different diagrams, the successful/failed calls, the call failure rates, the response time, and the call length.

Apart from displaying real-time updated diagrams during the test, this thread also exports the final diagrams to image files for later use after the test has been completed. These images are saved in the working directory.

5 EasyVPaBX Performance Evaluation

In this chapter, we set up certain systems as *target systems*. We consider various configurations including single computer, and load balancing scheme with two computers. Various computers are used to see how EasyVPaBX performs in different environments.

This chapter starts with section 5.1 giving details of target system configurations as well as the user system which hosts the performance evaluation framework. Following that, lists of computer specifications used in the configurations are described in section 5.2. The performance results from multiple experiments are presented in section 5.3. Performance data from the target systems are then compared and discussed in section 5.4 and section 5.5 respectively.

5.1 System Configuration

We set up a *user system* which contains the performance evaluation framework for generating SIP traffics and capturing the evaluation results. In addition, two configurations are set up for testing the EasyVPaBX (SIP server) application as the *target systems*. The first setting consists of a single instance of the server running on one computer. This is for evaluating the performance of the server. The second configuration, on the other hand, aims to test a load balancing scheme. It adds another computer which hosts one more instance of the server. The details of these two settings are explained in the following subsections.

5.1.1 Single Computer

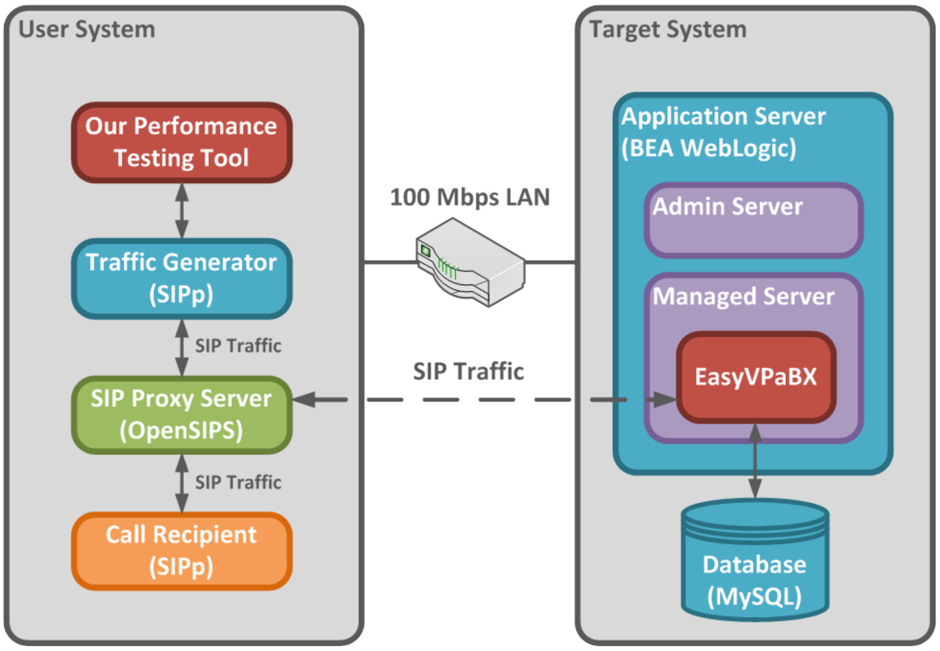


Figure 5-1: System Configuration for testing a single computer

The system configuration for performance evaluation of a single instance of the EasyVPaBX server is shown in Figure 5-1. On the left side of the figure, the user system hosts all components needed for performance measurements. This system contains the application, a traffic generator (a SIPp process), a SIP proxy (OpenSIPS), and a call recipient (another SIPp process). The application creates and controls the traffic generator. The SIP traffics generated are sent to the SIP proxy server which forwards them to the target system. The outbound traffics from the target system are sent to the call recipient through the SIP proxy.

On the right hand side of the figure lays the target system. The target system consists of one computer. An application server is installed in the computer for hosting the EasyVPaBX application. We also installed the Oracle WebLogic SIP Server version 3.1 [30] as the application server. The EasyVPaBX is deployed in a *managed server* which is controlled by the *administration server*. As for the data storage, we employ the MySQL [31] database. The server accesses application data in the local database installed on this computer.

Outbound and inbound SIP traffics take place between the SIP proxy and the SIP server (EasyVPaBX). The two systems are connected to the same local area network (LAN) with the Fast Ethernet standard supporting traffic at 100 megabits per second (100Mbps).

5.1.2 Two Computers with a Load Balancer

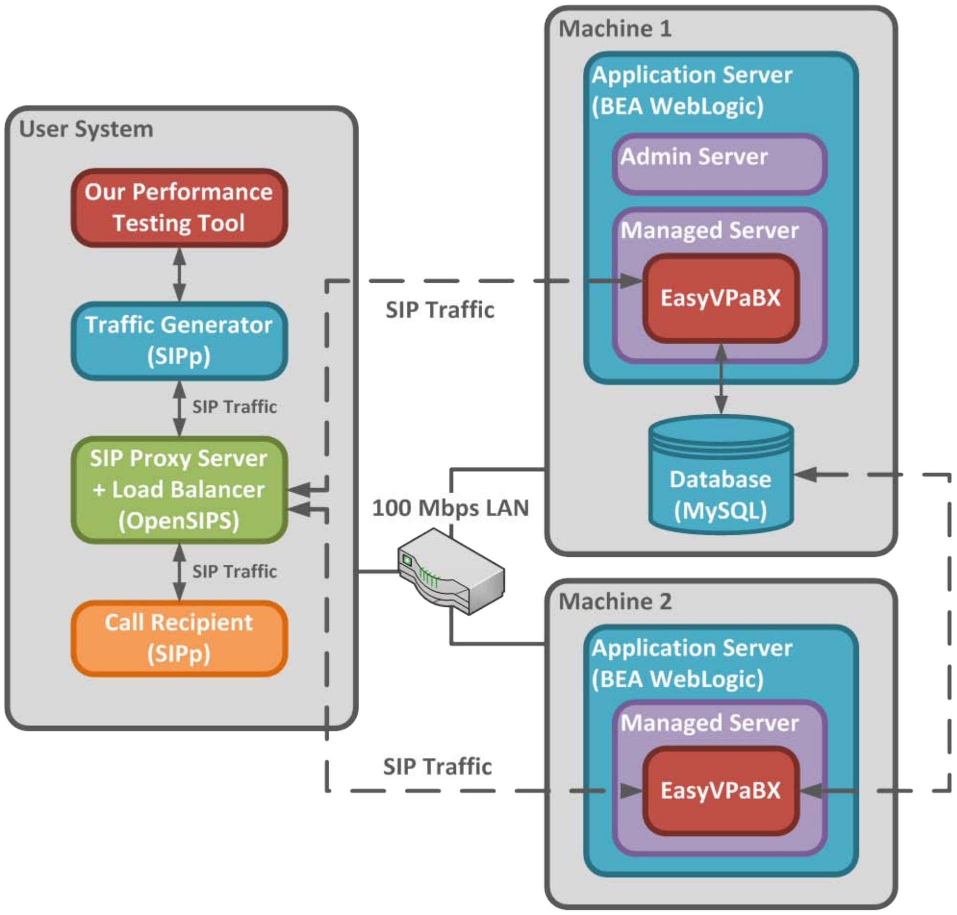


Figure 5-2: System Configuration for testing two computers

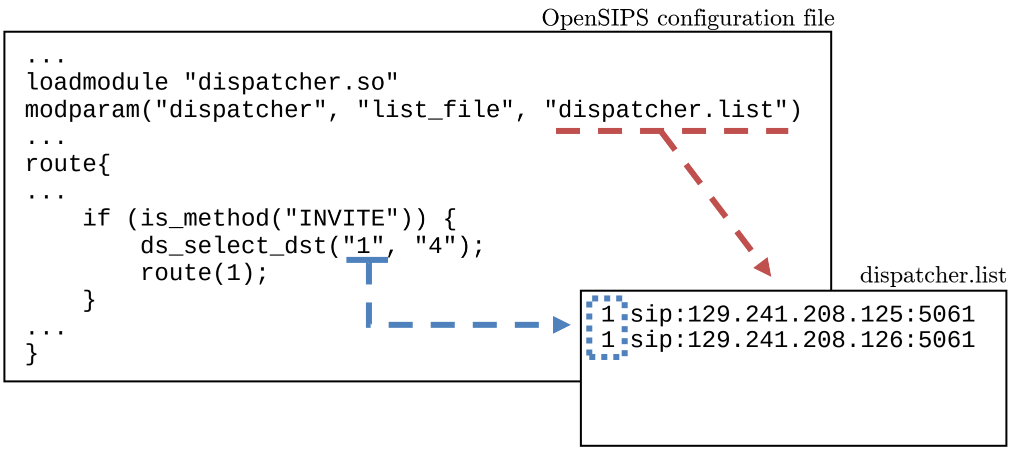
Figure 5-2 illustrates the system configuration for testing a load balancing scheme. In addition to the configuration of the single computer, the second computer is inserted. The two computers, *machine 1* and *machine 2*, in the target system process SIP requests simultaneously. Similar to the last configuration, machine 1 contains the application server as well as the database. Machine 2, which is newly added, is installed with the same application server. The application server contains a managed server hosting another instance of EasyVPaBX. This managed server shares the same administration server in machine 1. Furthermore, the application server in machine 2 also uses the database installed in machine 1.

Slight modification is made to the SIP proxy in the user system to support load balancing. The configuration files of the SIP proxy are edited so that the proxy forwards SIP requests to two SIP servers. Therefore, the SIP traffics happen between the SIP proxy and the SIP server in machine 1, as well as between the proxy and the SIP server in machine 2. The three components, the user system, machine 1, and machine 2 are connected to the same local area network (LAN) with the speed of 100 Mbps.

Figure 5-3 shows some parts of the OpenSIPS configuration file used in this setting as well as the file *dispatcher.list* used within the configuration file. In the configuration file, the module *dispatcher.so* is loaded in order to enable the *ds_select_dst* method.

The *ds_select_dst* method is used to dispatch received SIP requests to different destinations. A set of destination locations is defined in an external file specified in the *list_file* parameter of the dispatcher module. The first parameter of the method indicates the group of destination addresses. The technique for choosing the destination is identified by the second parameter. In our case, we apply the round-robin algorithm (*denoted as "4"*) which assigns SIP requests to destinations in circular order.

From Figure 5-3, the first request will be forwarded to the address in the first line (129.241.208.125 at port 5061), while the second request will go to the second address (129.241.208.126 at port 5061). The next request will again be destined at the first address, and so on.



```
...
loadmodule "dispatcher.so"
modparam("dispatcher", "list_file", "dispatcher.list")
...
route{
...
    if (is_method("INVITE")) {
        ds_select_dst("1", "4");
        route(1);
    }
...
}
```

```
1 sip:129.241.208.125:5061
1 sip:129.241.208.126:5061
```

Figure 5-3: Configuration files used for load balancing

5.2 Testing Environment

From the last section, there are two types of the target system explained, the *single computer*, and the *two computers with a load balancer*. This section defines in more details, the computers used in each configuration.

In order to see the performance testing results of the EasyVPaBX in different environment, we use various computers hosting the application. Four different computers were used as target systems. These computers have been set up in a similar way. Each computer is installed with the WebLogic SIP Server version 3.1 as an application server, and the MySQL version 5.0 as a database. The EasyVPaBX application is deployed in the application server. In addition, each computer is equipped with a network interface card supporting LAN at the speed of 100 megabits per second.

The four computers, *A, B, C,* and *D* are different in terms of hardware and software. Table 5-1 below lists the specifications of the four computers.

Machine	Processor	CPU Speed	Memory	Operating System
A	Intel® Core™ 2 Duo	1.73 GHz	1.5 GB	Ubuntu 8.10
B	Intel® Pentium® 4	3.0 GHz	2 GB	CentOS 4.6
C	Intel® Pentium® 4	2.6 GHz	1 GB	Ubuntu 8.10
D	Intel® Pentium® 4	2.6 GHz	1 GB	Ubuntu 8.10

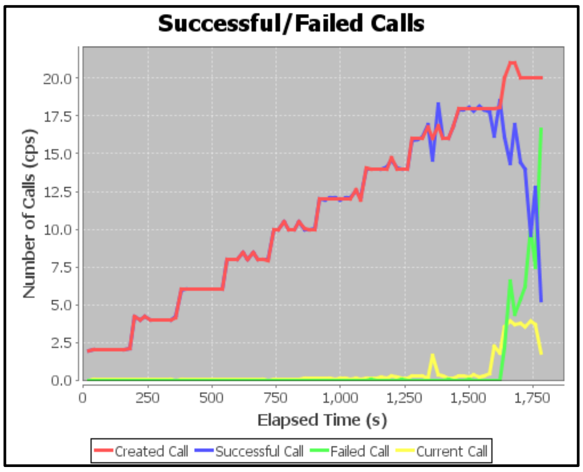
Table 5-1: Specifications of computers used as target systems

For the first configuration, the *single computer*, four computers are tested individually. The performance results of these computers are then evaluated. As for a better comparison of the performance testing results between the two configurations, two identical computers are needed. From Table 5-1, we can see that machine C and machine D share similar specifications both in hardware and software. For these reasons, the two computers are used in setting up the second configuration, the *two computers with a load balancer*.

5.3 Performance Evaluation Results

We use the performance evaluation framework to generate SIP traffic, and capture the performance results. The results presented in this section are averaged values from multiple experiments. For each configuration, we perform 10 rounds of evaluation. We use the increasing load for evaluating the throughput of the target system. The traffic rate starts from 2 calls per second (cps), and ends at 20 calls per second. The call rate is incremented by 2 cps step. For each call rate, the test lasts for 180 seconds. The statistics values are updated every 20 seconds. The results of different configurations are shown in the following subsections. All diagrams shown in this section are produced by the framework after the evaluation has been completed.

5.3.1 Single Computer: Machine A



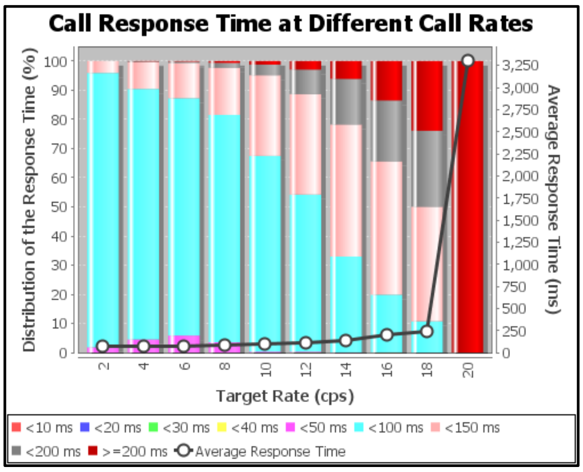


Figure 5-4: Number of calls and call setup time diagrams (Machine A)

Call Rate (cps)	Successful (cps)	Failed (%)	Call Setup Time (ms)
2	2	0	73
4	4	0	74
6	6	0.01	76
8	8	0.01	82
10	10	0.00	98
12	12	0.03	115
14	14	0.08	137
16	16	0.04	201
18	18	0.07	243
20	13.65	31.76	3289

Table 5-2: Average Statistics (Machine A)

Figure 5-4 shows two diagrams from one of the performance evaluation carried out on machine A. From the left diagram, we can see that the successful calls dropped after the call rate reaches 18 cps. In addition, after this call rate, the call response time increases significantly as shown in the diagram on the right hand side.

5.3.2 Single Computer: Machine B

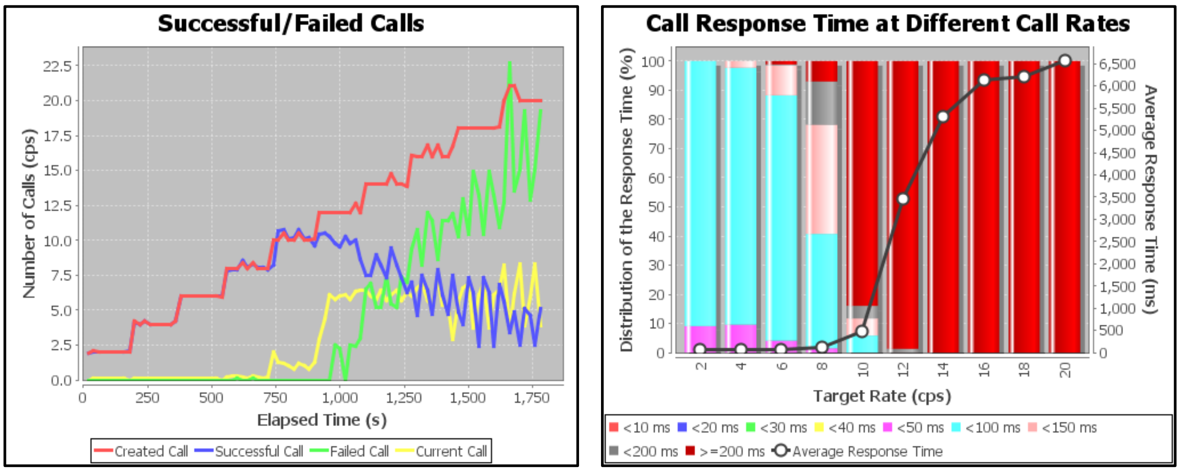


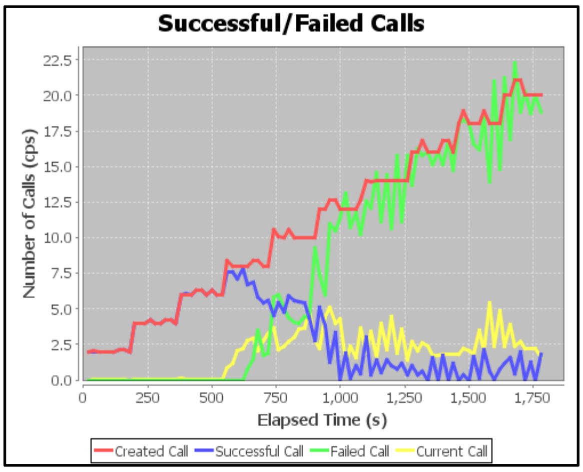
Figure 5-5: Number of calls and call setup time diagrams (Machine B)

Call Rate (cps)	Successful (cps)	Failed (%)	Call Setup Time (ms)
2	2	0	58
4	4	0	69
6	6	0.03	74
8	8	0.12	121
10	10	0.08	474
12	10.95	8.72	3451
14	8.24	40.83	5289
16	5.93	62.91	6077
18	5.38	70.14	6124
20	4.54	77.29	6608

Table 5-3: Average Statistics (Machine B)

From Figure 5-5, at 10 cps call rate the number of open call starts to rise. From this point on, the successful calls are decreasing. At 10 cps, most of the call setup time is larger than 200 milliseconds. The call setup time grows dramatically afterwards.

5.3.3 Single Computer: Machine C



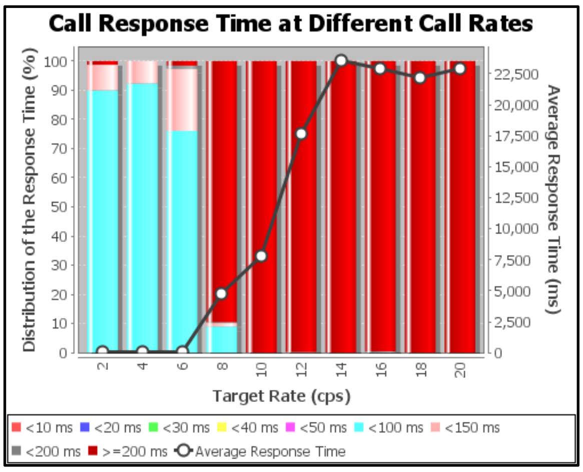


Figure 5-6: Number of calls and call setup time diagrams (Machine C)

Call Rate (cps)	Successful (cps)	Failed (%)	Call Setup Time (ms)
2	2	0	76
4	4	0.01	75
6	6	0.03	89
8	7.2	9.97	4735
10	5.67	43.28	7861
12	1.70	85.84	17583
14	1.36	90.28	18608
16	1.27	92.07	17569
18	0.87	95.19	17241
20	0.93	95.35	17813

Table 5-4: Average Statistics (Machine C)

With machine C, the unfinished call in the system starts increasing after 8 cps as displayed in Figure 5-6. The number failed call escalates at this point as well. After that, most of the calls are failed, only small number of calls can be processed by the target system.

5.3.4 Single Computer: Machine D

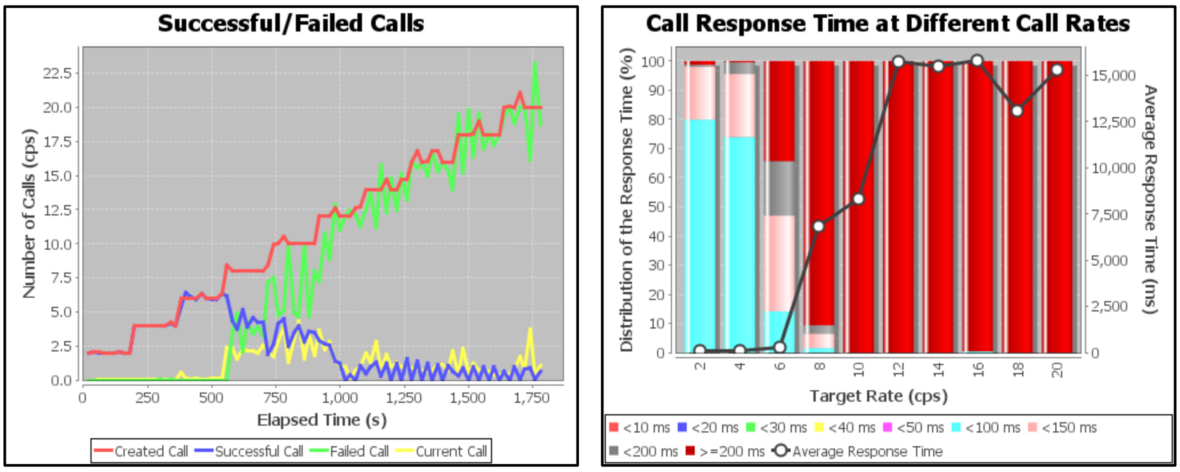


Figure 5-7: Number of calls and call setup time diagrams (Machine D)

Call Rate (cps)	Successful (cps)	Failed (%)	Call Setup Time (ms)
2	2	0	84
4	4	0.02	89
6	5.67	5.46	126
8	6.10	23.75	6476
10	4.42	55.93	8235
12	1.41	88.27	12685
14	1.13	91.93	13972
16	0.58	96.42	14391
18	0.56	96.89	13876
20	0.57	97.14	15197

Table 5-5: Average Statistics (Machine D)

Machine D has the same specifications as machine C. The performance results are also similar to the results of machine C. As shown in figure, most of the calls can be processed at 6 cps. The call setup time, and the number of failed call get considerably higher after 8 cps.

5.3.5 Load Balancing: Machine C and Machine D

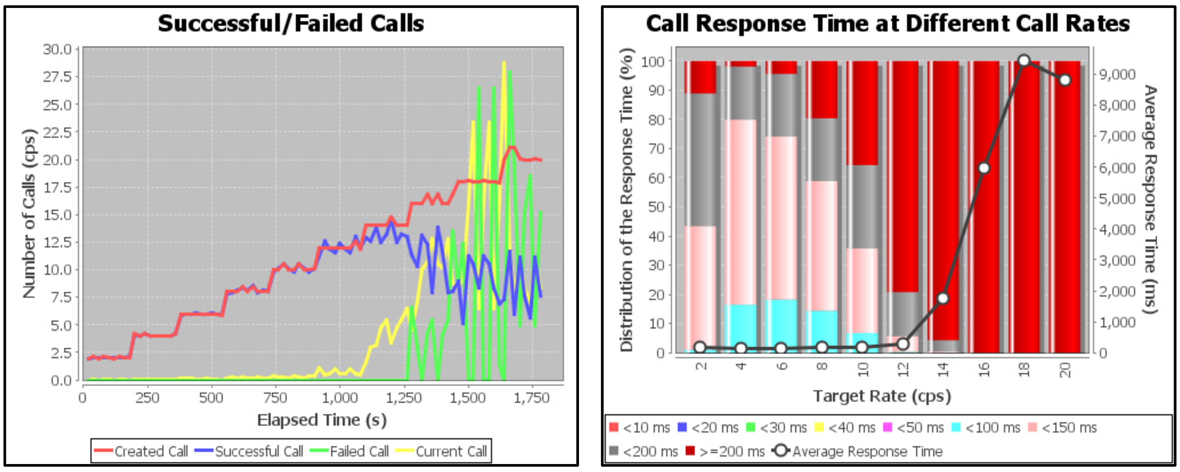


Figure 5-8: Number of calls and call setup time diagrams (Machines C, D)

Call Rate (cps)	Successful (cps)	Failed (%)	Call Setup Time (ms)
2	2	0	156
4	4	0.01	130
6	6	0.06	138
8	7.97	0.54	165
10	9.47	5.29	184
12	10.43	13.08	568
14	11.74	16.17	1743
16	12.52	21.75	5897
18	8.11	54.94	9256
20	8.33	58.37	9248

Table 5-6: Average Statistics (Machines C, D)

This target system is configured as explained in section 5.1.2. As presented in Figure 5-8, the number of unfinished call begins to climb up at 12 cps. At 10 cps, most of the requests can be processed by the system.

5.4 Results Comparison

We have evaluated the performance of EasyVPaBX on various configurations using the framework. The results of each configuration are presented earlier. In this section, we compare the gathered performance data and discuss about them.

From Table 5-2, Table 5-3, Table 5-4, Table 5-5, and Table 5-6 in section 5.3, we draw three comparison diagrams. Figure 5-9 and Figure 5-10 illustrate the number of successful calls and the failure percentage at different call rate respectively. From Figure 5-9, machine D started dropping some requests after 6 cps. Machine C, which has the same specification as machine D can process the requests at 6 cps, but the failure rate start rising after that.

The load balancing scheme which make use of both machine C and machine D performed well up to 8 cps. Small number of call rejections occurred after 10 cps and it grew afterwards.

With a faster processor compared to machine C and D, machine B can process most of the requests at 10 cps. However, at 12 cps it started rejecting a quantity of call requests.

Among the five configurations, machine A has the best result. It can handle call requests at 18 cps. Even though the clock speed of machine A' processor is lower than others, but the fact that the processor composes of two processing core is most probably be the reason for this.

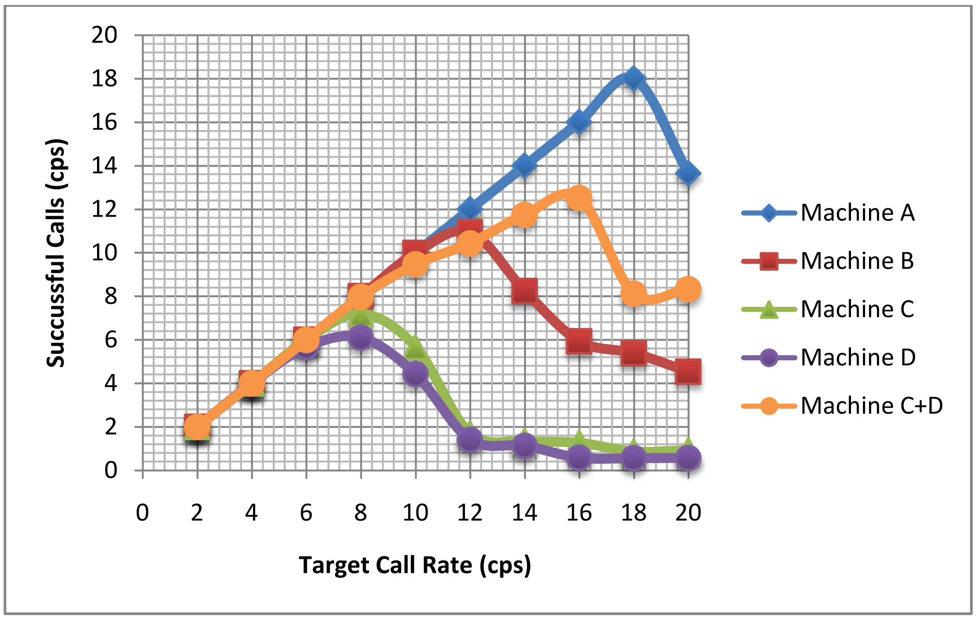


Figure 5-9: Successful call rate comparison

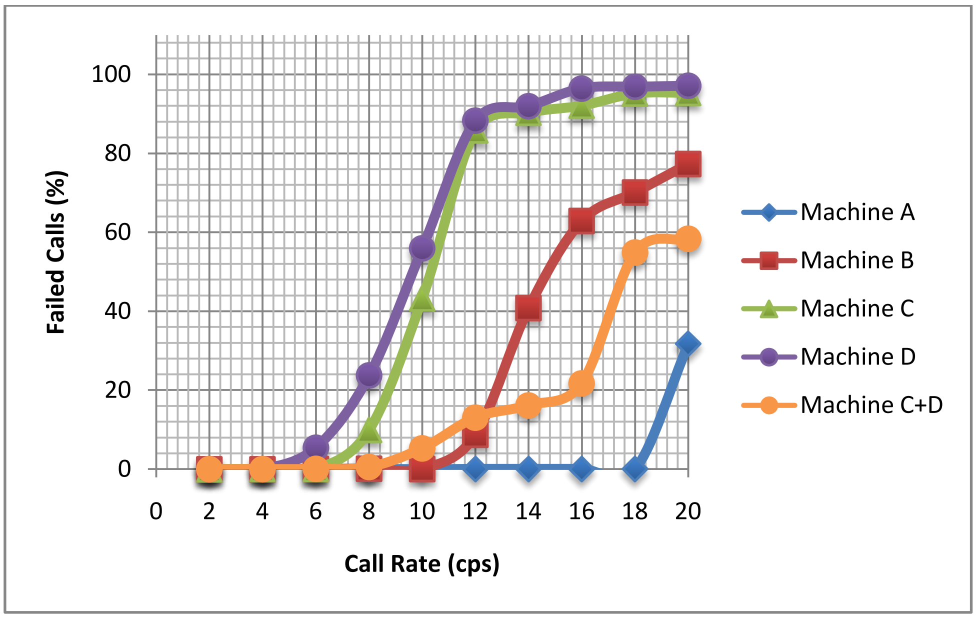


Figure 5-10: Failure percentage comparison

Apart from the successful and failure percentage, we also compare the call setup time of various systems. The comparison is presented in Figure 5-11 below. For all the single computer configurations, the call setup times at 2 cps are around 70 to 80 milliseconds. Then they get longer at higher request rates.

The load balancing setting, on the other hand, obviously has bigger response time (130 milliseconds) compared to the single computer settings. The reason behind this may due to the shared database which is resided on one of the computer. Access to data in remote database requires longer time than the local one.

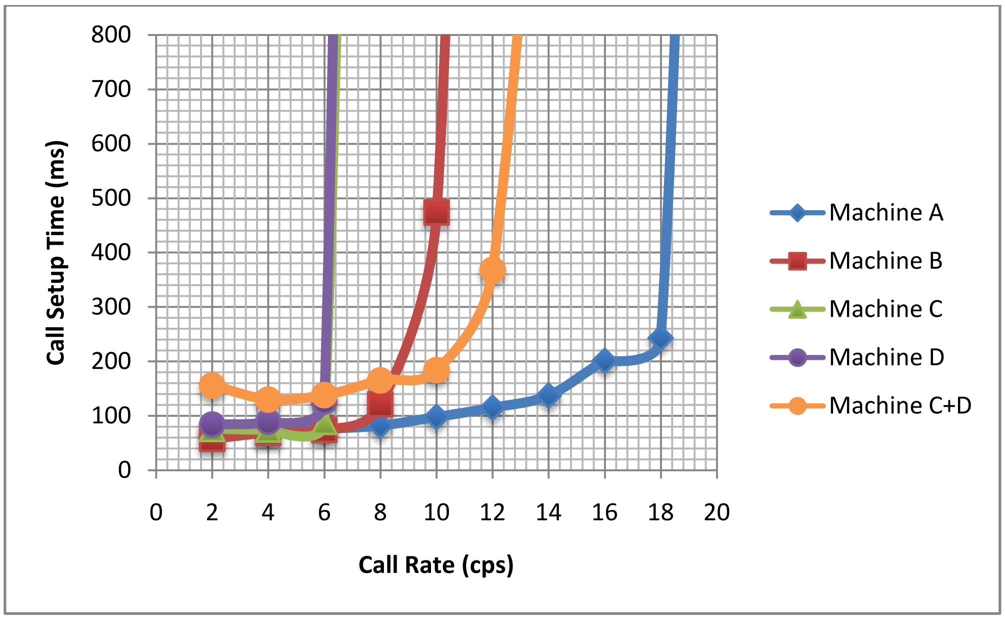


Figure 5-11: Call setup time comparison

For every configuration, the call setup time jumps dramatically after certain points. The huge increase of the call setup time indicates that the target system cannot process most of the call immediately. Some of the requests are then queuing up to be served by the system. After this point, waiting requests become more intense and the system then start rejecting certain amount of them. Considering Figure 5-10 and Figure 5-11, we can see that the rising of call setup time relates to the point where failed requests occurred.

5.5 Results Discussion

The successful/failure percentage as well as the call setup time measured by the framework can be used to determine the maximum capacity of the target system. For example, if we define the acceptable failure rate and the maximum call setup time to be 10% and 250 milliseconds[2] respectively, the system capacity for each configuration can be determined as highlighted rows in Table 5-7.

Machine	Call Rate (cps)	Failed (%)	Call Setup Time (ms)
A	16	0.04	201
	18	**0.07**	**243**
	20	31.76	3289
B	6	0.03	74
	8	**0.12**	**121**
	10	0.08	474
C	4	0.01	75
	6	**0.03**	**89**
	8	9.97	4735
D	4	0.02	89
	6	**5.46**	**126**
	8	23.75	6476
C+D	8	0.54	165
	10	**5.29**	**184**
	12	13.08	568

Table 5-7: Maximum capacity for different configurations

The capacity shown in the table is only one example of result interpretation. The appropriate criterion can vary from system to system.

[2] The default timer for retransmission of lost packets in SIP is 500 milliseconds. We choose 250 milliseconds as a time allowed for the target system to process the request. Since the experiments performed did not consider the network delay, we leave some space for this network delay in the real environment. However, the appropriate threshold must be defined based on thorough studies

The performance evaluations of EasyVPaBX performed in this chapter are done based on certain assumptions. The target systems are attached to the same local area network (LAN) as the user system which hosts the evaluation framework. As the two systems communicate through a high speed connection (100 Mbps), SIP messages travelling between them have a very small network delay. Consider the packet length is less than 1500 bytes and 14 packets per call session (section 4.2.2, Figure 4-2), so each session has no more than 20 kilobytes in total. For the traffic at 20 cps, SIP signaling packets require less than 3.2 Mbps. Therefore, the network is not considered as a factor in these experiments. However, in realistic circumstances, SIP packets require some amount of time in the network to reach the server. This network delay affects an overall performance perceived by users. For this reason, further experiments on a real network infrastructure hosting the product should be taken in consideration.

6 Framework Evaluation and Discussion

In this chapter, we evaluate the performance evaluation framework we have developed. We start with section 6.1 presenting the verification of the framework against the requirement specification. In addition, section 6.2 compares the framework and the tool SIPp. Then, some issues about the framework are discussed in section 6.3.

6.1 Requirement Verification

We designed and implemented the performance evaluation framework according to the requirement specified in section 4.2. The requirement specification includes supporting two performance metrics (i.e. system capacity and call setup time), two embedded scenarios, and user-friendly graphical interface.

Regarding the system capacity, the framework developed can generate SIP traffic at different rate. It can measure the number of call attempts successfully processed by a target system. These results from different rate are then used to determine the system capacity. As for the call setup time, the framework records the time taken from the first *INVITE* request until a corresponding *200 OK* reply is received. This is configured in the XML scenario files composed in section 4.2.2. Examples of the two measurements are presented earlier in section 5.3.

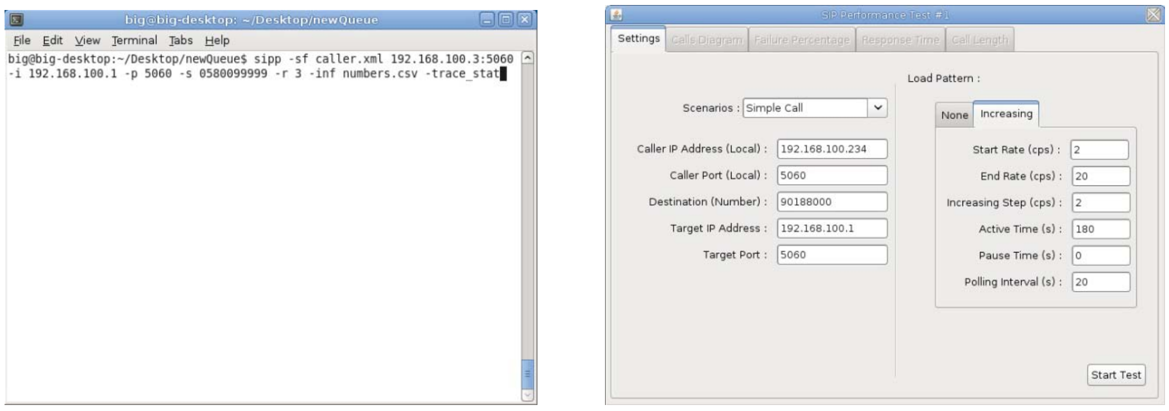
The two scenarios (i.e. *simple call*, and *queue*) were embedded in the framework. We created the two XML files based on SIP sequences illustrated in Figure 4-2 and Figure 4-3. These two files were contained in the specific directory in the framework. They are used as a parameter to the traffic generator.

For user-friendliness, we developed a graphical user interface as shown in Figure 4-5. Users can configure evaluation parameters through the interface. Moreover, the real-time resulting statistics are displayed in easy-to-understand diagrams as shown in Figure 4-6, Figure 4-7, Figure 4-8, and Figure 4-9.

6.2 Comparisons to SIPp

In this section, we present comparisons between the framework we developed and the existing tool, SIPp. In general, the framework offers better usability both in configuring startup parameters and results interpretation.

6.2.1 Evaluation Parameters Specification


big@big-desktop: ~/Desktop/newQueue
File Edit View Terminal Tabs Help

```
big@big-desktop:~/Desktop/newQueue$ sipp -sf caller.xml 192.168.100.3:5060
-i 192.168.100.1 -p 5060 -s 0580099999 -r 3 -inf numbers.csv -trace_stat
```

Scenarios : Simple Call
Caller IP Address (Local) : 192.168.100.234
Caller Port (Local) : 5060
Destination (Number) : 90188000
Target IP Address : 192.168.100.1
Target Port : 5060

Start Rate (cps) : 2
End Rate (cps) : 20
Increasing Step (cps) : 2
Active Time (s) : 180
Pause Time (s) : 0
Polling Interval (s) : 20

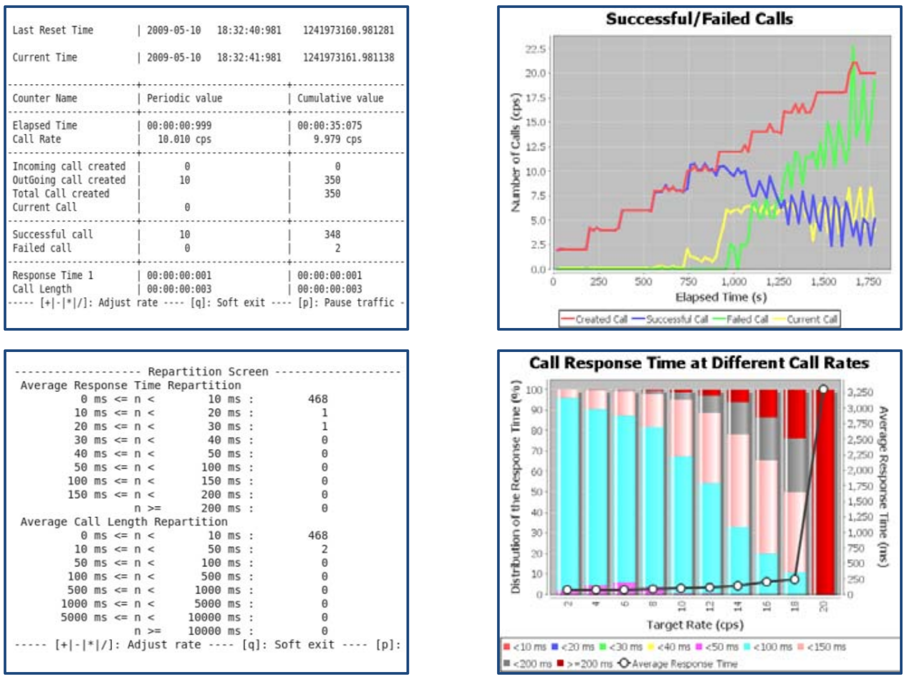
Figure 6-1: Comparison of parameter specification

For starting up the traffic generator, the startup parameters have to be specified. With SIPp, users are required to type all the parameters through a command line as shown in the left side of Figure 6-1. On the other hand, the framework provides a form where users can select and specify configurations for the evaluation.

In addition to the startup parameters, two embedded scenarios are provided by the framework. Therefore, users do not have to compose the XML scenario files themselves. Moreover, the framework can be configured to increase or pause the traffic generation during the evaluation while with SIPp, users have to explicitly insert the command to control the traffic.

6.2.2 Results Presentation

SIPp offers only the text-based interface. The performance results are presented in tabular form as in the left side of Figure 6-2. The developed framework improved the presentation by introducing graphical diagrams as shown in the right side of the figure. The graphical diagrams are easier and more convenient for result interpretation. In other words, this graphic interface promotes the user-friendliness of the performance evaluation framework.



```
Last Reset Time          | 2009-05-10   18:32:40:981    1241973160.981281

Current Time             | 2009-05-10   18:32:41:981    1241973161.981138

-------------------------+---------------------------+--------------------
Counter Name             | Periodic value            | Cumulative value
-------------------------+---------------------------+--------------------
Elapsed Time             | 00:00:00:999              | 00:00:35:075
Call Rate                |   10.010 cps              |    9.979 cps
-------------------------+---------------------------+--------------------
Incoming call created    |        0                  |        0
OutGoing call created    |       10                  |      350
Total Call created       |                           |      350
Current Call             |        0                  |
-------------------------+---------------------------+--------------------
Successful call          |       10                  |      348
Failed call              |        0                  |        2
-------------------------+---------------------------+--------------------
Response Time 1          | 00:00:00:001              | 00:00:00:001
Call Length              | 00:00:00:003              | 00:00:00:003
----- [+|-|*|/]: Adjust rate ---- [q]: Soft exit ---- [p]: Pause traffic -
```

```
------------------- Repartition Screen -------------------
 Average Response Time Repartition
          0 ms <= n <        10 ms :       468
         10 ms <= n <        20 ms :         1
         20 ms <= n <        30 ms :         1
         30 ms <= n <        40 ms :         0
         40 ms <= n <        50 ms :         0
         50 ms <= n <       100 ms :         0
        100 ms <= n <       150 ms :         0
        150 ms <= n <       200 ms :         0
                  n >=      200 ms :         0
 Average Call Length Repartition
          0 ms <= n <        10 ms :       468
         10 ms <= n <        50 ms :         2
         50 ms <= n <       100 ms :         0
        100 ms <= n <       500 ms :         0
        500 ms <= n <      1000 ms :         0
       1000 ms <= n <      5000 ms :         0
       5000 ms <= n <     10000 ms :         0
                  n >=    10000 ms :         0
----- [+|-|*|/]: Adjust rate ---- [q]: Soft exit ---- [p]:
```

Figure 6-2: Comparisons of result presentations

6.3 Discussions

6.3.1 Performance Evaluation based on SIP Signals

Multimedia call sessions can be separated into two planes, a control plane and a media plane. SIP defines a set of messages in the control plane. These SIP messages are used for signaling establishment, modification, and termination of call sessions. The media session is handled by other protocols (e.g. RTP) and is not a part of SIP. As mentioned in the scope of this thesis work in section 1.2, the performance evaluation framework only deals with SIP call sessions. SIP traffics generated by the framework do not contain any media data. Therefore, performance results produced by the framework are based on SIP signaling alone. Media data in each call session can create variations in the performance, thus additional study about the idea may be considered for more accurate results.

6.3.2 Platform Compatibility

Regarding platform compatibility, the current version of the performance evaluation framework has been developed to work on Linux operating systems. The interactions between the framework and the underlying application and files are implemented using Linux-based scripts. Hence, some modifications are needed in case of porting the framework to other operating systems.

6.3.3 Proxy Server and Call Recipient

SIP traffics generated are forwarded to the target system by a proxy server (e.g. OpenSIPS). The call session is established when the target system creates another SIP session with a final call recipient. Therefore, the proxy server and the call recipient are necessary parts of the performance evaluation framework.

Nevertheless, they are not included inside the developed application. Even though it seems more convenient embedding them to avoid additional steps for manual configurations of these components, excluding them can be more flexible. A system with its own proprietary SIP proxy or call recipients can also be evaluated using the framework.

6.3.4 Determination of the System Capacity

The evaluation result from the framework includes successful/failure percentage as well as the call setup time measured by the framework. The maximum capacity of the target system is not a direct output of the framework itself. Since the capacity of the system depends on a criterion of the service, failure rate and setup time thresholds must be defined. These threshold values are then used together with performance results from the framework to determine the capacity of the system.

6.3.5 Traffic Patterns

Concerning the traffic pattern used in the evaluations, a uniform distribution of requests is applied. Although the framework can control the rate of requests generation, the distribution cannot be configured. This is a limitation of the underlying traffic generator (SIPp). The current version of SIPp (v.3.1) supports only uniformly distributed traffic generation. For more realistic traffic pattern, some modifications to the framework and/or the underlying tools are needed.

Apart from the issues discussed above, the performance evaluation framework can facilitate the measurements of the successful/failure rate of call requests, and the call setup time of SIP-based telecommunication systems. These performance data are used for defining the capacity of such systems. In addition, the framework comes with an easy-to-use graphical interface which promotes usability and understandability in evaluating performance of a system.

7 Conclusions

7.1 Summary

Evaluating a performance of telecommunication systems can be a challenging task. It comprises of studying different scenarios, creating simulations, capturing statistics data, and interpreting results. Nevertheless, an accurate capacity of a system can be valuable for system developers as well as prospect customers. Therefore, the objective of this work was to develop a framework for performance evaluation of a SIP-based telecommunication system.

In this thesis, we studied the available tool which can be used for performance evaluation of a SIP system. In addition, the necessary components such as a SIP proxy and call recipient were also taken into account. SIPp and OpenSIPS were chosen as an evaluation tool, and a proxy respectively. Number of experiments had been carried out with these elements. They were then put together to found a basis of the performance evaluation framework.

We designed and implemented the performance evaluation framework using SIPp as a traffic generator. The framework facilitates measurements of number of calls that the system can process and also call setup times at different call rate. A user-friendly graphic interface (GUI) was built to promote the usability of the framework. Evaluation parameters such as call scenario, traffic generation rate, duration of the test, etc, can be configured through the GUI. Moreover, real-time updated graphical diagrams showing offer an easier way to interpret the results.

We also studied EasyVPaBX which is a SIP-based system providing controls over call behaviors. Different sets of computers hosting the EasyVPaBX application had been setup as the target systems. We performed several evaluations of these systems using the framework. The performance results gathered from the experiments were presented, compared, and discussed.

Concerning the capacity of the framework developed, we experienced in section 3.3.2 that number of calls which can be handled by OpenSIPS is dependent on the memory allocated. Since we used OpenSIPS as the SIP proxy in the performance evaluation framework, this constraint also applies. Consequently, the framework should not be used for evaluating systems which have higher capacity than the framework itself to avoid mistaken results. However, the EasyVPaBX system evaluated in chapter 5 has much less capacity than this limitation. Therefore, it does not affect the performance evaluation of the targets.

7.2 Future Work

The performance evaluation framework provides several features facilitating a dimensioning of SIP-based telecommunication systems. Nevertheless, there exists some functionality which can be added in the future. Automated test using a real-time result as a feedback is one good example. More intelligence may be introduced in decision making based on the feedback. For instance, the characteristic of generated traffics can be adjusted to suit the current condition of a target system. Furthermore, other call scenarios can be studied, implemented, and embedded in the framework for convenient usage.

Regarding the performance evaluation of EasyVPaBX, more thorough evaluations should be carried out. As discussed in section 5.5, the experiments performed in this thesis did not consider a network delay. In the actual working environment, this delay may not be negligible. Therefore, performance evaluation with the true user perception of the service should also be concerned. In addition to experiments with real network infrastructure, other system configurations may be studied and evaluated. This could be beneficial for future deployment plans of EasyVPaBX.

References

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

[2]. *SIPstone - Benchmarking SIP Server Performance.* **Henning Schulzrinne, Sankaran Narayanan and Jonathan Lennox.** Columbia University : s.n., April 2002.

[3]. *A Tutorial on SIP Applicaiton Server Performance and Benchmarking.* **Curtis Hrischuk, Ph.D. and Gary DeVal.** Reno, Nevada USA : The Computer Measurement Group (CMG), December 2006. 31st Annual International Conference of The Computer Measurement Group. Inc.

[4]. *SIPp.* [Online] http://sipp.sourceforge.net.

[5]. *Evaluating SIP Server Performance.* **Erich M. Nahum, John Tracey and Charles P. Wright.** San Diego, California, USA : ACM, June 2007. Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. ISBN 978-1-59593-639-4.

[6]. **Stephan Wanke, et al.** Measurement of the SIP Parsing Performance in the SIP Express Router. *Lecture Notes in Computer Science.* s.l. : Springer Berlin / Heidelberg, 2007, pp. 103-110.

[7]. *The Throughput Performance of SIP Session at Different Call Rates and Maximum Header Length.* **Ling-Feng Chiang, Cheng-Chi Yu and Jiang-Whai Dai.** 2005. International Symposium on Communications (ISCOM 2005) .

[8]. *SIP Parsing Offload: Design and Performance.* **Jia Zou, et al.** November 2007. Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE. pp. 2774-2779. ISBN 978-1-4244-1043-9.

[9]. **Alan B. Johnston.** *SIP: Understanding the Session Initiation Protocol.* 2nd Edition. s.l. : Artech House, 2004. ISBN 1580536557.

[10]. **Gonzalo Camarillo.** *SIP Demystified.* s.l. : McGraw-Hill Professional, August 2001. ISBN 0071373403.

[11]. **H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson.** *RTP: Real-time Transport Protocol, RFC 1889.* January, 1996.

[12]. **H. Schulzrinne, A. Rao, R. Lanphier.** *Real-Time Streaming Protocol (RTSP), RFC 2326.* April 1998.

[13]. **M. Handley, V. Jacobson.** *Session Description Protocol (SDP), RFC 2327.* April 1998.

[14]. OpenSIPS. *OpenSIPS.* [Online] http://www.opensips.org.

[15]. *Gintel AS.* [Online] http://www.gintel.com/.

[16]. **Gintel AS.** *EasyVPaBX Product Sheet.* s.l. : Gintel AS, 2008.

[17]. **Anu A. Gokhale.** *Introduction to Telecommunications.* 2nd Edition. s.l. : Cengage Learning, 2004. pp. 152-154. ISBN 9781401856489.

[18]. Advantages of Hosted PBX Solution. *Telecom Auditing Guide.* [Online] [Cited: April 2, 2009.] http://www.telecomauditguide.com/pbx/advantages-of-a-hosted-pbx-solution/.

[19]. **Gintel AS.** *EasyVPaBX - Complete Hosted Telephony Solution for Enterprises and SMEs : Service Description Document V2.0.* s.l. : Gintel AS, September 2008.

[20]. Apache Tomcat. *The Apache Software Foundation.* [Online] http://tomcat.apache.org/.

[21]. Oracle WebLogic Server. *Oracle.* [Online] http://www.oracle.com/appserver/.

[22]. Wireshark. [Online] http://www.wireshark.org.

[23]. SJphone. *SJ Labs.* [Online] http://www.sjlabs.com.

[24]. Java SE. *Sun Microsystems.* [Online] http://java.sun.com/javase/.

[25]. Swing Application Framework. [Online] https://appframework.dev.java.net/.

[26]. JFreeChart. [Online] http://www.jfree.org/jfreechart/.

[27]. IP Media Server. *Dialogic.* [Online] http://www.dialogic.com/products/ip_enabled/ip_media_server/.

[28]. Thread. *Java Platform SE 6.* [Online] http://java.sun.com/javase/6/docs/api/java/lang/Thread.html.

[29]. JFreeChart API Documentation. [Online] http://www.jfree.org/jfreechart/api/javadoc/index.html.

[30]. Orable WebLogic SIP Server 3.1. [Online] http://download.oracle.com/otn/bea/weblogic/B46920-01.zip.

[31]. MySQL - Open Source Database. [Online] http://www.mysql.com/.

[32]. **H. Sinnereich and A. Johnston.** *Internet Communications Using SIP: Delivering VOIP and Multimedia Services with Session Initiation Protocol.* s.l. : Wiley Publishing Inc., 2006.

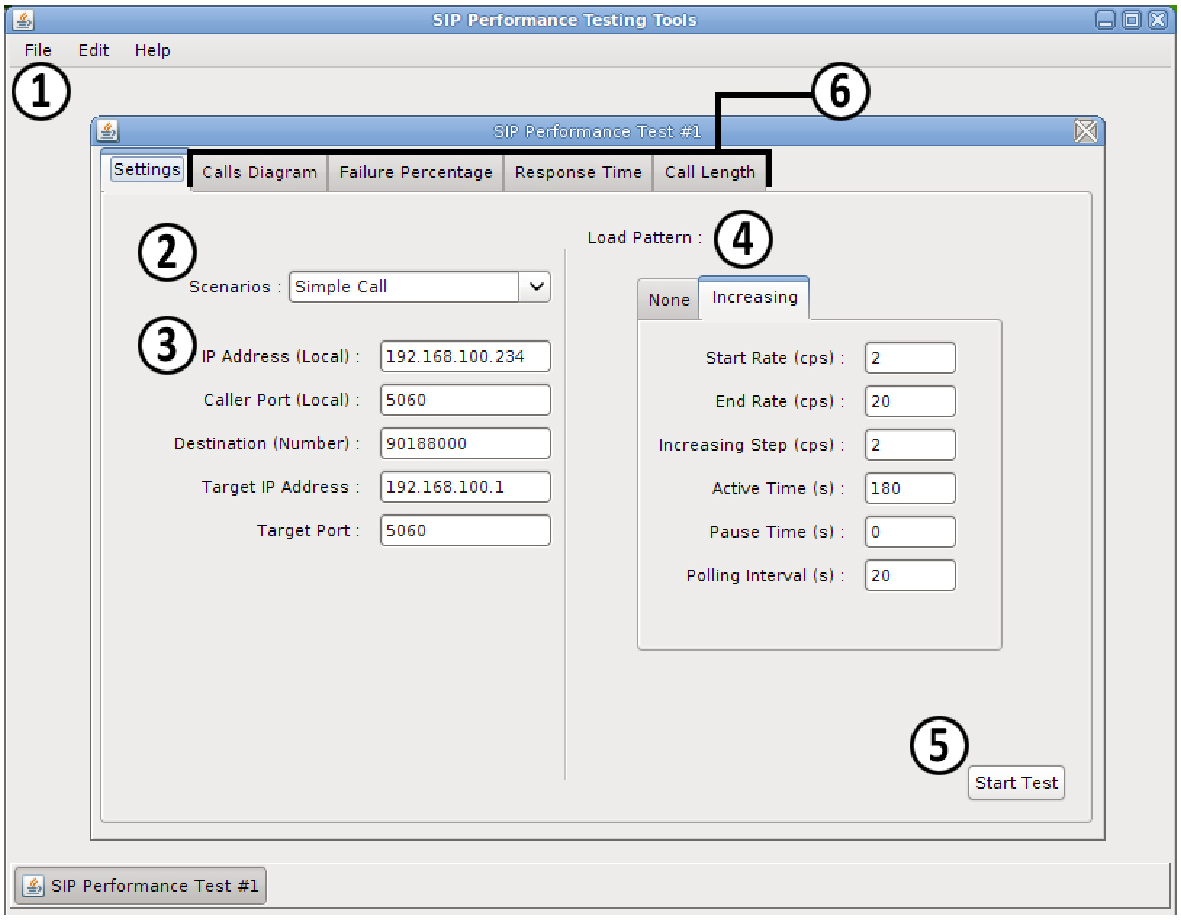
[33]. GNU Scientific Library. *GNU Project.* [Online] http://www.gnu.org/software/gsl/.

[34]. **Dorgham Sisalem, Jiri Kuthan.** *SIP Tutorial.* s.l. : iptel.org, Tekelec, March 2007.

Appendix A
User Manual

The performance evaluation framework is developed on Java 2 Platform standard edition. To use the framework, JRE 6 is recommended. Additionally, SIPp must be properly installed on the system. Other configured components (a SIP proxy and a call recipient) are also required.

To install the framework, one executable JAR file ($SIPPerformance.jar$) and two subdirectories ($/lib$ and $/predefined$) must be placed in a destination folder. In order to start the framework, execute the JAR file. The graphical user interface of the framework will appear as shown in Figure A-1.



Scenarios : Simple Call
IP Address (Local) : 192.168.100.234
Caller Port (Local) : 5060
Destination (Number) : 90188000
Target IP Address : 192.168.100.1
Target Port : 5060

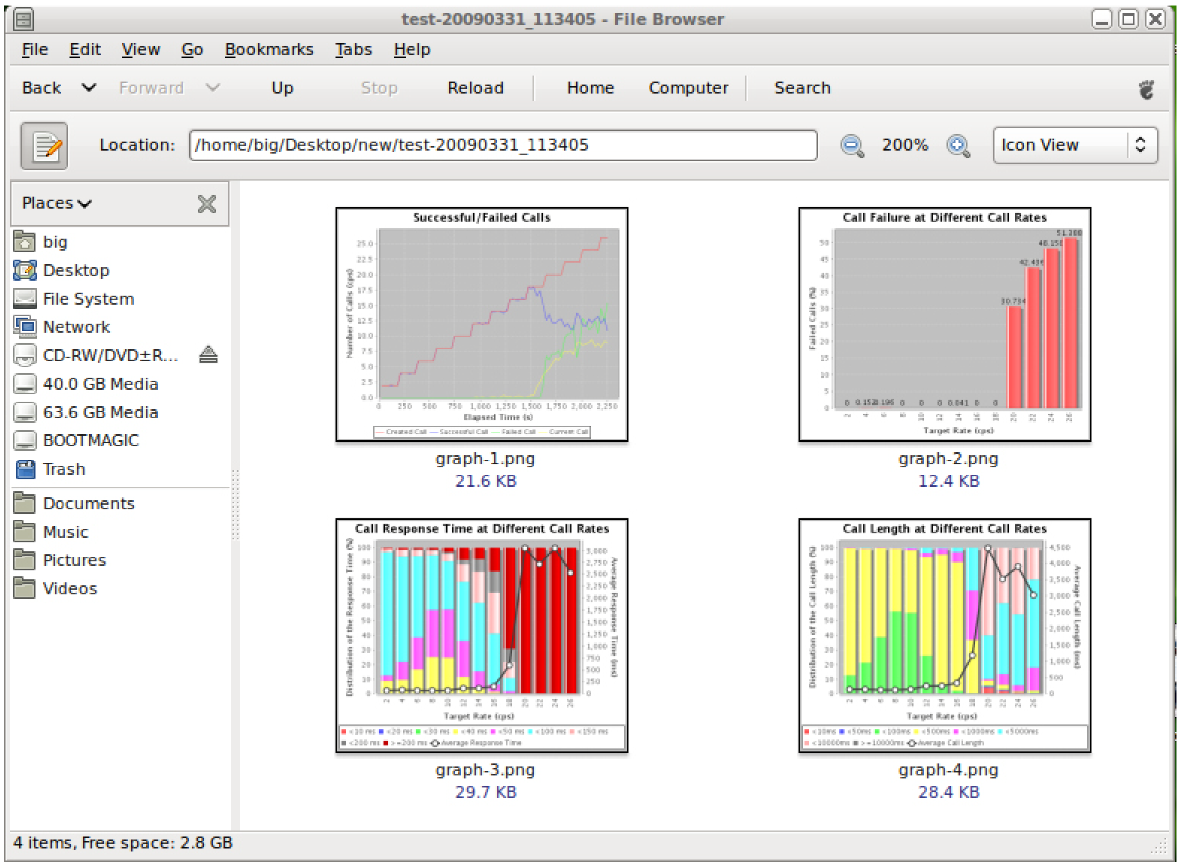
Load Pattern : Increasing
Start Rate (cps) : 2
End Rate (cps) : 20
Increasing Step (cps) : 2
Active Time (s) : 180
Pause Time (s) : 0
Polling Interval (s) : 20

Figure A-1: The performance evaluation framework (user manual)

To start an evaluation, follow these steps

1. Select *File* > *New Test*
2. Select a scenario (Simple Call or Queue).
3. Configure the IP addresses and port numbers of the traffic generator as well as the target system.
4. Choose a load pattern and configure the parameters.
5. Click *Start Test*.
6. Once the test has been started, the framework provides four panels for result monitoring. Users can switch between these panels to see the real-time updated graphs.

After the test has been completed, the four diagrams are written in the newly created folder. Figure A-2 shows the output image files from the framework.



Location: /home/big/Desktop/new/test-20090331_113405

200%

graph-1.png
21.6 KB

graph-2.png
12.4 KB

graph-3.png
29.7 KB

graph-4.png
28.4 KB

4 items, Free space: 2.8 GB

Figure A-2: Output image files from the framework

Appendix B
Simple Call Scenario File

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "[field0].dtd">
<scenario name="Basic Sipstone UAC">

  <send retrans="500">
    <![CDATA[
      INVITE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: [field0] <sip:[field0]@[remote_ip]:[remote_port]>;tag=[call_number]
      To: sut <sip:[service]@[remote_ip]:[remote_port]>
      Call-ID: [call_id]
      CSeq: 1 INVITE
      Contact: sip:[field0]@[local_ip]:[local_port]
      ServiceKey: 183
      Max-Forwards: 70
      Subject: Performance Test
      Content-Length: 0

    ]]>
  </send>

  <recv response="100" optional="true"></recv>
  <recv response="180" optional="true"></recv>
  <recv response="200" rrs="true" rtd="true"></recv>

  <send>
    <![CDATA[

      ACK [next_url] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: [field0] <sip:[field0]@[remote_ip]:[remote_port]>;tag=[call_number]
      To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
      Contact: sip:[field0]@[local_ip]:[local_port]
      Call-ID: [call_id]
      CSeq: 1 ACK
      Max-Forwards: 3
      Content-Length: 0

    ]]>
  </send>
  <send>
    <![CDATA[
      BYE [next_url] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: [field0] <sip:[field0]@[remote_ip]:[remote_port]>;tag=[call_number]
      To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
      Call-ID: [call_id]
      CSeq: 2 BYE
      Contact: sip:[field0]@[local_ip]:[local_port]
      Max-Forwards: 70
      Content-Length: 0

    ]]>
  </send>
  <recv response="200" crlf="true"></recv>

  <ResponseTimeRepartition value="10, 20, 30, 40, 50, 100, 150, 200" />
  <CallLengthRepartition value="10, 50, 100, 500, 1000, 5000, 10000" />

</scenario>
```

Appendix C
Queue Scenario File

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "[field0].dtd">
<scenario name="Basic Sipstone UAC">
  <User variables="q" />

  <send retrans="500">
    <![CDATA[
      INVITE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: [field0] <sip:[field0]@[remote_ip]:[remote_port]>;tag=[call_number]
      To: sut <sip:[service]@[remote_ip]:[remote_port]>
      Call-ID: [call_id]
      CSeq: 1 INVITE
      Contact: sip:[field0]@[local_ip]:[local_port]
      ServiceKey: 183
      Max-Forwards: 70
      Subject: Performance Test
      Content-Length: 0

    ]]>
  </send>

  <recv response="100" optional="true"></recv>

  <recv response="180" optional="true"></recv>

  <recv response="200" rrs="true" rtd="true">

    <action>
      <ereg regexp="snowshore" search_in="msg" case_idp="true" assign_to="q"/>
      </action>

  </recv>

  <send>
    <![CDATA[

      ACK [next_url] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: [field0] <sip:[field0]@[remote_ip]:[remote_port]>;tag=[call_number]
      To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
      Contact: sip:[field0]@[local_ip]:[local_port]
      Call-ID: [call_id]
      CSeq: 1 ACK
      Max-Forwards: 3
      Content-Length: 0

    ]]>
  </send>

<!-- If not in queue, jump to the real call -->

  <nop condexec="q" condexec_inverse="true">
    <action> <jump value="2" /> </action>
  </nop>

<!-- Otherwise, wait for an INVITE -->

  <recv request="INVITE" rrs="true" crlf="true"/>
  <send>
    <![CDATA[

      SIP/2.0 100 Giving a try
      [last_Via:]
      [last_From:]
      [last_To:];tag=[pid]SIPpTag01[call_number]
      [last_Call-ID:]
      [last_Record-Route:]
      [last_CSeq:]
      Contact: <sip:[local_ip]:[local_port];transport=[transport]>
      Content-Length: 0

    ]]>
  </send>

  <send>
    <![CDATA[
      SIP/2.0 200 OK
      [last_Via:]
      [last_From:]
      [last_To:]
      [last_Call-ID:]
      [last_Record-Route:]
      [last_CSeq:]
      Contact: <sip:[local_ip]:[local_port];transport=[transport]>
      Content-Length: 0

    ]]>
  </send>

  <recv request="ACK" crlf="true"/>

<!-- Start of the real call -->
  <label id="2" />
  <pause />
<!-- End of the real call -->

  <send>
    <![CDATA[
      BYE [next_url] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
      From: [field0] <sip:[field0]@[remote_ip]:[remote_port]>;tag=[call_number]
      To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
      Call-ID: [call_id]
      CSeq: 2 BYE
      Contact: sip:[field0]@[local_ip]:[local_port]
      Max-Forwards: 70
      Content-Length: 0

    ]]>
  </send>

  <recv response="200" crlf="true"></recv>

  <ResponseTimeRepartition value="10, 20, 30, 40, 50, 100, 150, 200" />

  <CallLengthRepartition value="10, 50, 100, 500, 1000, 5000, 10000" />

</scenario>
```

Appendix D
SIP Miscellaneous

Method	Description
INVITE	Session setup
ACK	Acknowledgement of final response to INVITE
BYE	Session termination
CANCEL	Pending session cancellation
REGISTER	Registration of a user's URI
OPTIONS	Query of options and capabilities
INFO	Mid-call signaling transport
PRACK	Provisional response acknowledgment
UPDATE	Update session information
REFER	Transfer user to a URI
SUBSCRIBE	Request notification of an event
NOTIFY	Transport of subscribed event notification
MESSAGE	Transport of an instant message body
PUBLISH	Upload presence state to a server

Table D-1: SIP methods (adapted from [32])

Class	Description
1xx	Provisional or Informational Request is progressing but not yet complete.
2xx	Success Request has been completed successfully
3xx	Redirection Request should be tried at another location
4xx	Client Error Request was not completed because of an error in the request
5xx	Server Error Request was not completed because of an error in the recipient
6xx	Global Failure Request has failed and should not be retried again

Table D-2: SIP responses (adapted from [32])

SIP Header	Description
Via	indicates the transport used for the transaction and identifies the location where the response is to be sent. It contains the transport protocol, the host name or IP address, as well as the port number (optional). e.g. *Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK-1*
To	shows the logical recipient of the request. It contains a name and a SIP or SIPS URI of the recipient. e.g. *To: Carol <sip:carol@chicago.com>*
From	indicates the logical identity of the initiator of the request. It contains a URI and optionally a display name. In addition, a new "tag" parameter chosen by the UAC is appended after the URI. e.g. *From: "Bob" <sips:bob@biloxi.com> ;tag=a48s*
Max-Forwards	the maximum number of hops a request can transit on the way to its destination. The number is decremented by one each hop. This is for loop prevention.
Call-ID	acts as a unique identifier to group a series of messages. It is required that the Call-ID remains the same for all requests and responses in a dialog. The UAC selects a globally unique identifier for a session. e.g. *Call-ID: f81d4fae-7dec-11d0-a765@foo.bar.com*
CSeq	identifies and orders the transaction. It consists of a sequence number and a method. e.g. *CSeq: 4711 INVITE*
Contact	provides a SIP URI which can be used to contact the UA for subsequent requests.
Content-Type	provides a description of the body part of the message. e.g. *Content-Type: SDP*
Content-Length	contains the length (byte) of the message content. The 0 value indicates no content in the message body.

Table D-3: SIP Headers (summarized from [1])

Appendix E
SIPp Miscellaneous

Parameter	Description
-bg	run SIPp in background mode
-ci	set the local control IP address for remote-controlling
-cp	set the local control port for remote-controlling
-d	set the length of the calls in millisecond, default is 0
-f	set the statistics report frequency, default is 1s
-fd	set the statistics dump frequency, default is 60s
-i	set the local IP, default is the primary host IP
-inf	inject values from an external CSV file
-l	set the maximum number of the simultaneous call If this number is reached, SIPp pause the traffic generation until open call number is less than this. Default value is 3 x call length x call rate.
-m	the number of total call to be generated
-max_retrans	maximum UDP retransmission Default is 5 for INVITE message, 7 for others.
-p	the local port number of this SIPp. Default is random free port
-r	the initial call rate
-rate_scale	the rate step used for hot keys
-rp	the rate period, default is 1000 ms
-sf	use the external XML scenario file
-sn	use the embedded scenario
-trace_msg	save sent and received SIP messages to the log file
-trace_err	save unexpected messages to the log file
-trace_stat	save all statistics to the file
-trace_counts	save individual message counts to the file
-trace_rtt	save all the response times to the file

Table E-1: SIPp startup parameters

Command	Description
dump tasks	Save the list of active tasks to the error log
set rate X	Set the call rate to X
set rate-scale X	Set the step of rate as used by hot keys +,-,*,/
set users X	Set the number of users
set limit X	Set the open call limit
set index <true|false>	Display message indexes in the scenario screen
set display <main|ooc>	Change the displayed scenario
trace <log> <on|off>	Turn log on or off

Table E-2: SIPp interactive commands

Hot Key	Description
+	Increase the call rate by one step
-	Decrease the call rate by one step
*	Increase the call rate by ten step
/	Decrease the call rate by ten step
c	Enter the command mode
p	Pause the traffic
q	Quit the SIPp (wait for ongoing calls to end)
Q	Force quit SIPp
s	Dump the screen
1-9	Switch between display screens

Table E-3: SIPp hot keys

Command	Attribute	Description
<send>	retrans	to adjust T1 timer used in UDP retransmission. For example, <send retrans=”500”> sets timer to 500 ms.
<recv>	response	to specify what SIP message code is expected. For example, <recv response=”200”> tells SIPp to wait for an answer with the code 200.
	request	to wait for a SIP request message. For instance, <recv request=”ACK”> will wait for an ACK message.
	optional	to indicate that the SIP message is optional. For instance, <recv response=”100” optional=”true”> tells SIPp that this 100 message can be received but not required.
	rrs	stands for record route set. With this attribute, SIPp will save the Record-Route header of the received SIP message for later use. For example, <recv response=”100” rrs=”true”> lets the received route be stored and is accessible by [routes] keyword.
	timeout	specify a timeout to wait for the message. For instance, <recv timeout=”10000”> tells SIPp to abort the call if the expected message is not received within 10 seconds.
<pause>	milliseconds	to specify the delay period in milliseconds. For example, <pause milliseconds=”5000”/> tells SIPp to pause the scenario for 5 seconds.
	variable	to pause for the number of milliseconds stored in the variable. For instance, <pause variable=”1”/> uses value in variable 1 as the pause time.
	distribution	to use certain statistical distributions to determine the delay. Available distributions include uniform, normal, exponential, etc. Some distributions require recompilation with GSL [33] support.

Table E-4: SIPp unique attributes for certain commands

Attribute	Description
start_rtd	starts one of the Response Time Duration timers
Rtd	stops one of the Response Time Duration timers
repeat_rtd	allows the timer to be counted more than once
Clrf	prints an empty line on the display screen
Next	jumps to another part of the script
test (used with *next*)	jumps to another part of the script only if the variable is set
chance (used with *test*)	adds probability to jump to another part
Condexec	executes the command only if the variable is set
condexec_inverse	executes the command only if the variable is not set
Counter	increments the specified counter

Table E-5: Attributes in XML scenario files

Keyword	Description
[service]	Service field, as passed in the -s service_name
[remote_ip]	Remote IP address, as passed on the command line.
[remote_port]	Remote IP port, as passed on the command line.
[transport]	Depending on the value of -t parameter, "UDP" or "TCP".
[local_ip]	Will take the value of -i parameter.
[local_port]	Will take the value of -p parameter.
[len]	Computed length of the SIP body. To be used in "Content-Length" header.
[call_number]	Index. The call_number starts from "1" and is incremented by 1 for each call.
[cseq]	Generates automatically the CSeq number. The initial value is 1. It can be changed by using the -base_cseq command line option.
[call_id]	A call_id identifies a call and is generated by SIPp for each new call. In client mode, it is mandatory to use the value generated by SIPp.
[routes]	If the "rrs" attribute in a recv command is set to "true", then the "Record-Route:" header received is stored and can be recalled using the [routes] keyword
[next_url]	If the "rrs" attribute in a recv command is set to "true", then the [next_url] contains the contents of the Contact header
[branch]	Provide a branch value which is a concatenation of magic cookie (z9hG4bK) + call number + message index in scenario.

Table E-6: Keywords in XML scenario files

Counters	Description
StartTime	Date and time when the test has started.
LastResetTime	Date and time when periodic counters where last reset.
CurrentTime	Date and time of the statistic row.
ElapsedTime	Elapsed time.
CallRate	Call rate (calls per seconds).
IncomingCall	Number of incoming calls.
OutgoingCall	Number of outgoing calls.
TotalCallCreated	Number of calls created.
CurrentCall	Number of calls currently ongoing.
SuccessfulCall	Number of successful calls.
FailedCall	Number of failed calls (all reasons).
OutOfCallMsgs	Number of SIP messages that cannot be associated.
Retransmissions	Number of SIP messages being retransmitted.
ResponseTime	Time between two commands specified in the XML file.
CallLength	Duration of an entire call.

Table E-7: SIPp statistics counters

File Name	Description	Parameter
<scenario>_<pid>_messages.log	Sent and received messages	-trace_msg
<scenario>_<pid>_shortmessages.log	Sent and received messages in CSV format	-trace_shortmsg
<scenario>_<pid>_screen.log	Final statistics screens before SIPp quits	-trace_screen
<scenario>_<pid>_errors.log	Unexpected messages	-trace_err
<scenario>_<pid>.csv	All statistics in CSV format	-trace_stat
<scenario>_<pid>_counts.csv	Individual message counts in CSV format	-trace_counts
<scenario>_<pid>_rtt.csv	Response times in CSV format	-trace_rtt
<scenario>_<pid>_logs.log	Log actions specified in the scenario file	-trace_logs

Table E-8: SIPp output files

Appendix F
OpenSIPS Miscellaneous

Keyword	Description
from_uri	the URI in the From header in the received SIP message. *e.g. if(from_uri~=".*@opensips.org")*
method	the SIP method of the message. *e.g. if(method=="INVITE")*
src_ip	the source IP address of the SIP message. *e.g. if(src_ip==127.0.0.1)*
src_port	the source port number of the SIP message. *e.g. if(src_port==5060)*
uri	the request URI of the SIP message. *e.g. if(uri~="[123]+.*@sipproxy.com")*

Table F-1: OpenSIPS important keywords

Parameter	Description
alias	to set alias hostnames for the server. *e.g. alias=sipserver.com:5060*
children	to set the number of children to fork for the UDP interfaces in the case that fork parameter is enabled. (default value is 8). *e.g. children=16*
debug	to set the debug level. Higher values result in more log messages. *e.g. debug=3*
fork	to enable or disable the daemon mode which allows one process for each network interface and each protocol. When disabled, the openSIPS is bound to the terminal as a single process. *e.g. fork=no*
listen	to set the network address that OpenSIPS should listen to. Multiple values can be assigned to this parameter to make the SIP server listens to more than one address. If it is omitted, the server will listen to all interfaces. *e.g. listen=udp:127.0.0.1:5060*
loadmodule	to load the module specified as the parameter. *e.g. loadmodule "db_mysql.so"*
mpath	to set the path for OpenSIPS to look for modules. *e.g. mpath="/usr/local/lib/opensips/modules"*
modparam	to modify the value of a parameter. *e.g. modparam("usrloc", "db_mode", 2)*
port	to set the port OpenSIPS should listen. (default value is 5060) *e.g. port=5061*

Table F-2: OpenSIPS key parameters

Function	Description
rewritehost()	to substitute the domain part of the request URI with the value specified. Other parts remain unchanged. *e.g. rewritehost("10.0.0.10");*
rewritehostport()	to change the domain part and the port number of the request URI. Other parts such as username remain unchanged. *e.g. rewritehostport("10.0.0.10:5090");*
rewriteport()	to change the port number of the request URI. *e.g. rewriteport("5090");*
rewriteuri()	to change the whole request URI. *e.g. rewritepath("sip:abc@sipserver:5090");*
setflag()	to mark the current SIP message for further processing. The flag parameter can be in the range from 1 to 31. *e.g. setflag(1);*
isflagset()	to test if the current SIP message is marked with the flag. *e.g. isflagset(1)*
strip()	to strip the first n characters from the username in the request URI. *e.g. strip(5);*
ds_select_dst(set, alg)	to choose the destination from the provided set of addresses based on different algorithms. The first parameter indicates the set number. The latter specifies the algorithm used in selecting one instance in the set. This method belongs to the dispatcher module (dispatcher.so). *e.g. ds_select_dst("2", "0");*

Table F-3: OpenSIPS important functions