Design of PSTN-VoIP Gateway with inbuilt PBX & SIP extensions for Wireless medium

Dissertation

submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Priyesh Wadhwa

(Roll no. 05329011)

under the guidance of

Prof. Sridhar Iyer

Department of Computer Science and Engineering

Indian Institute of Technology Bombay

2007

Dissertation Approval Sheet

This is to certify that the dissertation entitled

Design of PSTN-VoIP Gateway with inbuilt PBX & SIP extensions for Wireless medium

by

Priyesh Wadhwa

(Roll no. 05329011)

is approved for the degree of **Master of Technology**.

____

Prof. Sridhar Iyer

(Supervisor)

____

Prof. Anirudha Sahoo

(Internal Examiner)

____

Dr. Vijay T. Raisinghani

(External Examiner)

____

Prof. V. M. Gadre

(Chairperson)

Date: ______________

Place: ______________

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

CERTIFICATE OF COURSE WORK

This is to certify that **Mr. Priyesh Wadhwa** was admitted to the candidacy of the M.Tech. Degree and has successfully completed all the courses required for the M.Tech. Programme. The details of the course work done are given below.

Sr.No.	Course No.	Course Name	Credits
		Semester 1 (Jul – Nov 2005)	
1.	IT601	Mobile Computing	6
2.	HS699	Communication and Presentation Skills (P/NP)	4
3.	IT603	Data Base Management Systems	6
4.	IT619	IT Foundation Laboratory	8
5.	IT623	Foundation course of IT - Part II	6
6.	IT605	Computer Networks	6
		Semester 2 (Jan – Apr 2006)	
7.	HS700	Applied Economics	6
8.	IT630	Principles and Practices of Distributed Computing	6
9.	IT610	Quality of Service in Networks	6
10.	IT694	Seminar	4
11.	IT680	Systems Lab.	6
		Semester 3 (Jul – Nov 2006)	
12.	CS601	Algorithms and Complexity (Audit)	6
13.	IT608	Data Warehousing and Data Mining	6
14.	IT620	New Trends in Information Technology	6
		M.Tech. Project	
15.	IT696	M.Tech. Project Stage - I (Jul 2006)	18
16.	IT697	M.Tech. Project Stage - II (Jan 2007)	30
17.	IT698	M.Tech. Project Stage - III (Jul 2007)	42

I.I.T. Bombay Dy. Registrar(Academic)

Dated:

Acknowledgements

I take this opportunity to express my sincere gratitude for **Prof. Sridhar Iyer** for his constant support and encouragement. His excellent guidance has been instrumental in making this project work a success.

I would like to thank **Prof. Anirudha Sahoo** for his constant help throughout the project. I would also like to thank my colleague **Sravana Kumar** for helpful discussions in the initial part of the project, and for being supportive throughout the project. I would also like to thank the KReSIT department for providing me world class computing infrastructure.

I would also like to thank my **family** and **friends** especially the entire **M.Tech. Batch**, who have been a source of encouragement and inspiration throughout the duration of the project.

Last but not the least, I would like to thank the entire KReSIT family for making my stay at IIT Bombay a memorable one.

Priyesh Wadhwa

I. I. T. Bombay

July 03rd, 2007

Abstract

VoIP gateway enables voice communication between users of the IP network and the Public Switched Telephone Network(PSTN). The system setup requires a PC installed with Asterisk, and a Gateway to integrate with PSTN. The problem with this solution is the high cost, power consumption, and the involved setup of the system.

We have designed a single box solution for the PSTN-VoIP integration system. We studied detailed architecture of the gateway, the protocols used in the VoIP call setup and communication, the software used for the PBX systems, and the internal parts of the SPA3000 gateway. We used the Via motherboard, flash memory, and a normal data modem to create a improved and cost-effective system.

Next, we performed various studies on the Asterisk's response time in wired and wireless medium. We found a remarkable difference in the response time of Asterisk for both the mediums. The reason for the high response time of Asterisk in wireless medium is the slow call setup. SIP being a text-based protocol, is engineered for high data rate links, and so SIP message's size have not been optimized. With low bit rate IP connectivity of signaling channel, the transmission delays for call setup and feature invocation are significant.

We have extended the Session Initiation Protocol for improving its efficiency in wireless medium. We have implemented the compression and decompression mechanisms according to the SigComp standard, and integrated them to the Asterisk server. We have also developed a SigComp enabled client to run with Asterisk. We performed extensive testing of the system and obtained upto *90%* compression using SigComp and Deflate compression algorithm.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations and Notations

Abbreviations

SIP	: Session Initiation Protocol
SDP	: Session Description Protocol
SigComp	: Signaling Compression
FXO	: Foreign Exchange Office
FXS	: Foreign Exchange Station
POTS	: Plain old telephone
DAA	: Direct Access Arrangement
SLIC	: Subscriber Line Interface Circuit
SHA-1	: Secure Hash Algorithm

Chapter 1

Introduction and Motivation

In this work, we have focused on two problems. First, to reduce the cost and the power consumption of the PSTN-VoIP integrated system, and make it more suitable for use in rural environment. The other, is to make SIP protocol more efficient in wireless medium, to reduce the connection establishment time for Asterisk.

There are many villages where we have only single PSTN communication line working, and no appropriate power supply. We have tried to enable a single PSTN line to be used by multiple users, in order to increase the communication range. For this we require a PBX system for routing the calls appropriately. But the cost of switching devices are very high, and also the setup is quite involved. Also, because of the lack of appropriate power we have to make the device consume as much low power as possible. So the problem that we have worked on, is to reduce the setup cost of the system needed for PSTN-VoIP integration, in order to make it affordable for the rural environment.

The Session Initiation Protocol is designed for initiating, and managing multimedia sessions. SIP is a text based protocol engineered for bandwidth rich links. As a result, the messages have not been optimized in terms of size. Typical SIP messages range from a few hundred bytes up to several Kilobytes(upto 1200Bytes). When SIP is used in wireless handsets as part of 2.5G and 3G cellular networks, where the bandwidth and energy represent high cost resources, and the medium has potential high packet loss and collision rates, the large message size and the need to handle high number of messages per transaction becomes problematic.

1.1 Motivation

The emergence of VoIP technology has now made possible the use of data network for data, as well as voice communication. The data network is completely digital network and voice network is a completely analog one, so we need a medium that can encapsulate the analog signals to digital format, we call it a gateway. A gateway is used for converting the analog signals from Public Switched Telephone Network(PSTN) to digital signals in Packet Network(VoIP) and vise versa.

In a single user environment we don't need any switching equipment, but if multiple users are going to use the communication channel we need a PBX server to figure out which user is being called, the authorization of the user, and several other features specific to a user.

In the present technology, we use a gateway at the customer premises, and a PBX server is provided by the ISP. But in rural areas, where we have low power supply and less communication facilities available, we need to setup a single box solution for a facility like VoIP communication, as we can't afford to setup a PBX server. We need to build a single box solution that provides the functionality of both the gateway, and the PBX server. This solution reduces the cost of the overall setup.

For the second problem, viz making SIP more efficient in wireless medium, the motivation is the results we obtained in our first stage, while experimenting with Asterisk's response time in wired and wireless medium. From the graph 1.1, as we go on increasing the number of parallel calls in wireless medium the packet loss and error rate increases enormously. So there is a need to device mechanisms that make the SIP messages' transmission more robust in the wireless medium.

1.2 Problem Statement

The problem is to design a single box solution that integrates the functionality of the Asterisk PBX as well as the gateway. We should reduce the cost, power consumption and the intricacies of the system setup.

The other problem is to make Session Initiation Protocol more efficient in wireless medium, and to implement these extended features in Asterisk server. We have implemented Signaling compression (SigComp) mechanism in Asterisk. We also propose a

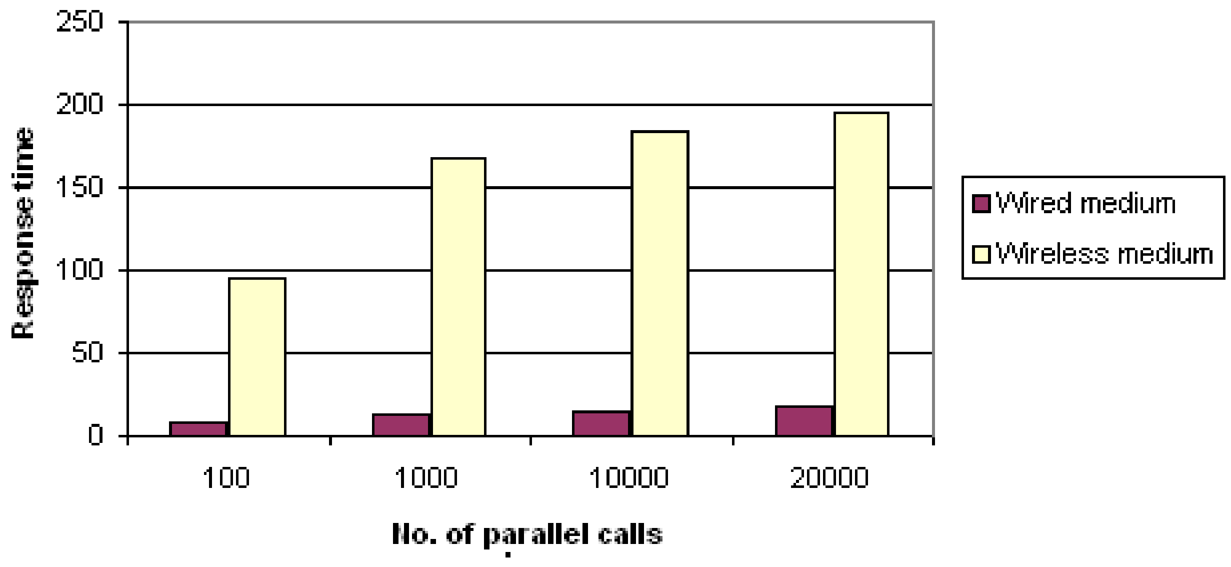

Figure 1.1: Response time of Asterisk in wireless and wired medium

solution to minimize the message size between SIP client and Edge-proxy by using data storage at the edge proxy. The data that we store at the edge proxy is usually transmitted again and again by the client. If we store that data on edge proxy the message size for all the communication between the client and proxy will be minimized.

1.3 Thesis outline

In chapter 2, we have presented the related work and the literature survey we have done. In this chapter, we have described the open source Asterisk PBX architecture, SPA3000 internals, the protocols we have used, and the related technologies. In the next chapter, we have described our solution for PSTN-VoIP integration, and the hardware devices we have worked with. Chapter 4 describes the SigComp standard from an implementation perspective. In chapter 5, we have described the implementation details of the SigComp for Asterisk and Yate. The next chapter concludes the thesis and describes the future work to continue the project.

Chapter 2

Literature Survey

In this chapter, we have discussed about the related work and the literature study done for the project. Here, we have presented the detailed study of the softwares involved, protocols we dealt with, hardware devices we studied and used, and different other related methods already implemented to handle similar problems.

2.1 Open PBX Asterisk

Asterisk [1, 2, 3] is an open source software PBX, created by Digium. Asterisk runs on Linux and other Unix platforms with or without hardware that connects the PBX server to the traditional global telephony network, the PSTN. Asterisk gives us real-time connectivity on both PSTN and VoIP networks.

Asterisk is much more than a standard PBX. With Asterisk as the telephony switching platform, we'll not only have a high-class PBX replacement, but also we can do telephony in new ways.

2.1.1 Asterisk Architecture

Asterisk consists of five base components:

- Dynamic Module Loader - When Asterisk is first started, the Dynamic Module Loader loads and initializes each of the drivers which provide channel drivers, file formats, call detail record backends, codecs, applications and more, linking them with the appropriate internal APIs.

- PBX Switching - The essence of Asterisk is a Private Branch Exchange Switching system, connecting calls together between various users and automated tasks. The

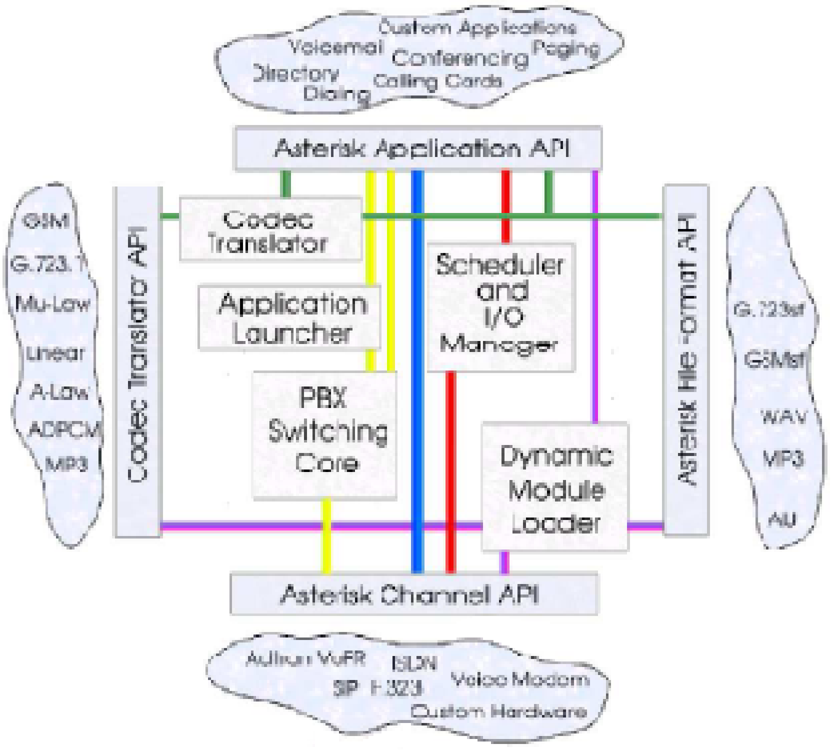

Figure 2.1: Asterisk Architecture

Switching Core transparently connects callers arriving on various hardware and software interfaces.

- Application Launcher - launches applications which perform services for users, such as voicemail, file playback, and directory listing.
- Codec Translator - uses codec modules for the encoding and decoding of various audio compression formats used in the telephony industry. A number of codecs are available to suit diverse needs and arrive at the best balance between audio quality and bandwidth usage.
- Scheduler and I/O Manager - handles low-level task scheduling and system management for optimal performance under all load conditions.

2.1.2 Loadable Module APIs

Four APIs are defined for loadable modules, facilitating hardware and protocol abstraction. Using this loadable module system, the Asterisk core becomes independent of the details like how a caller is connecting, what codecs are in use, etc.

- Channel API - the channel API handles the type of connection a caller is arriving on, be it a VoIP connection, ISDN, PRI, or some other technology. Dynamic modules are loaded to handle the lower layer details of these connections.
- Application API - the application API allows for various task modules to be run to perform various functions. Conferencing, paging, directory listing, voicemail, in-line data transmission, and any other task which a PBX system might perform now or in the future are handled by these separate modules.
- Codec Translator API - loads codec modules to support various audio encoding and decoding formats such as GSM, Mu-Law, A-law, and even mp3.
- File Format API - handles the reading and writing of various file formats for the storage of data in the file system.

2.2 YATE - Yet Another Telephone Engine

Yate is an open source soft phone which can be used as VoIP client, VoIP to PSTN gateway, PC2Phone and Phone2PC gateway, SIP router, SIP registration server, IAX server and/or client etc. We have used Yate as a VoIP client to make calls through Asterisk server. Yate provides many modules like 'callgen', and 'message sniffer' for measuring the performance of the PBX server. We have used callgen to generate parallel SIP calls to the server, to get the response time of the Asterisk server under heavy loads.

2.3 Sipura device internals

The LinkSys SPA 3000 is a residential gateway; we have used it to understand the implementation details of a gateway. The main ICs that are present in SPA 3000 are:

- Visba 3 Video CD Processor.
- SST39VF080 (Flash memory).
- RTL8019AS (Realtek Full-Duplex Ethernet Controller with Plug and Play Function).

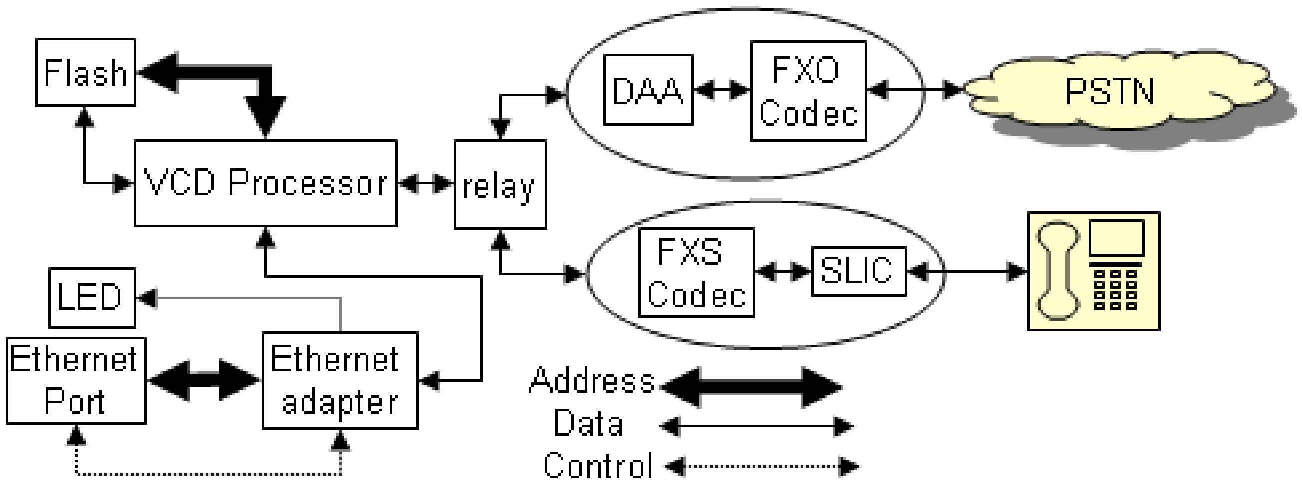

Figure 2.2: SPA3000 Block diagram

- Si3210 (ProSLIC).
- Si3050 (Direct Access Arrangement (DAA)).

2.3.1 Visba 3 Video CD Processor

The Visba3 ES3890 is a single chip Video CD Processor. The ES3890 integrates an audio ADC for microphone inputs, two video DACs for Composite and S-Video outputs, and digital echo circuitry. The ES3890 performs all of the functions such as video filtering, NTSC/PAL conversion, audio and video error concealment.

The Visba3 VCD processor in SPA3000 works as the processor of the system. It controls the web based monitoring/settings system of the device. It provides the web interface to the users for configuring the device. The processor interacts with the flash memory in order to store/retrieve the configuration information of the gateway.

2.3.2 RTL8019AS (Realtek Full-Duplex Ethernet Controller with Plug and Play Function)

The RTL8019AS is a highly integrated Ethernet controller which offers a simple solution to implement a plug and play adapter with full-duplex and power down features. The full-duplex function enables simultaneous transmission and reception on the twisted-pair link to a full-duplex Ethernet switching hub. This feature not only increases the channel bandwidth but also avoids the performance degrading problem due to the channel contention characteristics of the Ethernet CSMA/CD protocol.

2.3.3 Si3210 (ProSLIC)

The Si3210 ProSLIC provides a complete analog telephone interface, ideal for customer premise equipment (CPE) applications. The Si3210 integrates subscriber line interface circuit (SLIC), codec and battery generation functionality into a single low-voltage CMOS integrated circuit. The integrated battery supply continuously adapts its output voltage to minimize power and enables the entire solution to be powered from a single 3.3V or 5V supply.

2.3.4 Si3050 (Direct Access Arrangement (DAA))

The Si3050 connects to the PSTN line and emulates a POTS phone. Its main function is to remove the high voltage DC bias from the signals coming from the PSTN system, and pass only the analog AC signal.

2.4 Protocols

2.4.1 SIP

SIP (Session Initiation Protocol)[4] is an application-layer control protocol that can establish, modify, and terminate multimedia sessions such as Internet telephony calls (VOIP). SIP can also invite participants to already existing sessions, such as multicast conferences. Media can be added to (and removed from) an existing session. SIP transparently supports name mapping and redirection services, which supports personal mobility - users can maintain a single externally visible identifier regardless of their network location.

SIP supports five facets of establishing and terminating multimedia communications:

- **User location**: determination of the end system to be used for communication
- **User availability**: determination of the willingness of the called party to engage in communications
- **User capabilities**: determination of the media and media parameters to be used

- **Session setup**: “ringing”, establishment of session parameters at both called and calling party
- **Session management**: including transfer and termination of sessions, modifying session parameters, and invoking services

SIP is a component that can be used with other IETF protocols to build a complete multimedia architecture, such as the Real-time Transport Protocol (RTP) for transporting real-time data and providing QoS feedback, the Real-Time streaming protocol (RTSP) for controlling delivery of streaming media, the Media Gateway Control Protocol (MEGACO) for controlling gateways to the Public Switched Telephone Network (PSTN), and the Session Description Protocol (SDP) for describing multimedia sessions. Therefore, SIP should be used in conjunction with other protocols in order to provide complete services to the users. However, the basic functionality and operation of SIP does not depend on any of these protocols.

2.4.2 SDP

The Session Description Protocol (SDP)[5] describes multimedia sessions for the purpose of session announcement, session invitation and other forms of multimedia session initiation.

Session directories assist the advertisement of conference sessions and communicate the relevant conference setup information to prospective participants. SDP is designed to convey such information to recipients. SDP is purely a format for session description - it does not incorporate a transport protocol, and is intended to use different transport protocols as appropriate including the Session Announcement Protocol (SAP), Session Initiation Protocol (SIP), Real-Time Streaming Protocol (RTSP), electronic mail using the MIME extensions, and the Hypertext Transport Protocol (HTTP) .

SDP is intended to be general purpose so that it can be used for a wider range of network environments and applications than just multicast session directories. However, it is not intended to support negotiation of session content or media encodings. SDP communicates the existence of a session and conveys sufficient information to enable participation in the session.

Many of the SDP messages are sent by periodically multicasting an announcement packet to a well-known multicast address and port using SAP (session announcement protocol). These messages are UDP packets with a SAP header and a text payload. The text payload is the SDP session description.

The SDP text messages include:

- Session name and purpose
- Time for which the session is active
- Media comprising the session

2.4.3 SigComp

SigComp is the method described to compress the SIP and RTP messages for efficient use of the low bandwidth channels. The method works by two basic principles:

- Store the state of the previous messages and use it for further compression.
- Use of UDVM for decompression which can run with any compression algorithm. It makes the SigComp compression algorithm independent.

As our main focus is the implementation of SigComp for Asterisk, we have described SigComp standard from an implementation perspective in Chapter 4.

2.4.4 RTP

RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. These services include payload type identification, sequence numbering, timestamping and delivery monitoring. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services. The data transport is augmented by a control protocol (RTCP) to allow monitoring of the data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality. RTP and RTCP are designed to be independent of the underlying transport and network layers. The protocol supports the use of RTP-level translators and mixers.

2.5 Compression Algorithms

2.5.1 LZ77 (Lempel-Ziv 1977)

Principle

The algorithm searches the window for the longest match with the beginning of the lookahead buffer and outputs a pointer to that match. Since it is possible that, not even a one-character match can be found, the output cannot contain just pointers. LZ77 solves this problem this way: after each pointer it outputs the first character in the lookahead buffer after the match. If there is no match, it outputs a null-pointer and the character at the coding position.

The encoding algorithm

1. Set the coding position to the beginning of the input stream.
2. Find the longest match in the window for the lookahead buffer.
3. Output the pair (P,C) with the following meaning:
    - P is the pointer to the match in the window;
    - C is the first character in the lookahead buffer that didn't match;
4. If the lookahead buffer is not empty, move the coding position and the window L+1 characters forward and return to step 2.

Decoding

The window is maintained the same way as while encoding. In each step the algorithm reads a pair (P, C) from the input. It outputs the sequence from the window specified by P and the character C.

2.5.2 DEFLATE

Deflate is a lossless data compression algorithm that uses a combination of the LZ77 algorithm and Huffman coding. The deflate algorithm finds duplicate strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length). Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes. Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. There are three modes of compression that the compressor has available:

1. No compression at all: This is used, when data is already compressed. Data stored in this mode will expand slightly, but not by as much as it would if it were already compressed and one of the other compression methods was tried upon it.

2. Compression: first with LZ77 and then with a slightly modified version of Huffman coding. The trees that are used to compress in this mode are defined by the Deflate specification itself, and so no extra space needs to be taken to store those trees.

3. Compression: first with LZ77 and then with a slightly modified version of Huffman coding with trees that the compressor creates and stores along with the data. The data is broken up in "blocks", and each block uses a single mode of compression. If the compressor wants to switch from non-compressed storage to compression with the trees defined by the specification, or to compression with specified Huffman trees, or to compression with a different pair of Huffman trees, the current block must be ended and a new one is started.

2.5.3 LZW

It is a lossless 'dictionary based' compression algorithm. Dictionary based algorithms scan a file for sequences of data that occur more than once. These sequences are then stored in a dictionary and within the compressed file, references are put where-ever repetitive data occurred.

LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, it just adds every new string of characters it sees to a table of strings. Compression occurs when a single code is output instead of a string of characters.

The code that the LZW algorithm outputs can be of any arbitrary length, but it must have more bits in it than a single character. The first 256 codes (when using eight bit characters) are by default assigned to the standard character set. The remaining codes are assigned to strings as the algorithm proceeds.

2.6 Related work

2.6.1 Data storage on Edge Proxy

We have proposed a solution to reduce the size of SIP messages exchanged between a mobile client and the edge proxy. The basic concept is to utilize the feature of repetition of the same content transmission in SIP message. If we make the edge proxy stateful and also store the call profile information then, we can reconstruct the message on the edge proxy. In this case, the client needs to send only the minimal message content that make it reach the edge proxy and there the message is reconstructed with the help of the stored state. Now, the client doesn't need to send full SIP message for the communication. In

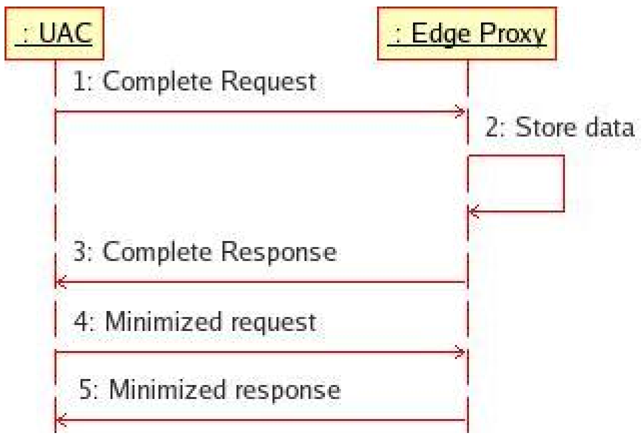

Figure 2.3: Data storage on edge proxy

this approach the first message is sent completely. Using this message the edge proxy creates the state for the client and stores the data used for communication. Further messages from client and edge proxy contain only minimum information. This way the data size in the SIP message is reduced. The advantage of this approach is that, it is transparent to the other SIP mechanisms, i.e, it is fully compatible with the existing SIP protocol and its extensions.

When combined with the compression mechanism it further improves the utilization of the wireless channel. The compression mechanism provides the compress up to ratio

1:8. When combined with data storage on edge proxy we expect further improvement in message compression.

Illustration: We have identified some of the parameters that can be stored on the edge proxy. For example, from the SIP message we can store parameters like Organization, Subject, Accept-Encoding, Accept, Accept-Language, Date, and Content-Type. From the SDP message we can store parameters like v(protocol version), o(owner/creator and session identifier), c(connection information), m(media name and transport address), a(media attribute) etc.

2.6.2 ROHC

Robust Header Compression (ROHC) is a standardized method to compress the IP, UDP, RTP, and TCP headers of Internet packets. It performs well over links where the packet loss rate is high, such as wireless links. In streaming applications, the overhead of IP, UDP, and RTP is 40 bytes for IPv4, or 60 bytes for IPv6. For VoIP this corresponds to around 60% of the total amount of data sent. Such large overheads may be tolerable in wired links where capacity is often not an issue, but are excessive for wireless systems where bandwidth is scarce.

ROHC compresses these 40 bytes or 60 bytes of overhead typically into only 1 or 3 bytes by placing a compressor before the link that has limited capacity, and a decompressor after that link. The compressor converts the large overhead to only a few bytes, while the decompressor does the opposite.

2.6.3 IP Header Compression [RFC 2507]

IP header compression is the process of compressing excess protocol headers before transmitting them on a link and uncompressing them to their original state on reception at the other end of the link. It is possible to compress the protocol headers due to the redundancy in header fields of the same packet as well as consecutive packets of the same packet stream. This is done by saving the state of TCP connections at both ends of a link, and only sending the differences in the header fields that change. This makes a very big difference for interactive performance.

2.6.4 Traffic payload Compression provided by Transport or Framing protocols

The data in the payload is compressed before transmission. HTTP provides such functionality of compression. Some well known industry algorithms from Cisco are 'Stacker' and 'Predictor'. Stacker uses an encoded dictionary of symbols and tokens to replace redundant strings of characters in the data stream. Predictor tries to predict the next sequence of characters in a data stream with an index to look up in a compression dictionary. If a match is found, Predictor replaces the matched sequence with the sequence that was looked up in the dictionary. Predictor is memory intensive but less CPU intensive. On the other hand, Stacker uses less memory. Predictor is generally considered more efficient than Stacker because of lower CPU requirements.

Chapter 3

Gateway with inbuilt Asterisk

3.1 Hardware components

We have designed a single box solution for the VoIP-PSTN integrated system. We performed various experiments with different telephony devices. In this chapter, we have described the hardware components used for the experiments and the final solution we designed.

3.1.1 Via motherboard

The VIA PC1500 provides enhanced performance and built-in security features to provide an energy-efficient, feature-rich solution. Via PC1500 is fully compatible with Windows and Linux operating systems. The VIA PC1500 Platform is the most cost-effective processor platform available. It delivers all the necessary performance for running applications while maintaining low levels of power consumption and effective heat dissipation.

Figure 3.1: Via PC1500

3.1.2 IDE Flash Memory

Flash memory is many times more reliable than hard drives due to the lack of moving parts. The IDE flash memory plugs directly into a standard 40-pin IDE port on the mainboard to replace a hard drive. However, flash memory has a limited number of write cycles, so extra care has to be taken when

Figure 3.2: IDE Flash

running softwares from compact flash cards.

3.1.3 Telephony Devices

3.1.3.1 Sipura SPA3000

The SPA-3000 is a PSTN-VoIP gateway designed to provide VoIP (Voice over IP) capabilities by interfacing with a normal analog telephone and a standard PSTN line.

Figure 3.3: Sipura SPA3000 Gateway

Sipura has both FXS and FXO interfaces. The FXS interface allows a normal telephone to be turned into an IP phone, and the FXO interface provides connectivity to the PSTN line. These interfaces can be configured independently using the Sipura's on-board web interface. It has several parameters which help us in fine tuning the device for specific environment.

Once setup and working, apart from allowing us to make VoIP calls, the SPA-3000 provides the following functions:

- *A PSTN to VoIP gateway* - this allows us to make a call using our PSTN phone line to a VoIP user.

- *A VoIP to PSTN gateway* - a VoIP phone user can make a call over the PSTN phone line.

- *Power Cut Protection* - if the SPA-3000 looses either the power or it's network connection, it can be configured to directly connect the two interfaces together - so the phone will be effectively directly connected to the phone line, so we can still make calls over the phone line during a power cut as if the device was not connected. If the power comes back on while a call is in progress, normal operation will not be resumed until the existing call has ended.

- *Complex dial plans can be constructed* - we can create dial plans suitable for the environment (e.g, enterprise, office, home etc).

- *Asterisk* - we can use the SPA-3000 as an FXS and FXO interface to the Asterisk open source PBX system to get more flexibility and features.

3.1.3.2 Linksys PAP2 ATA

The Linksys Phone Adapter enables high-quality, feature-rich telephone service through an Internet connection. With an appropriate Internet telephone service provider, we can get clear telephone reception, even while using the Internet at the same time for normal data operations. The Linksys Phone Adapter also provides us with the other special telephone features that are available from the telephone service provider, such as caller-id, call waiting, voicemail, call forwarding etc.

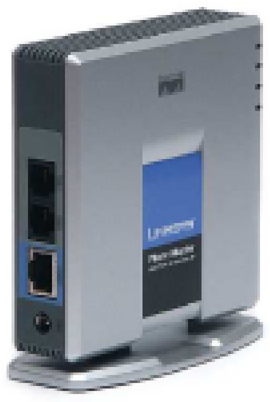
Figure 3.4: Linksys PAP2 Analog Telephone Adapter

3.1.3.3 Digium card

The X100P is the standard single port FXO Interface for Asterisk. It provides a single, full featured FXO port for connecting the open source Asterisk PBX server to PSTN.

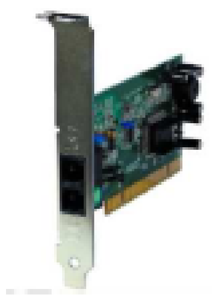
Figure 3.5: X100P PCI card from Digium

Digium X100P allows Asterisk to make calls to or receive calls from a traditional analog phone line. The X100P is an affordable and ideal component for building Interactive Voice Response (IVR) and voicemail applications. It also supports all standard enhanced call features including caller-id, call conferencing, and call waiting.

By combining the X100P and the open source Asterisk PBX, we can easily and economically implement sophisticated and very flexible call services. Such services range from multi-menued IVR, multi-protocol VoIP gateways, directory services to business class voicemail.

3.2 Different solutions for PSTN-VoIP integration

We have performed various experiments with different hardware devices that are used in the integration of PSTN & VoIP networks. In this section, we have described the setup of the experiments and presented a comparison of their cost, advantages, and disadvantages.

3.2.1 Server side setup

- **Experiment 1: Sipura SPA3000 with Normal PC**

  We performed the first experiment with a normal PC, and Sipura SPA3000. The Asterisk server is installed and configured on the computer system. SPA3000 as defined in 3.1.3.1 is the gateway that enables PSTN-VoIP integration. The SPA3000 needs to be configured to work along with Asterisk on the network.

  Advantages:

  – This setup is easy to install.

  – Sipura provides a nice web interface for its configuration.

  – SPA3000 provides us the facility for fine tuning the system(like callerId, call blocking, dial planning etc).

  Disadvantages:

  – This setup is the most expensive in terms of cost and power consumption.

  – Asterisk server is installed on a computer system, causing wastage of computing resources.

- **Experiment 2: Sipura SPA3000 with Via motherboard**

  In the previous solution, we were using a costly and more faster processor and thus also were wasting computational resources, so we replaced the processing unit with a inexpensive motherboard. We used the Via motherboard along with SPA3000 to build the system.

  Advantages:

  – In this setup, we have made efficient usage of computational resources.

  – The cost and power consumption of the system is reduced by using Via motherboard.

Disadvantages:

  – The cost of SPA3000 is still high, compared to the Digium card.
  – The power consumption of the setup is still high because of the use of hard disk.

- **Experiment 3: Digium X100P with Via motherboard**

  Next, we focused on reducing the cost of the gateway. We replaced the SPA3000 with the Digium X100P PCI card (see section 3.1.3.3). The Digium card provides the functionality of the gateway, however we can not fine tune it like the SPA3000.

  Advantages:

  – This setup requires no extra effort to configure the gateway. Asterisk provides us the Zaptel drivers to communicate with Digium card. We just need to configure the zaptel.conf file to make the communication possible.

  Disadvantages:

  – The X100P card provides only the functionality of FXO and FXS ports. No fine tuning of the system is possible unlike SPA3000.
  – The cost of X100P is high, compared to data modem.

- **Experiment 4: Normal Data modem with Via motherboard**

  In order to reduce the cost further, we used the normal data modem in place of the Digium card. This requires some code modification in the Asterisk's Zaptel driver's code. The normal data modem provides us the FXO and FXS ports just like X100P after the code modification in Asterisk. We have described the code modifications done in the drivers in section 3.3.2.

  Advantages:

  – Cost of the system is reduced by the use of data modem.

  Disadvantages:

  - Code modification in Asterisk is required to make Asterisk work with the modem.
  - Power consumption of the system is still high, because of the use of hard disk.

- **Experiment 5: Flash memory with Via motherboard**

  Next, we tried to reduce the power consumption of the system. We replaced the hard-disk of the system with a 40-pin flash IDE. Flash IDE is just like a hard disk that is connected to the motherboard on its 40-pin slot used to connect hard-disk data bus. We used AstLinux as our platform for the system. We have described this solution in more detail in section 3.3.

  Advantages:

  - This setup makes efficient utilization of resource.
  - The setup is low power consuming and less costly.

  Disadvantages:

  - The life time of the system is reduced because of the use of flash memory.
  - Data retrieval/storage is slow in flash memory.
  - We need to make code modifications in Linux and Asterisk to stop the logging.

3.2.1.1 Cost analysis of the experiments

Table 3.1 shows, the cost analysis of all the experiments we have done. As on December 2006, the approximate cost of the devices were like, SPA300(Rs 7,000), POTS(Rs 500), X100P(Rs 2,500), data modem(Rs 500), Via motherboard(5,000), and Flash IDE(Rs 1,000).

3.2.2 Client side setup

On the client side, we experimented with different devices like simputer, laptop with softphone, and ATA adapter with POTS phone. Each setup has its advantages and disadvantages. The major differentiating factors are the cost and the availability of the technology. POTS with adapter is the most common solution we have tried on. It is the most inexpensive solution, but can be used only for communication purpose. On the

Table 3.1: Server setup cost

S.No.	System setup	Setup Cost(Rs.)
1.	Sipura SPA3000 with Normal PC	8,030
2.	Sipura Spa3000 with VIA motherboard	7,030
3.	Digium X100P with Normal PC	7,530
4.	Digium X100P with VIA motherboard	6,530
5.	Normal Data Modem with VIA motherboard	5,360

other hand, if we use a softphone on a laptop we don't have to make any new investments. Simputer can also be used to make and receive calls. Simputer has most of the functionality which is similar to a softphone installed on PC.

Table 3.2: Cost comparison of client side devices

S.No	Client side solutions	Cost
1.	ATA + POTS	Rs. 4,000
2.	Simputer	Rs. 15,000
3.	Laptop with Softphone	Rs. 30,000
4.	Desktop PC with Softphone	Rs. 20,000

3.3 Final Solution

3.3.1 Conventional setup

In the conventional setup for getting the VoIP functionality with PSTN, we need a separate computer system with Asterisk server configured on it, and gateway for the VoIP-PSTN communication. In this setup, the cost and power consumption of the system is very high. Gateways are usually expensive and the use of a computer system for asterisk make the system unsuitable for use in rural areas where we don't have electricity supply, rather we have some solar cells to work with. In this setup we also get lot of other functionalities which are of no use for the end user.

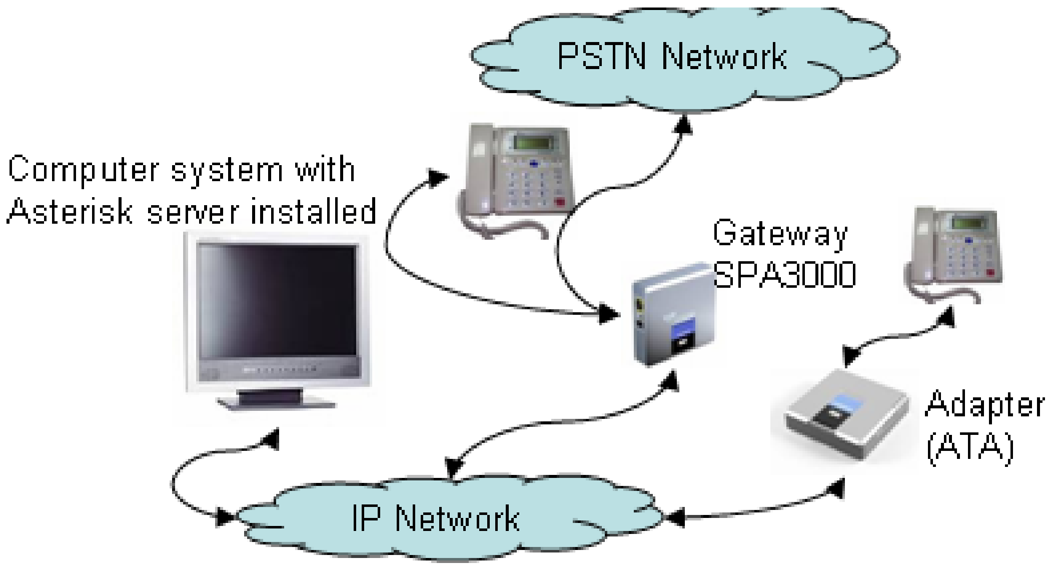

Figure 3.6: Conventional setup of asterisk system

3.3.2 Single box solution

We have designed a single box solution for the PSTN-VoIP integrated system. The hardware components we used are Via motherboard, data modem, and flash memory. We used AstLinux as the PBX system. AstLinux is a ‘CentOS Linux plus Asterisk’ combined package that can be installed directly on any system. AstLinux has the minimum required features of Linux that are needed for Asterisk to run properly. We installed AstLinux on the flash drive, so as to avoid the use of the hard-disk. The goal is to remove the use of hard-disk and avoid the power consumption by it, and so the SMPS can be removed.

The main problem was to stop the logging functionality of Linux and Asterisk, which they perform for error handling. For that, we compiled the Linux and Asterisk with logging disabled and then created a single package for installation by the user. Now after the bootup, Linux and Asterisk both are loaded in main memory and there is no need for any external storage.

After installing the modified AstLinux on flash memory, we used the Via motherboard for the processing needs of the system. Via motherboard as described in section 3.1.1 is a very inexpensive processor, with low power consumption.

For the gateway we first used the Digium card, then later we replaced the Digium card with a normal data modem. Data modem is quite inexpensive compared to the Digium PCI card or any other external gateway (like SPA3000). Asterisk is not designed to work with the data-modems, so we modified the ZAP channel files in order for the system to work with data-modem.

```
We modified the zaptel/wcfxo.c file as follows:
Existing code:
static struct pci_device_id wcfxo_pci_tbl[] __devinitdata = {
{ 0xe159, 0x0001, 0x8085, PCI_ANY_ID, 0, 0, (unsigned long) &wcx101p },
{ 0x1057, 0x5608, PCI_ANY_ID, PCI_ANY_ID, 0, 0, (unsigned long) &wcx100\$ };
Now change the wcfxo_pci_tbl[] in zaptel/wsfxo.c to:
static struct pci_device_id wcfxo_pci_tbl[] __devinitdata = {
{ 0xe159, 0x0001, 0x8085, PCI_ANY_ID, 0, 0, (unsigned long) &wcx101p },
{ 0xe159, 0x0001, 0x8086, PCI_ANY_ID, 0, 0, (unsigned long) &wcx101p },
{ 0x1057, 0x5608, PCI_ANY_ID, PCI_ANY_ID, 0, 0, (unsigned long) &wcx100\$ };
```

3.3.3 Improvements from conventional setup

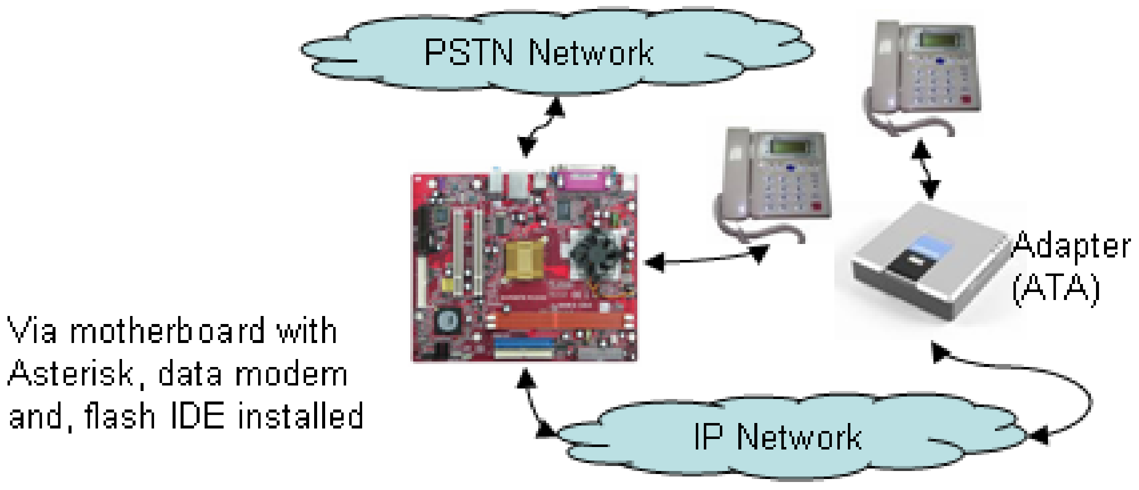
PSTN Network
Adapter (ATA)
Via motherboard with Asterisk, data modem and, flash IDE installed
IP Network

Figure 3.7: Improved setup of asterisk system

The resulting system was a small single box device. As compared to the previous solution in which we have to use a separate gateway and computer system, it is very easy to install and already configured for use. The cost of the system was greatly reduced as shown in table 3.1. The power consumption of the system was also reduced. As shown in figure 3.7, the motherboard contains the data modem which is connected to the PSTN network and POTS. The Ethernet port on the motherboard connects to the Internet.

Chapter 4

SigComp

4.1 Signaling Compression (SigComp)

In this chapter, we have described the SigComp (Signaling Compression)[6] mechanism from an implementation perspective. SigComp defines the mechanism to compress and decompress the SIP messages in end-to-end VoIP applications. Using SigComp we have obtained a compression ratio between 1:5 and 1:8. The important thing about SigComp is that it is totally independent of compression algorithm used.

4.2 SigComp Architecture

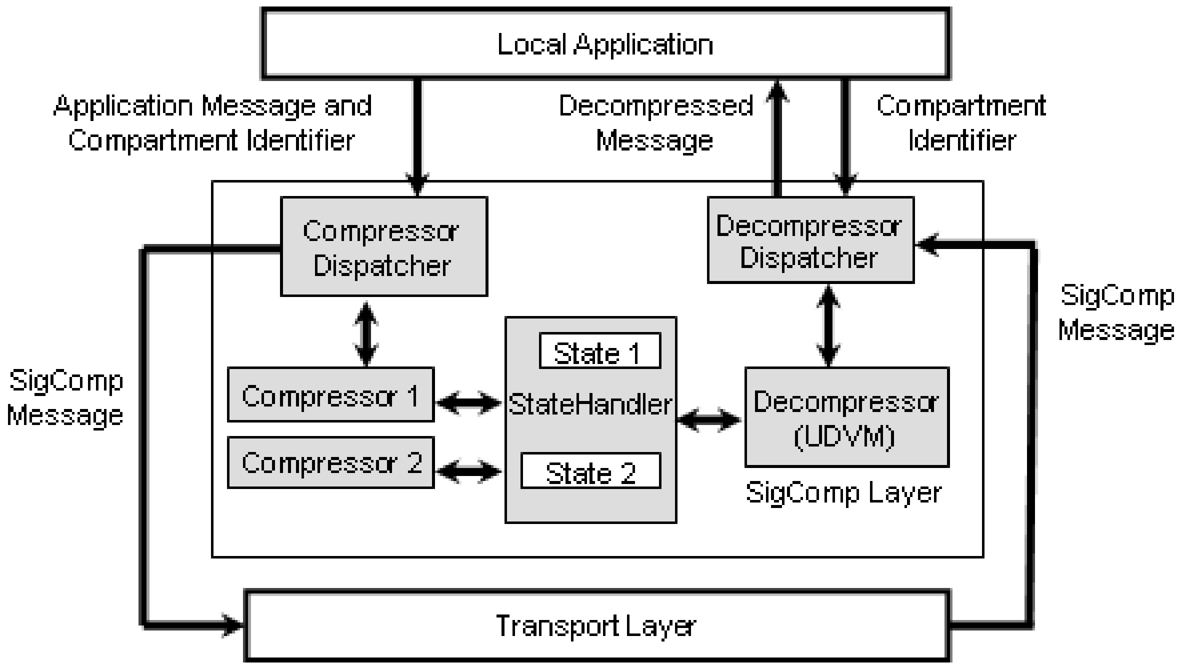

Figure 4.1: SigComp Architecture

The major components of the SigComp are:

- Compressor dispatcher: SigComp invokes compressors on a per-compartment basis, so when the application provides a message to be compressed it also provides a compartment identifier. The compressor dispatcher forwards the application message to the correct compressor based on the compartment identifier. The compressor returns a SigComp message that can be passed to the transport layer.
- Decompressor dispatcher: The decompressor dispatcher receives a SigComp message and invokes an instance of the Universal Decompressor Virtual Machine (UDVM). It then forwards the resulting decompressed message to the application, which may return a compartment identifier, if it wishes to allow state to be saved for the message.
- Compressor: The Compressor is the component that compresses the messages and uploads the ByteCode for the corresponding decompression algorithm in the UDVM as part of the SigComp message.
- Decompressor (UDVM): UDVM provides a mechanism to uncompress messages by interpreting the corresponding ByteCode. The UDVM can be used to decompress the output of various compressors such as DEFLATE.
- State Handler: The State Handler retains information between received SigComp messages, and thus eliminates the need to send decompression instructions with each of the compressed message.

4.3 SigComp Compressor

An important feature of SigComp is that decompression functionality is provided by a Universal Decompressor Virtual Machine (UDVM). This means that the compressor can choose any algorithm to generate compressed SigComp messages, and then upload bytecode for the corresponding decompression algorithm to the UDVM as part of the SigComp message.

For robustness, we have to use CRC of the message, and for security we have to perform SHA1 hash of the message.

The compressor is also responsible for forwarding the requested feedback items returned by the state handler, and also for uploading the local SigComp parameters to the remote endpoint.

4.4 SigComp State Handler

The function of the state handler is to retain information between received SigComp messages. To provide security against the malicious insertion or modification of SigComp messages, a separate instance of the UDVM is invoked to decompress each message. This ensures that damaged SigComp messages do not prevent the successful decompression of subsequent valid messages.

The UDVM can only create a state item when a complete message has been successfully decompressed and the application has returned a compartment identifier under which the state can be saved.

SigComp protects state access by creating a `state_identifier` that is a hash over the item of state to be retrieved. This `state_identifier` must be supplied to retrieve an item of state from the state handler.

4.5 SigComp Message Format

A SigComp message takes one of two forms depending on whether it accesses a state item at the receiving endpoint, or it is the first message that is uploading the bytecode. The `T-bit` controls the format of the returned feedback item.

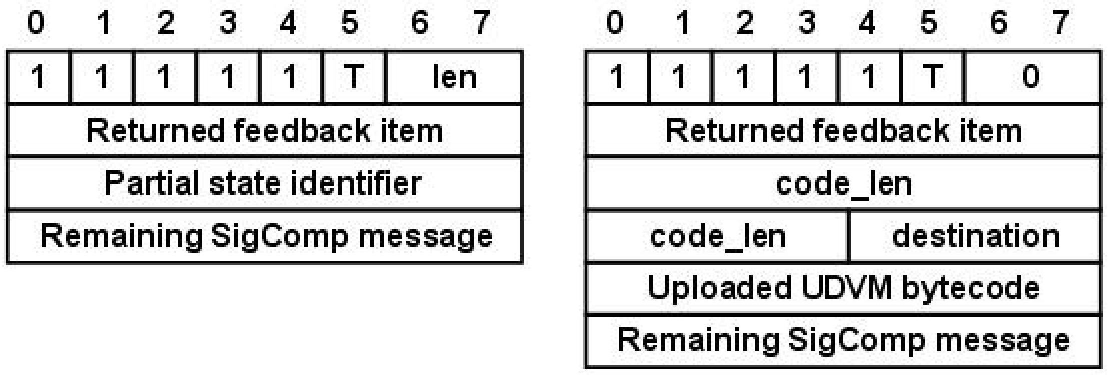

Figure 4.2: SigComp message format

For both variants of the SigComp message, the T-bit is set to 1 whenever the SigComp message contains a returned_feedback_item. The returned_feedback_length specifies the size of the returned feedback field.

The len field of the SigComp message determines which fields follow the returned feedback item. If the len field is non-zero, then the SigComp message contains a state identifier to access a state item at the receiving endpoint. The partial state identifier is passed to the state handler, which compares it with the most significant bytes of the state_identifier in every currently stored state item. If a state item is successfully accessed then the state_value byte string is copied into the UDVM memory beginning at state_address. The 12-bit code_len field specifies the size of the uploaded UDVM bytecode.

4.6 UDVM: Universal Decompressor Virtual Machine

4.6.1 UDVM Architecture

A new UDVM is instantiated and initialized for each received incoming SigComp message. The memory layout of the UDVM is shown in the figure 4.3.

Address 0-63 is initialized after receiving the message. Addresses 0 to 5 indicate the resources available to the receiving endpoint. The UDVM memory size is expressed in bytes modulo 2^{16}, so in particular, it is set to 0 if the UDVM memory size is 65536 bytes.

The cycles_per_bit is expressed as a 2-byte integer taking the value 16, 32, 64 or 128. The UDVM then begins executing instructions at the memory address contained in state_instruction, which is part of the retrieved item of state.

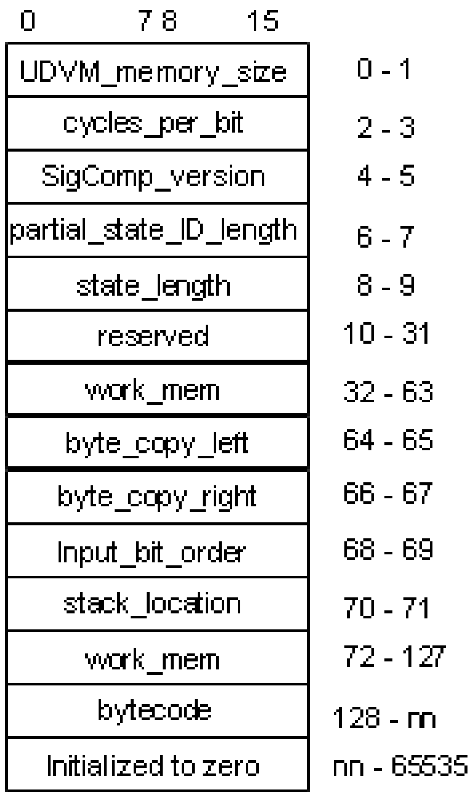

Figure 4.3: UDVM architecture

`SigComp version` is either 0 or 1. Addresses 6 to 9 are initialized to the length of the p`partial state identifier`, followed by the `state_length` from the retrieved state item. Addresses 10 to 31 are reserved and are initialized to 0 for Version 0x01 of SigComp.

The addresses 32-63 are used as working space by the UDVM for example storing stack. The next 8 bytes (64-71) are used as four registers by the UDVM. To provide bit-wise compatibility with various well-known compression algorithms, the `input_bit_order` register can modify the order in which individual bits are passed within a byte. The `P` flag in the `input_bit_order` determines the way of interpreting the bits received as input. If set to 0, it indicates that the bits within an individual byte are passed to the INPUT instructions in MSB to LSB order. The `stack_location` register stores the base address of the stack. The PUSH, POP, CALL and RETURN instructions use the stack.

4.6.2 UDVM Instruction Set

There are in all 35 different instructions defined in the standard for the UDVM. The instructions are highly optimized for the general purpose compression algorithms. Because of this major compression algorithms can be expressed in less than 100 bytes in UDVM byte-code. The instructions can be classified into categories of arithmetic, bitwise, memory management, program flow, I/O instructions.

Chapter 5

Implementation of SigComp for Asterisk & Yate

5.1 Implementation Description

We have implemented the SigComp[6] standard with Deflate compression algorithm for Asterisk server. The aim is to improve the Asterisk's response time for the signaling messages. The available implementation of Asterisk doesn't provide the SigComp compression algorithms. We have also integrated SigComp to the Yate, a SIP client, to work with Asterisk server. In this chapter, we have presented a detailed description of the implementation intricacies of the standard and its integration with Asterisk and Yate. We have shown the main data structures, the flow of control, the integration points in both Asterisk and Yate.

5.1.1 Data Structures used in SigComp implementation

The main data structures we designed for the implementation are shown in the table 5.1 with a short description of each.

Table 5.1: Data Structures used in SigComp implementation

S.No.	Data Structures	Description
1.	BitBuffer	Used by the UDVM to read input and write output in a bit-oriented fashion.
2.	Buffer	Used for encapsulating buffers of bytes.

Continued on next page...

Table 5.1 – continued from previous page

S.No.	Data Structures	Description
3.	Compartment	Tracks information for a single remote endpoint. Compartment contains SigComp state objects, information about the amount of memory and advertised states available at the remote end, and any information that the compressor must store.
4.	CompartmentMap	Hash map of compartments.
5.	CompartmentBucket	Bucket for storing compartments after performing the hash in CompartmentMap.
6.	Compressor	Interface for the compressors in SigComp.
7.	CompressorData	A generic interface for all the compressor data. As we can use any compression algorithm with SigComp we have defined a generic interface that all the compression algorithms must follow.
8.	DeflateCompressor	A SigComp compressor based on the Deflate algorithm.
9.	DeflateData	Compartment state information used by the DeflateCompressor.
10.	MultiBuffer	Used to combine arrays so that they can be used as a single array.
11.	MutexLockable	Mutex functionality for multithread synchronization.
12.	NackMap	Hash map used to correlate from NACK messages to the proper compartment.
13.	NackNode	NackNode allows storage of NACK records.
14.	ReadWriteLockable	Provides Read/Write Lock functionality for multithread synchronization.
15.	Sha1Hasher	Performs SHA-1 hashing.

Continued on next page...

Table 5.1 – continued from previous page

S.No.	Data Structures	Description
16.	SigcompMessage	Encapsulates a SigComp message, and provide methods for accessing information from the message.
17.	SipDictionary	Contains the SIP/SDP dictionary defined in RFC 3485
18.	Stack	The main interface between the application and the SigComp
19.	State	Tracks the states stored and used by the UDVM and Compressors
20.	StateChanges	Represents the state changes requested by a successfully decompressed SigComp message
21.	StateHandler	Stores state for the UDVM and tracks NACK messages and compartments
22.	StateList	Tracks a list of states associated with a compartment
23.	StateNode	Tracks state associated with a single state in a StateList
24.	StateMap	Hash map of State objects, keyed by State ID
25.	UDVM	The UDVM is the main component of the decompression mechanism. It takes the compressed message, accesses the state associated with it and decompresses the message. UDVM is capable of using any compression mechanism; we just need to upload the bytecode for the algorithm. For now we have fixed the algorithm to be Deflate.

5.1.2 Pseudocode: Compression and Decompression mechanism

In this section, we have presented the algorithm that we designed to implement the SigComp. This the core algorithm that defines the compression and decompression according to the standard. We have left out the internal details in order to maintain the clarity of the algorithm.

Compression: If this is the first message, we add the bytecode for decompression in the outgoing message, otherwise we extract the data from the state of previously sent messages and use it. Now, we add the message to be sent to the dictionary, and perform the string matching, and encoding of length and distance. If no match is found we add the literal after encoding it.

Algorithm 1 Deflate dictionary

```
1: DeflateDictionary provide methods to encode the length, distance, and literals in
   message. These methods are used by the compress() method while performing com-
   pression.
2: findNextLenghtAndDistance(len,dis) /*Get the next (length,distance) pair.*/
3: encodeLength(len) /*Encode the length of match.*/
4: encodeDistance(dis) /*Encode the distance where pattern is found.*/
5: encodeLiteral() /*Encode the literals, for which no pattern was found.*/
```

Algorithm 2 Deflate compression

```
6: /*Initializations*/
7: Create & initialize the stateHandler
8: Create & initialize stack
9:     Initialize stack's UDVM
10:    Create & add stateChanges to UDVM
11: Create & initialize deflateCompressor
12: Add deflateCompressor to stack
13: /*Initializations complete*/
14:
15: /*CompressMessage*/
16: compartment ⇐ Get the Compartment from stateHandler
17: if !compartment then
18:     Create compartment and add it to stateHandler
19: end if
20: Increment compartment.retainCounter
```

```
if $stateHandler.nackCount$ > threshold then
    trim the $stateHandler.nackMap$ to remove old Nacks from $stateHandler$
end if
//start the compression
$deflateData \Leftarrow$ Get DeflateData from $compartment$
if !$deflateData$ then
    Create and add $deflateData$ to $compartment$
end if
$oldstate \Leftarrow$ Get most recently acked state from $compartment$
//Now create a SigComp message that we will be writing into
if !$oldstate$ then
    $sm \Leftarrow$ Create SigComp message with $oldState$'s data.
else                                                    ▷ this is the first message
    $sm \Leftarrow$ Create SigComp message with bytecodes added.
end if
if sending bytecodes then
    Include our local capabilities with outgoing message.
    Add $CpbDmsSms$
end if
Advertise: $statememory$, $SIPdictionary$, and $decompressionbuffersize$ in $sm$.
Add 4-bit $serialnumber$ to $sm$.
$dictionary \Leftarrow$ Create and initialize DeflateDictionary
/*Now we will perform the actual compression.*/
if not the first message then
    Extract the data from the $oldState$ and add it to $dictionary$.
end if
Add the new message's data to the $dictionary$.
/*Now we will encode the message using dictionary.*/
while $dictionary$.isfinished() do
    if $dictionary$.findNextLengthAndDistance(len,dis) then
        $dictionary$.encodeLength(len);
```

```
52:         dictionary.encodeDistance(dis);
53:     else
54:         dictionary.encodeLiteral(dictionary.getCurrent());
55:     end if
56: end while           ▷ After this encoding of the message the compression is complete.
57: //Check to ensure we're not about to overflow the decompression memory size on the
    remote stack
58: if sm.getDatagramLength() > maxSigcompMessageSize then
59:     //We have exceeded the size
60:     //Report error
61: end if
62: newState ⇐ Generate new state using oldState and new data.
63: Release oldState
64: Add newstate in stateHandler
65: Add newstate in compartment for future use
66:      Add newState in compartment.remoteStateList
67: Store newstate's id as current stateId in deflateData
68: Release newstate
69: if sm is not valid then
70:     Create SigComp message without compression, and mark the header indicating
    the uncompressed message
71: end if
72: Add the requested feedback to outgoing sm
73: Hash the outgoing SigComp message in stateHandler.nackMap ▷ If we get a NACK
    for this message we can refer to this message
74: Release compartment
         return sm
```

Decompression: On receiving the compressed SigcompMessage, we first check for its validity. Next, we load either the stored state or the received bytecode in UDVM, depending on the message content. Then, we execute the code and create the stateChanges object, which defines the changes in the compartment to get the uncompressed message.

Algorithm 3 Decompression

```
1: /*Decompression*/
2: buffer ⇐ Read the message from the socket
3: sm ⇐ Create the SigcompMessage from buffer
4: if sm is Nack then
5:     compartment ⇐ Find the corresponding Compartment in stateHandler
6:     Remove all the remote-states that the NACKed message was expected to create
       from compartment
7:     Release compartment
8: end if
9: Create an outputBuffer to extract the uncompressed message in it
10: /*Initialize the UDVM for decompression*/
11: if sm is not valid then
12:     Put the failure reason in UDVM
13:     break
14: end if
15: Initialize the UDVM memory size                    ▷ Different for TCP/UDP
16: if bytecode is present in the message then
17:     Load the code in UDVM
18: else
19:     Load the state in UDVM
20: end if
21: Initialize the UDVM's memory space parameters: memorySize, cyclesPerBit, and
    sigcompVersion
22: /*UDVM Initialization complete*/
23: if sm is valid then
24:     Execute the code and prepare the stateChanges object accordingly
25: end if
26: if UDVM fails & sigcompVersion > 2 then
27:     stack.nack ⇐ Get the Nack from UDVM & return
28: end if
29: stateChanges ⇐ Get proposed states from the UDVM
```

```
30: Add the *returnedfeedback* from *sm* into *stateChanges*
31: /*Here the decompression of the message is complete. We have the uncompressed
    message in UDVM's output buffer*/
32: //Provide compartment Id
33: **if** no *stateChanges* **then**
34:     return
35: **end if**
36: *compartment* ⇐ Get compartment from *stateHandler*
                                                    ▷ Process stateChanges
37: **if** S-bit is set **then**
38:     Remove *compartment* from the *stateHandler*
39: **end if**
40: **for** all the operations in *stateChanges* **do**
41:     **if** operation == ADD_STATE **then**
42:         Get *state* from *stateChanges*
43:         Add *state* in *compartment*
44:     **end if**
45:     **if** operation == REMOVE_STATE **then**
46:         Remove *state* from *compartment*
47:     **end if**
48:     Set *compartment.CpbDmsSms*
49:     Reset remote advertised states of *compartment*
50:     Add the remote advertised states in *compartment* from *statechanges*
51:     **if** I-bit is valid **then**
52:         Set I-bit
53:     **end if**
54:     Set Requested Feedback and Returned Feedback in *compartment*
55: **end for**
56: //Handle feedback
57: *feedback* ⇐ Get returned feedback from *compartment*
58: *stateId* ⇐ Get stateId from *deflateData*
```

```
59: Ack remote state.
60: Release compartment
```

5.2 SigComp with Asterisk

5.2.1 Integration of SigComp with Asterisk

In Asterisk, we have different channels for each type of communication, like for SIP we have chan_sip.c, for ZAP channel chan_zap.c. For integrating SigComp, we have modified the Asterisk's SIP channel. We have made it thread safe, in order make it work for different calls from different users. Now, when a call is made through Asterisk, a thread is created, to transfer messages for both the clients. The Asterisk compresses & decompresses the messages similar to the end user. We have inserted the compression code in its `__sip_transmit()` method, and the decompression code in `sipsock_read()` method.

5.2.2 Test Cases

- Working of SigComp as stand alone application.
  - Underlying protocol (TCP/UDP):

    We have tested the working of SigComp as a stand alone application with TCP and UDP. TCP being a stream oriented protocol uses the TCPStream data structure to add the stream in to the ongoing connection. It also uses the stream to extract the SigComp messages on the receiving end. We need to insert the message-end markers in the stream to indicate the message boundaries.

    With UDP we used the datagram to send and receive the SigComp messages. After receiving the SigComp message we decompress the message using decompressMessage() method. Along with the uncompressed SIP message, it also provides the StateChanges object which provides a number of states to be added and removed as requested by the bytecodes.
  - Compression test

    We have implemented the Deflate compressor as per the SigComp standard. We have to test for the state extraction from the compartment, compression of message, and inclusion of bytecode and state identifier in the outgoing message.

– Decompression test

  After receiving the compressed message we need to perform the decompression. For both UDP and TCP, the application examines the decompressed message. If it finds it to be valid, it calls “provideCompartmentId” on the Stack; this method takes the compartmentId associated with the decompressed message and the “StateChanges” object that was returned from the earlier call to “uncompressMessage”. The stack then updates the StateManager according to the instructions in the “StateChanges” object. If the application decides that the message is invalid, it simply destroys the StateChanges object.

– CrcComputer test

  The CRC is added to every outgoing SigComp message for reliability. We have tested the CRC computation by giving a message to it and comparing it with the expected output.

– Sha1Hasher test

  We have used the SHA1 hasher as specified in the SigComp standard for hashing the messages. We keep on adding the message data to SHA1 hasher and then call getSha1hash() method to get the hash of the message.

– Stack test

  Stack is the user-end interface to the SigComp. It provides all the methods to perform operation on the SIP message to compress and decompress it. We have tested the working of Stack by running it for sending messages as given in SIP standards example.

– Torture tests

  We have followed the torture tests described in RFC 4465[7] to implement the torture tests for our implementation.

  Stack: We have performed the torture tests on the system by compressing and decompressing 1000 messages back to back.

  UDVM: We did the torture test for UDVM by performing all the operation defined for UDVM in the standard.

StateHandler: The torture test for State Handler includes the test for SigComp feedback mechanism, state memory management, multiple compartments, bytecode state creation.

- Asterisk and SigComp.

  We also performed test on the SigComp's working with Asterisk server in following scenarios:

  – Single client: Only one client connected to the Asterisk, making no calls. It checks that the client registration with Asterisk is working fine. Then we also tested it when one client is making call to itself.
  – Multiple clients: Multiple clients from one machine connected to Asterisk and making calls to each other. It checks that threading is done properly for each client and each call.
  – Torture test: We used the Yate callgen to generate around 20,000 calls to the Asterisk server. It checks the load handling of the system and also shows how much improvement is made by using the SigComp with the Asterisk.

5.3 SigComp with Yate

Yate is a open source SIP client. We have integrated SigComp with the Yate softphone to make calls using the Asterisk server. Yate provides us the callgen facility to make multiple parallel calls, which we have used to test the working of Asterisk server on heavy loads.

5.3.1 Integration of SigComp with Yate

We have integrated SigComp with Yate in its SIP-channel (ysipchan.c) file. As shown in the figure 5.1, we first create and initialize the Statehandler, the Stack, DeflateCompressor and give an Id to the client. Now, whenever the client sends a SIP message we first compress it and the send it to the other user. On reception of the SigComp message, the user first checks for the message's validity. Now it decompresses the message, then again check the uncompressed message. If the message is valid it sends it to the upper layer, else it sends a NACK message to the other user to inform it about the error.

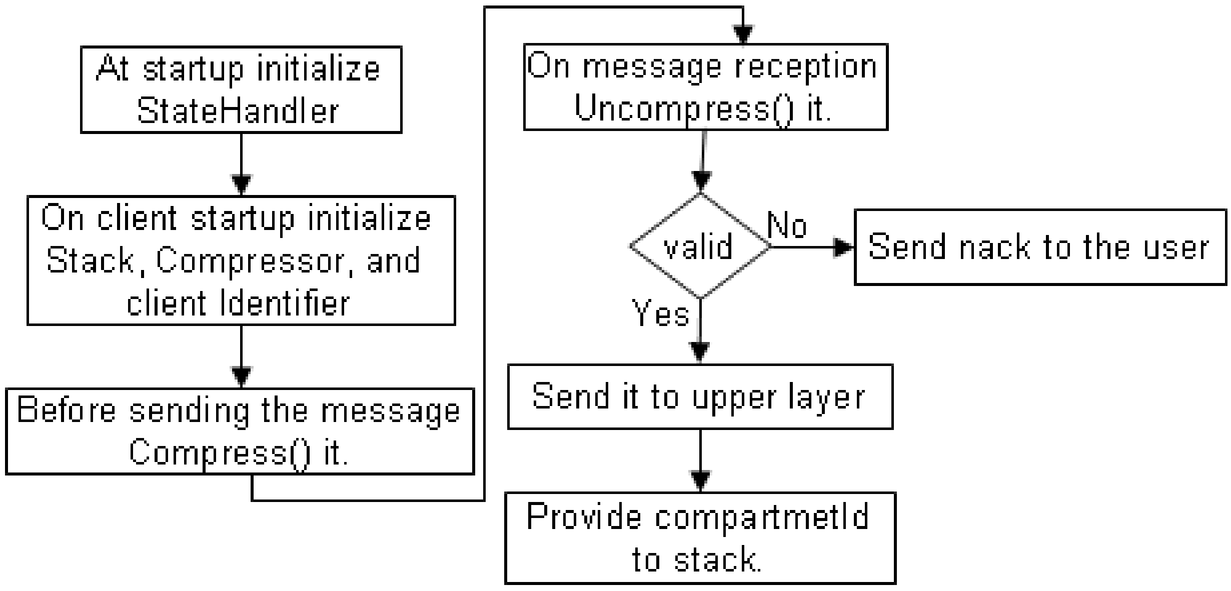
At startup initialize StateHandler
On client startup initialize Stack, Compressor, and client Identifier
Before sending the message Compress() it.
On message reception Uncompress() it.
valid
No
Send nack to the user
Yes
Send it to upper layer
Provide compartmetId to stack.

Figure 5.1: SigComp integration with Yate

5.3.2 Test Cases

- Single instance one call

  Create only one Yate client on a machine and make call to itself. This verifies that on calling the same client we are able to differentiate between incoming and outgoing calls.

- Single instance multiple calls

  Create single instance of Yate and make multiple calls to the itself. This verifies that each call is being treated separately.

- Two instances on different machines with one call

  Now make a call from a different machine. This verifies that the socket level programming is working fine.

- Two instances on different machines with multiple calls

  Create two instances and make multiple calls. This gives us the improvement in Asterisk's response time. We have done this test rigorously for 20,000 parallel calls, and shown our results in figure 5.5.

5.4 Experiments and Results

In this section, we have presented the results of the experiments we have done with SigComp. We have shown the improvement in the response time of Asterisk that we have obtained by using SigComp. We have also shown the improvement in session establishment while making direct SIP-to-SIP call.

5.4.1 Packet drop probability vs Packet size

We have made the study of packet drop probability with varying packet sizes for different bandwidths. We have made this study by sending 200 packets of a particular size and calculating the number of packets received correctly.

We made this study because, the compression mechanism is of no use, if even after compression the packet size doesn't fall in the safe range.

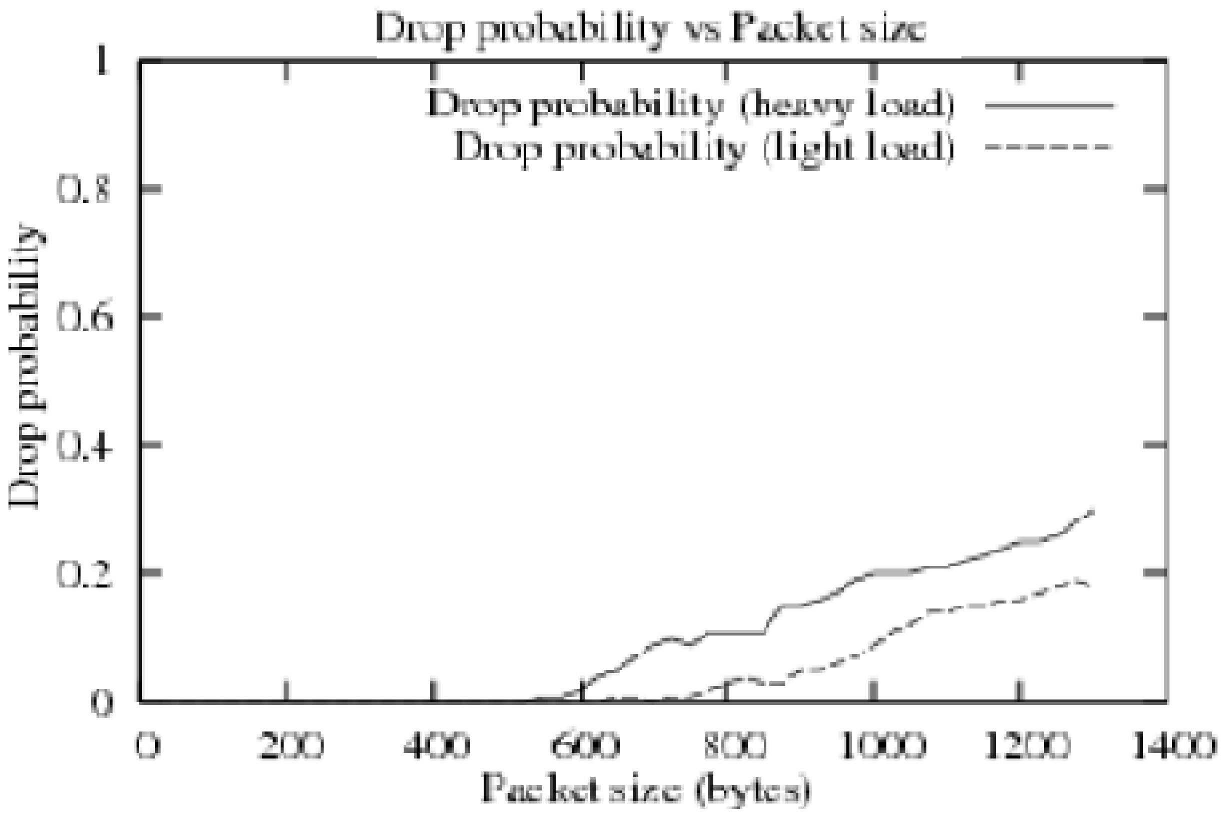

Figure 5.2: Packet drop probability vs Packet size for different bandwidths.

Analysis: From the graph 5.2, we can see that, on heavy load the packets starts dropping more rapidly as compared to light load. The heavy load is representative of low available bandwidth. So we can conclude that if the message size is less in low bandwidth medium we can get good throughput and faster response time.

5.4.2 Compression Ratio with UDP

In this experiment, we have calculated the compression ratio of the SIP messages that we have obtained using SigComp. When the Yate starts up it sends a REGISTER request to the server. Starting from the REGISTER method till BYE we have observed the size of the compressed SIP messages being exchanged. The graph 5.3, shows the percentage compression we have obtained by using SigComp.

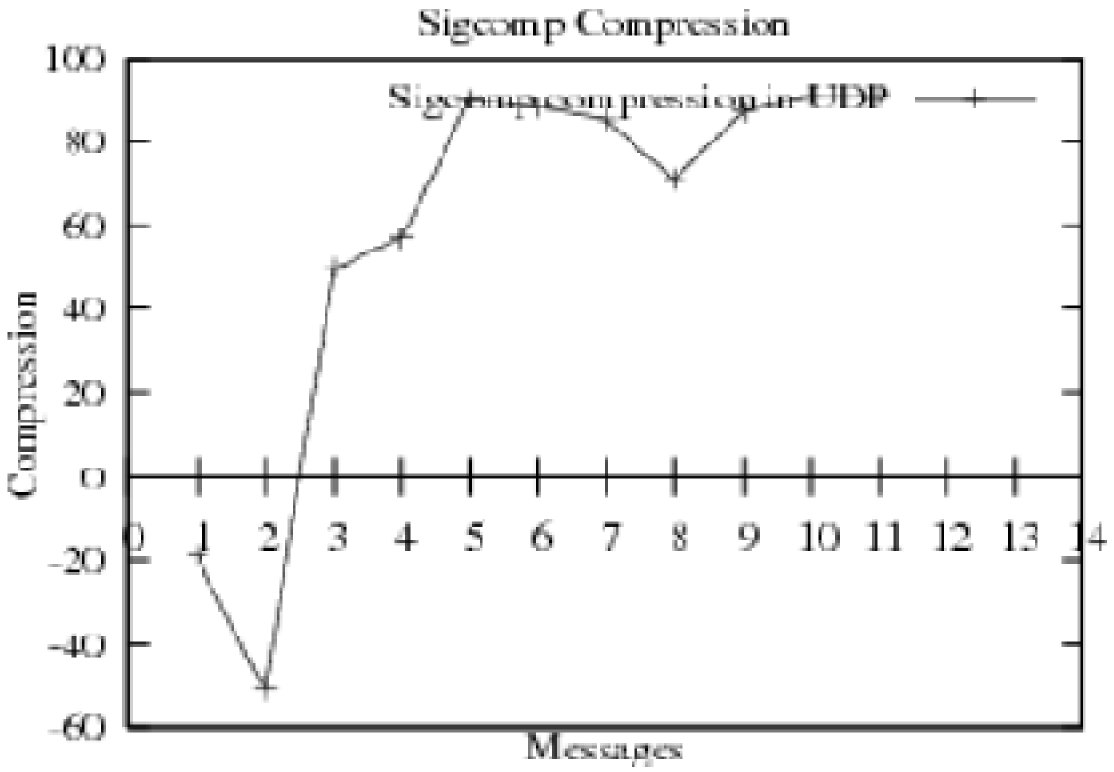

Figure 5.3: UDP: Deflate compression (Compression ratio vs Packet sequence number)

Table 5.2: Message compression obtained by using SigComp

Message No.	Original size	Compressed size	% compression
1	850	1017	119
2	468	710	**151**

Continued on next page...

Table 5.2 – continued from previous page

Message No.	Original size	Compressed size	% compression
3	1168	586	50
4	483	208	43
5	418	42	10
6	1360	105	**7**
7	393	59	15
8	632	189	29
9	405	55	13
10	405	39	**9**
11	384	38	**9**

Analysis: We have obtained upto 90% compression ratio by using the SigComp with Asterisk. The first few messages become larger because of the transmission of the bytecode along with the message. We can reduce the size of these messages too if we fix the algorithm before hand.

5.4.3 Improvement in Asterisk response time

The graph 5.4, shows the improvement we have obtained by using SigComp with Asterisk. We have made 20,000 parallel calls and recorded the response time of Asterisk. Here, we have compared the response time of Asterisk server in wireless medium, for two cases: 1) with compression mechanism and 2) without compression.

Analysis: By using the SigComp with Asterisk we have obtained the improvement of about 15% in its response time. The gain is highly varying, sometimes the response time is more for some calls. But on an average the response time is much improved for multiple parallel calls.

5.4.4 Improvement in SIP to SIP communication

In this experiment, we have measured the improvement in SIP-to-SIP connection establishment that we obtain by using SigComp.

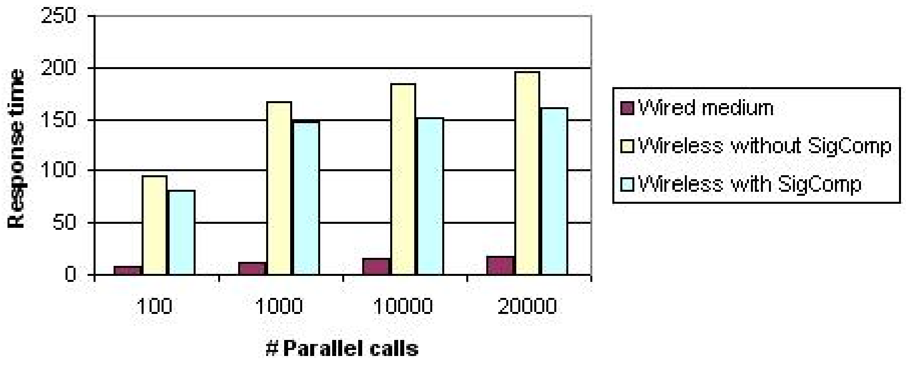

Figure 5.4: Asterisk response time with compression.

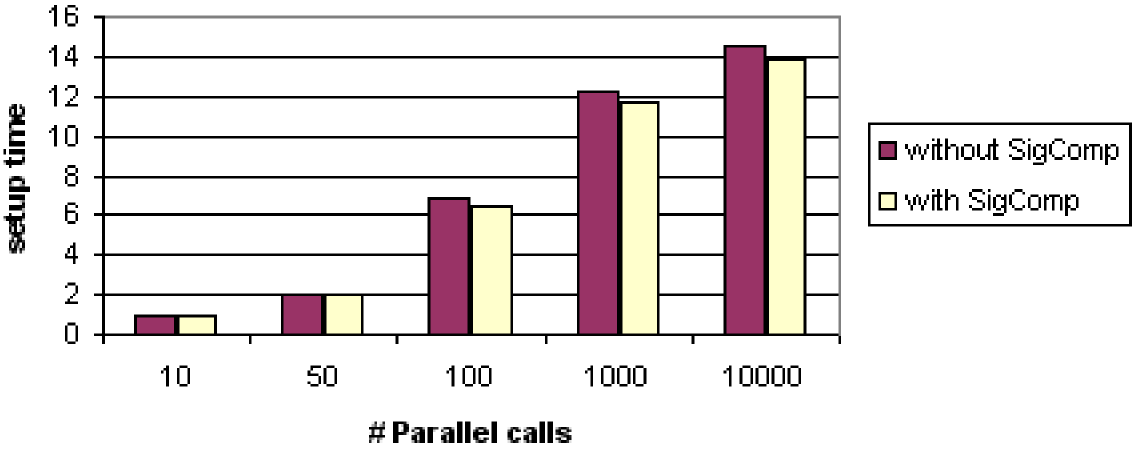

Figure 5.5: SIP to SIP connection improvement.

Analysis: In this experiment we make calls from Yate to Yate without using Asterisk, and we have got about 4-5% improvement in the connection establishment by using SigComp with Yate.

Summary

In this section, we have described the SigComp implementation and its integration with Asterisk and Yate. The improvement in session establishment via Asterisk server shows the potential benefits of using the compression methods with SIP. The computational load on the system is not much affected by using compression, given the use of efficient bytecodes for compression and very large number of users.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have done a lot of experiments with different hardware devices that are used in VoIP and PSTN integration. From the discussion in the previous chapters, we have seen that the design of the single box solution is the most inexpensive, easy to install, robust and, is a low power consuming device. It is the best we can get out by using off-the-shelf components.

Using SigComp we have achieved a compression of about 90% in the SIP messages. The Deflate compression algorithm reduces the size of messages using the pattern of bytes being transferred. We have improved Asterisk's response time by about 10-15%. We have also reduced the session establishment time in direct SIP-to-SIP calls. We have planned to integrate the SigComp implementation with the main branch of Asterisk's open source code repository.

6.2 Future Work

The project can be enhanced in the following ways:

- Hardware implementation of the solution

  We can implement the architecture design of the single box solution on a hardware unit. That will give us better cost reduction and power efficiency.

- More compression algorithms

  We have currently implemented only Deflate compression algorithm for Asterisk and Yate. More compression algorithms can be implemented and compared with the current results we have obtained.

- Data storage on Edge Proxy

  We have proposed the solution of data storage on Edge Proxy, for reducing the message size. But the implementation of the proposal is still undone. Along with the compression mechanisms, the implementation of stateful Edge Proxy is expected to produce much more better results.

- 3GPP2 standard

  3GPP2 IMS project also proposes the use of SigComp for compression in mobile clients. Its standards are still in the making phase. Once its standards are available we can integrate the SigComp implementation with its applications.

Bibliography

[1] J. V. Meggelen, J. Smith, and L. Madsen, *Asterisk The Future of Telephony.* O'Reilly, August 15, 2007.

[2] M. Spencer, M. Allison, and C. Rhodes, *The Asterisk Handbook.* Asterisk Documentation Team, 2003.

[3] L. Madsen, J. Smith, S. Sokol, W. Baig, D. Heinzen, J. Rollyson, P. Grace, N. Bachmann, M. Preston, M. List-Petersen, W. Suffill, J. V. Meggelen, and C. Tooley, *The Hitchhiker's Guide to Asterisk*, 2003.

[4] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session Initiation Protocol.* RFC 3261, June 2002. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3261.txt

[5] M. Handley and V. Jacobson, *SDP: Session Description Protocol.* RFC 2327, April 1998. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2327.txt

[6] R. Price, S. Manor, C. Bormann, T. Bremen, J. Christoffersson, H. Hannu, Z. Liu, and J. Rosenberg, *Signaling Compression (SigComp).* RFC 3320, January 2003. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3320.txt

[7] A. Surtees and M. West, *Signaling Compression (SigComp) Torture Tests.* RFC 4465, June 2006. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4465.txt

[8] H. Hannu, J. Christoffersson, S. Forsgren, K.-C. Leung, Z. Liu, and R. Price, *Signaling Compression (SigComp) - Extended Operations.* RFC 3321, January 2003. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3321.txt

[9] P. Carden, *Building Voice over IP*, May 8, 2000. [Online]. Available: http://www.networkcomputing.com/netdesign/1109voip.html

[10] ShenKai, XiangYong, and S. MeiLin, "Design of lan based ip phone," in *Communication Technology Proceedings, 2000. WCC - ICCT 2000. International*, 2000.

[11] R. Balbinot, J. Guedes-Silveira, and P. Franco, *Desktop Voice Over IP Development with PSTN integration*, 2001.

[12] W. E. Witowsky, *IP Telephone Design and Implementation Issues*, July 1998.

[13] Y. Zhang, *SIP-based VoIP network and its interworking with the PSTN*, Dec 2002.

[14] *Sipura User Guide*, July 2004.

[15] M. Lad, M. Jalan, D. Patil, and S. Sule, *IP based Single-line VoIP Residential Gateway*, 2002.

[16] "Asterisk - open source pbx." [Online]. Available: http://www.asterisk.org/

[17] "Yate - yet another telephony engine." [Online]. Available: http://yate.null.ro

[18] "Deflate compression algorithm." [Online]. Available: http://www.gzip.org/algorithm.txt

[19] T. Y. Chan, D. Greenstreet, G. Yancey, K. Devlin-Allen, D. Jarrett, K. Buchanan, S. Scoggins, M. Harvill, and D. Jobson, "Building residential voip gateways: A tutorial," in *White paper, Texas instruments*, 2000.

[20] H. Fathi, S. S. Chakraborty, and R. Prasad, "Optimization of sip session setup delay for voip in 3g wireless networks," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, 3 Dec 2004, pp. 4092– 4096 Vol.6. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1379135