HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Electrical and Communications Engineering
Networking Laboratory

Jouni Mäenpää

Performance of Signalling Compression in the Third Generation Mobile Network

Thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Engineering

Espoo, Finland, June, 7, 2005

Supervisor	Professor Raimo Kantola
Instructor	Harri Reiman, M.Sc.

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF THE MASTER'S THESIS**

Author:	Jouni Mäenpää		
Name of thesis:	Performance of Signalling Compression in the Third Generation Mobile Network		
Date:	7.6.2005	**Number of pages:**	193
Faculty:	Department of Electrical and Communications Engineering		
Professorship:	S-38 Networking Technology		
Supervisor:	Prof. Raimo Kantola		
Instructor:	Harri Reiman, M.Sc. (Tech)		

The use of Session Initiation Protocol (SIP) as the call control protocol in the third generation mobile network, starting from the Third Generation Partnership Project (3GPP) release 5 onwards, results in long call setup times. This is because the large size of SIP signalling messages increases the transfer delay over the narrowband radio interface. Since users will see little sense in switching to a service that does not provide at least the same quality of service as the existing systems, a solution is needed to reduce the call setup time. One such solution is the Signalling Compression (SigComp) protocol designed by the Internet Engineering Task Force (IETF). SigComp provides a framework for the compression of application-layer signalling between two network elements.

This master's thesis examines the performance of the SigComp protocol. In addition, the architecture and operation of the SigComp prototype used in the performance evaluation are presented.

The first part of the thesis introduces some theory and literature related to the subject. The architecture of the third generation mobile network is presented and the central concepts of the SigComp protocol are described. The way SigComp can be applied to SIP is explained. Also an introduction to computer and operating system architecture is given.

In the second part of the thesis, the architecture and operation of the SigComp prototype are presented. A modified version of the Lempel-Ziv-Storer-Szymanski (LZSS) compression algorithm is introduced. The algorithm belongs to the class of dictionary compression algorithms and is used throughout the measurements.

In the third and final part of the thesis, the results of the measurements performed on the SigComp prototype are presented and analysed. Also the way the measurements were carried out is described. Because it can be expected that most SigComp implementations will use dictionary compression algorithms like the modified LZSS used in this thesis, the results presented in the third part are applicable to a wide range of compression algorithms.

Keywords: signalling compression, performance, SigComp, SIP

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Jouni Mäenpää	
Työn nimi:	Signalling compression -protokollan suorituskyky kolmannen sukupolven matkapuhelinverkossa	
Päivämäärä:	7.6.2005	**Sivumäärä:** 193
Osasto:	Sähkö- ja tietoliikennetekniikan osasto	
Professuuri:	S-38 Tietoverkkotekniikka	
Työn valvoja:	Prof. Raimo Kantola	
Työn ohjaaja:	Harri Reiman, DI	

SIP (Session Initiation Protocol) –protokolla on valittu kolmannen sukupolven matkapuhelinverkon puhelunhallintaprotokollaksi 3GPP:n (Third Generation Partnership Project) viidennestä julkaisusta eteenpäin. SIP-merkinantoviestien suuren koon vuoksi SIP-protokollan käytöllä on negatiivinen vaikutus puheluiden aloitusviiveeseen. Koska kolmannen sukupolven matkapuhelinverkon käyttäjät eivät tule kokemaan houkuttelevana järjestelmää, jonka tarjoama palvelunlaatu on huonompi kuin olemassa olevien järjestelmien, on syntynyt tarve ratkaisulle, joka voisi pienentää puhelunaloitusviivettä SIP-protokollaa käytettäessä. Tärkein tällainen ratkaisu on IETF:n (Internet Engineering Task Force) SigComp (Signalling Compression) –protokolla. SigComp tarjoaa viitekehyksen sovellustason merkinantoliikenteen pakkaamiselle kahden verkkoelementin välillä.

Tässä diplomityössä tutkitaan SigComp-protokollan suorituskykyä. Lisäksi työssä esitetään suorituskykymittauksissa käytetyn SigComp-prototyyppitoteutuksen arkkitehtuuri ja toiminta.

Työn ensimmäinen osa käsittelee aiheeseen liittyvää teoriaa ja kirjallisuutta. Ensimmäisessä osassa esitetään kolmannen sukupolven matkapuhelinverkon arkkitehtuuri ja SigComp-protokollan keskeiset käsitteet. Ensimmäisessä osassa käydään myös läpi SigComp-protokollan soveltaminen SIP-liikenteen pakkaamiseen ja annetaan johdanto tietokoneiden ja käyttöjärjestelmien arkkitehtuuriin.

Työn toisessa osassa esitetään SigComp-prototyypin arkkitehtuuri ja toiminta. SigComp-prototyypissä käytetään muokattua versiota Lempel-Ziv-Storer-Szymanski (LZSS) –pakkausalgoritmista. Tämä algoritmi kuuluu sanakirjapohjaisten pakkausalgoritmien luokkaan ja sitä käytetään kaikissa työhön liittyvissä mittauksissa.

Työn kolmannessa osassa esitetään SigComp-prototyypin suorituskykymittausten tulokset ja niiden analyysi. Myös mittausjärjestelyt esitellään. Koska voidaan odottaa, että suuri osa SigComp-toteutuksista tulee käyttämään sanakirjapohjaisia pakkausalgoritmeja, tässä työssä esitellyt tulokset ovat sovellettavissa moniin eri pakkausalgoritmeihin.

Avainsanat: merkinantoliikenteen pakkaus, suorituskyky, SigComp, SIP

Preface

This thesis was written at Oy LM Ericsson Ab Finland.

I would like to thank my supervisor, professor Raimo Kantola and my instructor, Harri Reiman. I would also like to thank Christer Holmberg and Tomas Mecklin of Ericsson Finland for their guidance.

Finally, I would like to thank my family and my girlfriend Johanna for their support.

Espoo, 1.6.2005,

Jouni Mäenpää

Table of Contents

Abbreviations

3G	Third Generation
3GPP	Third Generation Partnership Project
ACK	Acknowledgement
AN	Access Network
AS	Application Server
BGCF	Breakout Gateway Control Function
BS	Base Station
CAMEL	Customised Application for Mobile network Enhanced Logic
CDR	Call Detail Record
CN	Core Network
CPU	Central Processor Unit
CS	Circuit Switched
CSCF	Call Session Control Function
DDR	Double Data Rate
DoS	Denial-of-Service
DMS	Decompression Memory Size
DRAM	Dynamic Random Access Memory
FIFO	First-In-First-Out
FTP	File Transfer Protocol
GGSN	Gateway GPRS Support Node
GMSC	Gateway Mobile Switching Center
GPRS	General Packet Radio Service
GSM	Global System for Mobile communication
HSS	Home Subscriber Server
HTTP	Hypertext Transfer Protocol
I-CSCF	Interrogating Call Session Control Function
IM	IP Multimedia
IMS	IP Multimedia Subsystem
IMS-MGW	IMS Media Gateway Function
IM-SSF	IP Multimedia Service Switching Function
IP	Internet Protocol
IPComp	IP Compression
JVM	Java Virtual Machine
LZ77	Lempel-Ziv 1977
LZSS	Lempel-Ziv-Storer-Szymanski
MGCF	Media Gateway Control Function
MO	Mobile Originated
MRFC	Multimedia Resource Function Controller
MRFP	Multimedia Resource Function Processor
MSC	Mobile Switching Center
MT	Mobile Terminated
NACK	Negative Acknowledgement
OMA	Open Mobile Alliance
OSA	Open Service Access
P-CSCF	Proxy Call Session Control Function
PDP	Packet Data Protocol

PDU	Protocol Data Unit
PoC	Push-to-talk over Cellular
PS	Packet Switched
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RAN	Radio Access Network
RFC	Request For Comments
RNC	Radio Network Controller
ROHC	RObust Header Compression
RTSP	Real Time Streaming Protocol
RTT	Round-Trip Time
S-CSCF	Serving Call Session Control Function
SDP	Session Description Protocol
SDRAM	Synchronous Dynamic Random Access Memory
SGSN	Serving GPRS Support Node
SHA-1	Secure Hash Algorithm 1
SigComp	Signalling Compression
SIP	Session Initiation Protocol
SLF	Subscription Locator Function
SRAM	Static Random Access Memory
SS7	Signalling System No. 7
TCP	Transmission Control Protocol
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
UDVM	Universal Decompressor Virtual Machine
UE	User Equipment
UML	Unified Modelling Language
UMS	UDVM Memory Snapshot
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USD	User-Specific Dictionary
UTF	Unicode Transformation Format
UTRAN	UMTS Terrestrial Radio Access Network
WLAN	Wireless Local Area Network
XML	eXtensible Mark-up Language

1 Introduction

1.1 Background

Session Initiation Protocol (SIP) is the protocol used for call control in the third generation mobile network starting from the Third Generation Partnership Project (3GPP) release 5. SIP uses textual encoding, which makes it easier to build services based on SIP, design extensions to SIP and debug the protocol. However, the textual encoding of SIP also has a serious drawback; it is well-known that SIP messages are considerably larger than those of the protocols used for instance in GSM call control. Large message sizes result in increased call setup delay because more data needs to be transmitted over the low-bandwidth radio interface. This observation created a need to develop a solution which could reduce the call setup time. One such solution is the Signalling Compression (SigComp) protocol designed by the Internet Engineering Task Force (IETF). SigComp provides a framework for the compression of application-layer signalling between two network elements. The central piece of SigComp architecture is the Universal Decompressor Virtual Machine (UDVM), which is a virtual machine optimised for running decompression algorithms. Because of the UDVM, SigComp can support a wide range of compression algorithms instead of dictating a single algorithm to be supported by all SigComp endpoints.

SigComp is a mandatory part of the 3GPP release 5 IP Multimedia Subsystem (IMS). It is applied over the interface between a terminal and Proxy Call Session Control Function (P-CSCF), which is the first contact point for the terminal within the IMS. SigComp improves the quality of service the user perceives by reducing the idle time at call setup. It also allows the network to support a greater number of users by reducing the amount of resources consumed per subscriber.

The concept of the compression of protocol information is certainly not a new one and has already been applied in other contexts. Well-known examples include the File Transfer Protocol (FTP) [RFC 959], which defines a compressed transmission mode, and IP payload compression (IPComp) [RFC 3173], which can be used to reduce the size of Internet Protocol (IP) datagrams. Yet another related approach to protocol compression is the compression of signalling protocol headers. An example of this is the Robust Header Compression (ROHC) [RFC 3095] scheme of IETF. However, these approaches are not suitable for the compression of application protocol payload in a mobile network.

Like the compression of signalling protocols, also the use of virtual machines has a long history. A virtual machine is an abstract computer that is implemented in software and executed on a real hardware platform and an operating system. A well-known example of a virtual machine is the Java Virtual Machine (JVM) [Lindholm 1997], an abstract computer that executes compiled Java programs. The SigComp UDVM is a virtual machine much like the Java Virtual Machine, but with the key difference that it has been optimised for running decompression algorithms.

The primary target for SigComp is cellular systems, where the mobile terminals have varying capabilities and undetected errors may be introduced on the cellular link. SigComp is not the optimal choice in short-range wireless networks, such as the Wireless Local Area Network (WLAN), in which the throughput is usually high enough and latency low enough to support uncompressed messages. When throughput and latency are not an issue, unnecessary compression and decompression can even decrease performance. SigComp is also not the best compression scheme on wired links, such as those between network servers. Because of the design decisions made in the development of SigComp, other compression solutions are more efficient for these purposes.

1.2 The Goals and Objectives of the Thesis

Because SigComp is a new feature, it is important to study its performance, including the achievable compression ratios and the amount of resources it consumes in the network elements performing compression and decompression of SIP messages. So far little research has been carried out on this topic and the focus has been on estimating the achievable compression ratios. This thesis describes the performance and architecture of a SigComp prototype implementation. The main goal is to examine the performance of the SigComp protocol through measurements performed on the prototype. The secondary goals are (1) to describe the way SigComp functionality can be implemented and (2) to examine the way the load placed by SigComp compression and decompression can be reduced.

1.3 Scope of the Thesis

This thesis concentrates on the general class of dictionary compression algorithms, i.e. algorithms that compress data via textual substitution. The algorithm used in the measurements is a modified version of the Lempel-Ziv-Storer-Szymanski (LZSS) compression algorithm. The application-layer protocol being compressed is the Session Initiation Protocol (SIP). The performance of the following SigComp mechanisms defined in [RFC 3320], [RFC 3321] and [RFC 3485] is studied: basic compression, static SIP and Session Description Protocol (SDP) dictionary, dynamic compression and shared compression.

1.4 The Structure of the Thesis

This thesis is divided into three parts. The first six chapters form the first part and they cover the theory and background information that are related to the SigComp protocol and to the implementation of the prototype. The chapters of the first part are based on literature research.

Chapter 2 describes the central concepts related to the third generation mobile network.

Chapter 3 covers the most important concepts of the SigComp protocol.

Chapter 4 describes the way SigComp compression is applied to the Session Initiation Protocol (SIP).

Chapter 5 gives an introduction to computer and operating system architectures.

Chapter 6 presents some previous research on SigComp.

In the second part, consisting of Chapter 7, the architecture and operation of the SigComp prototype that was implemented as a part of the thesis work are described. We implemented the prototype from scratch for the purposes of this thesis.

In the third and final part of the thesis, consisting from chapters 8 to 12, the results of the measurements carried out on the SigComp prototype implementation are presented and analysed. The measurements are divided into three phases.

Chapter 8 explains the way the measurements will be carried out. A systematic approach to performance evaluation is used.

Chapter 9 describes the first phase of the measurements. The aim of the first phase is to study various factors that affect the performance of SigComp.

Chapter 10 presents the second phase of the measurements. In the second phase, the performance of the SigComp protocol is studied in case of different SIP message sequences.

Chapter 11 consists of the third and final phase of the measurements. In the third phase, the performance of the system performing SigComp compression and decompression is evaluated.

Chapter 12 concludes the thesis, describes the advantages and limitations of the SigComp protocol, presents the considerations made in the thesis, and suggest ideas for further work.

2 Third Generation Mobile Network

The purpose of this chapter is to give an overview of the environment in which SigComp is used. The architecture of the third generation mobile network, the IP Multimedia Subsystem (IMS) and the Session Initiation Protocol (SIP) are presented. Also the reasons why a compression scheme like SigComp is required is demonstrated by comparing the call setup delay in the second and third generation mobile networks.

2.1 Comparison of Call Setup Delay in Second and Third Generation Mobile Networks

To make it clear why a compression scheme like SigComp is needed, the call setup delay in a GSM network and in a UMTS network using SIP as the call control protocol is compared in this section.

The worst-case estimate regarding the maximum combined message size of a GSM call setup is calculated in [Nortel 2000] to be approximately 1050 bytes. [Foster 2002] presents results derived from a UMTS system simulator. The results obtained for an initial SIP call setup delay in a UMTS network applying SIP are shown in Table 1. Compression was not applied in the simulations. MT stands for mobile terminated i.e. from a fixed user in the Public Switched Telephone Network (PSTN) to a mobile user. MO stands for mobile originated.

Table 1 - SIP call setup delay in a UMTS network applying SIP but not compression [Foster 2002]

Delay	MT SIP call setup	MO SIP call setup
RAN delay	4,228 s	4,169 s
Core delay	2,397 s	2,692 s
Total delay	6,624 s	6,861 s

The results presented in [Foster 2002] include also estimates for call delays in a GSM network and a UMTS release 99 network, which does not use SIP for call control. These results are shown in Table 2.

Table 2 - GSM and UMTS Release 99 call setup delay [Foster 2002]

Delay	MT call	MO call	Mobile to mobile call
GSM	2,2 s	2 s	4,0 s
UMTS Release 99	2 s	1,7 s	3,4 s

We can see from the results presented above that for example the mobile originated call setup delay in a UMTS network applying SIP is over 4.8 seconds longer than the mobile originated delay calculated for the GSM network. It can also be seen that the call setup delay in a UMTS release 99 network not applying SIP is of the same level as in the case of a GSM network. In addition, one can observe that in a UMTS network applying SIP, the radio access network (RAN) delay seems to contribute a significant portion (68%) of the total delay.

The main reason behind the increased call setup delay in a UMTS network applying SIP compared to a GSM or release 99 network is the use of SIP for call signalling. SIP is not an efficient protocol regarding message size. The textual encoding of SIP makes SIP messages grow dramatically as soon as several extensions are used at the same time. GSM uses Signalling System No. 7 (SS7) protocols for signalling. These protocols use bit-wise encoded messages that make more efficient use of the air interface where bandwidth is limited.

Table 3 – SIP messages

	SIP Message	Size uncompressed [bytes]
1	INVITE	1437
2	100 Trying	254
3	183 Session Progress	1440
4	PRACK	1318
5	200 OK	904
6	UPDATE	1291
7	200 OK	865
8	180 Ringing	563
9	PRACK	717
10	200 OK	260
11	200 OK	1133
12	ACK	458
	Total	10640

Table 3 shows example message sizes for a mobile originated SIP session establishment in a release 5 network. The flow of messages between the UE and the P-CSCF is taken from [3GPP TS 24.228] and is illustrated in Figure 1. The message sizes were also calculated from [3GPP TS 24.228 2004]. We can observe from Table 3 that the combined message size for the setup of a mobile originated call in a release 5 network is 10640 bytes, which is over ten times more than the corresponding worst-case value for GSM call setup, 1050 bytes. The one-way RAN delay for each message can be calculated as follows:

$$One\ way\ RAN\ delay = \frac{Size\ of\ message[bits]}{link\ speed\,[bits/s]} + \frac{RTT}{2}. \quad (2.1)$$

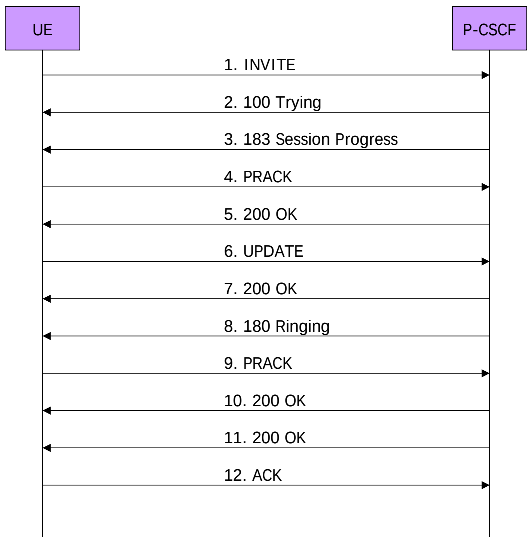


Figure 1 - SIP messages between UE and P-CSCF during session initiation

Because there is pressure to minimize capacity allocations on grounds of cost, it can be expected that relatively low bandwidth is provided for signalling. According to [Nordberg 2003] it is reasonable to assume that a bit rate of the order of 9.6 or 12.2 kbps will be allocated for SIP signalling in the UMTS. The one-way RAN for a SIP session establishment using the messages of Table 3 delay is depicted in Figure 2 for the following signalling link bandwidths: 9.6, 12.2, 16, 32, 64, 128 and 256 kbps. It is assumed that the Round-Trip Time (RTT) of the signalling link is 70 ms [RFC 3322, Nordberg 2003]. The values presented in Figure 2 do not include the overhead added by IP and transport protocol headers. In addition, the values do not include the radio bearer setup delay and the delay introduced by resource reservation and session management signalling before the INVITE message and during the SIP message sequence.

From Figure 2, we can observe that the one-way RAN delay for a mobile originated call is 9.7 seconds if the bit rate of the signalling link is 9.6 kbps. Thus, according to these calculations, the mere RAN delay part of the call setup time can be almost five times longer than the entire mobile originated call setup time in a GSM network. To get the RAN delay close to the entire call setup time in a GSM network, 2.0 seconds, a bit rate of 64 kbps or more is required.

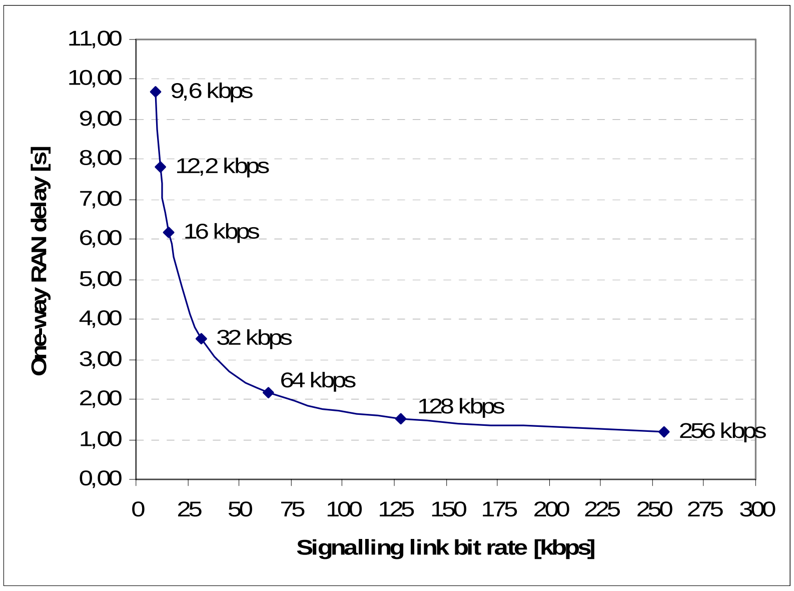


Figure 2 - One-way RAN delay for session establishment signalling flow in a 3GPP release 5 network

It is clear that a call setup time that is considerably longer than in the GSM network is not acceptable. Users will see little sense in switching to a service that does not provide at least the same quality of service as GSM. Therefore, solutions for decreasing the call setup time are needed. Three most intuitive candidates are increased bandwidth per user, reduced round-trip time or smaller message sizes.

If the bit rate of one user is increased, the number of users that a cell can support will decrease. This is not desirable especially in dense urban areas. Decreasing the RTT may not be possible, because it is likely to require considerable changes in the current system architecture. In addition, it is unlikely to have a sufficient impact since only a small portion of the total delay is due to the RTT. In fact, the only one of the three alternatives that is feasible is reduced message sizes. One way to achieve this is through the compression of SIP messages, e.g. by using a compression scheme like SigComp.

2.2 Third Generation Mobile Network Architecture

In this section, the architecture of the third generation mobile network is presented. The Third Generation Partnership Project (3GPP) release 5 network architecture [3GPP TS 23.228, 3GPP TS 23.002] is logically divided into a Core Network (CN) and an Access Network (AN) infrastructure. The CN is logically divided into a Circuit Switched (CS) domain, a Packet Switched (PS) domain and IP Multimedia Subsystem (IMS). The AN is called the UMTS Terrestrial Radio Access Network (UTRAN) and is formed by a hierarchical Radio Network Subsystem (RNS), whose elements are Radio Network Controller (RNC), Node B and User Equipment (UE). A Node B is a logical network component which serves one or more cells. It is the radio transmission/reception unit for communication in the radio cells. A RNC is a network component with the functions for control of one or more Node B elements. It handlers protocol exchanges between UTRAN interfaces. The RNC provides centralised operation and maintenance of the radio network system including access to an operations support system. Among other things, the functions of the RNC include radio resource control, admission control, channel allocation and handover control. The entities specific to the circuit switched domain are Mobile Switching Centre (MSC) and Gateway Mobile Switching Centre (GMSC). The MSC constitutes the interface between the radio system and the fixed networks. The GMSC is an MSC which performs routing to the actual location of the mobile station. The entities specific to the packet switched domain are Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN). The SGSN and GGSN handle packet traffic. The SGSN delivers packets to mobile stations within its service area. It performs mobility management functions such as handing off a roaming subscriber from the equipment in one cell to the equipment in another. The GGSNs are used as interfaces to external IP networks such as the public Internet, other mobile service provider's GPRS services, or enterprise intranets. The GGSNs maintain routing information that is necessary to tunnel protocol data units (PDUs) to the SGSNs that service particular mobile stations. The IMS entities are discussed in Section 2.3. Figure 3 illustrates the architecture of the 3GPP release 5 network.

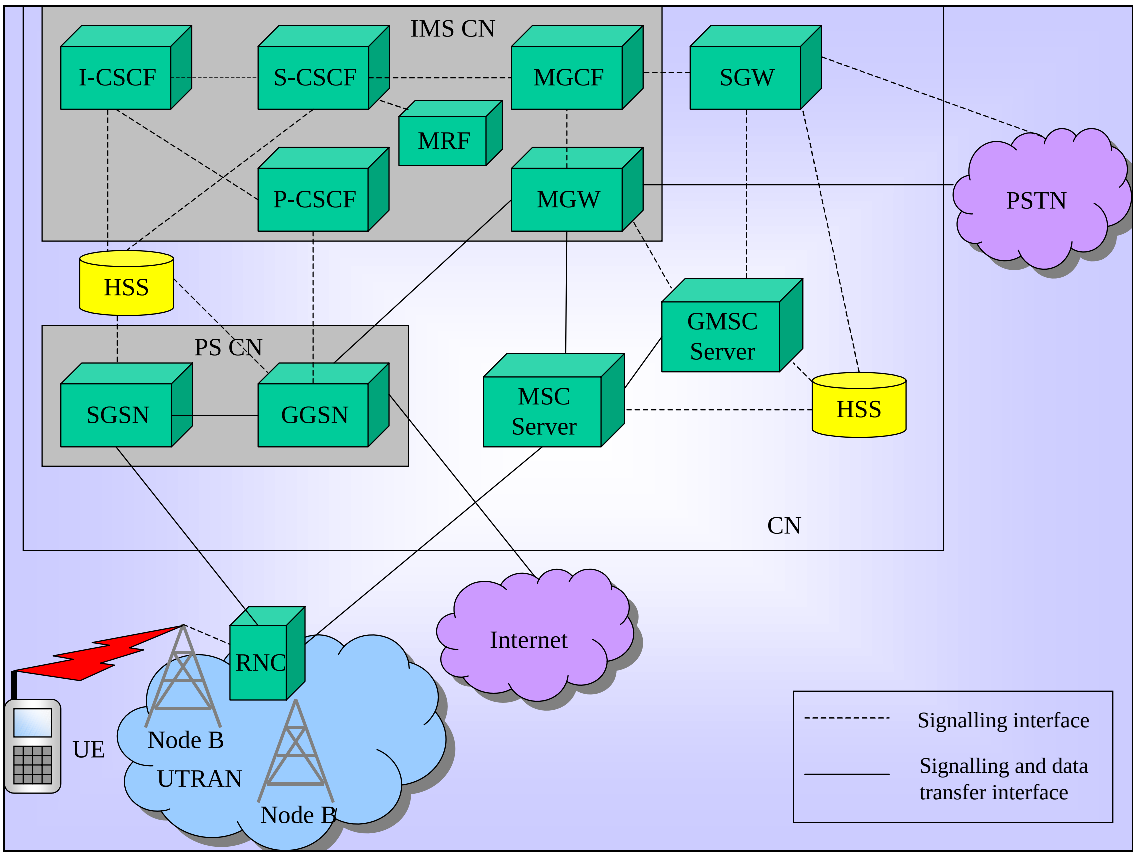


Figure 3 - 3GPP release 5 network architecture

Other new features besides IMS that are included in release 5 include bearer independence, separation of transport and control and Home Subscriber Server (HSS), which contains user and terminal profiles [3GPP TS 23.002].

2.3 IP Multimedia Subsystem

Third Generation Partnership Project (3GPP) release 5 introduces among other things the IP Multimedia Subsystem (IMS) [3GPP TS 23.228, Andreadis 2003, Camarillo 2004]. The IMS aims at combining the latest trends in technology and make the mobile Internet paradigm come true. It is an attempt to create a common platform to develop diverse multimedia services. One aim is also to create a mechanism to boost margins due to extra usage of mobile packet-switched networks. The IMS comprises all core network elements for the provision of IP multimedia (IM) services, for example Call Session Control Function (CSCF) and Media Gateway Control Function (MGCF). When exploring the architecture in the IMS, one should keep in mind that 3GPP does not standardize nodes, but functions. The IMS architecture is a collection of functions linked by standardized interfaces. Implementers are free to combine two functions into a single node or to split a single function into two or more nodes. The IMS is a new core network domain that controls voice and multimedia calls and sessions as well as the interconnection to other networks like PSTN and other UMTS networks. It has a signalling plane and a media plane that traverse different paths. SigComp is a mandatory part of the IMS and it is used to compress SIP signalling traffic.

The IM domain enables cost reductions and introduction of new services, e.g. voice telephony, video telephony, multimedia conferencing, instant messaging and real-time interactive games. IMS should enable the convergence of, and access to, voice, video, messaging, data and web-based technologies for the wireless user, and combine the growth of the Internet with the growth in mobile communications. IMS makes it possible for PLMN operators to offer their subscribers multimedia services based on and built upon Internet applications, services and protocols. It utilises the packet switched domain to transport multimedia signalling and bearer traffic. The packet switched domain maintains the service while the terminal moves and hides the movement from the IMS. IMS is independent of the circuit switched domain. The IM domain enables users and applications to control sessions and calls between multiple parties. It controls and supports network resources to provide the functionality, security and quality required for the calls. The IM domain provides for registration of users so that they can access their own services from any UMTS network. One additional role of the IM is to generate Call Detail Records (CDRs), which contain information on call participants, time, duration and volume of data sent and received. CDRs are used for charging purposes.

IMS entities [3GPP TS 23.228] include CSCF, MGCF, IMS Media Gateway Function (IMS-MGW), Multimedia Resource Function Controller (MRFC), Multimedia Resource Function Processor (MRFP), Subscription Locator Function (SLF), Breakout Gateway Control Function (BGCF) and Application Server (AS). The configuration of IMS entities is shown in Figure 4. In the figure, interfaces supporting user traffic are shown as bold lines and interfaces supporting signalling are drawn as dashed lines.

The roles of IMS entities are described in [3GPP TS 23.228]. The CSCF, which is a SIP server, can act as a Proxy CSCF (P-CSCF), Serving CSCF (S-CSCF) or Interrogating CSCF (I-CSCF). The P-CSCF is the UE’s first contact point for the IMS. The P-CSCF is also of special importance to SigComp, since it is the core network element that has been selected for performing compression and decompression of SigComp messages. For this, the P-CSCF includes a compressor and a decompressor (IMS terminals include both as well). The S-CSCF handles the session states in the network while the role of the I-CSCF is to find the proper S-CSCF for a particular user. The MGCF performs protocol conversion, receives out of band information, communicates with the CSCF, selects the CSCF and controls parts of call state. The IMS-MGW terminates bearer channels from a switched circuit network and media streams from a packet network. It handles media conversion, bearer control and payload processing. The task of the MRFC is to control media stream resources in the MRFP, generate CDRs and interpret information coming from an AS and an S-CSCF and control MRFP accordingly. The MRFP provides resources that are controlled by the MRFC, controls bearers on the Mb reference point shown in Figure 4 and mixes, sources and processes media streams. The SLF provides the name of the HSS containing the required subscriber specific data when requested by the I-CSCF during registration and session setup. It is also queried by the S-CSCF during the registration process. The BGCF selects the network in which PSTN breakout is to occur and chooses the MGCF that is used. The AS can be a SIP Application Server, an Open Service Access (OSA) Application Server or a Customized Application for Mobile Enhanced Logic (CAMEL) IP Multimedia Service Switching Function (IM-SSF). It offers value added IM services. The interface between the S-CSCF and the AS is used to provide services residing in the AS.

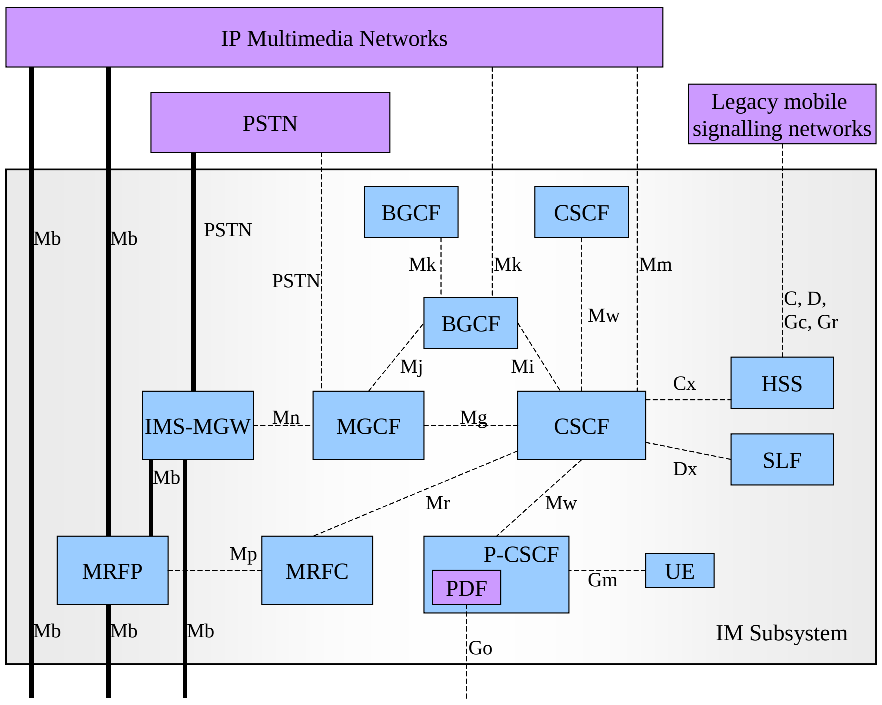


Figure 4 - Configuration of IM subsystem entities [3GPP TS 23.228]

The IP multimedia subsystem attempts to be conformant to IETF Internet standards in order to achieve access independence and to maintain a smooth operation with wireline terminals across the Internet [3GPP TS 23.228]. The signalling protocol that is used for registration and call control in the IM domain is the Session Initiation Protocol (SIP). SIP is the single protocol that is applied between UE and CSCF.

2.4 Location of Signalling Compression Functions

The entity that compresses messages sent to a terminal and decompresses messages received from the terminal is the P-CSCF. This is illustrated in Figure 5, in which a SIP signalling flow from a UE to the S-CSCF is shown. SIP messages that are compressed with SigComp in the UE flow through the radio interface, Base Station (BS) and Radio Network Controller (RNC) of UMTS Terrestrial Radio Access Network (UTRAN). From the UTRAN they traverse through the Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (SGSN) all the way to the P-CSCF, where the SigComp messages are decompressed. From the P-CSCF onwards, the SIP messages are sent uncompressed. The reasons behind selecting the entity performing SigComp compression and decompression from the network core rather than from the radio access network are discussed below [West 2002]. First of all, the location of traffic encryption and decryption functionalities also affects the location of compression functionality, because compression has to be applied out bound from the points of encryption and decryption and it must be transparent. The packet content of some traffic types is authenticated, integrity protected or encrypted. The trusted party that decrypts traffic from and encrypts traffic to a terminal is in the mobile network core. If the endpoint was chosen from the radio access network, network design and performance would suffer from the complexity that would be added by transferring message keys within the mobile network. Another important issue that affects the location of signalling compression is handover. In SigComp, a relatively large amount of historical state is built up to enable efficient compression. If the endpoint performing decompression changed, this state would need to be transferred to the new entity to maintain compression efficiency. This kind of solution would add complexity to the network. When the decompression is performed in the P-CSCF, the decompressing endpoint remains stable for the duration of the application layer session.

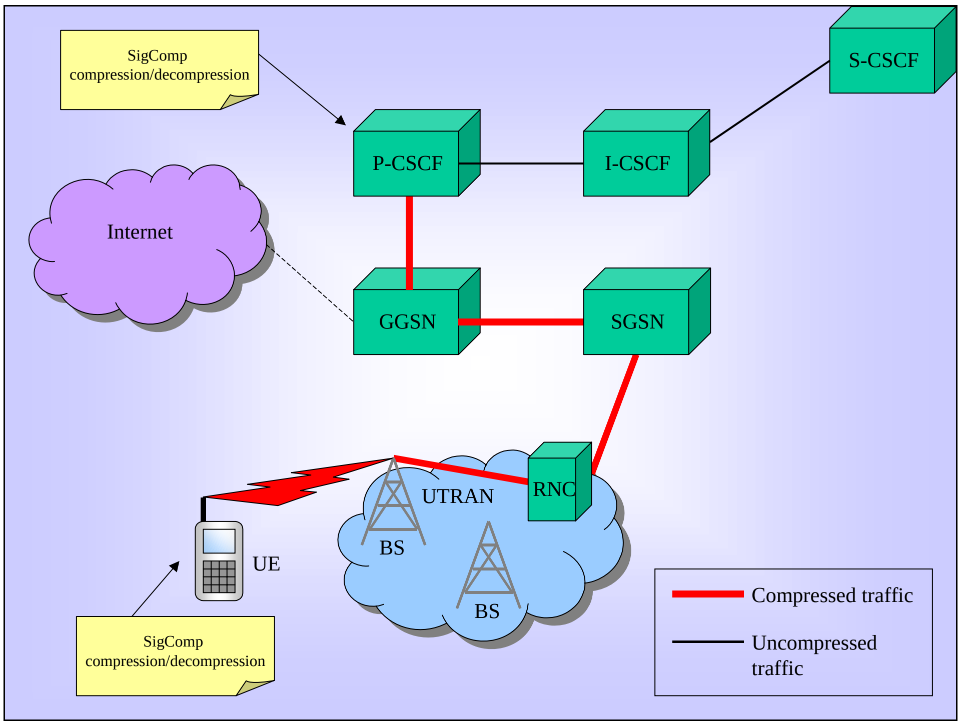


Figure 5 - Location of SigComp functions

Thus, the location of SigComp functions is in the mobile terminal and in the interior of the network, namely in the IMS. This approach contrasts with header compression [RFC 3095], in which the compression functions are located in the terminal and in the radio access network. In the case of SigComp, messages are application level messages that do not contain routing information. They are carried in the payload of transport layer protocols, which in turn leave routing issues to IP. SigComp does not compress the headers of transport layer protocols. Only the entities interested in the content of the transport layer protocol payload, namely the two communicating endpoints, need to decompress SigComp messages.

It should be emphasized that the reason SIP signalling is sent compressed between the terminal and the P-CSCF is not to save a few bytes over the air interface. It is not worth saving a few bytes of signalling when the terminal will be establishing a multimedia session that will use much more bandwidth. The main motivation for compression is to reduce the time required to transmit SIP messages over the air interface.

2.5 Session Initiation Protocol

Perhaps the most important component of the signalling plane is the protocol that performs session control. In the IMS, this is the task of the Session Initiation Protocol (SIP) [RFC 3261]. SIP was originally used to invite users to existing multimedia conferences, but today it is mainly used to create, modify and terminate multimedia sessions. Although SigComp can be used to compress the messages of any text-based protocol, the main focus is currently on the compression of SIP messages.

SIP is independent of the type of multimedia session handled and of the mechanism used to describe the session. The most common format to describe multimedia sessions is the Session Description Protocol (SDP). SDP is simply a textual format that is carried in the body of SIP messages. This is the reason SigComp has to be able to efficiently compress both SIP and SDP. The SIP/SDP static dictionary [RFC 3485] was defined for this purpose.

SIP protocol defines several entities [Camarillo 2002], which are user agents (UAs), redirect servers, proxy servers, registrars and location servers. All 3G terminals supporting 3GPP Release 5 or later releases contain a SIP UA. Also 3GPP2 has adopted SIP. SIP makes use of proxy servers to help route requests to the user’s current location, authenticate and authorise users for services, implement provider call-routing policies, and provide features to users. Redirect servers help in the location of SIP UAs by providing alternative locations where the user can be reachable. A registrar accepts registrations. It is usually co-located with a redirect server or a proxy server. A location server is not a SIP entity, but is an important part of any architecture that uses SIP. Location servers store and return possible locations of users.

SIP is a request/response protocol like the Hypertext Transfer Protocol (HTTP), on which it is based. SIP User Agent Clients (UACs) send requests and User Agent Servers (UASs) return responses. The start line of a request declares a method name, which indicates the purpose of the request. The methods that are currently defined in SIP are shown in Table 4 [Camarillo 2004].

Table 4 - SIP methods [Camarillo 2004]

Method name	Meaning
ACK	Acknowledges the establishment of a session
BYE	Terminates a session
CANCEL	Cancels a pending request
INFO	Transports PSTN telephony signalling
INVITE	Establishes a session
NOTIFY	Notifies the user agent about a particular event
OPTIONS	Queries a server about its capabilities
PRACK	Acknowledges the reception of a provisional response
PUBLISH	Uploads information to a server
REGISTER	Maps a public URI with the current location of the user
SUBSCRIBE	Requests a notification about a particular event
UPDATE	Modifies some characteristics of a session
MESSAGE	Carries an instant message
REFER	Instructs a server to send a request

The start line of a response is referred to as the status line. The status line contains the protocol version and the status of the transaction. The latter is given in a numerical format using a status code and also in a human readable format. Responses are classified by their status codes, which are listed in Table 5. Status codes indicate the status of a transaction.

Table 5 - Status codes of SIP responses

Range	Response class	Example
100-199	Informational	180 Ringing
200-299	Success	200 OK
300-399	Redirection	380 Alternative service
400-499	Client error	401 Unauthorized
500-599	Server error	500 Internal server error
600-699	Global failure	600 Busy everywhere

The exchange of a set of SIP messages between two user agents is referred to as a SIP dialog. In Figure 6, there is an example of a dialog, which is established by an INVITE-200 OK transaction and terminated by a BYE-200 OK transaction. The contents of messages (2) and (3) of the dialog are shown in Figure 7.

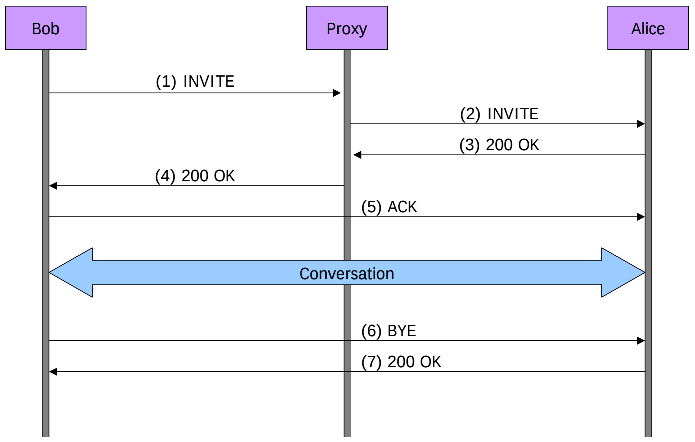


Figure 6 - Session establishment through a proxy

```
INVITE sip:Alice@domain.com SIP/2.0
Via: SIP/2.0/UDP p1.domain.com:5060;branch=xyz
Via: SIP/2.0/UDP c1.domain2.com:5060;branch=abc;
      received=123.0.100.4
Max-Forwards: 69
From: Bob <sip:Bob@domain2.com>;tag=123
To: Alice <sip:Alice@domain.com>
Call-ID: 123456789@123.0.100.4
Cseq: 1 INVITE
Contact: <sip:Bob@123.0.100.4>
Content-Type: application/sdp
Content-Length: 120

v=0
o=Bob 2890844526 2890844526 IN IP4 c1.domain2.com
s=-
c=IN IP4 123.0.100.4
t=0 0
m=audio 20000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

(2) INVITE

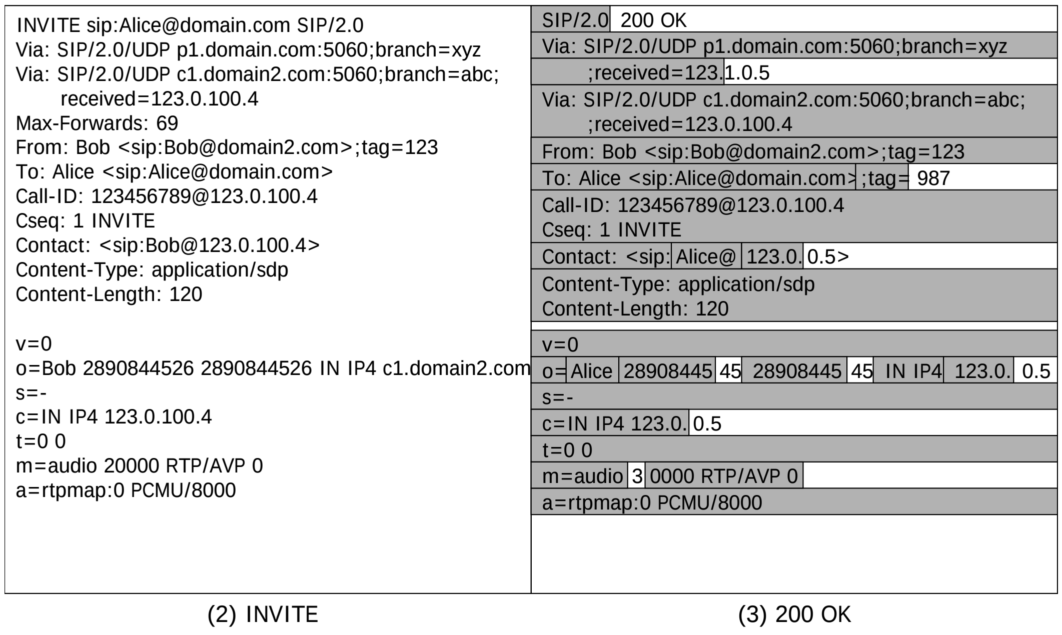
```
SIP/2.0 200 OK
Via: SIP/2.0/UDP p1.domain.com:5060;branch=xyz
      ;received=123.1.0.5
Via: SIP/2.0/UDP c1.domain2.com:5060;branch=abc;
      received=123.0.100.4
From: Bob <sip:Bob@domain2.com>;tag=123
To: Alice <sip:Alice@domain.com>;tag=987
Call-ID: 123456789@123.0.100.4
Cseq: 1 INVITE
Contact: <sip:Alice@123.0.0.5>
Content-Type: application/sdp
Content-Length: 120

v=0
o=Alice 2890844545 2890844545 IN IP4 123.0.0.5
s=-
c=IN IP4 123.0.0.5
t=0 0
m=audio 30000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

(3) 200 OK

Figure 7 - Messages (2) and (3) of the example dialog

From Figure 7, we can observe that most of the rows or substrings of rows in the 200 OK response can also be found from the INVITE request. The matches are shown in Figure 8 using grey background colour. Matches shorter than three characters are not shown. All coloured rows or substrings contain information that was already present in the INVITE message. The purpose of the figure is to show that usually SIP messages belonging to the same dialog contain much information that is present also in other messages of the same dialog. This is good news for a compression scheme like SigComp, since this redundant information can be compressed efficiently.



```
INVITE sip:Alice@domain.com SIP/2.0
Via: SIP/2.0/UDP p1.domain.com:5060;branch=xyz
Via: SIP/2.0/UDP c1.domain2.com:5060;branch=abc;
      received=123.0.100.4
Max-Forwards: 69
From: Bob <sip:Bob@domain2.com>;tag=123
To: Alice <sip:Alice@domain.com>
Call-ID: 123456789@123.0.100.4
Cseq: 1 INVITE
Contact: <sip:Bob@123.0.100.4>
Content-Type: application/sdp
Content-Length: 120

v=0
o=Bob 2890844526 2890844526 IN IP4 c1.domain2.com
s=-
c=IN IP4 123.0.100.4
t=0 0
m=audio 20000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP p1.domain.com:5060;branch=xyz
      ;received=123.1.0.5
Via: SIP/2.0/UDP c1.domain2.com:5060;branch=abc;
      ;received=123.0.100.4
From: Bob <sip:Bob@domain2.com>;tag=123
To: Alice <sip:Alice@domain.com>;tag=987
Call-ID: 123456789@123.0.100.4
Cseq: 1 INVITE
Contact: <sip:Alice@123.0.0.5>
Content-Type: application/sdp
Content-Length: 120

v=0
o=Alice 2890844545 2890844545 IN IP4 123.0.0.5
s=-
c=IN IP4 123.0.0.5
t=0 0
m=audio 30000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

(2) INVITE (3) 200 OK

Figure 8 - Comparison of the content of messages (2) and (3)

2.5.1 Compressibility of Session Initiation Protocol

The average number of binary symbols needed to code the output of a source is called entropy [Sayood 1996]. In case of first-order entropy, nothing is known about the structure of the data and only single characters are being coded, not the whole message. The first-order entropy of a typical SIP message is about 6.7 bits [Fidrich 2003]. This means that the best scheme that could be found to code a SIP message could only code it at 6.7 bits/character. Knowing that the amount of bits that is used to present a character is 8, we can get the achievable compression ratio calculated below:

$$\frac{compressed}{original} = \frac{6.7 bit/char}{8 bit/char} = 0{,}83 .$$

If something about the structure of the data is known, the entropy can be reduced. Better compression ratios can be achieved if dictionaries that make use of probability models and codewords for symbols are used. An example of this approach is the static SIP/SDP dictionary. One further means is to compress messages relative to previously sent messages taking advantage of the redundancy in the contents of the messages. This was illustrated in Figure 8.

3 Signalling Compression

In this chapter, an introduction is given to SigComp, and the central concepts related to the protocol are explained. The chapter begins with a description of SigComp requirements. We continue by going through the architecture, message structure and most important mechanisms of the SigComp protocol. Finally, an example is given on the operation of SigComp.

3.1 Requirements

SigComp requirements, as stated in [RFC 3322], are:

- Transparency
- Header compression coexistence
- Compatibility
- Ubiquity
- Generality
- Support for unidirectional routes
- Operation over both unreliable and reliable transport
- Performance requirements
    - Scalability
    - Must not add delay noticeably
- Robustness
    - Low probability of incorrect decompression caused by errors undetected by lower layers
    - Minimised error propagation
    - Ability to operate under all expected delay conditions
- Compression efficiency
    - Message loss should not affect later messages
    - Toleration of moderate message disordering

Transparency requirement states that when a message is first compressed and then decompressed, the result must be bitwise identical to the original message. SigComp must be able to coexist with header compression, which is likely to be needed together with it to reduce bandwidth usage even further. The compression scheme must be compatible, i.e. it has to allow the upper layer protocols' mechanisms to negotiate whether the compression scheme is used or not. Even if only one of the two communicating entities supports SigComp, the entities have to be able to communicate with each other. SigComp should not require modifications to the protocols generating the messages that are to be compressed. The compression scheme should also be general. It must not be limited to certain protocols, traffic patterns or sessions. SigComp must be able to operate on unidirectional routes without explicit feedback messages from the compressor. It has to work for both reliable and unreliable transport protocols. A primary target for SigComp is cellular systems, where the mobile terminals have varying capabilities. Therefore it must be scalable; it must be flexible enough to accommodate a range of compressor/decompressor pairs with varying processor and memory capabilities. It must not noticeably add to the delay experienced by the end user. The probability that errors, which are not detected by lower layers, cause incorrect decompression should be low. The compression of later messages should not be affected by the loss or damage of earlier messages. SigComp should also be able to operate under all expected delay conditions. It should allow for the correct decompression of moderately disordered messages between compressor and decompressor.

3.2 Architecture

The layout of a SigComp endpoint is illustrated in Figure 9. It includes the following entities: compressor dispatcher, one or more compressors, state handler, Universal Decompressor Virtual Machine (UDVM) and decompressor dispatcher. Each of these entities is described in the following subsections.

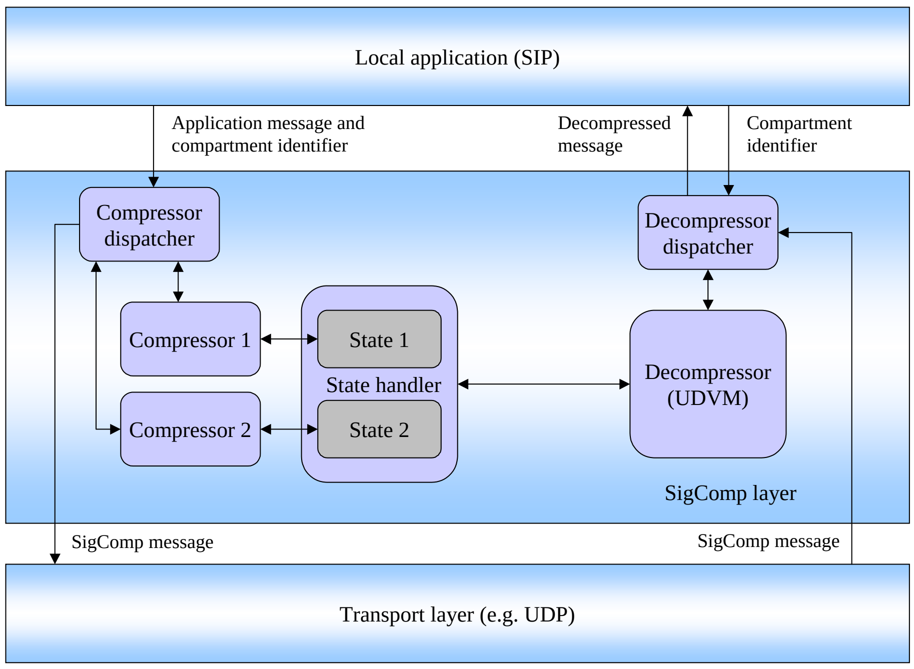


Figure 9 - Architecture of a SigComp endpoint [RFC 3320]

3.2.1 Compressor Dispatcher

The task of the compressor dispatcher [RFC 3320] is to receive messages from the application and pass the compressed version of each message to the transport layer. The application has to provide the compressor dispatcher a compartment identifier together with each message. A compartment is an application specific grouping of messages that relate to a peer endpoint. In case of SIP, a compartment is formed by all messages belonging to a SIP dialog. The compartment identifier uniquely identifies a compartment. SigComp invokes compressors on a per-compartment basis, which means that a compartment identifier can also be used to identify a compressor. For this, a mapping between compartment identifiers and compressors has to be maintained. By providing a compartment identifier together with the application message, the application ensures that the compressor dispatcher can locate an appropriate compressor. Each time a new compartment identifier is encountered, a new compressor is invoked. Once the compressor has compressed the application message, a SigComp header is created and attached to it. After this, the compressor dispatcher can pass the SigComp message to the transport layer. When the application wishes to close a compartment, e.g. after receiving a BYE message and sending the final response, it should indicate this to the compressor dispatcher.

3.2.2 Compressor

The compressor [RFC 3320] implements a certain compression algorithm that is used to compress application messages. One of the fundamental ideas of SigComp is that the standard does not dictate the use of one compression algorithm that should be used by all endpoints. Instead, the choice of the algorithm is left as an implementation decision. What follows is that each endpoint should be able to decompress the output of a variety of compression algorithms. This is made possible by the use of a virtual machine to take care of the decompression functionality. When a compressor creates a SigComp message containing a compressed application message, it includes a decompression algorithm to the header of the message. This decompression algorithm is called the bytecode, and it has been compiled to a form that can be executed on the virtual machine.

A number of requirements are placed on the compressor in [RFC 3320]. First of all, it needs to be transparent, i.e. it must not send bytecode which causes the UDVM to incorrectly decompress a SigComp message. The compressor should supply some form of integrity check over the application message to ensure that successful decompression has occurred. It must ensure that the message can be decompressed using the resources available at the remote endpoint. If the transport is message-based, as it is in the case of User Datagram Protocol (UDP), the compressor must map each application message to exactly one SigComp message. In case the transport is stream-based, but the application defines its own internal message boundaries, the compressor should also map each application message to exactly one SigComp message.

The compressor of the SigComp prototype that we implemented as part of this thesis uses a compression algorithm called Lempel-Ziv-Storer-Szymanski (LZSS). The LZSS algorithm compresses data via textual substitution. It uses an adaptive dictionary, which consists of a portion of the previously encoded sequence. When the input sequence is encoded, the LZSS algorithm performs searches to its adaptive dictionary, trying to find as long substitutions as possible for the *n* next bytes in the portion of the input sequence that has not yet been encoded. The *n* next bytes are called the look-ahead buffer. All matches, i.e. reoccurrences of strings that are already in the dictionary are substituted in the output of the algorithm using offset/length pairs. This is illustrated in Figure 10, in which the adaptive dictionary contains the string *abracad*. The look-ahead buffer contains the four next bytes of the portion of the input sequence that has not yet been encoded, in this case the string *abra*. By comparing the content of the look-ahead buffer to the content of the adaptive dictionary, the LZSS compressor notices that the string *abra* can be found in the adaptive dictionary. Therefore, it can use an offset/length pair to encode the string *abra*. In this case, the offset is seven positions to the left and the length is four.

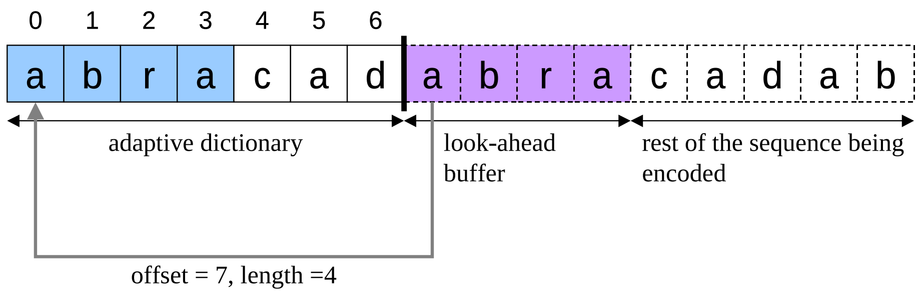


Figure 10 - Operation of the LZSS compression algorithm

3.2.3 Decompressor Dispatcher

The role of the decompressor dispatcher [RFC 3320] is to receive SigComp messages from the transport layer, invoke a new instance of the UDVM to decompress each message, and pass the resulting uncompressed message to the application. Once the application has received the message, it maps the message to a compartment and returns the compartment's identifier to the decompressor dispatcher. The decompressor dispatcher then hands the identifier to the state handler, which uses the identifier to save state information and forward feedback information to an appropriate compressor. By supplying a compartment identifier, the application grants the dispatcher a permission to do this.

3.2.4 Universal Decompressor Virtual Machine

The Universal Decompressor Virtual Machine (UDVM) [RFC 3320] is the central piece of the SigComp architecture. It is the entity that decompresses SigComp messages. The decompression process is carried out by executing a special compiled program called the bytecode on the virtual machine. The UDVM is a virtual machine much like the Java Virtual Machine, but with the difference that it has been optimised for running decompression algorithms. In the case of SigComp, the source code that is compiled to bytecode is called the UDVM assembly and the entity compiling it is called the UDVM interpreter. The bytecode can be thought of as the machine language of the UDVM.

The UDVM provides flexibility when choosing how to compress a given application message: the compressor implementer has the freedom to select an algorithm of his choice. The compressed data is combined with a bytecode containing a set of UDVM instructions. These instructions are carried in the header of the SigComp message and they allow the original data to be extracted at the receiving endpoint.

Because SigComp can run over an unsecured transport layer, a separate instance of the UDVM is invoked on a per-message basis to ensure that damaged messages do not affect the decompression of later messages. However, during the decompression process the UDVM may invoke the state handler to access an existing state. This way the state of the UDVM instance that decompressed the previous message can be restored by a later UDVM instance.

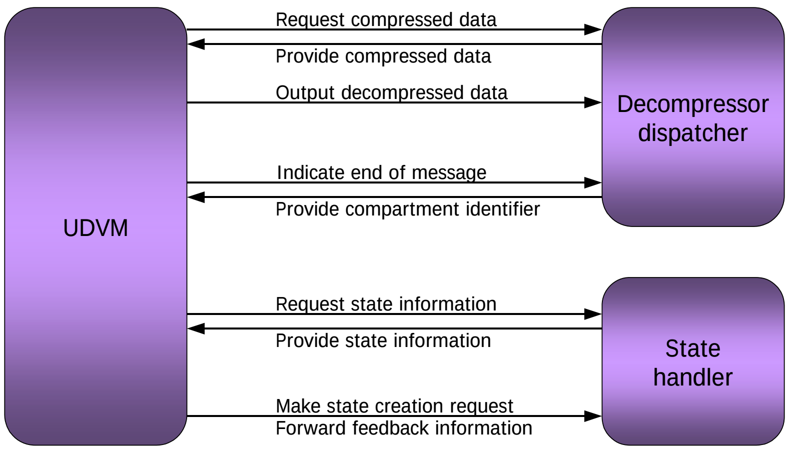

UDVM
Request compressed data
Provide compressed data
Output decompressed data
Indicate end of message
Provide compartment identifier
Decompressor dispatcher
Request state information
Provide state information
Make state creation request
Forward feedback information
State handler

Figure 11 - Interfaces between UDVM and its environment [RFC 3320]

The interfaces between the UDVM and its environment are illustrated in Figure 11. When the UDVM has been initialised, it can receive additional compressed data from the decompressor dispatcher or state information from the state handler only upon request. As the decompression proceeds, the UDVM outputs decompressed data to the decompressor dispatcher. When it encounters the end of a message, it indicates this to the dispatcher, which provides it with a compartment identifier. This identifier is passed to the state handler in a state creation request. The state handler uses the compartment identifier to store the state information in a location in the state memory that is reserved for the corresponding compartment. The UDVM also forwards the feedback information that may be piggybacked to a SigComp message to the state handler.

To ensure that the decompression of a single message cannot consume excessive processing resources, the concept of UDVM cycles in introduced in [RFC 3320]. A UDVM cycle is a measure of the amount of CPU power that is required to execute a UDVM instruction. A UDVM cycle limit is used to restrict the number of UDVM cycles that can be used to decompress each bit in a SigComp message. The amount of cycles a bytecode uses must be monitored because malicious users can send bytecodes containing looping code. However, the cycle limit only reduces the amount of damage that can be caused, but does not remove the problem.

In SigComp, the size of the decompressor memory is negotiable [RFC 3320]. The decompressing side advertises the size of the decompressor memory to the compressing side. The default size is two kilobytes. To improve the efficiency of the compression, a memory size of four or eight kilobytes or even more can be used. The decompressor memory is divided into two sections, the first of which is used to store the decompressed message. The other section is used for the UDVM to hold the bytecode and a circular buffer, which enables the use of states that are larger than the UDVM memory. This is possible because as soon as the buffer fills, the UDVM can start to overwrite content at the beginning of the buffer.

The UDVM instruction set specified in [RFC 3320] contains 36 instructions chosen to support the widest possible range of compression algorithms with the minimum possible overhead. These instructions, their bytecode values and their cost in UDVM cycles are presented in Appendix A.

3.2.5 State Handler

Because a separate instance of the UDVM is invoked to decompress each message that arrives, a way is needed to retain information between messages. This is the task of the SigComp state handler [RFC 3320], which stores information between received SigComp messages. Thanks to the state handler, the compression ratio is improved since messages can be compressed relative to the information contained in previous messages. The state handler makes it possible to create state items for access when a later message is being decompressed. The state items typically contain either a snapshot of a UDVM instance's memory or an uncompressed message.

The state handler manages state memory on a per-compartment basis. As well as storing the state items themselves, it maintains a list of the state items created by a particular compartment and ensures that no compartment exceeds its allocated memory.

3.2.6 UDVM Interpreter

The entity that translates the UDVM instructions and their operands listed in UDVM assembly to the bytecode form is the UDVM interpreter [Draft Price]. The operation of the UDVM interpreter is illustrated in Figure 12. The interpreter takes as an input a file containing UDVM assembly source code and compiles it to a bytecode, which can be executed on the virtual machine.

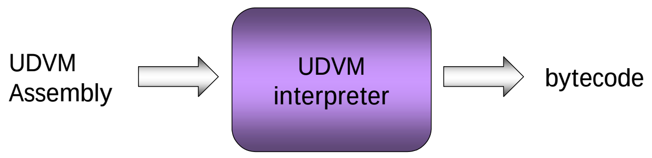


Figure 12 - UDVM interpreter

As an example, let's suppose the following piece of UDVM assembly is provided as an input to the interpreter:

```
:start    pad(8)
MULTILOAD(start, 4, 10, 20, 30, 400)
```

The assembly contains one instruction, MULTILOAD, which simply loads four consecutive two-byte blocks to the location specified by the label *start*, which in this case has been assigned to memory address zero. The output of the interpreter, i.e. the bytecode would look as follows:

```
0x0f 0x00 0x04 0x0a 0x14 0x1e 0xa1 0x90
```

The first byte, 0x0f, contains the operation code of the MULTILOAD instruction and the remaining seven bytes encode the six operands of the instruction. The five first operands can be encoded using only one byte per operand, but two bytes are required to encode the last operand, 400. The operands are encoded using variable length encoding as defined in [RFC 3320].

3.3 Messages

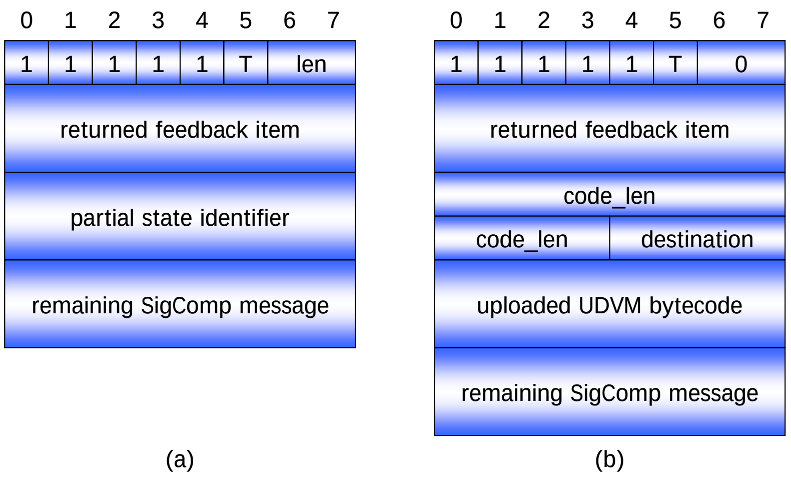


Figure 13 - Format of a SigComp message [RFC 3320]

The format of a SigComp message [RFC 3320] depends on whether it accesses a state item at the receiving endpoint or not. A message that contains a partial state identifier, which is used to load a previously stored state item, is shown in Figure 13 (a). Figure 13 (b) presents a message that does not access a state item, but instead contains the UDVM bytecode needed to decompress the message. An example of a situation in which the bytecode is supplied with the message is when the message is the first message of a compartment, meaning that there are not any stored state items to access yet.

The SigComp header is formed by the fields other than the field *remaining SigComp message*. All SigComp messages contain a special prefix, which does not occur in Unicode Transformation Format 8 (UTF-8) encoded text messages: the five most significant bits of the message are set to 1. The prefix makes it possible to receive uncompressed messages and SigComp messages on the same port. The T-bit of the header is set to 1 whenever the SigComp message contains a returned feedback item. The *len* field of the header determines which fields follow the returned feedback item. If it is non-zero, the message contains a partial state identifier to access a state item at the receiving endpoint. The length of the *partial state identifier* field can be 6, 9 or 12 bytes. If the *len* field is set to 0, then the bytecode needed to decompress the message is supplied as part of the message itself. The *code_len* field specifies the size of the uploaded UDVM bytecode. The *destination* field specifies the starting memory address to which the bytecode is copied. The format of the field *remaining SigComp message* is an implementation decision by the compressor that supplies the UDVM bytecode. It can contain for example a compressed SIP message and a field specifying its length.

3.4 Extended Operations

SigComp extended operations are specified in [RFC 3321]. They can significantly improve the compression efficiency compared to per-message compression, which is the mechanism offered by RFC 3320. Extended operations include the following mechanisms:

- dynamic compression
- shared compression
- maintenance of state data across application sessions
- use of user-specific dictionary
- checkpoint states
- implicit deletion for dictionary update

Dynamic compression, shared compression and user-specific dictionary are discussed below in their own subsections. In the method of maintaining state data across application sessions, the lifetime of a compartment is made longer than the duration of a single application session. Checkpoint state can be used to avoid decompression failure due to reference to a non-existing state. A compressor can indicate that a state is a checkpoint state by setting parameter *state_retention_priority* to the highest value. This parameter is set when a state item is created. In implicit deletion some parts of the dictionary are deleted using a well-defined algorithm, which can be part of the predefined UDVM bytecode. When implicit deletion approach is used, there is no need to signal explicitly which parts of the dictionary need to be deleted on a per-message basis. The content of the dictionary needs to be deleted in order to keep an upper bound on the memory consumption of e.g. in a low-end mobile terminal.

3.4.1 Dynamic Compression

In dynamic compression [RFC 3321], compression is done relative to messages sent prior to the current compressed message. The use of previously sent messages is efficient because the entropy of a message flow is better than the entropy of a single message. Dynamic compression makes use of the similarity of consecutive messages. As an example, let’s suppose endpoints A and B exchange messages using dynamic compression. If both the first and the second messages that A sends to B contain the string “john.doe@domain.com”, there is no need to resend this information in the second message, provided that A knows that B has received and saved the first one. Instead of this string, the output of A’s compressor can contain a pointer, which points to this string in the memory of B’s UDVM. In the dynamic compression approach, information from previously decompressed messages is maintained as a dictionary in the memory of the UDVM. After a message has been decompressed, the contents of the UDVM’s memory are saved. This UDVM memory snapshot is then retrieved when a new message is to be decompressed, and the memory of the new UDVM instance is initialised using it.

In order to be able to utilise information from previously sent messages, the compressor has to gain knowledge about the reception of these messages. In case of unreliable transport, the SigComp feedback mechanism can be used to provide a means for a SigComp endpoint to confirm which states it has established during the lifetime of a compartment. When a reliable transport layer protocol such as TCP is used, explicit acknowledgements are not necessary.

3.4.2 Shared Compression

In shared compression [RFC 3321], messages are compressed relative to messages received by the endpoint prior to the current compressed message. The compressing endpoint saves the uncompressed version of the compressed message as a state. In addition to sent and acknowledged messages, also received messages are used to update the dictionary and to compress new messages. It is efficient to use received messages because acknowledgements are not needed. Instead of acknowledging a state item, endpoint A signals to endpoint B that it has saved the uncompressed version of message X it just sent by setting a special bit on the SigComp header. When endpoint B checks the bit, it immediately knows that the dictionary entry corresponding to message X is available at endpoint A. Therefore, endpoint B can compress the next message it sends relative to message X. Shared states are saved in the same memory as the normal states created by the particular remote compressor.

3.4.3 User-specific Dictionary

The idea behind the use of a user-specific dictionary [RFC 3321] is that for protocols such as SIP, a given user and device combination produces some messages containing fields that are always populated with the same data. For example, the capabilities of SIP endpoints tend not to change unless the capabilities of the devices change. Also the user’s name, email address and Uniform Resource Locator (URL) constitute information that does not change frequently. When this approach is used, the SigComp compressor includes the user-specific dictionary to the initial message that is sent to the remote decompressor. This increases the compression efficiency once the messages start to flow.

3.4.4 Impacts on SigComp Messages

[RFC 3321] suggests a format for SigComp messages carrying information required by SigComp extended operations. To support dynamic and shared compressions, the SigComp messages need to convey additional information: acknowledged state identifiers and shared state identifiers. There is no need to specify a message format to carry the information necessary for the extended features, because the format of the field *remaining SigComp message* is an implementation choice by the compressor which supplies the UDVM bytecode. An example of what the *remaining SigComp message* field with support for shared compression and dynamic compression could look like is illustrated in Figure 14 [RFC 3321].

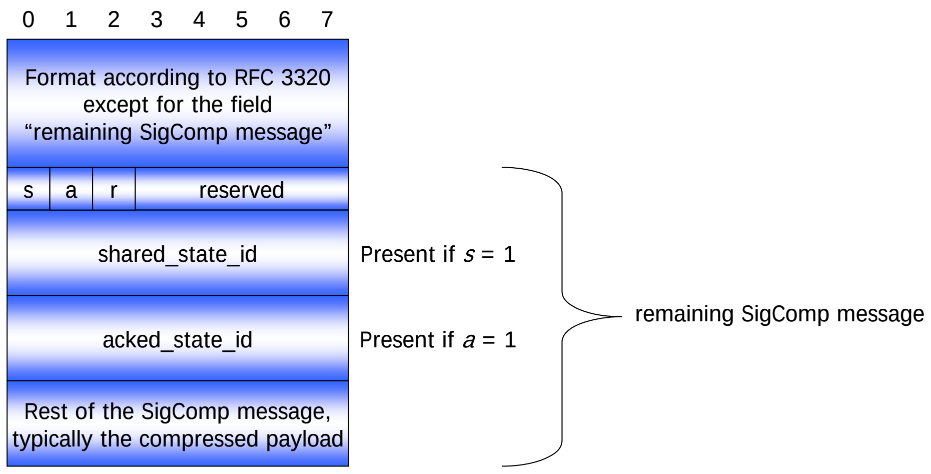

0 1 2 3 4 5 6 7
Format according to RFC 3320 except for the field "remaining SigComp message"
s
a
r
reserved
shared_state_id
Present if s = 1
acked_state_id
Present if a = 1
Rest of the SigComp message, typically the compressed payload
remaining SigComp message

Figure 14 - SigComp extended operations message format [RFC 3321]

If the *s* bit of the message is set, the message contains a *shared_state_id* field. If the *a* bit is set, the message contains an *acked_state_id* field. If the *r* bit is set, a state corresponding to the decompressed version of the compressed message was saved at the compressor. The lengths of the *shared_state_id* and the *acked_state_id* fields are the same as in the case of the partial state identifier, i.e. 6, 9 or 12 bytes depending on the length of the partial state identifier.

3.5 Feedback Mechanism

SigComp feedback mechanism is specified in [RFC 3320]. If SigComp endpoints both send and receive SigComp messages and there is a one-to-one relationship between the compartments exchanged between them, it is possible to send feedback information that monitors the behaviour of an endpoint and helps to improve the efficiency of the compression. Two types of feedback data exist: requested feedback data and returned feedback data.

Feedback is done on a request/response basis: a compressor makes a feedback request and receives some feedback in return from the remote endpoint. Requests and responses are always piggybacked to SigComp messages carrying compressed data. The feedback data is retained between SigComp messages and is considered to be part of the overall state. It cannot be forwarded if not accompanied by a valid compartment identifier. Size of the returned feedback item is 1-128 bytes.

By using the feedback mechanism, the receiving endpoint is informed about the capabilities of the sending endpoint and additional resources available can be advertised. Remote endpoints can also indicate their interest in receiving a list of some of the state items available locally at an endpoint. Thanks to the feedback mechanism, it is possible for a compressor to check that the state item it wants to access is not rejected because there is not enough state memory available at the remote endpoint. This is done by checking a special *state_memory_size* parameter. Successful decompression can also be acknowledged in case of unreliable transport such as UDP. This needs to be done, because when unreliable transport is used, messages can be lost or disordered on the path between the compressor and a remote decompressor.

3.6 Negative Acknowledgement Mechanism

A negative acknowledgement mechanism for SigComp is described in [Draft Roach]. It allows the reporting of precise error information upon reception of a message that cannot be decompressed. The negative feedback can be used by the endpoint that originally sent the message to make adjustments to the compressed message before retransmitting it. The negative acknowledgement mechanism is needed, because there are situations in which a sender's view of a shared state differs from the receiver's view. Examples of such situations are discarding of compartments without explicit signalling in case of client failures, loss of connectivity, mobile terminal restarts and server failover. The only solution the basic SigComp offers to these situations is to signal that all states have been lost. Thus, even though only one state is corrupted or missing, all states belonging to a compartment are erased. In addition, this requires a message in the reverse direction that the application will authorize.

SigComp implementations that use the negative acknowledgement (NACK) mechanism need to calculate and store a hash value for each SigComp message they send. When a SigComp message that is received causes a decompression failure, the recipient forms and sends a SigComp NACK message. The message contains a hash of the message that could not be decompressed, the exact reason why the decompression failed and any additional details that might assist the NACK recipient to correct the problems. Figure 15 shows the format of a SigComp NACK message. Only the content of the fields that are new compared to those of Figure 13 are described here. *Version* gives the version of the NACK mechanism being used. *Reason code* is a one-byte value that indicates the nature of the decompression failure. *OPCODE of failed instruction* is a one-byte value that includes the operation code to which the Program Counter (PC) of the UDVM was pointing when the failure occurred. *PC of failed instruction* is a two-byte field containing the value of the program counter when failure occurred. *Hash of failed message* is simply a hash of the message that could not be decompressed. *Error Details* provides any additional information that might be useful in correcting the problem that caused the decompression failure.

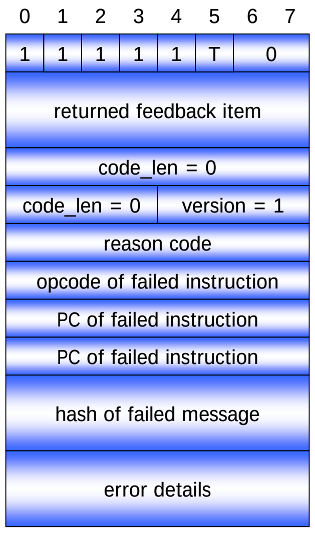


Figure 15 - SigComp NACK message format [Draft Roach]

When a NACK message is received, the receiver performs a search using the hash contained in the message. Then, if unreliable transport is used, SigComp uses the information included in the message to make adjustments to the compressor associated with the compartment in question. The next transmission of a message that the application makes will take advantage of the adjustments. When NACK is received for a message that was sent over reliable transport, the SigComp layer must indicate the error to the application. The application, e.g. a SIP application, should react in the same way as it does for any other transport layer error.

3.7 SigComp Operation

An example is given on the process of sending and receiving a SigComp message in Figure 16. It is assumed that state information has already been saved at both endpoints, i.e. messages have already been exchanged between them. Two communicating endpoints, A and B, are shown. The sequence of interactions between different SigComp entities is explained below.

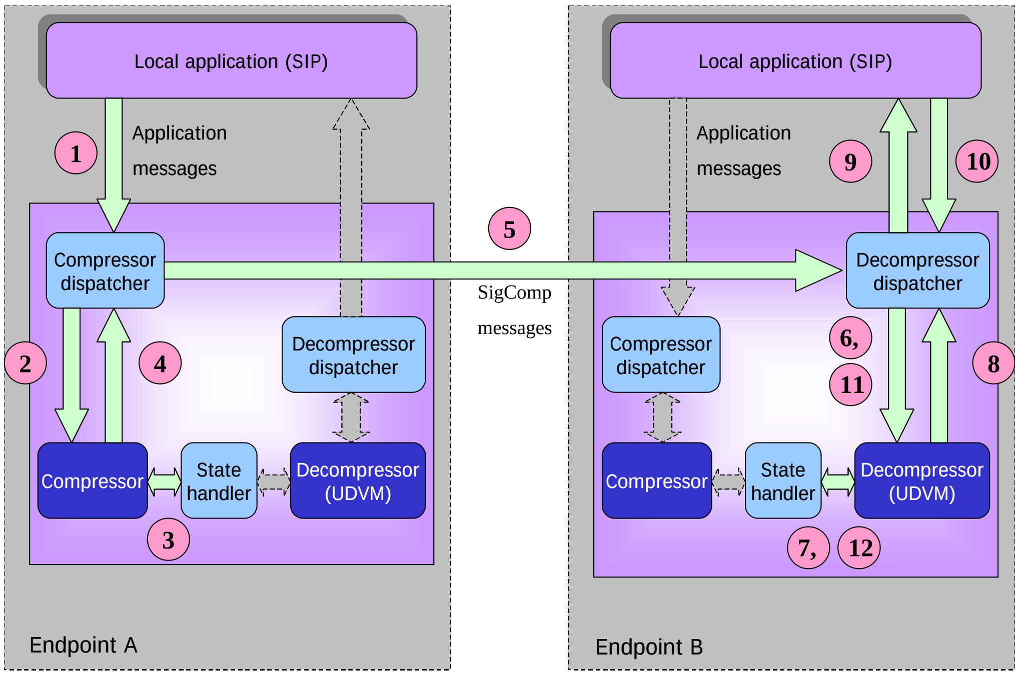


Figure 16 - SigComp operation

(1) The local application at endpoint A sends a SIP message together with a compartment identifier to the compressor dispatcher. Also an IP address and a port number of the destination need to be supplied.

(2) The compressor dispatcher figures out the compressor that is associated with this compartment and forwards the SIP message, the destination address and port number to it.

(3) The compressor requests state information from the state handler. This state information can contain for example a shared state, provided that SigComp extended operations are used. The compressor uses the loaded state information to compress the message, creates a SigComp message header and puts the output from the compression process to the payload of the SigComp message. The compressor may also save state to store the uncompressed version of the message that is being processed. This will later become the next

shared state.

(4) The compressor hands the SigComp message that was created in the previous step to the compressor dispatcher.

(5) The compressor dispatcher passes the SigComp message to the transport layer together with the IP address and the port number that were received from the application. At endpoint B, SigComp decompressor dispatcher is invoked when the UDP datagram containing the SigComp message arrives to the port the decompressor dispatcher has been assigned. The decompressor dispatcher inspects the five first bits of the UDP payload and concludes that the payload contains a SigComp message.

(6) The decompressor dispatcher reads the fields of the SigComp header. It creates a new UDVM and initialises the memory of the UDVM using a previously saved state, the state identifier of which was included in the header. The bytecode that is needed to decompress the payload of the SigComp message is a part of this state item. Next, the UDVM starts executing the bytecode.

(7) While executing the bytecode, the UDVM retrieves the shared state item that endpoint A used in the compression process from the state handler.

(8) When it has finished decompressing the message, the UDVM sends the decompressed SIP message to the decompressor dispatcher.

(9) The decompressor dispatcher forwards the SIP message to the SIP application.

(10) The SIP application maps the message to a certain compartment and returns the corresponding compartment identifier to the decompressor dispatcher.

(11) The decompressor dispatcher forwards the compartment identifier to the UDVM.

(12) The UDVM hands the compartment identifier to the state handler. Now the state handler can save state, i.e. store the contents of the UDVM’s memory, the shared state and the returned feedback item that the SigComp message possibly contained. These are saved in the state memory reserved for the compartment of the message.

4 Applying Signalling Compression to the Session Initiation Protocol

In this chapter, the way SigComp is applied to the session initiation protocol is explained. The chapter begins by listing the requirements SigComp places on SIP. Also the mechanism that is used to signal that compression is required is presented. Finally, the static SIP/SDP dictionary that can be used to improve the efficiency of SigComp compression is described.

4.1 Requirements of Signalling Compression on the Session Initiation Protocol

A compression scheme like SigComp cannot be implemented without support from the application level protocols using it. [RFC 3320] lists a number of requirements on the applications using SigComp. SigComp requirements especially on SIP are discussed in [RFC 3486].

First of all, the negotiation of whether to use compression or not must be handled by the SIP application. When receiving messages, both SigComp messages and uncompressed SIP messages are first inspected by the SigComp layer. If the first five bits of the first byte of a transport protocol message payload are '11111', the message is identified to be a SigComp message. If another bit pattern is encountered, the message is considered to be a SIP message and is immediately forwarded to the SIP application. This makes it possible to multiplex compressed and uncompressed messages on the same port.

It is not enough for the SIP application to hand only the message to be compressed to the SigComp layer. In addition, a compartment identifier, destination IP address and a port number need to be supplied at a minimum. To be able to provide a compartment identifier, the SIP application should handle the mapping between a SIP dialog and a pair of peer SigComp compartments. When the application receives a decompressed message from the SigComp layer, it maps the message to a certain compartment and returns the compartment's identifier. Distinct compartments must be assigned to distinct endpoints. The application should also use an authentication mechanism to securely map decompressed messages to compartment identifiers. When the application wishes to close a particular compartment, it should indicate this to the compressor dispatcher, so that the resources taken by the compartment can be reclaimed.

Applications should agree on any limits to the lifetime of a compartment in order to avoid the case in which an endpoint accesses state information that has already been deleted. It should also be kept in mind that not all endpoints will understand SigComp. If a server, which does not support SigComp, receives a compressed message, it has no means to indicate this to the client. Thus, if a SIP client has initiated a transaction by sending a compressed request, and the client does not receive a response during the transaction timeout period, the client should resend the same request uncompressed.

A SIP user agent needs a way to declare that it wishes to receive incoming requests compressed. On the other hand, it must also be able to send requests, preferably even the initial INVITEs, compressed. Mechanisms for this are discussed in section 4.2.

In order to avoid asymmetric compression, proxies need to rewrite their record-route entries in the responses. The Record-Route header field is inserted by proxies in a request to force future requests in the dialog to be routed through the proxy. If the URI of the next upstream hop in the route set contains the parameter *comp=sigcomp*, which indicates SigComp compression, the proxy should add the same parameter to its entry. If the URI does not contain the parameter, the proxy should remove the *comp=sigcomp* parameter from its entry in the Record-Route header field. Also the user agent servers need to observe the presence of the *comp=sigcomp* parameter. If the URI of the next upstream hop in the route set contains the *comp=sigcomp* parameter, the UAS should add the same parameter to the contact header field of the response.

4.2 A Mechanism to Signal That Compression Is Required

A mechanism for signalling that SigComp compression is required is described in [RFC 3486]. In SIP, clients send requests to the host part of a URI. Servers send responses to the host specified in the sent-by parameter of the Via header field. To signal that a SIP message needs to be compressed, a *comp=sigcomp* parameter is used in URIs if a request is to be compressed or in Via entries if a response is to be compressed. An example of a URI containing this parameter is as follows:

```
          sip:bob@hut.fi;comp=sigcomp
```

This indicates that the request has to be compressed using SigComp. An example of a Via header field indicating that the SIP entity is willing to receive compressed messages is presented below:

```
          Via:SIP/2.0/UDP
computer.hut.fi:5060;branch=XXX;comp=sigcomp
```

Clients other than proxies add the parameter *comp=sigcomp* to the URI in the contact header field, whereas proxies add the parameter to their URI in the record-route header field. Figure 17 shows the way the *comp=sigcomp* parameter can be used to signal SIP traffic compression between a user agent and a proxy. In the figure, the Route header field in the messages that the User Agent Client (UAC) sends indicates that the request (INVITE and ACK) needs to be compressed. Via header field in the messages the proxy sends indicates that the response (200 OK and 180 ringing) needs to be compressed. All SIP messages between the UAC and the proxy are sent compressed in Figure 17.

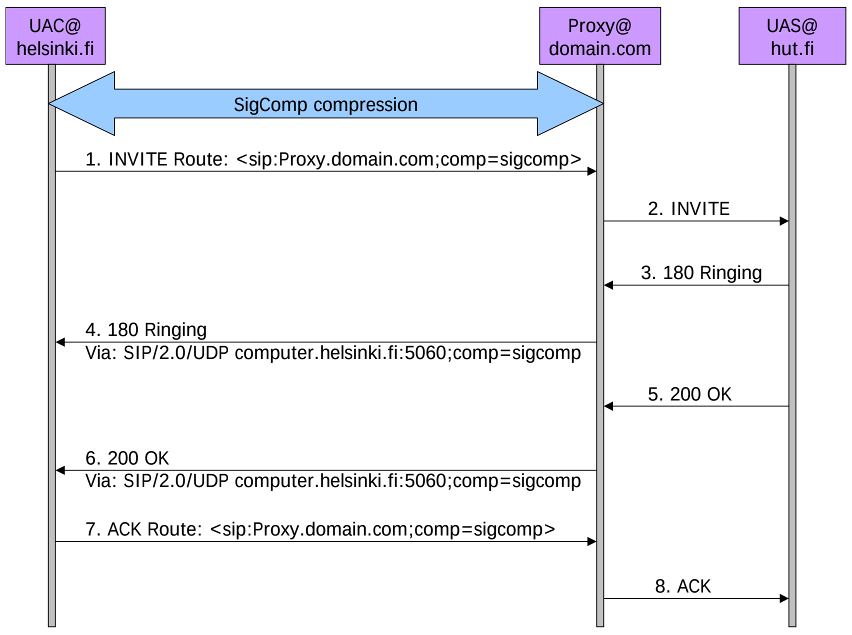

UAC@
helsinki.fi
Proxy@
domain.com
UAS@
hut.fi
SigComp compression
1. INVITE Route: <sip:Proxy.domain.com;comp=sigcomp>
2. INVITE
3. 180 Ringing
4. 180 Ringing
Via: SIP/2.0/UDP computer.helsinki.fi:5060;comp=sigcomp
5. 200 OK
6. 200 OK
Via: SIP/2.0/UDP computer.helsinki.fi:5060;comp=sigcomp
7. ACK Route: <sip:Proxy.domain.com;comp=sigcomp>
8. ACK

Figure 17 - The comp=sigcomp parameter

An example of the format of the INVITE message, which is the first message sent in Figure 17, is presented below. The SDP content is not shown.

```
INVITE sip:UAS@hut.fi SIP/2.0
Via: SIP/2.0/UDP computer.helsinki.fi:5060;comp=sigcomp
Route: <sip:Proxy.domain.com;comp=sigcomp>
From: UAC <sip:UAC@helsinki.fi>
To: UAS <sip:UAS@hut.fi>
Call-ID: 123456789@computer.helsinki.fi
CSeq: 1 INVITE
Contact: UAC <sip:UAC@computer.helsinki.fi;comp=sigcomp>
Content-Type: application/sdp
Content-Length: 200
```

It is assumed that the UAC is configured to send compressed traffic to the Proxy, which is the reason it sends the INVITE compressed. The UAC adds the parameter *comp=sigcomp* to the Via and the Contact header fields so that it can receive future requests and responses compressed. An example of the format of the first response sent from the Proxy to the UAC, the 180 Ringing, is shown below.

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP computer.helsinki.fi:5060;comp=sigcomp
Record-Route: Proxy.domain.com;comp=sigcomp
From: UAS <sip:UAS@helsinki.fi>
To: UAC <sip:UAC@hut.fi>;tag=543210
Call-ID: 123456789@computer.helsinki.fi
CSeq: 1 INVITE
Contact: UAS <sip:UAC@hut.fi>
```

The *comp=sigcomp* parameter in the Via header field of the 180 Ringing message indicates that the message should be sent compressed to the UAC. It is assumed that the Proxy wants to remain in the signalling path. This is the reason the message contains the Record-Route field, which was originally added to the INVITE sent from the Proxy to the UAS in step (2). When the Proxy sends the 180 Ringing response to the UAC, it assumes that because the UAC wants to receive compressed requests, it would also like to send compressed requests. Therefore the proxy adds the *comp=sigcomp* parameter to its entry in the Record-Route field.

Receiving incoming requests, even initial INVITEs compressed is not a problem, since user agents can register a SIP URI with the *comp=sigcomp* parameter in their registrar [RFC 3486]. All incoming requests for the user will be sent to this SIP URI using compression. Sending of compressed messages is slightly more complicated. It is, of course, easy for the client to get a next-hop URI with the *comp=sigcomp* parameter from a record-route header field or contact header field, but in this case the client has to wait until it receives a response from the user agent server. To send the initial INVITE compressed, the client needs to get a *comp=sigcomp* URI from its outbound proxy before it decides to establish a session. To do this, the client can send an uncompressed OPTIONS request to its outbound proxy [RFC 3486]. The proxy can then provide an alternative URI with the *comp=sigcomp* parameter to the client. A client should never send a compressed request to a server if it does not know whether or not the server supports SigComp.

4.3 The Static Session Initiation Protocol and Session Description Protocol Dictionary

[RFC 3485] defines the Session Initiation Protocol (SIP) and Session Description Protocol (SDP) specific static dictionary, which can be used to achieve higher compression efficiency. The dictionary is compression algorithm independent and must be available in all SigComp implementations for SIP/SDP. It is static, i.e. it will stay as it is forever, even though some minor errors in the dictionary have been reported [Draft Surtees]. In general, the use of a static dictionary technique like the static SIP/SDIP dictionary is most appropriate when considerable prior knowledge about the source is available. When compressing SIP, it is known ahead of time that certain words such as Via, From, To or Contact are going to appear in almost all of the messages.

When SIP/SDP messages are compressed, the first few messages are only partially compressed because there are not previous states to compress against. The compression becomes more efficient only after a few messages have been exchanged. For example, the INVITE request, which is the first message sent, cannot take advantage of any previously built dictionaries. This problem is reduced by the static SIP/SDP dictionary [RFC 3485], which constitutes a SigComp state that can be referenced already in the first SIP message that a compressor sends out. This is possible because both endpoints are guaranteed to have the same version of the dictionary. The use of the dictionary enables more efficient compression of the first messages.

The static SIP/SDP dictionary is a collection of well-known strings that appear in most SIP and SDP messages. Table 6 shows examples of the input strings that have been used in generating the library. Also their priorities are shown. The examples are taken from the complete list of SIP/SDP strings presented in [RFC 3485] that were used when building the dictionary. A low number in the priority field indicates that the string can be found with high probability in a SIP message. The dictionary that is actually included in the SigComp implementation is in binary form.

Table 6 - The static SIP/SDP dictionary

String	Priority
“sip:”	1
“To: “	1
“tel:”	3
“SIP/2.0”	1
“SIP/2.0/UDP “	1
“terminated”	4
“INVITE”	1
“ Dec “	4
“ACK”	1
“Via: “	1
“OPTIONS”	4
“algorithm=”	2
“BYE”	2
“Record-Route: “	2
“CANCEL”	4
“100 Trying”	2
“REGISTER”	2
“180 Ringing”	2
“INFO”	4
“Contact: “	5

The binary SigComp dictionary [RFC 3485] is comprised of two parts: a string subset and a table subset. The string subset contains all strings in the contributing collections as a substring. There are two collections: the first is a collection of strings that SIP contributed to the dictionary and the second one a collection of strings that SDP contributed to the dictionary. The table subset contains pairs of length and offset values for all the strings in the contributing collections. All compression algorithms are able to use the string subset and some compression methods can also use the table subset. The idea is that a compressor can choose to reference either a string in the string subset or an entry in the table subset. The compressor that is used in the SigComp prototype described in this thesis always references entries in the string subset.

5 Computer and Operating System Architectures

In order to better understand the results of the performance measurements presented in this thesis, it is good to know something about the way a computer is organised. In this chapter, the memory hierarchy of a computer system is described. Because the SigComp prototype presented in this thesis is implemented as a multithreaded application, also an introduction to multithreaded programming is given.

5.1 Memory Hierarchy

In an ideal situation, a computer would have a memory with an unlimited size and extremely fast access times. However, since no current technology satisfies these goals, a computer system can only provide an illusion of a large memory than can be accessed as fast as a very small memory. This illusion is provided by organising the memory into a memory hierarchy, which takes advantage of the principle of locality. The principle of locality [Patterson 1997] states that programs access a relatively small portion of their address space at any instant of time. The memory hierarchy of a computer is depicted in Figure 18.

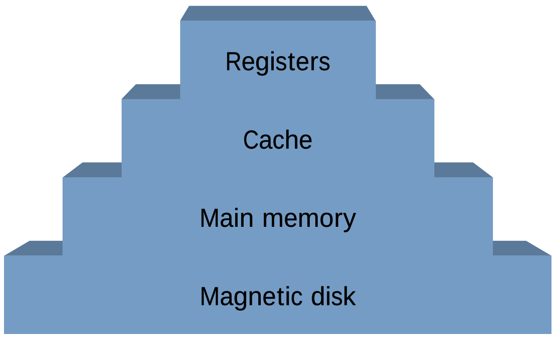


Figure 18 - Memory hierarchy

A memory hierarchy [Patterson 1997, Tanenbaum 2001] consists of multiple levels of memory with different speeds and sizes. On the top of the hierarchy are registers that are internal to the Central Processor Unit (CPU). They are just as fast as the CPU and can be accessed with no delay. The next level in the hierarchy is the cache memory, which uses static random access memory (SRAM) with an access time of about 5-25 ns. The main memory is divided up into cache lines, the most heavily used of which are kept in a high-speed cache that is located inside or very close to the CPU. There can be two or even three levels of cache, each one slower and bigger than the one before it. If a program wants to read a memory word and the needed line happens to be in the cache, the situation is called a cache hit. This means that the main memory does not need to be accessed and the word is returned to the program with only a little delay. On the other hand, if the needed line is not found in the cache, a cache miss occurs. This means that the main memory must be accessed with a substantial time penalty. The main memory [Patterson 1997] is implemented from dynamic random access memory (DRAM). DRAM is cheaper per bit than SRAM, but it is also substantially slower, having an access time of about 60-120 ns. All CPU requests that cannot be satisfied out of the cache go to main memory.

The use of virtual memory [Patterson 1997, Tanenbaum 2001] means that the combined size of the program, data and stack may exceed the amount of physical memory available for it. The parts of the program that are in use are kept in the main memory, while the rest are kept on disk, which is on the bottom of the memory hierarchy. In a multiprogramming system, it is sufficient that the main memory contains the active portions of the programs in execution. A virtual memory block is called a page, and a virtual memory miss is called a page fault. It takes millions of CPU cycles to process a page fault since the missing page has to be brought in from the disk. This may take even 10-20 million nanoseconds. Therefore, the miss penalty can be considered enormous.

5.2 Multithreaded and Parallel Programming

5.2.1 Processes and Threads

A process [Tanenbaum 2001] is an executing program, including the current values of the program counter, registers and variables. A process is essentially a way to group related resources together; this makes their management easier. In the process model, all runnable software on the computer is organised into a number of sequential processes. If there are multiple processes running on the same Central Processor Unit (CPU), the CPU switches back and forth from process to process, meaning that the processes take turns executing on the CPU. A process has also a thread of execution. While processes are used to group resources together, threads are the entities that are scheduled for execution on the CPU. A thread has a program counter keeping track of which instruction to execute next, registers to holds its current working variables and a stack containing the execution history. In multithreading [Tanenbaum 2001], multiple threads are allowed in the same process. The threads share the address space and resources of the process. When a multithreaded process is executed on a single-CPU system, its threads take turns running, in a similar way as in the case that there are multiple processes running on the same CPU. If there are for instance ten compute-bound threads in a process, the threads appear to be running in parallel, because the CPU switches rapidly back and forth among the threads. However, each thread gets only one-tenth of the speed of the CPU.

Threads are useful, because they make the programming model simpler. In the thread model, parallel entities are allowed to share an address space and all of its data. Threads are also easier to create and destroy than processes. Having multiple threads is most useful when both substantial computing and substantial Input/Output (I/O) are present. The use of multiple threads allows overlapping of activities, thus speeding up the application. This is because when one thread becomes blocked while waiting for an I/O event to occur, CPU time can be allocated to another thread performing computing work. If the system had only one thread, the CPU would be idle for the duration of the I/O event. On the other hand, when all threads do heavy computing on a single-CPU system, having multiple threads yields no performance gain.

Threads are also useful on systems with multiple CPUs. In these systems, real parallelism is possible; if there are for instance two threads and two CPUs, both threads can execute simultaneously on their own CPUs.

5.2.2 Processors and Multiprocessors

A single-CPU system executes multiple threads by switching between them. Rapid switching provides the illusion that the threads are running in parallel. There are many approaches to multithreading. In time-slice multithreading [Marr 2002], the processor switches between software threads after a fixed time period. In switch-on-event multithreading, the threads are switched on long latency events such as cache misses. In simultaneous multi-threading [Marr 2002], multiple threads can execute on a single processor without switching. The threads execute simultaneously and make better use of resources than in the case of time-slice or switch-on-event multithreading. For instance, Intel’s Hyper-Threading Technology [Marr 2002] uses the simultaneous multi-threading approach. The Hyper-Threading Technology makes a single physical processor appear as multiple logical processors. In a Hyper-Threading processor, there is one copy of the architecture state for each logical processor, and the logical processors share a single set of physical resources.

True multiprocessor systems have two or more CPUs sharing the same physical memory. Adding more processors allows applications to get performance improvements, because multiple threads can be executed on multiple processors at the same time.

6 Previous Research on Signalling Compression

In this chapter, some previous research on SigComp is presented. Because SigComp is a fairly new feature, manufacturers are still in the middle of the process of implementing the protocol. Therefore, the research that has been published so far has mainly focused on the estimated performance of the protocol. At least to the author's best knowledge, this thesis is the first to study the performance of a full-blown SigComp implementation and the load it places on the core network element performing compression and decompression of SIP messages. SigComp architecture plans have neither been published before.

The compression ratios achieved by SigComp extended operations are studied in [Nordberg 2003]. The compression ratios were measured for a compression algorithm called LZBS. Compression and decompression times or the overhead added by the extended mechanisms were not measured. In addition, some simplifying assumptions were made and therefore the results obtained do not reflect the performance of a complete SigComp implementation. A maximum compression ratio (size compressed/size uncompressed) of 22.7% was achieved for the entire call setup sequence using a buffer size of 4096 bytes and a ratio of 35.7% using a buffer size of 2048 bytes. This thesis studies the effects of using larger buffer sizes, namely 8192 and 16384 bytes in addition to the smaller buffer sizes. The use of a buffer size of 2048 bytes is not feasible in most cases, because the buffer is too small to hold the static SIP/SDP dictionary, let alone the biggest SIP messages. In fact, it makes it completely impossible to send certain messages compressed [Draft Surtees]. A buffer of size 4096 is able to hold the string subset of the static SIP/SDP dictionary, but parts of the dictionary typically need to be overwritten already when the first message is decompressed.

Fidrich, Bilicki, So'gor and Sey [Fidrich 2003] studied the compression ratios, compression times and decompression times of some well-known algorithms. These algorithms include Deflate, Subexponential encoder, Synth, LZ77 and Huffman coding. The algorithms supported only basic compression together with a static dictionary, which was not the same one as the static SIP/SDP dictionary defined in [RFC 3485]. For these two reasons, the resulting compression ratios are quite modest, about 46% in the case of the best-performing algorithm. The compression and decompression times were estimated using a theoretical approach. They were calculated for a CPU of a mobile phone having a clock rate of 100 MHz. This thesis focuses on the performance of SigComp on the core network side. Fidrich, Bilicki, So'gor and Sey conclude that most well-known compression algorithms cannot be used without adapting or even re-designing them. If carefully built dictionaries are not used, the compression ratios are typically over 100%, i.e. the compressed messages are actually larger than the original ones.

SigComp can also be applied to the Push-to-talk over Cellular (PoC) service, because PoC uses SIP based signalling for session control. The effect of SigComp on the end-to-end delay in PoC session control is estimated in [Balazs 2004]. The results imply that if SigComp were able to reduce the size of the SIP message sequence by 80%, the end-to-end session setup delay would be reduced by almost 56% from 3.04 seconds to 1.34 seconds.

7 Signalling Compression Implementation

The purpose of this chapter is to describe the SigComp prototype that was implemented as a part of this thesis. We implemented the SigComp prototype from scratch. We did not use any program code written by other parties, but implemented all the functionality ourselves. The techniques used and the design decisions made are discussed. The chapter begins by describing the way that code from a single-threaded SigComp implementation was ported to the multi-threaded SigComp prototype. The test configuration and the process interaction paradigms used are presented. The implementation of shared resources and the data structures used are described. The architecture and operation of the SigComp prototype are explained through Unified Modelling Language (UML) class, state and sequence diagrams. The compression and decompression algorithms used by the prototype are presented. Finally, the way the functionality of SigComp extended operations was implemented is described.

7.1 From Single-threaded to Multi-threaded Code

In the first stage of the implementation work, we implemented the SigComp prototype as a single-threaded application. This allowed us to concentrate on the core functionality of the protocol, since the multi-threaded issues were left to a later stage. The things that had to be kept in mind when the single-threaded code was modified to support multiple threads are discussed below.

First of all, it has to be noted that choices that enable efficient parallelisation have to be made during the design of a program. It is most often not possible to introduce them to the program code afterwards. For this reason, the program code that was used to test the single-threaded SigComp prototype was thrown away, redesigned and rewritten when the switch to multiple threads was made. This program code is the one that binds the different components together, e.g. listens to the socket interface, creates and stores the compressors, invokes the decompressors and manages the state handler. However, many of the individual components of the protocol like the compressor and the decompressor could be ported to the new design with only minor, if any modifications.

The code written for multi-threaded programs has to be re-entrant and thread-safe to protect resource integrity. A re-entrant function cannot hold any static data over successive calls. This is because all statically allocated data will be overwritten by the next call to the function. Instead of the function holding the data, its caller must provide all the data that needs to be maintained over successive calls. The second requirement for a re-entrant function is that it should not return a pointer to static data. This can be achieved either by returning dynamically allocated data or by using caller-provided storage. In the latter case, the interface to the function needs to be modified so that for example a pointer to the storage can be passed to the function. In the former case, the caller must be modified in such a way that it frees the dynamically allocated storage when it is no longer required. Both of these approaches were used in the SigComp prototype. Finally, also all library procedures that multithreaded programs use should be re-entrant.

Thread-safety means that entities using shared resources, such as a shared database, must take turns using the resource. This can be achieved through the use of mutexes and semaphores, which are special kinds of shared variables used for synchronization. The approach that was taken in the SigComp prototype is to use a separate wrapper class for each shared resource. These classes use the Singleton design pattern [Gamma 1995] to ensure that only one instance of each such class can be created. Furthermore, each class takes care of its own lock variables. All the functions that are visible to outsiders are guaranteed to be multi-thread safe. In practise, this means that the caller of the function does not need to pay special attention to mutual exclusion; it can call the function just as if the code was single-threaded. Therefore, no modifications to the callers of such functions are required. For example, the interface to the state handler remains unchanged despite of the fact that in a multi-threaded code, there are multiple compressors and decompressors accessing the state handler.

7.2 Test Configuration

The SigComp prototype process acts as a P-CSCF in the measurements carried out to collect the results analysed in this thesis. In addition to the SigComp prototype, also another program was implemented to generate SigComp traffic for the P-CSCF process. The purpose of this program is to simulate mobile users that establish and terminate various kinds of sessions. From now on, this process is referred to as the UE process. The use of simple scripts on the UE side to generate the signalling traffic was not sufficient, because the UEs must generate and receive compressed traffic.

7.3 Process Interaction Paradigms

In this section, the process interaction paradigms used by the P-CSCF and UE processes are presented. These paradigms include the bag-of-tasks paradigm and the producers and consumers paradigm.

7.3.1 Bag-of-tasks

The UE process uses the bag-of-tasks paradigm [Andrews 2000] as the underlying process interaction paradigm. The bag-of-tasks paradigm is used to implement parallel computations. Its benefits include that it is scalable, meaning that it is easy to vary the number of workers that carry out the parallel computations. It is also relatively easy to ensure that each worker does about the same amount of work. The bag is a data structure that is shared by worker threads, while a task is an independent unit of work. There can be multiple worker threads executing on the CPU at a given time, each carrying out a separate task. A manager thread listens to the socket and generates a new receive task each time a SigComp message arrives. The worker threads fetch these tasks from the bag, execute them and possibly generate new send tasks to the bag. This is illustrated in Figure 19.

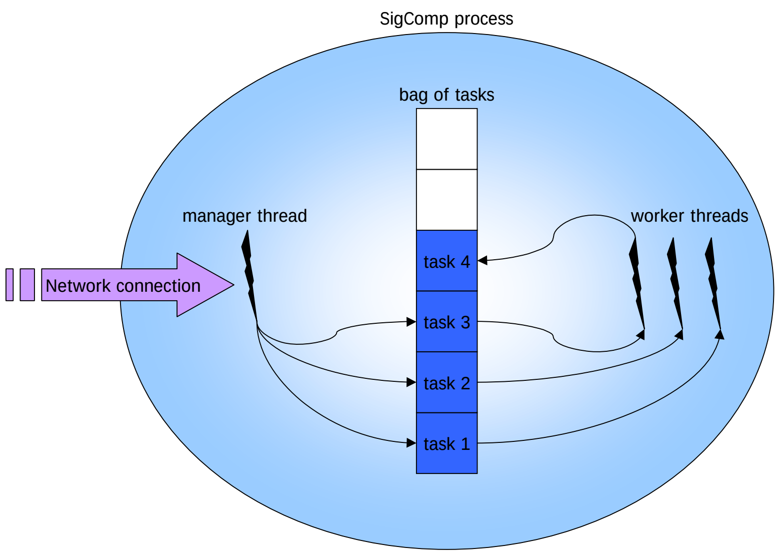


Figure 19 - The bag-of-tasks paradigm

As it was already mentioned, there are two kinds of tasks: receive tasks and send tasks. A receive task consists of a SigComp message that is to be decompressed, while a send task contains a SIP message that should be compressed and sent. A worker thread has two different roles: it can be either a receive worker or a send worker. Each worker is capable of handling both receive and send tasks; it cannot know which one of these two tasks it will get when it accesses the bag. The worker runs in a loop, which is illustrated by the following pseudo-code:

```
while(true) {
    if(the bag is empty) {
        wait until a task appears to the bag;
    }
    get a task from the bag;
    if(the task is a send task) {
        compress the message and create a SigComp message;
        send the SigComp message;
    }
    else {
        decompress the message;
        hand the decompressed SIP message to the SIP user agent, which returns
        a compartment identifier and possibly adds a new send task to the bag;
        provide the compartment identifier to the decompressor;
    }
}
```

A worker carrying out a receive task goes through three stages. First it decompresses the payload of the SigComp message it received. In the second stage it acts as a SIP user agent. It parses the SIP message and possibly generates a new send task to the bag. In the final stage, the worker terminates its UDVM instance by providing the UDVM the compartment identifier that was returned by the SIP user agent.

7.3.2 Producers and Consumers

The P-CSCF process uses producers and consumers [Andrews 2000] as the underlying process interaction paradigm. In the producers and consumers paradigm, a producer process produces tasks that are carried out by a consumer process. Communication between producers and consumers is conducted by means of a shared buffer. Access to the shared buffer is synchronized using a bounded buffer technique [Andrews 2000], which uses semaphores as resource counters. A bounded buffer is a multislot communication buffer. Producers deposit tasks to the rear of the buffer while consumers fetch messages from the front of the buffer. Because of this, deposit and fetch operations can execute concurrently as long as there are both empty slots and stored tasks in the buffer. Most importantly, a continuous stream of fetch operations initiated by consumers cannot prevent a producer from depositing new tasks to the buffer.

The P-CSCF process has one producer thread whose task is to listen to the socket interface and generate new tasks. It receives SIP messages from the core network side and SigComp messages from the access network side and inserts them to the buffer as new tasks. In addition to the producer, there are multiple worker threads that consume the generated tasks. A task containing a SIP message is executed by reading the message, compressing it and sending it to the access network. On the other hand, a task containing a SigComp message is carried out by decompressing the message, parsing it and sending the decompressed SIP message to the core network.

7.4 Shared Resources

Due to the nature of the SigComp protocol, the SigComp prototype has to use certain shared resources: the state handler and the compressor array. In addition to these, the shared buffer used by the producer and consumers must be implemented as a shared resource as well. All of these structures need to be multithread-safe because multiple workers use them simultaneously

The state handler, compressor array and the shared buffer have two kinds of users: those performing read operations and those carrying out write operations. When there are both readers and writers accessing the shared resource, a classical synchronization problem, the readers/writers problem, needs to be taken into account. In the case of a shared resource like the state handler, readers execute transactions that examine the records in the state item table of the state handler, while writers execute transactions that both examine and update the records. To ensure that each transaction transforms the state handler from one consistent state to another, the writer processes must have exclusive access to the state item table. If no writer is accessing the state item table, any number of readers may concurrently execute transactions.

The SigComp prototype uses the technique of passing the baton [Andrews 2000] to solve the readers/writers problem. This technique employs split binary semaphores both to provide exclusion and to signal delayed threads. Its virtues include that the order in which delayed threads are awakened can be controlled precisely and thus different scheduling policies can be implemented between readers and writers. When a thread is executing within a critical section, it can be thought that it is holding a baton that signifies permission to execute. When the thread no longer needs the baton, it passes the baton to one other process. To ensure fair access to the shared data structures of the state handler of the SigComp prototype, the guards of the technique of passing the baton use the following rules:

- If a reader is waiting, delay a new writer
- If a writer is waiting, delay a new reader

- When a reader finishes, awaken, i.e. pass the baton to one waiting writer, if any.
- When a writer finishes, awaken all waiting readers. If there is more than one delayed reader, one is awakened first and the others are awakened in cascading fashion. If there are not any readers waiting, awaken one waiting writer, if any.

The purpose of these rules is to force readers and writers to alternate turns when both are waiting. Both the state handler and the compressor array use this scheduling policy.

7.5 Data Structures

Different components of the SigComp prototype make use of different kinds of data structures. The bag of the bag-of-tasks paradigm is implemented as a priority queue. Each task on the queue has a timestamp indicating the time when the task should be carried out. The tasks are ordered based on the timestamp value in such a way that the task with the smallest timestamp is placed at the top of the queue. The shared buffer of the producers and consumers paradigm is implemented as a fixed-length array. The compressor array uses a map, which is a data structure containing key/value pairs. The key to the compressor array consists of a SigComp compartment identifier.

The compressor uses a hash table to implement the search buffer of the modified LZSS algorithm. The state handler uses two hash tables to store state items and feedback items. The key to the compressor’s hash table is formed by the first three bytes of the sequence to which the value of the key/value pair points. The key to the state handler’s state item table is the state identifier, which is a 20-byte Secure Hash Algorithm One (SHA-1) cryptographic hash, and the key to state handler’s feedback item table is a SigComp compartment identifier. The use of hash tables [Weiss 1999] is called hashing and it is a technique used for performing insert, delete and find operations in constant average time. Constant time can be achieved because the operations can be carried out without performing a search. Instead, the position where an item is stored can be determined directly from its value. This is enabled by the use of a hash function, the task of which is to map the position of an item and its value. A hash function distributes the keys evenly among the cells of the array the hash table uses. Each key is mapped into some number in the range 0 to *size of table*-1 and placed in the appropriate cell. A good hash function is easy to compute, spreads out keys evenly in the array and avoids collisions, i.e. situations in which distinct keys produce same hash values. Hashing is useful for any problem where the entries have real names instead of numbers.

7.6 Classes of the Signalling Compression Prototype

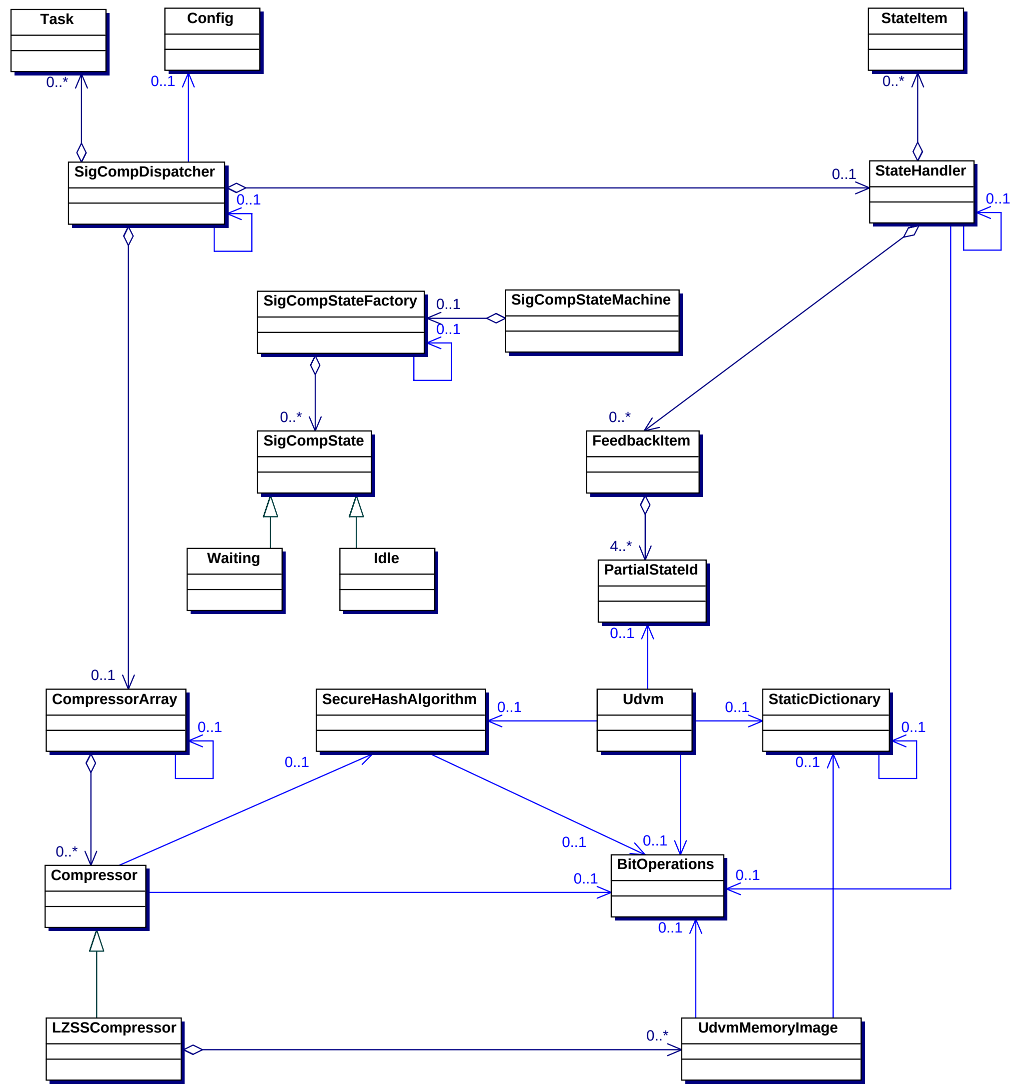


Figure 20 - Class diagram of SigComp prototype

The class diagram of the SigComp prototype is shown in Figure 20. The role of each class in the diagram is discussed in the following subsections.

7.6.1 BitOperations

The class BitOperations is a collection of functions that are used to manipulate binary sequences: to get and set bits, shift bits, carry out logical operations and so forth. The compressor and the UDVM use these functions heavily.

7.6.2 Compressor

The compressor is a super class for different compression algorithms. One Compressor object is associated with each SigComp compartment. The identifier of the compartment uniquely identifies a compressor. The most important task of the Compressor class is to provide an interface for the classes inheriting it. Compressor objects also construct the SigComp message header and attach the payload to it.

7.6.3 CompressorArray

Since the lifetime of a compressor is the same as the lifetime of a compartment, compressors need to be stored between the sending of SigComp messages. This is the task of the class CompressorArray. CompressorArray uses the Singleton design pattern described in [Gamma 1995]. The purpose of this pattern is to ensure a class only has one instance, and provide a global point of access to it. When the first message of a compartment is to be compressed, a new Compressor object is created. After the message has been compressed, the compressor object is stored in the compressor array for future use. The key of the compressor array is the compartment identifier, which has to be unique. After the initial message has been sent, the compressor is fetched from the compressor array each time a new message belonging to its compartment is sent. When the compartment is closed, the compressor has to be removed from the compressor array.

7.6.4 Config

The class Config implements an Extensible Mark-up Language (XML) reader that reads configuration information from an XML file. The configuration information includes various parameters used by the other classes of the SigComp prototype.

7.6.5 FeedbackItem

FeedbackItem is a class whose instances are used to convey information between compressors and decompressors. For example, after having decompressed a message and received a compartment identifier from the SIP application, the UDVM saves the feedback data that was included in the SigComp message header in a FeedbackItem object and forwards it to the state handler. One feedback item is associated with each compartment and the data that is stored in it is not compression or decompression algorithm specific. A feedback item holds a list of locally available states, a list of states that the remote endpoint has established, the most recently acknowledged state identifier, the shared state identifier and the state identifier of the most recent local UDVM memory snapshot state. It also contains values of different SigComp parameters that may have been returned by the remote endpoint.

7.6.6 LZSSCompressor

LZSSCompressor inherits the Compressor class. It contains a modified implementation of the Lempel-Ziv-Storer-Szymanski (LZSS) compression algorithm, which is described in Section 7.8.

7.6.7 PartialStateId

PartialStateId is a simple class whose instances are used to hold the values and lengths of state identifiers. The value of a state identifier contains either a complete SHA-1 hash or its first six, nine or twelve bytes.

7.6.8 SecureHashAlgorithm

Class SecureHashAlgorithm implements the Secure Hash Algorithm 1 (SHA-1) [RFC 3174, FIPS 180-1], which is used to calculate the hash or message digest values identifying SigComp state items. The SHA-1 hash is a condensed representation of the content of a SigComp state item, which typically holds a SIP message or the contents of UDVM memory. The SHA-1 algorithm always produces an output of 20 bytes despite the length of the input sequence. The SHA-1 is called secure because it is computationally infeasible to find a message, which corresponds to a given hash, or to find two different messages, which produce the same hash value. Any change to a transmitted message, for example a decompression or compression error or corruption of the message content during transport, will, with very high probability, result in a different message digest than the one calculated for the original message.

For example, when endpoint A sends a message to endpoint B, it calculates an SHA-1 hash over the uncompressed SIP message X before compressing it. It includes the SHA-1 hash in the returned parameters of the SigComp message that carries the compressed version of message X in its payload. If shared compression is used, then after having received and decompressed the message, endpoint B calculates an SHA-1 over it. By comparing the hash it calculated to the one in the returned parameters, endpoint B can deduce whether the message was received correctly. If the hashes match, endpoint B can use the received message, i.e. the shared state, in the compression process of the next message it sends.

7.6.9 SigCompDispatcher

The class SigCompDispatcher is responsible for implementing the producers and consumers paradigm. It uses the Singleton design pattern. The main function of the SigComp prototype is included in the class file of SigCompDispatcher. The main function is responsible for creating the pool of worker threads. It also creates the socket and listens to it. As new messages arrive to the socket, new receive tasks are generated and added to the shared buffer.

7.6.10 SigCompState, Idle, Waiting SigCompStateFactory and SigCompStateMachine

Classes SigCompState, Idle, Waiting SigCompStateFactory and SigCompStateMachine use the Flyweight design pattern defined in [Gamma 1995]. The Flyweight is a structural design pattern and its intent is to use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. Therefore, flyweights cannot make any assumptions about the context in which they operate. They can only store intrinsic state information, which is independent of the flyweight’s context. When a client object uses a flyweight, it has to pass any required extrinsic state information to the flyweight as an argument. The use of flyweight objects results in storage space savings. This is because sharing reduces the total number of instances that have to be maintained and thus also the amount of intrinsic state that has to be stored.

The class SigCompState acts as a flyweight. It declares an interface through which flyweights can receive and act on extrinsic state. It has two subclasses: Idle and Waiting, which implement the interface and add storage to intrinsic state. Idle and Waiting are sharable, which means that all clients share the same instances of these two classes. The task of the class SigCompStateFactory is to create and manage flyweight objects. It ensures that they are shared properly. SigCompStateFactory uses the Singleton design pattern.

The client that uses the flyweight objects is the class SigCompStateMachine. It uses the State design pattern described in [Gamma 1995] together with SigCompState and its subclasses. The State is a behavioural design pattern and its purpose is to allow an object to alter its behaviour when its internal state changes. SigCompStateMachine maintains an instance of one of the subclasses of SigCompState. This instance defines the current state of the protocol state machine. Each of the subclasses of SigCompState implements a behaviour associated with a state of the state machine. The use of the State pattern has the following consequences: (1) all behaviour associated with a particular state is located in one object, (2) state transitions are explicit, because separate objects are used for different states and (3) state objects can be shared.

7.6.11 SipParser

The class SipParser implements a simple SIP message parser. It is used to extract information from SIP messages.

7.6.12 StateHandler

Class StateHandler implements the SigComp state handler described in [RFC 3320]. The task of the state handler is to store state items, which are instances of the class StateItem, and compartment-specific feedback items, which are instances of the class FeedbackItem. StateHandler uses the Singleton design pattern.

7.6.13 StateItem

StateItem objects hold SigComp state items used to store for example the messages used in shared compression or the contents of the UDVM memory. Each state item is identified by a 20-byte SHA-1 message digest, which is a hash over the contents of the state item. More specifically, the hash is calculated over the byte string formed by concatenating the fields *state_length*, *state_address*, *state_instruction*, *minimum_access_length* and *state_value* of the state item. These fields contain the length of the state item, the address to which the value of the state item should be copied, the memory address from which execution should continue when the state item is used, the minimum number of bytes to use when the state identifier is compared to other state identifiers, and the value of the state item, respectively.

7.6.14 StaticDictionary

StaticDictionary is a class holding the static SIP/SDP dictionary described in [RFC 3485]. It includes functions for retrieving subsets of the dictionary and fields for holding its state parameters.

7.6.15 Task

The producer thread creates a new Task object each time it receives a message from the socket interface. Task objects are placed to the shared buffer, from which they are fetched by the worker threads.

7.6.16 UdvmDecompressor

Class UdvmDecompressor implements the Universal Decompressor Virtual Machine (UDVM) specified in [RFC 3320]. Since the SigComp implementation described here uses the LZSS compression algorithm, the decompression algorithm whose bytecode is run on the virtual machine is as well that of the LZSS algorithm. The UdvmDecompressor is written entirely in the C/C++ language.

7.6.17 UdvmMemoryImage

The class UdvmMemoryImage is used to generate and hold an image of the memory of the receiving endpoint’s UDVM. By maintaining this image, the sending endpoint knows the contents and state of the remote UDVM’s circular buffer and the values of the UDVM’s registers.

An example of the use of UDVM memory images is given in Figure 21, in which SigComp endpoint A sends a compressed INVITE request in the payload of SigComp message *m1* to endpoint B in step (1). Endpoint B sends back a compressed 180 Ringing response in the payload of SigComp message *m2* in step (2).

Before endpoint A starts to compress the INVITE in step (1), it generates a remote UDVM memory image, which is called *UMS_B* in Figure 21. The name stands for Udvm Memory Snapshot B. The image reflects the contents of the remote UDVM’s memory at the moment it has finished decompressing the INVITE message. Figure 21 shows only the dictionary part of the UDVM’s memory. Other data in the memory, e.g. registers and bytecode are omitted from the figure for simplicity. At the moment the INVITE has been decompressed, the dictionary of endpoint B’s UDVM will contain only the static SIP/SDP dictionary and the decompressed INVITE message. Therefore, the compressor of endpoint A uses the static dictionary to compress the INVITE message. Endpoint A stores *UMS_B* and calculates an SHA-1 hash over it. Endpoint A also records that state *UMS_B* has not been acknowledged and thus cannot be used to compress messages.

After endpoint B has decompressed the message *m1*, the contents of the memory of its UDVM correspond exactly to *UMS_B*. Therefore, the SHA-1 hash that endpoint B calculates over its memory matches the one that was calculated at endpoint A. Next, endpoint B saves a snapshot of its memory for future use. Since this snapshot corresponds to the memory image that was calculated at endpoint A, it is also called *UMS_B* in Figure 21. When endpoint B sends the message *m2* to endpoint A in step (2), it acknowledges the creation of *UMS_B* in the header field *acked_state_id* of message *m2*. The message *m2* carries a compressed 180 Ringing message in its payload. When endpoint A receives the message, it records that *UMS_B* has been acknowledged and can be used to compress the next message endpoint A sends. This means that when the next message is sent, endpoint A can initialise the dictionary of its compressor with the dictionary that is retrieved from *UMS_B*.

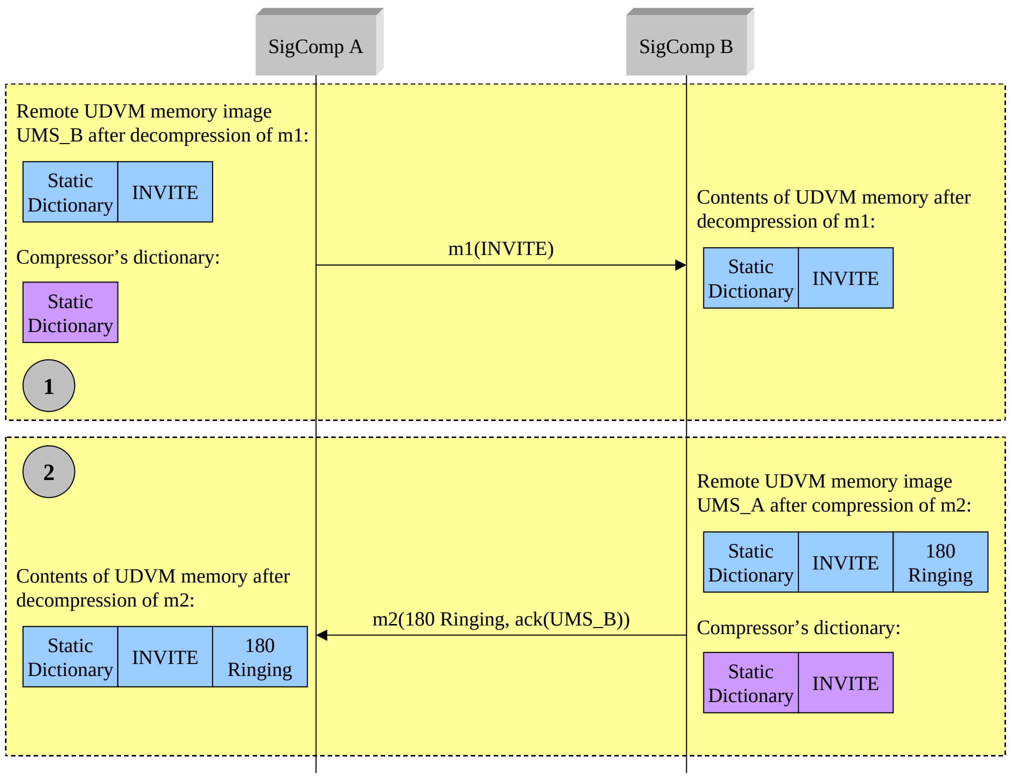

SigComp A
SigComp B
Remote UDVM memory image UMS_B after decompression of m1:
Static Dictionary
INVITE
Compressor's dictionary:
Static Dictionary
1
m1(INVITE)
Contents of UDVM memory after decompression of m1:
Static Dictionary
INVITE
2
Contents of UDVM memory after decompression of m2:
Static Dictionary
INVITE
180 Ringing
m2(180 Ringing, ack(UMS_B))
Remote UDVM memory image UMS_A after compression of m2:
Static Dictionary
INVITE
180 Ringing
Compressor's dictionary:
Static Dictionary
INVITE

Figure 21 - Use of UDVM memory images

7.7 Classes of Universal Decompressor Virtual Machine Interpreter Implementation

In this section, the architecture of the UDVM interpreter that was implemented as a part of this thesis is described. The class diagram of the interpreter is presented in Figure 22.

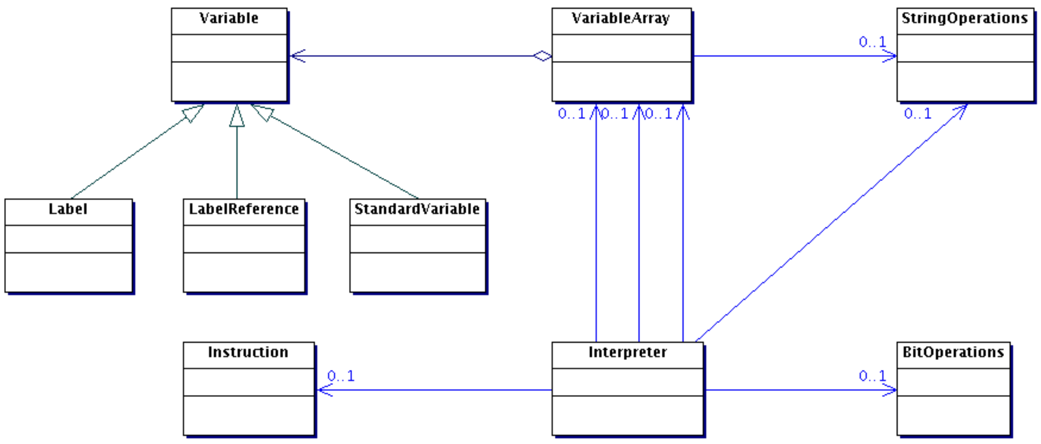


Figure 22 - Class diagram of the UDVM interpreter

7.7.1 BitOperations

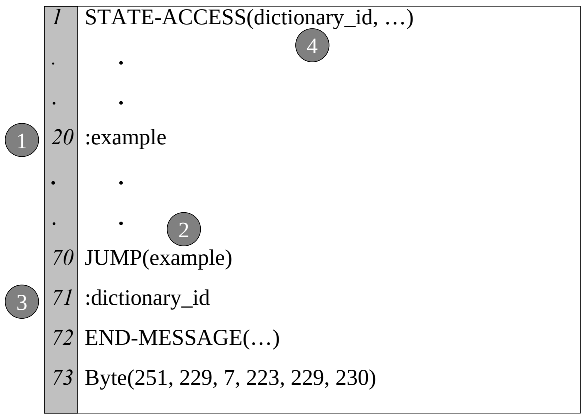
The class BitOperations is a collection of functions used to manipulate binary sequences, i.e. to get and set bits, shift bits, carry out logical operations and so forth. This class is the same one that the SigComp prototype uses.

7.7.2 Instruction

Instruction objects store the instructions that are read from the UDVM assembly being interpreted. Each Instruction object contains the name of certain instruction and the operation code and operands of the instruction.

7.7.3 Interpreter

The UDVM interpreter is responsible for compiling the human readable UDVM assembly to the binary bytecode that can be executed on the virtual machine. The Interpreter class contains the main function of the UDVM interpreter. The Interpreter maintains three instances of the class VariableArray, one for labels, one for label references and one for variables. The main function of the UDVM interpreter program first reads an input file containing the UDVM assembly and then stores all lines containing instructions, directives and declarations for labels and variables to separate string arrays. After this, it processes the lines that contained variables and for each declared variable, creates a Variable object to hold the variable. Variable objects are stored in an instance of the class VariableArray. In the next step, the instruction lines are encoded. When this step is over, a preliminary version of the bytecode has been constructed. The reason for the version being only preliminary is that at this point, the final positions of different labels cannot be known for sure. An example illustrating this is shown in Figure 23. In the figure, the STATE-ACCESS instruction is the first instruction in the assembly and the label *dictionary_id* is the last label in the assembly. The *dictionary_id* is referenced by the STATE-ACCESS instruction. Between the STATE-ACCESS and the *dictionary_id*, there can be an arbitrary number of lines containing instructions, label declarations, variable declarations and directives. When the STATE-ACCESS instruction is initially encoded, the final value of the *dictionary_id* label is unknown, because the length of the bytecode between the instruction and the label is not known. Therefore, the interpreter has to use an estimate when it encodes the reference to the *dictionary_id*. After all the remaining instructions have been encoded in a similar manner, the UDVM interpreter enters a loop in which, during each iteration, it recalculates the values of the labels and updates their references. The loop terminates after the first iteration during which the values of all the labels remain unchanged. Multiple iterations are typically needed, because when the value of a label changes, the new value cannot always be encoded using the same number of bytes as the old value. This is because each operand of a UDVM instruction is compressed using variable-length encoding. The problem is illustrated in Figure 23. In step (1) of the figure, the value of the label *example* changes. Because of this, the length of the encoded value of the reference to *example* in instruction *JUMP* changes also in step (2). This, in turn, has an effect on the position of the label *dictionary_id* in the bytecode in step (3). And because the position of the *dictionary_id* changes, the value of its reference in STATE-ACCESS (4) needs to be updated. This may result in a situation in which the length of the reference to *dictionary_id* changes and thus also the positions of labels *example* and *dictionary_id* have to be shifted.



```
1   STATE-ACCESS(dictionary_id, …)
.       .
.       .
20  :example
.       .
.       .
70  JUMP(example)
71  :dictionary_id
72  END-MESSAGE(…)
73  Byte(251, 229, 7, 223, 229, 230)
```

Figure 23 - Processing of UDVM assembly

7.7.4 StringOperations

The class StringOperations is a collection of functions that are used to process the data that are read from the file containing the UDVM assembly.

7.7.5 Variable, Label, LabelReference and StandardVariable.

Variable is a super class for the classes Label, LabelReference and StandardVariable. Label objects are used to store the labels that are declared in the UDVM assembly. A label assigns a memory address to a text name. The position of a label may change as the assembly is interpreted to bytecode. A LabelReference object is created for each reference to a label that is encountered in the assembly. This is done in order to track the position of the reference as the interpretation proceeds. Finally, instances of the StandardVariable objects are used to store variables declared by *set* directives. The *set* directive is used to assign values to text names in the UDVM assembly language. The values of such text names remain constant during the interpretation process, i.e. their values do not depend on the position in which they are declared.

7.7.6 VariableArray

VariableArray is a class whose instances store StandardVariable, Label and LabelReference objects, i.e. the objects that inherit the Variable class.

7.8 Compression Algorithm

The compression algorithm that is used in the SigComp implementation is based on the Lempel-Ziv-Storer-Szymanski (LZSS) algorithm. LZSS in turn is a variation of the Lempel-Ziv 1977 (LZ77) algorithm, which is based on a paper by Jacob Ziv and Abraham Lempel in 1977 [Ziv 1977]. Before the operation of the modified LZSS algorithm is explained, an introduction is given to the LZ77 and LZSS algorithms. Also the basics of dictionary compression techniques are covered.

7.8.1 Dictionary Techniques

In many applications, the output of the source consists of recurring patters. This is also the case with SIP messages, in which certain patterns recur constantly. A very reasonable approach to encoding such sources is to keep a list or dictionary of frequently occurring patterns. When the patterns reappear in the input, they are encoded with a reference to the dictionary.

There are two approaches to dictionary compression: a static approach and an adaptive approach. The use of a static dictionary [Sayood 1996] is most appropriate when considerable prior knowledge is available about the source. The static dictionary approach is most suitable for use in specific applications. Since our task is to compress the session initiation protocol, the static dictionary approach is an efficient solution because we know ahead of time that certain words such as “FROM”, “SIP” and “CALL-ID” are going to appear in almost all of the messages to be compressed. This is the very purpose of the static SIP/SDP dictionary; to offer a collection of strings relative to which SIP messages with SDP content can be compressed.

However, using the static dictionary approach alone is unlikely to result in high compression ratios. After all, a great deal of the content of SIP messages is specific to a single session; it would not be feasible to include for instance the addresses of all of the potential callers and callees to the dictionary. The adaptive dictionary technique provides an answer. It is an efficient approach when sufficient prior knowledge about the source is not available. An example of an algorithm that uses the adaptive dictionary technique is the LZ77.

7.8.2 LZ77

LZ77 [Ziv 1977, Sayood 1996] uses the adaptive dictionary technique. In LZ77, the adaptive dictionary is a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window, which consists of two parts: a search buffer and a look-ahead buffer. The search buffer contains a portion of the recently encoded sequence, and the look-ahead buffer contains the next portion of the sequence to be encoded. To encode a sequence in the look-ahead buffer, the encoder moves a search pointer back through the search buffer until it encounters a match to the first symbol in the look-ahead buffer. The distance of the pointer from the start of the look-ahead buffer is called the offset. Next the encoder examines the symbols following the symbol at the pointer location to see whether they match consecutive symbols in the look-ahead buffer. The number of consecutive symbols in the search buffer matching consecutive symbols in the look-ahead buffer is called the length of the match. The search buffer is searched for the longest match that can be found. Once such a match has been found, it is encoded with a triple $<o, l, c>$, where o is the offset, l is the length and c is the codeword corresponding to the symbol in the look-ahead buffer that follows the match. The reason why the third element in the triple is used is to take care of the situation in which no match for the symbol in the look-ahead buffer can be found in the search buffer. In this case, the offset and length values are set to zero and the third element of the triple is the code for the symbol itself. An example of the operation of the LZ77 algorithm is illustrated in Figure 24 and Figure 25.

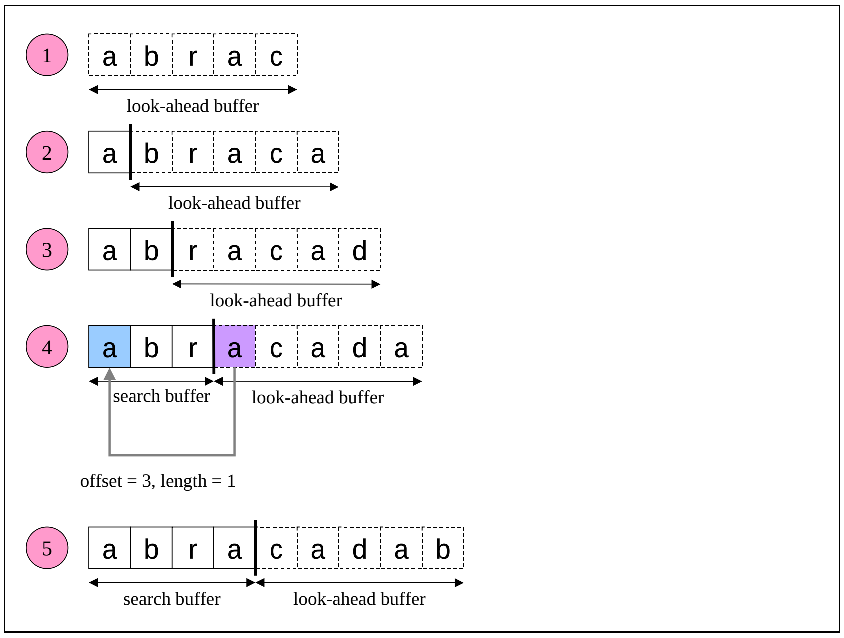


Figure 24 - Operation of the LZ77 algorithm, part I

In step (1) of Figure 24, the search buffer is initially empty and the look-ahead buffer contains the first five characters of the input sequence, which is the string ‘abracadabra’. The size of the search buffer is 7 symbols and the size of the look-ahead buffer is 5 symbols. In practice, the sizes of the buffers would be significantly larger. The first encoded character is the first character in the look-ahead buffer, i.e. *a* in step (1). Because the search buffer is empty, no match can be found. Therefore, the encoder outputs the triple *<0,0,a>* and moves the sliding window one position right, ending up to the situation shown in step (2). In step (2), symbol *b* is not found from the search buffer, and code for the symbol itself is output in the same way as in the previous step. Symbol *r* is encoded in a similar manner in step (3). In step (4), the first symbol in the look-ahead buffer is *a*, which is also present in the search buffer. Therefore, instead of outputting the symbol *a*, the encoder outputs the position of the match, namely the triple *<3,1, ->*. In step (5), no match is found from the search buffer for symbol *c* and the encoding proceeds in a similar way as in step (2). The remaining steps of the encoding process are shown in Figure 25.

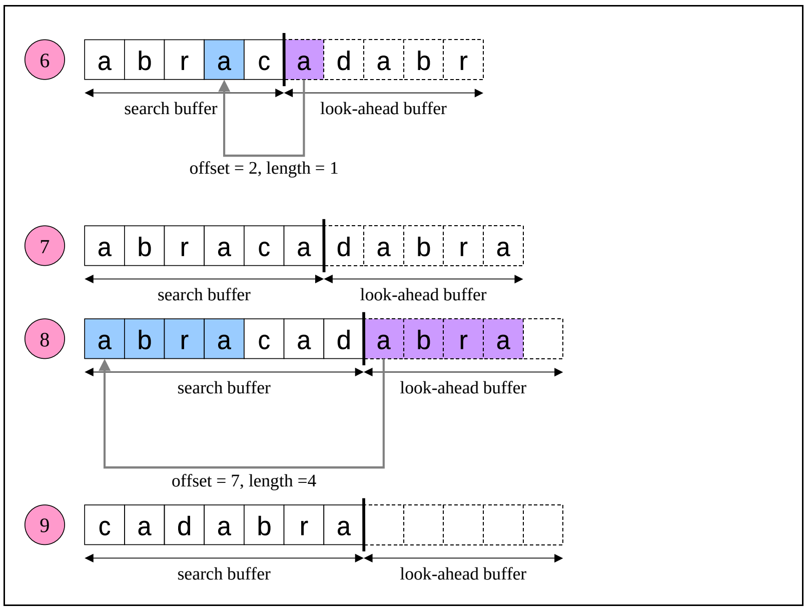


Figure 25 - Operation of the LZ77 algorithm, part II

In step (6), two matches for symbol *a* are found in the search buffer. In the case of both of these matches, the next symbol is different from the symbol that follows *a* in the look-ahead buffer. Therefore, it does not matter which one of the matching symbols is encoded. The encoder chooses the nearest symbol and outputs the triple *<2,1,->*. In step (7), no match is found for symbol *d*. In the next step, the first symbol in the look-ahead buffer is symbol *a*. Two matches can be found in the search buffer. However, in the case of the first match, the three consecutive symbols in the look-ahead following *a* are the same as the three consecutive symbols following *a* in the search buffer. Therefore, we can encode the sequence *abra* using the triple *<7,4,->*. In step (9), the look-ahead buffer is empty, since we have encoded the entire sequence. Note that when the sliding window was moved four positions left after the sequence *abra* was encoded in step (8), the four first symbols in the search buffer got pushed out, since the capacity of the search buffer is only 7 symbols.

When the LZ77 algorithm is used, three different possibilities may be encountered during the coding process:

- There is no match for the next character to be encoded in the sliding window
- There is a match
- The matched string extends inside the look-ahead buffer

The third case occurs when the last character of the matching sequence is in the last slot of the search buffer and the next symbol or symbols of the sequence in the look-ahead buffer match to the first symbol or symbols in the look-ahead buffer. This is illustrated in Figure 26. In the first step of the figure, a match for sequence *cab* is found in the search buffer. In the second step, the search is continued in the look-ahead buffer. It is observed that instead of coding a match of length three, the triple can actually code a match of five characters by extending the match to the look-ahead buffer.

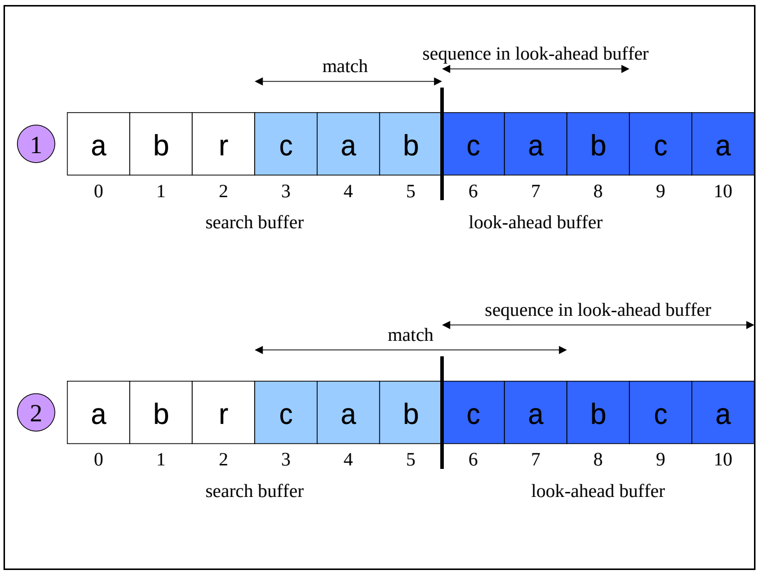


Figure 26 - Match that extends to look-ahead buffer

7.8.3 LZSS

The LZ77 algorithm is very inefficient when it comes to encoding references to single symbols. For example, in step (4) of Figure 24, the symbol *a* is encoded using the triple *<3,1,->*. This is highly inefficient, since the encoded triple is actually longer than the code for the symbol itself. For example, the ASCII-representation of *a* takes only 8 bits, while the triple might require 16 bits assuming that 12 bits are used to encode the offset value and 4 bits to encode the length value. Using triples to encode single symbols is highly inefficient if a large number of characters occur infrequently. The LZSS algorithm eliminates the situation in which a triple is used to encode a single character by using a flag bit to indicate whether what follows is a codeword or a single symbol. The flag bit also makes it possible to get rid of the third element in the triple.

7.8.4 The Modified LZSS Algorithm

In the SigComp prototype, the following modifications were made to the LZSS algorithm:

- In addition to the adaptive dictionary approach, also the static dictionary approach is used
- Support for external dictionaries was added
- A circular buffer is used instead of a linear buffer
- 14 bits are used to encode offset values (to enable the use of a decompression memory of up to 16384 bytes)
- The number of bits that is used to encode length values can be configured
- Hashing is used to speed up searches from the circular buffer

The static dictionary approach of the static SIP/SDP dictionary assumes that we have prior knowledge about the source, while the adaptive dictionary technique of the LZSS algorithm assumes no prior knowledge of the source. These two approaches are combined in the modified LZSS algorithm. By using the static dictionary approach, it is possible to encode in an efficient way the patterns that are common across all SIP messages. On the other hand, by using the adaptive dictionary approach together with the support for external dictionaries, we can make use of the similarities between the messages that belong to a single SIP dialog, for example the user's name, email address and URL.

The adaptive dictionary of the unmodified LZSS algorithm is the previously encoded sequence, i.e. the portion of the SIP message that has already been compressed. Each message is encoded independently. The modified LZSS algorithm has the ability to load external dictionaries to its search buffer and thus compress the new message relative to their contents. This allows the use of acknowledged and shared states in the compression process.

The search buffer and thus also the sliding window of the modified LZSS algorithm is circular, which means that once the right boundary of the buffer is reached, characters will be added to the front of the buffer. Because of this, the offset values may be greater than the value of the current search pointer. This is illustrated in Figure 27, in which the current value of the search pointer is 3 and the offset of the match is 7. The use of a circular buffer is necessary, since when using dynamic compression, the contents of the search buffer are saved between the compressions of consecutive messages. When the next message is compressed, the search buffer is initialised by using the contents of a previous search buffer. This also means that eventually the search buffer will fill up, provided that the combined size of the messages is greater than the capacity of the buffer. This will usually be the case; it is not feasible to maintain a large buffer firstly because the use of a large buffer typically results in longer compression and decompression times, and secondly because the memory capacity can sometimes be a restrictive factor, especially in mobile terminals. When the buffer fills up, it is far more reasonable to start again from the front of the buffer than to start again with an empty buffer. The use of a circular buffer is also necessitated by the fact that the UDVM uses a circular buffer as well. In fact, the size of the search buffer of the modified LZSS algorithm has to be the same as that of the UDVM to make sure that both of them see their buffers in a similar state.

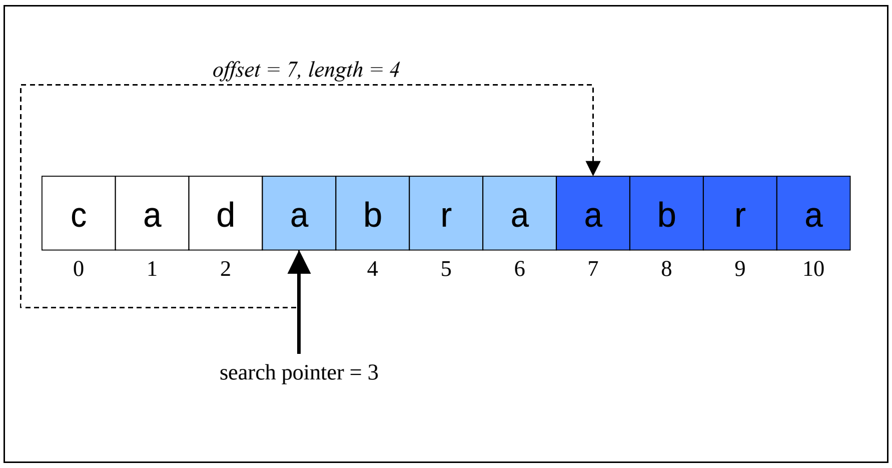


Figure 27 - Circular buffer

The number of bits that the modified LZSS algorithm uses to encode offset values depends on the size of the decompressor's circular buffer. 12 bits are sufficient for a circular buffer of the size $2^{12} = 4096$ bytes, 13 bits for a buffer of the size $2^{13} = 8192$ bytes and 14 bits for a buffer of the size $2^{14} = 16384$ bytes. In a similar way, the number of bits that are used to encode length values restricts the maximum length of a match that can be taken from the dictionary. If 5 bits are used to encode length values, then the maximum length of a match is $2^5 + 3 = 35$ bytes. The factor 3 is added, because LZSS never encodes matches whose length is two bytes or less. If a match of length two were encoded as an offset/length pair, then the number of bits needed to encode the value would be 1+14+5 = 20, provided that 1 bit is used by the flag bit, 14 bits are used to encode the offset and 5 bits to encode the length. If the two characters are encoded using their own codes and the flag bits, then their length is 1+8+1+8=18. Therefore, it would be wasteful to encode matches of length two or less as offset/length pairs.

7.8.5 Hash Function of the Modified LZSS Algorithm

The hash function *h* that the compression algorithm uses is implemented as a composition of two functions *f* and *g*. Given a set of keys, *K*, and a hash table of size *M*, a hash function is a function of the form

$$h : K \mapsto \{0,1,\ldots,M-1\}. \qquad (7.1)$$

The function *f* maps keys into integers:

$$f : K \mapsto \mathrm{Z}^+, \qquad (7.2)$$

where Z+ is the set of non-negative integers. The function g maps non-negative integers into the range from 0 to M-1:

$$g : \mathrm{Z}^+ \mapsto \{0,1,\ldots,M-1\}. \qquad (7.3)$$

The hash function *h* is defined as follows:

$$h = g \circ f \;, \qquad (7.4)$$

meaning that the hash value of a key *x* is given by *g(f(x))*. The function *f* [Preiss 1997] is shown below:

$$f(s) = \left(\sum_{i=0}^{n-1} B^{n-i-1} s_i \right) \bmod W \;, \qquad (7.5)$$

where *s* is a character string, $W = 2^w$ such that *w* is word size of the machine and $B = 2^b$ such that *b* is the number of bits in the input characters. Assuming the value 32 for *w* and the value 6 for *b*, the following code [Preiss 1997] presents an optimised way to implement the function *f*:

```
unsigned const int shift = 6;
unsigned const int mask = ~0U << (32 – shift);

unsigned int hash = 0, i = 0;
for(i=0; s[i] != ‘\0’; ++i)
          hash = (hash & mask) ^ (hash << shift) ^ s[i];
```

```
        return hash;
```

Value 6 can be used for b if it is assumed that letters and digits are the most common characters in strings; all the information in the ASCII codes of these characters is in the six least significant bits. The code contains one additional step compared to function f: the six most significant bits are retained and inserted back to the shifted *hash* variable through an exclusive or (^) operation.

The task of function g is to map the values produced by function f into the interval [0, M-1]. For this, a method called the Fibonacci hashing [Preiss 1997] is used. In Fibonacci hashing, the hash function has the form

$$g(x) = \left\lfloor \frac{M}{W}(ax \bmod W) \right\rfloor, \tag{7.6}$$

where x is the result of function f and a is a carefully chosen constant, whose value is closely related to the number called golden ratio. The value of the golden ratio is

$$\phi = \frac{1+\sqrt{5}}{2}. \tag{7.7}$$

The value of factor a in Equation (7.6) is given by $a = \phi^{-1}W$. When this value is used, the function g has two favourable properties. First of all, each subsequent hash value divides the interval into which it falls according to the golden ratio. Second of all, the hash value for each subsequent key falls between the two widest spaced hash values already computed. This is illustrated in Figure 28 for a hash table whose size is 100. We can observe from the figure that consecutive keys spread out quite efficiently.

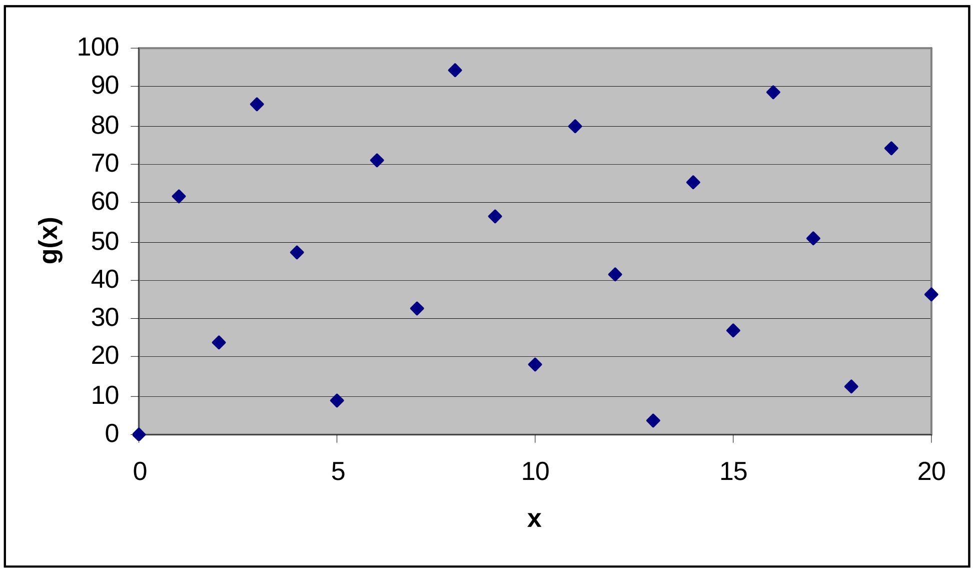


Figure 28 - Fibonacci hashing

7.9 Decompression Algorithm

In this section, the decompression algorithm used in the SigComp prototype is described. The way the bytecode of the decompression algorithm organizes the memory of the UDVM is illustrated in Figure 29. In the figure, it is assumed that the size of the decompression memory is 8192 bytes. The first 128 bytes of the memory are reserved for the values of various registers. Addresses 128 - 400 are reserved for the bytecode. Rest of the memory is used as a circular buffer. The static SIP/SDP dictionary is considered to be a part of the buffer, meaning that once the buffer reaches its right boundary (i.e. *byte_copy_right*), the static dictionary will be overwritten. This is because in the SigComp prototype, the content of previous messages is considered to be more valuable than the information in the static dictionary; it is more likely to help in achieving higher compression ratios. The string subset of the static dictionary is initially loaded to the beginning of the circular buffer. The first byte of the string subset is copied to address 400, which is the left boundary of the circular buffer and is pointed to by the register *byte_copy_left*.

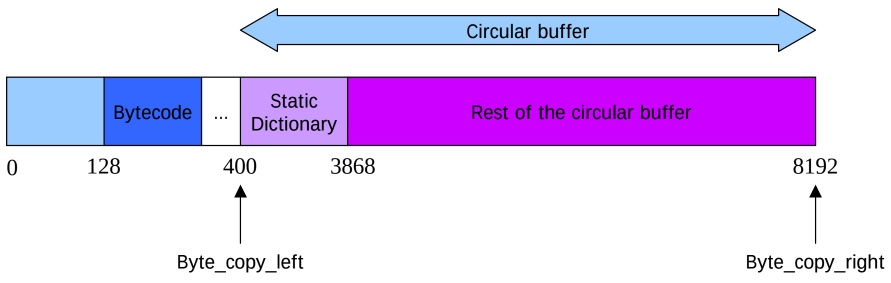


Figure 29 - Organization of the UDVM's memory

The UDVM assembly of the LZSS decompression algorithm is presented in [Draft Price]. This assembly is included in Appendix B. The assembly that is used in the SigComp prototype includes a few modifications to the original assembly, including support for the modified LZSS algorithm, changes to the way the UDVM memory is organized, changes to the use of shared states and other minor modifications. A description of the original assembly is given below.

Rows 1-34 of the assembly are used to reserve registers in the memory of the UDVM. As an example, on line 3 the label *index* is associated with memory addresses 32-33. The *at(32)* directive on line 1 appends 32 padding bytes to the bytecode. Because *index* is the first label that is declared after the *at(32)* directive, *index* is associated with memory address 32. The *pad(2)* directive on line 3 appends two padding bytes to the bytecode starting from address 32. This means that the length of the *index* field becomes two bytes.

The STATE-ACCESS instruction on line 40 loads the string subset of the static SIP/SDP dictionary to the memory of the UDVM starting from address 1024. The partial state identifier of the static dictionary is read from the memory address specified by the operand *dictionary_id*, which will, by the time the bytecode is loaded to the memory of the UDVM, contain the six bytes that are declared on line 126 of the assembly. These bytes present the partial state identifier of the static dictionary.

The MULTILOAD instruction on line 45 initialises the values of the registers in memory locations 64-71, i.e. registers *byte_copy_left*, *byte_copy_right*, *input_bit_order* and *decompressed_pointer*. The registers *byte_copy_left* and *byte_copy_right* specify the bounds of the circular buffer, while the register *decompressed_pointer* contains a value that points to the location to which the next uncompressed byte should be copied.

The INPUT-BYTES instruction on line 49 reads the *s*-bit of the SigComp header. The value of the *s*-bit is checked on line 50 by the COMPARE instruction. If the value is zero, the UDVM continues execution on line 57. On the other hand, if the value is '1', then the shared state identifier is read by the INPUT-BYTES instruction on line 54 and the state with this state identifier, i.e. the shared state, is loaded to the UDVM memory on line 55.

The MULTILOAD instruction on line 59 loads new values to registers *minimum_access_length*, *announcement_location*, *decompressed_start* and *decompressed_length*.

On line 60, the COPY-LITERAL instruction appends eight padding bytes to the front of the area to which the message will be decompressed. Later in the assembly, the length of the decompressed message and the contents of the register *minimum_access_length* will be written to these eight bytes. The purpose is to leave room for the eight bytes that are a part of all state items: *state_length*, *state_address*, *state_instruction* and *minimum_access_length*. When an SHA-1 hash is created for the decompressed message, these eight bytes will be included in the string over which the hash is calculated.

The instructions on lines 62 and 63 read the *a*-bit of the SigComp header and take the following actions depending on its value: if the value is zero, the execution of the bytecode continues from line 71. If the value is one, execution moves to line 67. In the latter case, the UDVM starts to write information to the location of returned parameters, which contains the list of returned remote partial state identifiers. The instructions on lines 67 and 68 write the length and value of the state identifier specifying the remote UDVM memory snapshot state to the returned parameters location. The LOAD instruction on line 69 ensures that the shared state identifier, which is calculated later in the assembly, is stored in the next free byte after the state identifier of the remote UDVM memory snapshot.

The actual decompression algorithm starts from line 75. The INPUT-HUFFMAN instruction on line 75 reads the first nine bits of the decompressed message. It goes through one or two iterations depending on the value of the sequence of 9 bits that are read. As was explained earlier, in the LZSS compression algorithm, each compressed character begins with a 1-bit indicator flag specifying whether the character is a literal or an offset/length pair. If the flag is '0', then the eight-bit sequence that follows the flag contains a literal. A literal is an uncompressed ASCII character and its value is within the range 0-255. If the sequence following the flag is a literal, the INPUT-HUFFMAN instruction returns after the first iteration. On the other hand, if the flag is '1', the sequence of sixteen bits that follows the flag contains an offset/length pair. In this sequence, 12 bits are reserved for the offset and 4 bits for the length value. If the flag is '1', the instruction reads the indicator flag and the first eight bits of the offset in the first iteration, and the six remaining bits of the offset during the second iteration.

If the indicator flag was '0', the execution continues from line 82. The instructions on lines 82 and 83 output one character, and also copy the character to the decompressed message. The JUMP instruction on line 84 moves execution back to line 75, i.e. to the beginning of the decompression algorithm.

If the indicator flag was ‘1’, the execution continues from line 88. The INPUT-BITS on that line reads the length value of an offset/length pair from the compressed message. The ADD instruction on line 89 increments the length by 3. This is done because the LZSS algorithm never encodes sequences whose length is less than 3 bytes as offset/length pairs. If length 0 is read from the message, it should be interpreted as length 3, whereas length 1 should be interpreted as length 4 and length *n* as length *n+3*. The COPY-OFFSET instruction on line 91 counts backwards a total of offset memory addresses, starting from the next byte following the most recent decompressed byte. Starting from the resulting address, it appends a total of *length* bytes to the decompressed message. This happens for example whenever a byte string is fetched from the static SIP/SDP dictionary. On line 92, the fetched byte string is output and on line 93, the JUMP instruction moves execution back to the beginning of the decompression algorithm.

The instructions on lines 97 and 98 read the *r*-bit of the SigComp header. Next, one of the following actions is taken depending on the value of the *r*-bit: value ‘0’ indicates that the sender did not save a state corresponding to the decompressed message. In this case, the execution continues from line 118. If the *r*-bit is ‘1’, then the sender saved a state corresponding to the decompressed message. If this is the case, the shared state identifier is calculated on lines 102-112.

The LOAD instruction on line 118 stores some flag bits and the length of the requested feedback field to the location of requested feedback in the memory of the UDVM. The MULTILOAD instruction on the next line loads four two-byte blocks to the front of the state value of the UDVM memory snapshot that will be calculated. These two-byte blocks include the length of the sequence over which the hash is calculated, the address to which the snapshot value should be loaded, the instruction from which the execution should continue once the snapshot is loaded and the minimum access length of the state item. On line 120, a hash is calculated over the UDVM memory. This hash value is placed to the field *acked_state_id* of the SigComp header when the next message is sent by this endpoint.

Finally, the END-MESSAGE instruction on line 122 terminates the UDVM. It ensures that the feedback data, the shared state and the UDVM memory snapshot will be saved once the SIP application provides a valid compartment identifier.

7.10 State and Sequence Diagrams

In this section, the state diagram and various sequence diagrams of the SigComp prototype are presented by means of Unified Modelling Language (UML) diagrams [Doldi 2003].

7.10.1 State Diagram

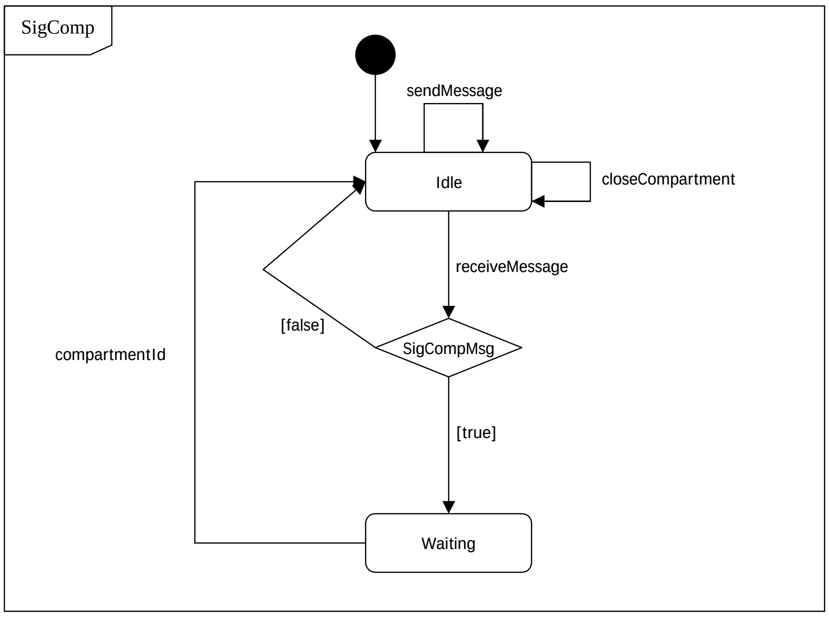


Figure 30 - SigComp state diagram

The state diagram of the SigComp state machine is shown in Figure 30. It illustrates the states that a worker thread can occupy during its lifecycle. The execution begins at the initial pseudo-state: the state machine goes to state *Idle*. In this state, three events can be received: *sendMessage*, *receiveMessage* and *closeCompartment*. The events *sendMessage* and *closeCompartment* do not change the state of the state machine. However, the event *receiveMessage* moves the state machine to state *Waiting* if the received message is a SigComp message. If the message is a SIP message, the state machine stays in the state *Idle*. When the event *compartmentId* is received in the *Waiting* state, the state machine returns to the *Idle* state.

The event *sendMessage* is used to request the sending of a new message. When it is received, a message is compressed and placed into the payload of a SigComp message, which is passed to the transport layer. The event *closeCompartment* occurs when the SIP application wants to close a compartment and release its resources. When the event *receiveMessage* is received, a SigComp message is decompressed and the resulting message is passed to the SIP application. After this, the state machine makes a transition to the state *Waiting*, in which it waits for the SIP application to provide a compartment identifier. The identifier arrives in the form of a *compartmentId* event. Once this event occurs, state information can be saved, after which the state machine returns to the state *Idle*. The following subsections describe in detail what happens when each of the events is received.

7.10.2 Event SendMessage

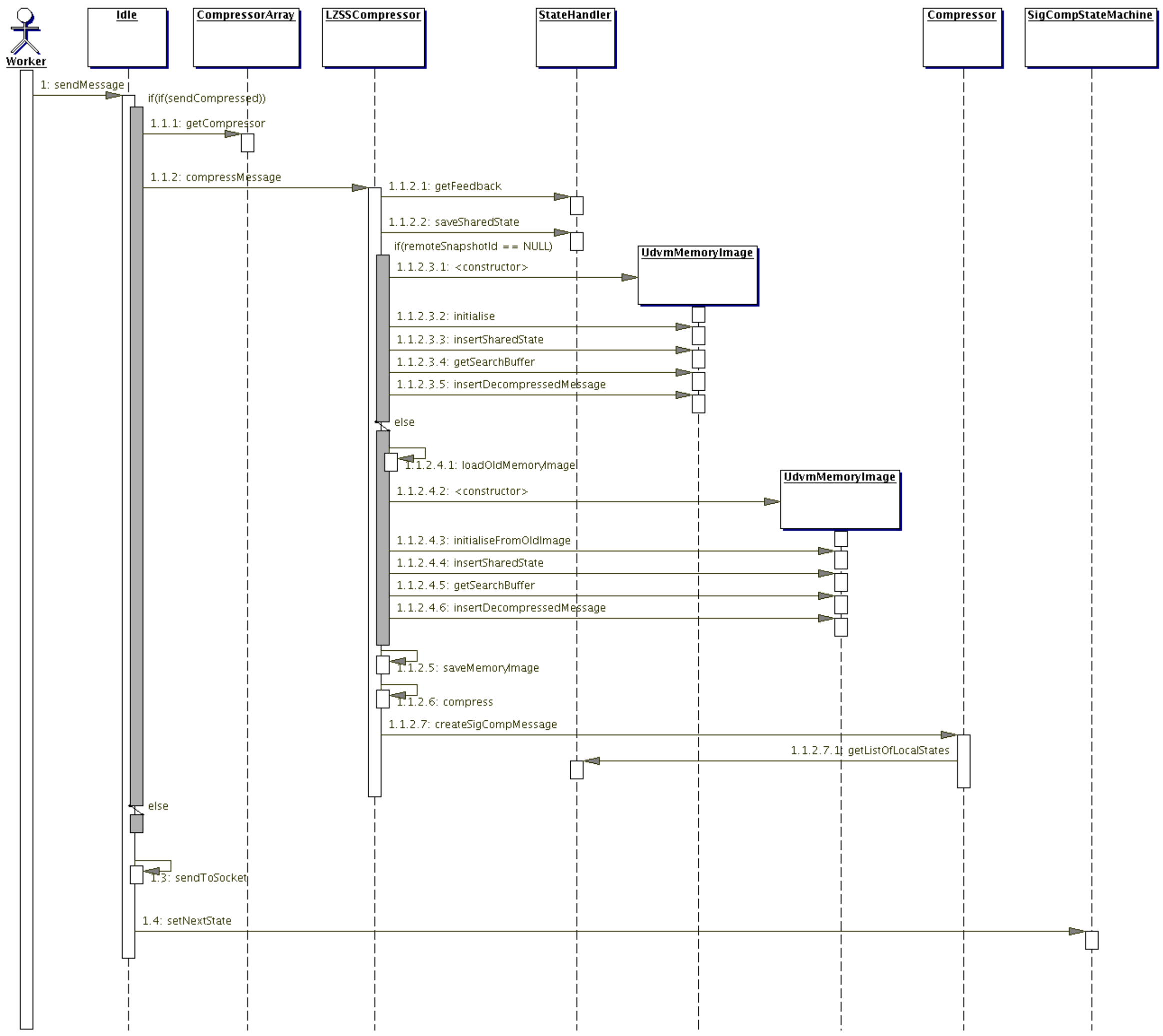


Figure 31 – Sequence diagram for event sendMessage

A Unified Modelling Language (UML) diagram illustrating the actions taken when the event *sendMessage* is received is shown in Figure 31. Only the most important messages are included in the figure. The purpose of each message is explained below. In the figure, it is assumed that shared compression and dynamic compression are used and that the compartment has already saved state items in the state handler.

1: *sendMessage*	The worker thread calls the function *sendMessage* of the state object *Idle*.
1.1.1: *getCompressor*	If the message is to be sent compressed, the *Idle* object fetches a compressor from the compressor array using the compartment identifier provided as an argument to the function *sendMessage*.
1.1.2: *compressMessage*	The state *Idle* orders the compressor, which is an instance of the class *LZSSCompressor*, to compress the SIP message provided by the worker thread.
1.1.2.1: *getFeedback*	The LZZS compressor retrieves a feedback item from the state handler. The correct feedback item is identified

	using the compressor's compartment identifier.
1.1.2.2: *saveSharedState*	The LZSS compressor orders the state handler to save the uncompressed SIP message as a shared state.
1.1.2.3.1: *<constructor>*	If the remote UDVM snapshot identifier that is read from the feedback item is empty, the opposite endpoint has not yet acknowledged any UDVM memory snapshots. Therefore, the LZSS compressor creates a new instance of the class *UdvmMemoryImage*, the purpose of which is to present the contents of the remote UDVM's memory.
1.1.2.3.2 *initialise*	The *UdvmMemoryImage* object is initialised by setting the values of UDVM registers and loading the static SIP/SDP dictionary to the circular buffer of the memory image.
1.1.2.3.3: *insertSharedState*	The contents of the shared state are inserted to the UDVM memory image.
1.1.2.3.4: *getSearchBuffer*	The contents of the search buffer of the compression algorithm are read from the memory image. The search buffer is initialised using the retrieved value.
1.1.2.3.5: *insertDecompressedMessage*	The memory image is updated to reflect the situation in which the message this endpoint is about to send has been decompressed. Also the values of affected registers are updated.
1.1.2.4.1: *loadOldMemoryImage*	If the state identifier of the remote UDVM memory snapshot that was read from the *FeedbackItem* object contained a value, the corresponding memory image is retrieved from the compressor's memory image table.
1.1.2.4.2: *<constructor>*	A new *UdvmMemoryImage* object is created.
1.1.2.4.3: *initialiseFromOldImage*	The contents of the old memory image are copied to the new one.
1.1.2.4.4: *insertSharedState*	The memory image is updated to contain the new shared state.
1.1.2.4.5. *getSearchBuffer*	The compressor's search buffer is fetched from the memory image.
1.1.2.4.6: *insertDecompressedMessage*	The memory image is updated to reflect the situation in which the new message has been decompressed.
1.1.2.5: *saveMemoryImage*	The new UDVM memory image is stored in the compressor's memory image table
1.1.2.6: *compress*	The SIP message is compressed using the modified LZSS algorithm.
1.1.2.7: *createSigCompMessage*	The LZSS compressor calls the function *createSigCompMessage* of its super class, *Compressor*. This function creates a SigComp message header and places the compressed SIP message into the payload of the message.
1.1.2.7.1: *getListOfLocalStates*	The *Compressor* object accesses the state handler to get a list of locally available state identifiers. This list is included in the header of the SigComp message.
1.3: *sendToSocket*	The SigComp message that was created is sent to a UDP socket.

1.4: *setNextState*	The *Idle* object sets the next state of the state machine.

7.10.3 Event ReceiveMessage

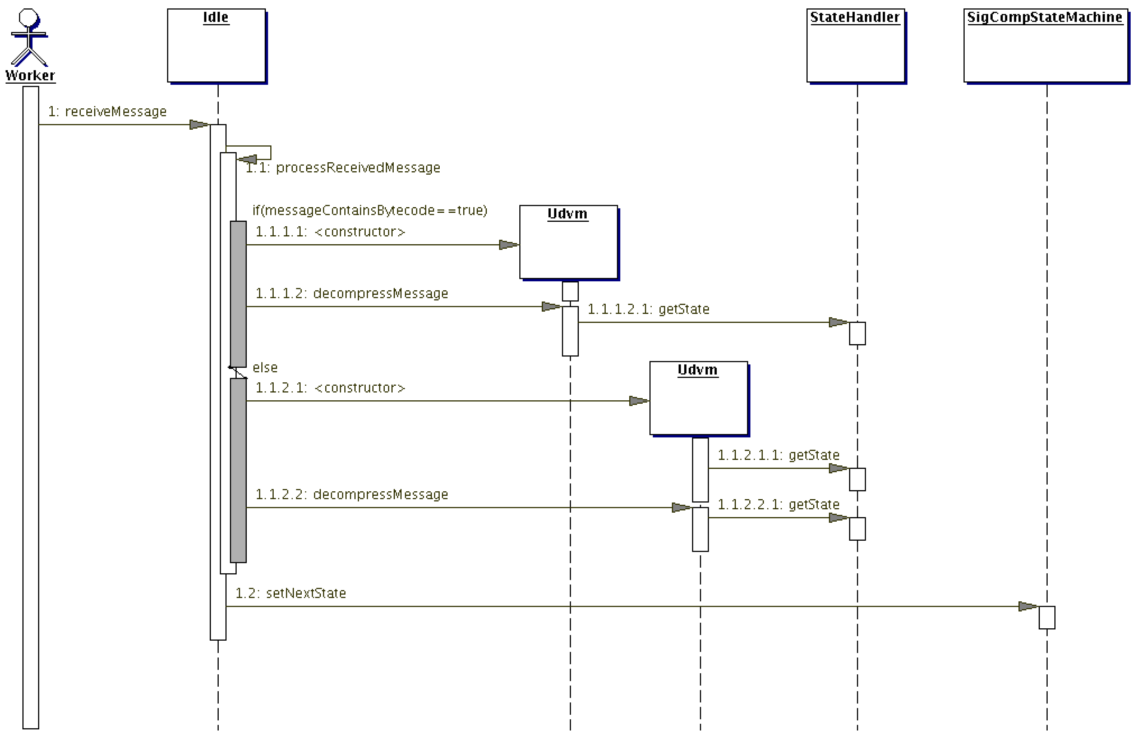


Figure 32 – Sequence diagram for event receiveMessage

Figure 32 illustrates the sequence of actions that takes place when the event *receiveMessage* is received by the state machine of the SigComp prototype.

1: *receiveMessage*	The worker thread calls the function *receiveMessage* of the class *Idle* after having fetched a receive task from the shared buffer.
1.1: *processReceivedMessage*	The function *processReceivedMessage* of the class *Idle* is called. This function either reads the header fields and the payload of the SigComp message or identifies the message to be an uncompressed SIP message. In the latter, case the message requires no processing. Whether a message is a SIP or a SigComp message is determined by inspecting the first five bits of the first byte of the message. If the first five bits are all ‘1’s, the message is guaranteed to be a SigComp message, since this bit sequence never occurs in UTF-8 encoded text messages. If some other bit sequence is found, the message is considered to be a SIP message, in which case the next action taken is the function call 1.2: *setNextState*. Otherwise the next action is the creation of a new *Udvm* object in step 1.1.1.1 or 1.1.2.1.
1.1.1.1: *<constructor>*	If the SigComp message header contained bytecode, a new *Udvm* object is created by calling the constructor that takes

	the bytecode as an argument. This constructor initialises the UDVM's memory from scratch.
1.1.1.2: *decompressMessage*	The payload of the SigComp message is passed to the *Udvm* object created in the previous step. The *Udvm* object starts executing the bytecode.
1.1.1.2.1: *getState*	The bytecode contains an instruction that orders the UDVM to load the static SIP/SDP dictionary from the state handler.
1.1.1.2.2: *getState*	The state handler is accessed to load the shared state.
1.1.2.1: *<constructor>*	If the message does not contain bytecode but instead a partial state identifier of an earlier UDVM memory snapshot, a new *Udvm* object is created by calling the second constructor of the class *Udvm*. This constructor initialises the memory using a previous UDVM memory snapshot, which is loaded from the state handler.
1.1.2.2: *decompressMessage*	The compressed message is handed to the UDVM. The UDVM starts executing the bytecode included in the snapshot loaded from the state handler.
1.1.2.2.1: *getState*	The state handler is accessed to load the shared state. There is no need to load the static dictionary, because it was included in the snapshot loaded by the constructor of the class *Udvm* in step 1.1.2.1.
1.2: *setNextState*	The *Idle* object sets the next state of the state machine, which is the state *Waiting*.

7.10.4 Event ReceiveCompartmentId

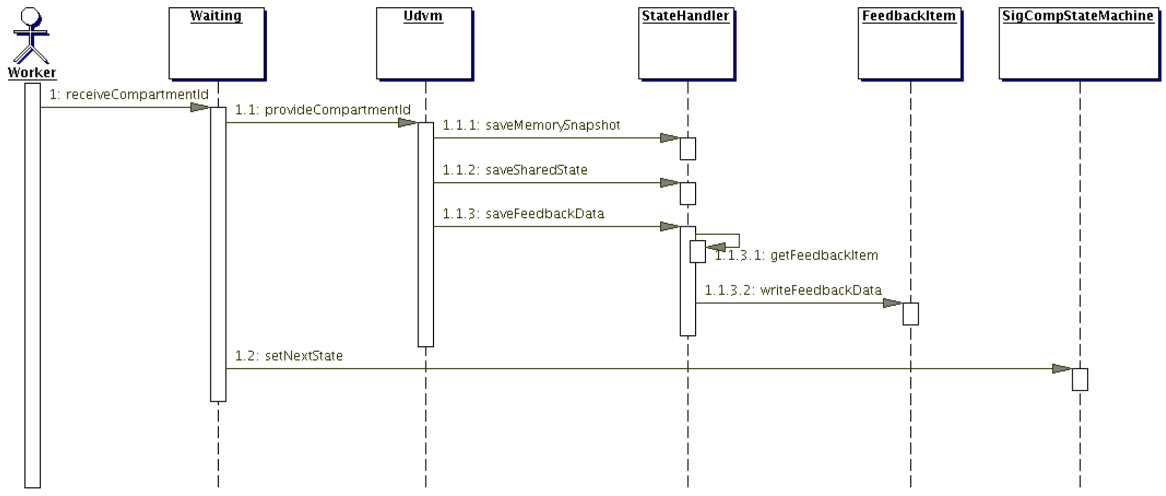


Figure 33 – Sequence diagram for event receiveCompartmentId

Figure 33 shows what happens when the state machine receives the event *receiveCompartmentId*. It is assumed that the compartment already has a feedback item.

1: *receiveCompartmentId*	A worker thread calls the function *receiveCompartmentId* of a *Waiting* object. A pointer to a *Udvm* object and a compartment identifier are passed to the function as arguments.
1.1: *provideCompartmentId*	The compartment identifier is provided to the UDVM.

	This allows the UDVM to save state information.
1.1.1: *saveMemorySnapshot*	The UDVM orders the state handler to create a new state item, which contains the contents of its memory.
1.1.2: *saveSharedState*	The UDVM saves the message that it decompressed in the state handler as a new state item.
1.1.3: *saveFeedbackData*	Next, the UDVM orders the state handler to save the feedback information that the message possibly contained.
1.1.3.1: *getFeedbackItem*	The state handler fetches the feedback item of the compartment.
1.1.3.2: *writeFeedbackData*	The feedback item is updated to contain the most recent feedback information.
1.2.setNextState	The *Waiting* object sets the next state of the state machine, which is the state *Idle*.

7.10.5 Event CloseCompartment

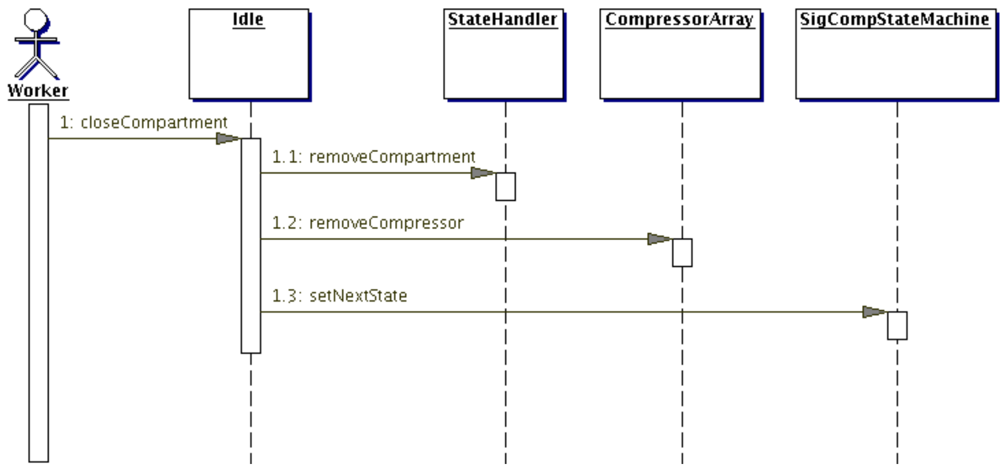


Figure 34 - Sequence diagram for the event closeCompartment

The sequence of actions that takes place when the state machine receives the event *closeCompartment* is illustrated in Figure 34. The purpose of each message in the diagram is explained below.

1: *closeCompartment*	A worker thread calls the function *closeCompartment* of the object *Idle*. The identifier of the compartment that is to be closed is passed to the function as an argument.
1.1: *removeCompartment*	The state items and the feedback item that were created by the specified compartment are removed from the state handler.
1.2: *removeCompressor*	The compressor of the specified compartment is deleted from the compressor array.
1.3: *setNextState*	The next state of the state machine is set.

7.11 Implementation of Extended Operations

In this section, the way the SigComp prototype implements the SigComp extended operations is illustrated through an example of a SIP session establishment message sequence, in which all messages are compressed using SigComp and extended operations. The example is depicted in Figure 35, which contains messages 1-3 and in Figure 36, which includes messages 4-6. The notation used in the figures is presented in Table 7.

Table 7 - Notation that is used in the figures

Notation	Meaning
m1	Message 1
SD	The SIP/SDP Static Dictionary
200 OK (N)	The Nth 200 OK message that is exchanged between the endpoints
ums_A(m1)	UDVM Memory Snapshot (UMS) of a UDVM created at endpoint A, reflecting the situation in which message 1 has just been decompressed.
m2: 180 Ringing [SD + m1]	m2, which contains a compressed 180 Ringing message, is compressed using the SIP/SDP static dictionary and information from message 1

It is assumed both in Figure 35 and Figure 36 that the circular buffers of the UDVMs of endpoint A and B will not reach their maximum sizes and thus all data that are written to the buffer remain there. It is also assumed that the state memories are large enough to hold all the states that are established and that UDP is used for transport.

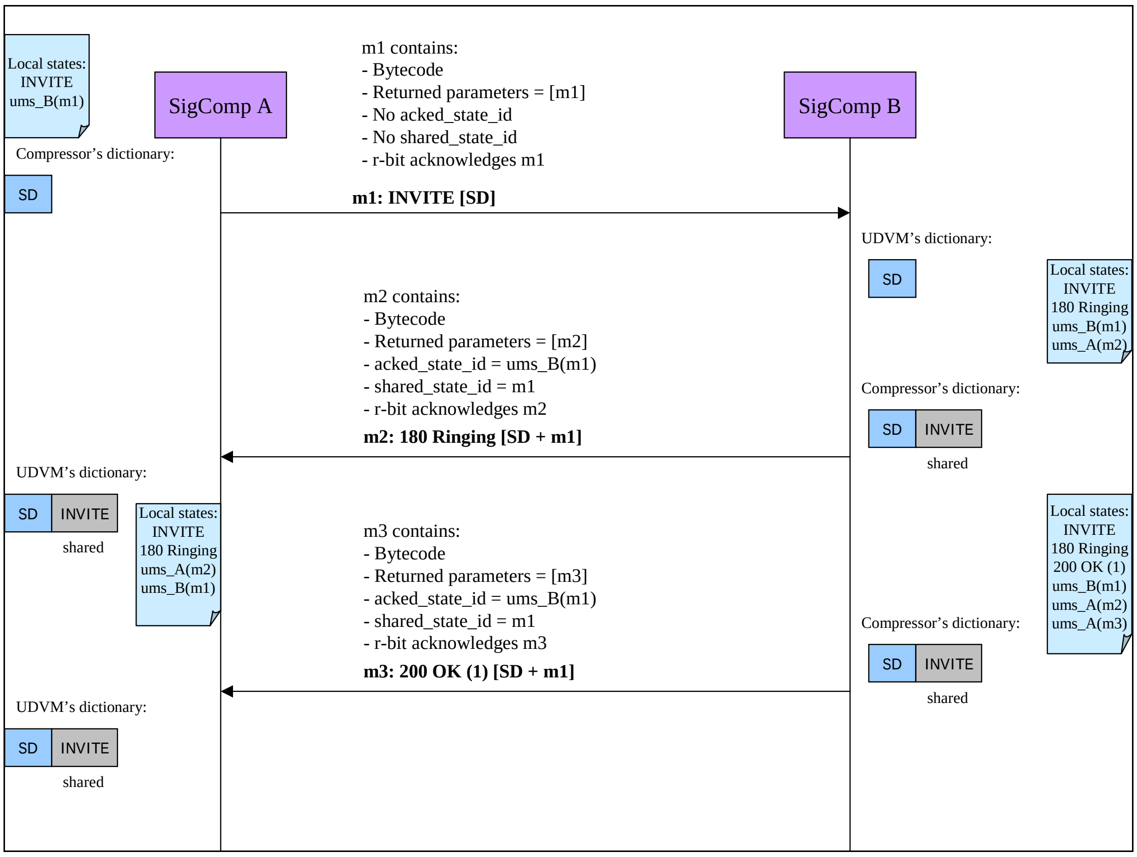


Figure 35 - Extended operations part I

m1 The first message shown in Figure 35 is a compressed INVITE message sent from endpoint A to endpoint B. The process of sending the INVITE begins when the SIP application of endpoint A hands an INVITE message to the SigComp service running at the same endpoint. Since the INVITE is the first message of the compartment, the SigComp message that is created has to contain bytecode, i.e. the decompression algorithm uploaded to endpoint B. The field *acked_state_id* is empty, because there are no states to acknowledge yet, i.e. this endpoint has not saved any UDVM memory snapshots. Shared compression cannot be applied yet because endpoint A has not received any messages from endpoint B. Therefore, the field *shared_state_id* of the SigComp message does not contain a state identifier. The *r*-bit of the SigComp header is set to indicate that the INVITE message was saved at endpoint A. The state identifier of the INVITE message is also included in the list of returned parameters. This allows endpoint B to check the integrity of the shared state by comparing the hash it calculated over the message to the value in the returned parameters. Before compressing the INVITE, endpoint A has to construct an image of the memory of endpoint B's UDVM. This memory image reflects the moment endpoint B has decompressed the message *m1*. Since there are not any shared or acknowledged states available, the only states that are loaded to the memory image are the static SIP/SDP dictionary and the INVITE message. Next, the memory image, called *ums_B(m1)*, is saved by endpoint A. The search buffer of endpoint A's compressor is initialised using the contents of the constructed memory image excluding the INVITE message. Upon receiving *m1*, endpoint B creates a new UDVM instance and initialises it with the bytecode that was included in *m1*. The bytecode loads the static SIP/SDP dictionary to the memory of the UDVM. It also instructs endpoint B to save the decompressed INVITE message and a snapshot of the memory of the UDVM as new states. The compressor of endpoint B will use the decompressed INVITE as a shared state when it compresses the next SIP message it sends. The snapshot can be used by endpoint B to initialise the contents of the memory of a UDVM instance that is invoked to decompress the next SigComp message endpoint B receives.

m2 The next message sent in Figure 35 is *m2*, which contains a compressed 180 Ringing message. Since endpoint B may use a different compression algorithm than endpoint A, a bytecode containing endpoint B's decompression algorithm is included in the message. The only state identifier that is included in the list of returned parameters is that of the next shared state (i.e. the message that is being sent), because (1) the creation of snapshot *ums_B(m1)* is already acknowledged by including its state identifier in the field *acked_state_id* and (2) the creation of shared state, i.e. the INVITE message, is acknowledged by including its state identifier in the field *shared_state_id*. The presence of a state identifier in the field *shared_state_id* also indicates to endpoint A that shared compression was used to compress the 180 Ringing message. The state identifiers are calculated by using the SHA-1 algorithm. Since both endpoints use this algorithm and the same data to calculate the identifiers (i.e. in this case the INVITE message and the UDVM memory image/snapshot), both endpoints are guaranteed to have similar state identifier values, assuming there are no transmission or decompression errors. To utilise shared compression, endpoint B loads the INVITE message, i.e. the most recent message it has received, to the remote UDVM memory image *ums_A(m2)* it constructs. Naturally, also the static SIP/SDP dictionary is loaded to the memory image as well as the 180 Ringing message. Since the search buffer is initialised using the memory image, the search buffer will also contain the shared state. Therefore, the INVITE message will be used in the compression process of the 180 Ringing message. Before sending the message *m2*, endpoint B saves the uncompressed message and the memory image *ums_A(m2)* it created.

Upon receiving *m2*, endpoint A initialises a UDVM instance with the bytecode that is included in the message and starts executing the bytecode. The bytecode loads the static SIP/SDP dictionary to the memory of the UDVM. It also reads the field *shared_state_id*, which contains the state identifier of the INVITE message, and loads this message to the circular buffer of the UDVM. Having decompressed the message, endpoint A saves the uncompressed 180 Ringing message. It also records to the feedback item of this compartment the acknowledged state and shared state identifiers and the indication that the sender has saved the uncompressed message (as indicated by the *r*-bit).

m3 The process of sending the third message, which is a 200 OK sent from endpoint B to endpoint A, is presented in Figure 35. When endpoint B starts to compress the message 200 OK, it cannot know whether endpoint A has successfully received the previous message, *m2*, and created state *ums_A(m2)*, assuming that unreliable transport is used. This is because it has not received anything from endpoint A after it sent the message *m2*. Therefore, endpoint B has to include the bytecode also in message *m3*. The contents of the message are identical to those of *m2*, except for the payload, which this time contains the compressed 200 OK message.

m4 The process of sending the fourth message, *m4*, is shown in Figure 36. Endpoint B acknowledged the creation of the UDVM memory snapshot state *ums_B(m1)* by including the state identifier of this state to the field *acked_state_id* of message *m3* (and message *m2*). Because of this, endpoint A can be sure that this state is available at endpoint B. It uses the state identifier of *ums_B(m1)* to find the corresponding memory image, which was saved when message *m1* was created. Endpoint A updates the image by loading the new shared state, i.e. the 200 OK message, and the message that is to be compressed and sent, ACK, to the circular buffer of the memory image. Because endpoint A utilises an old image in the compression process, endpoint A has to indicate to endpoint B that endpoint B should initialise the UDVM it creates using an old memory snapshot. This is done by writing the state identifier of the old memory image, *ums_B(m1)*, to the field *partial_state_id*. The creation of state *ums_A(m3)* is acknowledged in the field *acked_state_id* and the identifier of the new shared state, 200 OK, is included in the field *shared_state_id*. Returned parameters contain only two state identifiers, that of *ums_A(m2)* and the identifier of the ACK message; all other states available at endpoint A get acknowledged in the other fields. When the ACK message is compressed, the search buffer of the compressor contains the static SIP/SDP dictionary, the INVITE message and the 200 OK message. These are the states that are used in the compression process. Message *m3* does not contain bytecode, because by acknowledging state *ums_B(m1)*, endpoint B also indicated that it has received the bytecode. The reason for this is that because *ums_B(m1)* is a snapshot of the memory of endpoint B's UDVM, it also contains the bytecode. When the state *ums_B(m1)* is loaded to the memory of endpoint B's UDVM, the bytecode comes together with it.

When endpoint B receives the message *m4*, it notices the state identifier the field *partial_state_id* contains. This is the state identifier of the state *ums_B(m1)*. Endpoint B creates a new UDVM instance, retrieves the snapshot state corresponding to the state identifier of *ums_B(m1)* from the state handler, and initialises the UDVM's memory using the snapshot. Therefore, at this point, the contents of the memory of the UDVM contain an exact copy of the memory from the time when message *m1* was decompressed. The circular buffer of the UDVM contains the static SIP/SDP dictionary and the decompressed INVITE. When the UDVM starts executing, the bytecode loads the shared state, which is the decompressed version of message *m3*, to the circular buffer directly after the INVITE message. Now the contents of the memory match exactly to those used when compressing the ACK at endpoint A, and the UDVM can successfully decompress the message. Finally, the bytecode saves the decompressed ACK message and a snapshot of the UDVM's memory, *ums_B(m4)*, as new states.

m5 When endpoint B decides to terminate the session, it sends a BYE message to endpoint A. The BYE is carried in the payload of message *m5* shown in Figure 36. It is assumed that the BYE request and the 200 OK response belong to the same compartment as the previous messages.

Endpoint A acknowledged the UDVM memory snapshot state *ums_A(m3)* in the previous message it sent to endpoint B. It also acknowledged the creation of a state holding the payload of message *m4*, namely the ACK message, by setting the *r*-bit. Therefore, endpoint B can order endpoint A to use the old snapshot state by setting the field *partial_state_id* to contain the state identifier of *ums_A(m3)*. In addition, endpoint B can order endpoint A to load the shared state, ACK, to the UDVM's circular buffer. These are also the states endpoint B uses to construct a new remote UDVM memory image *ums_A(m5)*, and to initialise its compressor's search buffer. Since the snapshot state *ums_A(m3)* contains the static SIP/SDP dictionary and messages *m1* and *m3* (i.e. INVITE and 200 OK), the BYE is compressed using these three states and the shared state, ACK. The state that is acknowledged in message *m5* is *ums_B(m4)*. The returned parameters contain the state identifier of the message that is being sent and the identifier of the state *ums_B(m1)*.

Upon receiving message *m5*, endpoint A initialises its UDVM instance with *ums_A(m3)* and the shared state. Having decompressed the message it saves a new snapshot state *ums_A(m5)* and the decompressed message.

m6 The last message exchanged between the two endpoints is the 200 OK sent from endpoint A to endpoint B. It is carried in the payload of the message *m6* shown in Figure 36. Endpoint A orders endpoint B to initialise the memory of its UDVM with snapshot state *ums_B(m4)* by including the state identifier of this state to the field *partial_state_id*. It also announces that BYE is the new shared state, acknowledges state *ums_A(m5)* and indicates the creation of a state containing the 200 OK (2) message. Returned parameters that are carried in the message *m6* include the state identifier of the 200 OK (2) message and the identifiers of the memory snapshots that were acknowledged previously, *ums_A(m2)* and *ums_A(m3)*. The 200 OK (2) is compressed using the static SIP/SDP dictionary, the messages that are in the circular buffer of *ums_B(m4)*, i.e. INVITE, 200 OK (1), ACK and BYE, and, in addition to these, the new shared state.

When endpoint B receives the message *m6*, it initialises a new UDVM with the memory snapshot state *ums_B(m5)* and the shared state, i.e. BYE, and decompresses the message.

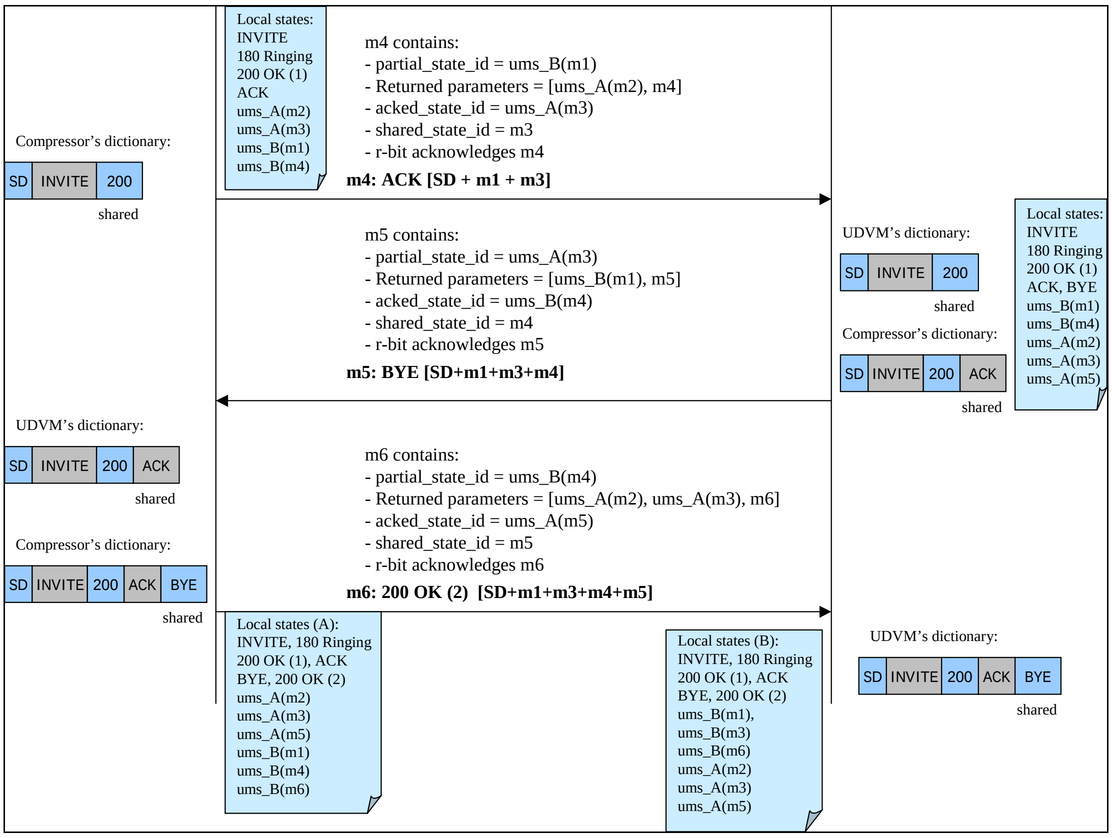


Figure 36 - Extended operations part II

8 Measurements

The approach used to describe the measurements carried out consists of a systematic approach to performance evaluation introduced in [Jain 1991]. This chapter presents the steps of the approach and describes the way they were applied to the measurements of this thesis.

8.1 System Definition

The goal of this study is to measure the performance of the SigComp protocol. The key components under study are SigComp compressor and the UDVM. The test configuration consists of three computers connected to a closed network via a hub as shown in Figure 37. Each of the computers executes either a UE process or a P-CSCF process. The UE process is used to generate compressed SIP signalling traffic initiating from the access network side. The P-CSCF process decompresses the traffic generated by the UE process and forwards the traffic to the core network side. The P-CSCF process also receives SIP traffic coming from the core network side, compresses the traffic and forwards it to the access network side. As shown in Figure 37, computer A acts as the access network side, computer C as the P-CSCF and computer B as the core network side. Both computer A and computer B execute the UE process, while computer C runs the P-CSCF process. The system under study consists of the P-CSCF process running on computer C.

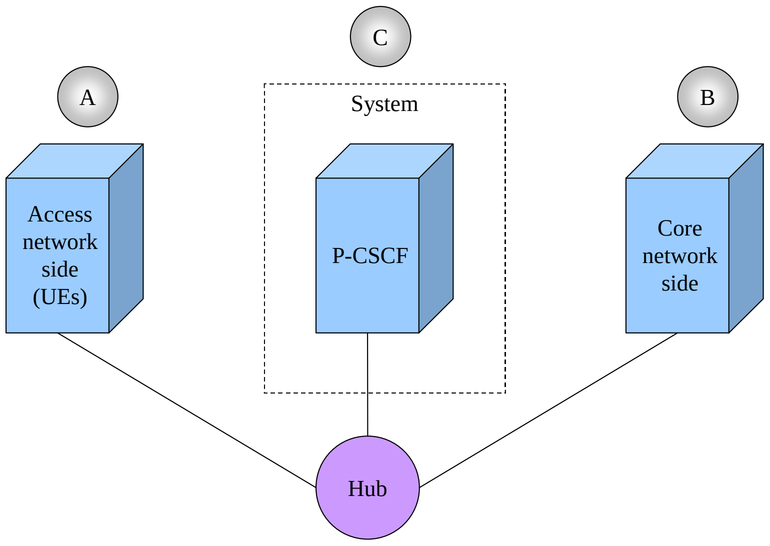


Figure 37 - System definition

8.2 Services

The system offers two services: compression of a SIP message and decompression of a SigComp message. The system either receives a SigComp message from the A-side (access network), decompresses the message and sends the resulting SIP message to the B-side (core network), or receives a SIP message from the core network, compresses the message and sends the resulting SigComp message to the access network. The traffic sent between the access network and the P-CSCF is always compressed, whereas all the traffic exchanged between the P-CSCF and the core network is uncompressed.

8.3 Metrics

For both of the services defined in the previous section, the following issues are studied: (1) responsiveness: the amount of CPU time consumed to compress and decompress a SIP message, (2) productivity: the rate at which the service can be performed, i.e. the throughput of the system, (3) resource utilization and (4) achievable compression ratios. This leads to the following performance metrics: CPU time per compressed message and decompressed message, compressed and decompressed bit rates per unit of time, memory utilization of the SigComp process, state memory utilization of different SigComp mechanisms and finally, the number of bytes sent per compressed message compared to the number of bytes sent per uncompressed message. The compressed/decompressed bit rate is equivalent to the time required to compress/decompress a sequence of *n* bits.

The definition of CPU time used is discussed below. CPU time [Patterson 1998] is the time the CPU spends computing for a particular task and does not include the time spent waiting for I/O or running other programs. It can be further divided into the CPU time spent in the program, called user CPU time and the CPU time spent in the operating system performing tasks on behalf of the program, called system CPU time. The system CPU time depends on the operating system on which the program is run. It may be inaccurate, because of the inaccuracy of an operating system’s self-measurement. However, no program runs without some operating system running on the hardware, so a case can be made for using the sum of user CPU time and system CPU time as the measure of program execution time. In this thesis, the term CPU time always refers to the sum of the user CPU time and the system CPU time.

8.4 Parameters

The system parameters that affect the performance are discussed below. Perhaps the single most important parameter is the speed of the CPU. On the other hand, also the multithreading technique used by the CPU is likely to have an impact, since the SigComp prototype uses multiple threads. Another interesting parameter is the decompression memory size (DMS) of the UDVM. The DMS dictates the length of the circular buffer on the compressor side, meaning that it has a major impact on the efficiency of the compression. Other parameters that are likely to have an effect on performance are the length of the compressor’s look-ahead buffer, the SigComp mechanisms used, the length of shared states, the search technique used by the compression algorithm, the size of the portion of the SIP/SDP static dictionary used and the number of messages that can be compressed and decompressed concurrently.

The workload parameters affecting the performance are time between successive messages, time between calls, total number of calls, the content and sizes of the messages, type of the message sequence, number of messages in the message sequence, and other loads on the CPU.

8.5 Factors

In this thesis, both the performance of the SigComp protocol in general and the performance of the SigComp prototype that was implemented are studied. The key factors chosen for the study of the SigComp protocol’s performance are the SigComp mechanisms used and decompression memory size. However, also the impact of the other factors mentioned in the previous section is studied. The following SigComp mechanisms and their combinations are compared: the basic SigComp protocol, the static SIP/SDP dictionary, dynamic compression and shared compression. The following sizes are used for the decompression memory: 4096, 8192 and 16384 bytes.

The key factors chosen for the study of SigComp prototype's performance are the type of the message sequence and the CPU type used. Two different CPUs are compared: Intel Pentium 4 2.66 GHz and Intel Pentium Hyper-Threading (HT) 3.0 GHz.

8.6 Evaluation Technique

Since a SigComp prototype is implemented as a part of this thesis, measurements will be used for evaluation. One of the most important metrics studied is the CPU time. The use of Linux system calls, like *getrusage()*, to record the CPU time is not enough for our purposes, because the accuracy with which the Linux kernel reports the CPU time consumed by a process is only one millisecond. Therefore, wall-clock time is used as an estimate of the real CPU time consumed. The wall-clock time is monitored using Linux command *strace*, the accuracy of which is one microsecond. Since all the measurements are performed in an unloaded system, the wall-clock time is very close to the actual CPU time. The measurement of memory consumption is carried out by examining the information in file */proc/<$pid>/statm*, which is maintained by the Linux operating system. The throughput of the system is monitored using Ethereal, which is a network protocol analyser.

8.7 Workload

The workload consists of synthetic SIP user agent clients and user agent servers exchanging messages. The user agent clients are located on computer A, while the user agent servers reside on computer B. Seven different session types are examined:

- Basic voice call
- Basic video call
- Registration in a 3GPP release 5 network taken from [3GPP TS 24.228]
- Voice and video call in a 3GPP release 5 network taken from [3GPP TS 24.228]
- An alternative message sequence for voice and video call in a 3GPP release 5 network with a RE-INVITE request
- An alternative message sequence for voice and video call in a 3GPP release 5 network with a RE-INVITE request and reliable delivery of provisional responses
- Push-to-Talk over Cellular (PoC) session

8.8 Experimental Design

The experiments are conducted in three phases. The first and second phases focus on the performance of the SigComp protocol, whereas in the third phase, the performance of the SigComp prototype is measured. In the first phase, the goal is to determine the relative effects of various factors and choose optimal values for these factors. A single-threaded version of the SigComp prototype is used. In the second phase, the factors determined during the first phase are used to measure compression time, decompression time and compression ratios of the session types defined in the previous section. In the third phase, the factors determined during the first phase are used to measure the performance of the multi-threaded SigComp prototype.

8.9 Data Analysis

Analysis of variability will be used where appropriate to take into account the variability of the results.

8.10 Data Presentation

The final results will be presented in graphic form and also in table form where necessary. The results of the measurements are included in Appendixes A-N.

8.11 Materials and Apparatus

The first and second phases of the measurements are conducted on an Intel Pentium Hyper-Threading 3.0 GHz platform. In the third phase of the measurements a three-computer 100 Mbit/s Ethernet network is constructed. The computers are connected via an OfficeConnect Dual Speed Hub 8 10/100 Mbit/s hub. One of these computers acts as a P-CSCF, one executes the UEs and one acts as the core network side. The CPU of the computer acting as the P-CSCF is either an Intel Pentium 3,0 GHz supporting the Hyper-Threading technology or an Intel Pentium 4 2.66 GHz. The CPU of the computer executing the UEs is an Intel Pentium Mobile 1.6 GHz and that of the computer acting as the core network side is an Intel Pentium III 600 MHz. The computer with the least CPU power is used on the core network side because it does not need to compress or decompress SigComp traffic. All the computers use SuSE Linux as the operating system. The C++ compiler that is used is GNU project C and C++ compiler version 3.3.3. The POSIX threads library is used to implement threads. The computers used in the measurements are presented in Table 8.

Table 8 - Computers used in the measurements

	Intel Pentium 4 Hyper-Threading 3,0 GHz	**Intel Pentium 4 2,66 GHz**	**Intel Pentium 4 1.8 GHz**	**Intel Pentium Mobile 1,6 GHz**	**Intel Pentium III 600 MHz**
Role in the measurements	P-CSCF	P-CSCF	-	UEs	Core network side
Operating system	SuSE Linux 9.2 professional	SuSE Linux 9.0 professional	SuSE Linux 9.1 professional	SuSE Linux 9.1 professional	SuSE Linux 9.0 professional
Linux kernel version	2.6.8-24.14-smp	2.4.21-99-default	2.6.4-52-default	2.6.5-7.151-default	2.4.21-99-default
Main memory	1024 MB	504 MB	1024 MB	512 MB	192 MB
Free main memory	783 MB	316 MB	33 MB	296 MB	8 MB
Type of memory	DDR 400 MHz	DDR 333 MHz	DDR 333 MHz	DDR 333 MHz	SDRAM
L2 Cache memory	1 MB	512 KB	512 KB	2 MB	256 KB
Front side bus	800 MHz	400 MHz	400 MHz	400 MHz	100 MHz

8.12 Assumptions

In this section, some common values and assumptions that are used throughout the measurements are listed. Firstly, it is always assumed that partial state identifiers are of the minimum length, i.e. six bytes. A complete list of locally available state items is always included in the SigComp messages whenever stateful compression is applied. State memory size, i.e. the number of bytes offered to a particular compartment for the creation of state, is set to the maximum value, 131072 bytes. The parameter cycles_per_bit, which specifies the number of UDVM cycles available to decompress each bit in a SigComp message is set to value 32.

9 Phase One – Effects of Different Factors

The goal of the measurements performed in the first phase is to determine the relative effects of various factors and choose optimal values for these factors. In the measurements, different combinations of the following SigComp mechanisms are used: the static SIP/SDP dictionary, dynamic compression and shared compression. The signalling flow is taken from [3GPP TS 24.228]. The flow is illustrated in Figure 1 and its messages are shown in Table 9. The messages are those exchanged between a UE and a P-CSCF during the establishment of a video call in a 3GPP Release 5 network. All the measurements of the first phase are performed on the Intel Pentium 4 3.0 GHz platform unless otherwise stated.

Table 9 - Message sequence

Message number	Message	Length [bytes]
1	INVITE	1437
2	100 Trying	254
3	183 Session progress	1440
4	PRACK (1) to 183 Session progress	1318
5	200 OK (1) to PRACK (1)	904
6	UPDATE	1291
7	200 OK (2) to UPDATE	865
8	180 Ringing	563
9	PRACK (2) to 180 Ringing	717
10	200 OK (3) to PRACK (2)	260
11	200 OK (4) to INVITE	1133
12	ACK	358

9.1 Linear Search versus Hashing

Two different versions of the LZSS compressor were implemented for the purposes of this thesis. The first version uses linear searching, while the second version uses a hash table to speed up searches. The goal of this measurement is to compare these two approaches.

The results of the measurements are shown in Table 10. The values for the compression times are averages calculated over seven measurements. The results imply that the hash table is 2.8 times faster than linear searching. However, it uses six times more memory than linear searching does.

Table 10 - Linear search versus hashing

Approach	Compression time [ms]	Memory requirement for N-byte message [bytes]
Linear search	32,12	N
Hashing	11,47	6*N

Before analysing the differences between hashing and linear search, the compressibility of the message sequence used in the measurements is examined. The compression ratio for each message in the sequence of 12 SIP messages is shown in Figure 38. Naturally, the compression ratios are identical for both linear searching and hashing since the compression algorithm does not change. The compression ratios shown in Figure 38 were calculated without taking into account the overhead added by the SigComp protocol. We can observe that there are two messages that cannot be compressed as well as the others, namely messages 1 and 3. Message 1 is the initial INVITE sent from endpoint A to endpoint B. The only information that can be utilised in the compression of the INVITE is the SIP/SDP static dictionary, which explains the low compressibility of the message. Message two, i.e. 100 Trying, is the first message sent from endpoint B to endpoint A. In addition to the static dictionary, also the INVITE message can be used when the 100 Trying is compressed. Because of the similarity of the content of the INVITE and the content of the 100 Trying, compression is efficient. Message number 3, i.e. 183 Session progress, sent from endpoint B to endpoint A is compressed exactly the same way as the 100 Trying. However, this time there are not as much similarities between the content of the INVITE and the content of the 183 Session progress. Therefore, the compression is less efficient than in the case of the second message. All the remaining messages starting from the fourth message achieve relatively good compression ratios. This is because dynamic compression and shared compression can be efficiently utilised as the amount of state information grows.

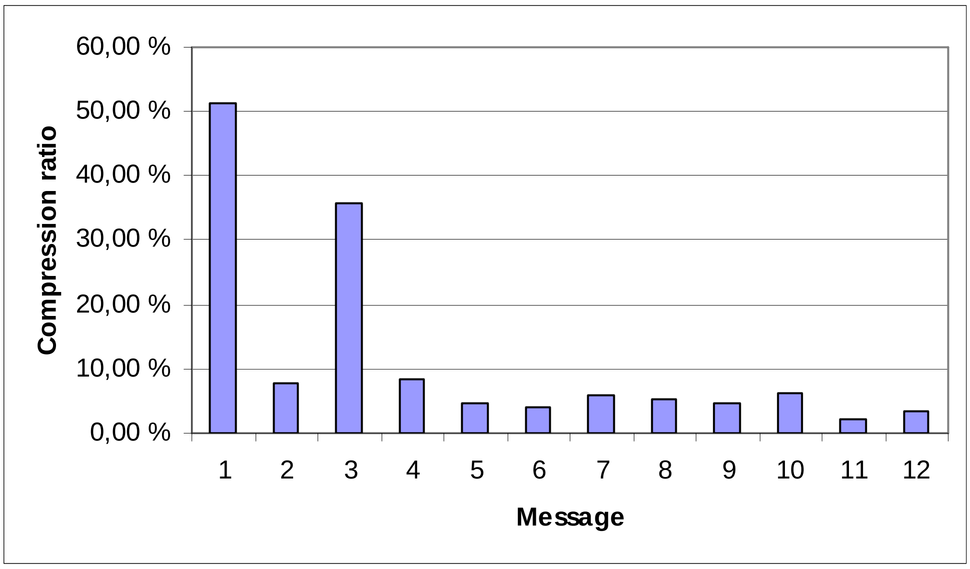


Figure 38 - Compression ratio of each message without SigComp overhead

The compression time of each message is shown in Figure 39. The values in Figure 39 are averages calculated over seven measurements and include both linear search and hashing. We can observe that the compression of the messages 1, 3, 6, 7 and 8 seems to constitute a bottleneck for linear searching. Linear search compares the look-ahead buffer with each of the positions in the search buffer and selects the maximum match. The worst case of linear searching is performed in *O(NM)*, where *N* is the size of the search buffer and *M* the size of the look-ahead buffer. However, in most cases linear searching can complete in *O(M+N)* time. Linear searching is inefficient when (a) there are only few matches in the search buffer, or (b) the matches are located in such a position that a lot of searching is required to find them (for instance near the end of the buffer). In case of the first and the third messages, most of the content in the search buffer is useless, which results in poor performance when linear searching is applied. When messages 6, 7 and 8 are compressed, most of the useful content, i.e. the most recent received and sent messages, are near the end of the search buffer. Therefore many comparisons are required and the compression consumes a lot of time. On the other hand, we can also observe that linear searching performs better than hashing in the case of messages 2, and 9-12. Messages 2 and 10 are fast to compress because they are the smallest ones in the sequence, 254 and 260 bytes. Linear searching performs well for messages 9, 11 and 12 firstly because most of the content that can be utilised in the compression is located at the beginning of the buffer. Secondly, these are also messages with very good compression ratios as can be seen from Figure 38. Compression is fast because long matches can be found in a short time.

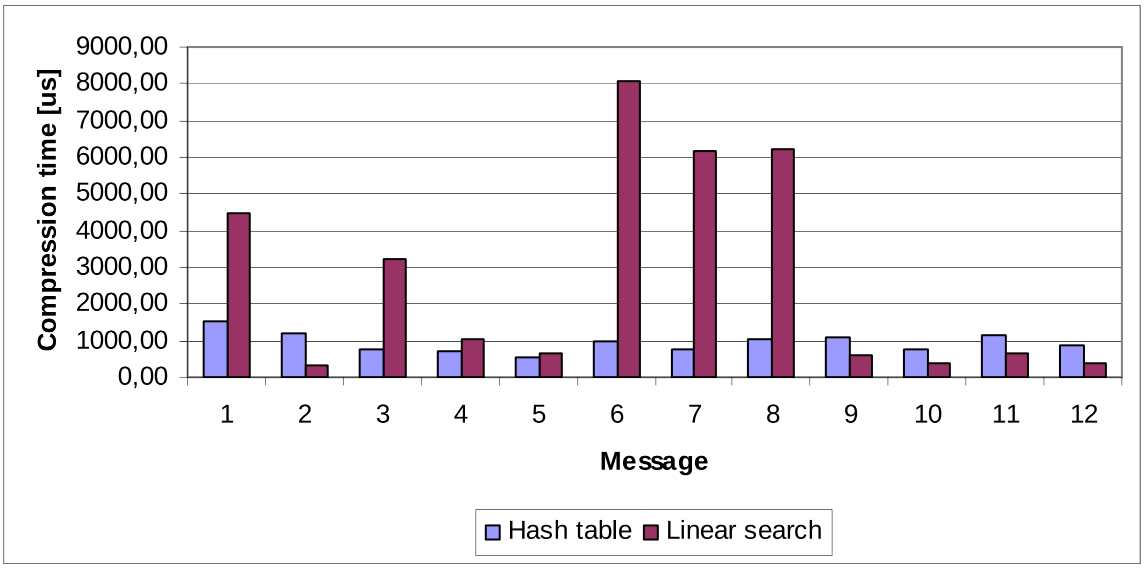


Figure 39 - Linear search versus hashing

Thus, in some cases linear searching can outperform hashing, which is rather surprising. The reason for this is that when a hash table is used, most of the compression time is spent in organizing the table, the average being roughly 60%. This is illustrated in Figure 40, which shows the building blocks of the total compression time for our SIP sequence. When messages 1 and 2 are compressed, the hash table must be constructed from scratch. Both the static dictionary and the compressed message need to be inserted, which takes a lot of time. The other case in which hash table updates require a lot of time is when content must be deleted in order to make room for new one. Such behaviour takes place when messages 8-12 are compressed and partially when messages 6-7 are encoded. Deletion of old content is the reason hashing becomes slower than linear searching. However, if the search buffer were bigger, hashing would most likely outperform linear searching under all circumstances. In the rest of the measurements presented in this thesis, the compression algorithm utilising hashing is used.

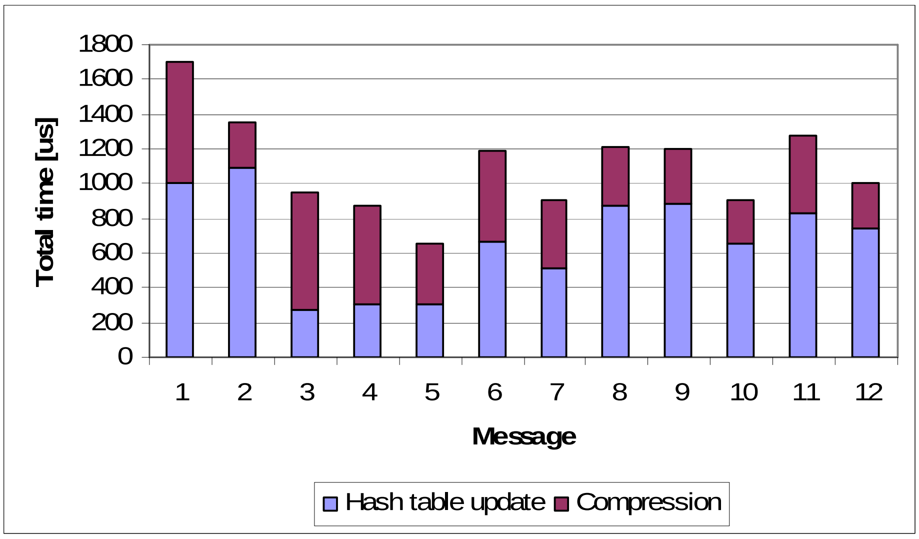


Figure 40 - Hash table updates

9.2 Length of Look-ahead Buffer

The purpose of this measurement is to determine the optimal size for LZSS compressor's look-ahead buffer. The average values for compression and decompression times were calculated over seven measurements. The size of the look-ahead buffer depends on the number of bits the compressor uses to encode the length of matches and can be calculated as follows:

$$2^n + 2, \qquad (9.1)$$

where n is the number of bits used to encode the length of matches. The maximum length of a match equals to the size of the look-ahead buffer. Thus, when 4 bits are used to encode length values, the maximum length of a match is 18 bytes. The more bits are used, the longer the offset/length pairs become. Therefore, it is wasteful to allow very long matches. Three look-ahead buffer lengths are examined: 18, 66 and 258 bytes. Matches longer than 258 bytes are unlikely to occur in SIP messages. On the other hand, if the maximum length of a match is restricted to less than 18 bytes, the compression becomes highly inefficient.

The results are shown in Table 11. We can observe that the compression ratio improves as the length of the look-ahead buffer grows. This is because we can encode longer matches using a single offset/length pair. Secondly, both the compression and decompression times decrease when the length of the buffer increases. When longer matches are allowed, the amount of UDVM cycles consumed to decompress a message is less than in the case of shorter matches. This is because the UDVM can retrieve longer sequences from its circular buffer during a single fetch operation. When compressing, the use of longer matches means that to find a match of maximum length, more bytes in the search buffer need to be compared with the bytes in the look-ahead buffer, i.e. more work needs to be done. However, the use of longer matches also means that less search operations are required in total. This has the effect of slightly reducing the compression time. Based on the results, a look-ahead buffer of length 258 appears to be the best choice. All the remaining measurements presented in this thesis use a look-ahead buffer of this size.

Table 11 - Length of the look-ahead buffer

Size of look-ahead buffer [bytes]	Bits in length values [bits]	Compression ratio	Compression time [ms]	Decompression time [ms]
18	4	0,224	11,97	14,47
66	6	0,164	11,58	11,75
258	8	0,155	11,47	11,27

9.3 Length of Shared States

The SigComp prototype described in this thesis uses fixed lengths for shared states in order to simplify the calculation of shared state lengths in the bytecode. In this measurement, the optimum length for the shared states is determined. Four different lengths are used: 500, 750, 1000 and 1500 bytes in such a way that if the maximum length of the shared state is *n* and the state that is used is longer than *n*, only *n* first bytes of the shared state are inserted to the circular buffer. On the other hand, if the shared state is shorter than *n*, the rest of the *n*-byte sequence in the circular buffer reserved for the shared state is left untouched. The results of the measurements are show in Table 12. The average values for compression and decompression times were calculated over seven measurements. The compression ratios presented in Table 12 were calculated without taking the SigComp overhead into account. By studying the results of Table 12, we can observe that as the size of the shared states increases, the compression ratios improve, compression time increases and decompression time decreases. Compression ratios are better when longer shared states are used because there is more previous data against which to compress. Decompression time decreases slightly because better matches are found and the UDVM needs to do less work. However, compression time increases because once the buffer has become full, previous data need to be deleted from the hash map to make room for new data. The longer shared states are used the more data must be deleted. If the maximum length of shared states is 1500 bytes, 1500 entries must be deleted from the hash map in the worst case. Based on these results, it seems that the most appropriate value for the maximum length of the shared state depends highly on the context of use. If CPU time is not an issue, large shared states should be used to obtain the best compression ratios. However, if CPU time is scarce, it is recommendable to use shorter shared states.

Table 12 - Shared state length

Shared state length [bytes]	Compression ratio	Compression time [ms]	Decompression time [ms]
500	0,155	11,47	11,27
750	0,148	13,40	10,76
1000	0,141	14,68	10,74
1500	0,128	17,85	10,62

9.4 Static Dictionary Priorities

Each string in the SIP/SDP static dictionary has a priority ranging from one to five. The LZSS algorithm offers an increased efficiency when the most commonly used strings are located at the bottom of the dictionary. The goal of this measurement is to determine the effects of different static dictionary priorities on compression ratios, compression time and decompression time. The average values for compression and decompression times were calculated over seven measurements.

The results of the measurements are reported in Table 13. The values presented for compression ratios do not include the SigComp overhead. The results indicate that as a bigger portion of the dictionary is used, compression ratios improve and compression time and decompression time increase. Based on these results, it appears that the best performance is achieved by using priorities 1-2 or 1-3, provided that we wish to minimise compression and decompression times and still achieve satisfactory compression ratios.

Table 13 - Static dictionary priorities

Priorities	Length of static dictionary [bytes]	Compression ratio	Compression time [ms]	Decompression time [ms]
1 only	218	0,194	7,03	11,43
1-2	1132	0,164	7,73	10,98
1-3	1492	0,161	8,11	10,81
1-4	3335	0,154	11,01	11,10
1-5	3468	0,155	11,47	11,27

The average compression time of each message in the sequence of 12 SIP messages is shown in Figure 41. The use of different priorities seems to have the greatest impact in the case of messages 1 and 2, the first messages sent in each direction. This is because when these messages are sent, the search buffer has to be constructed from scratch. The more priorities are used, the more insertions must be done to the hash table and the more time is required. We can also observe that the compression time of messages 6-12 is considerably longer for priorities 1-4 and 1-5 than for the other priorities. The reason for this is that when a bigger portion of the static dictionary is used, the circular buffer becomes full earlier; at endpoint A this occurs during the compression of message 6 and at endpoint B during message 7. As soon as the buffer becomes full, time must be spent deleting entries from the hash map to make room for new ones. Finally, although the use different static dictionary priorities has a clear impact on the compression time, the same is not true for the decompression time; it was observed that the effect of different priorities on the decompression time is negligible.

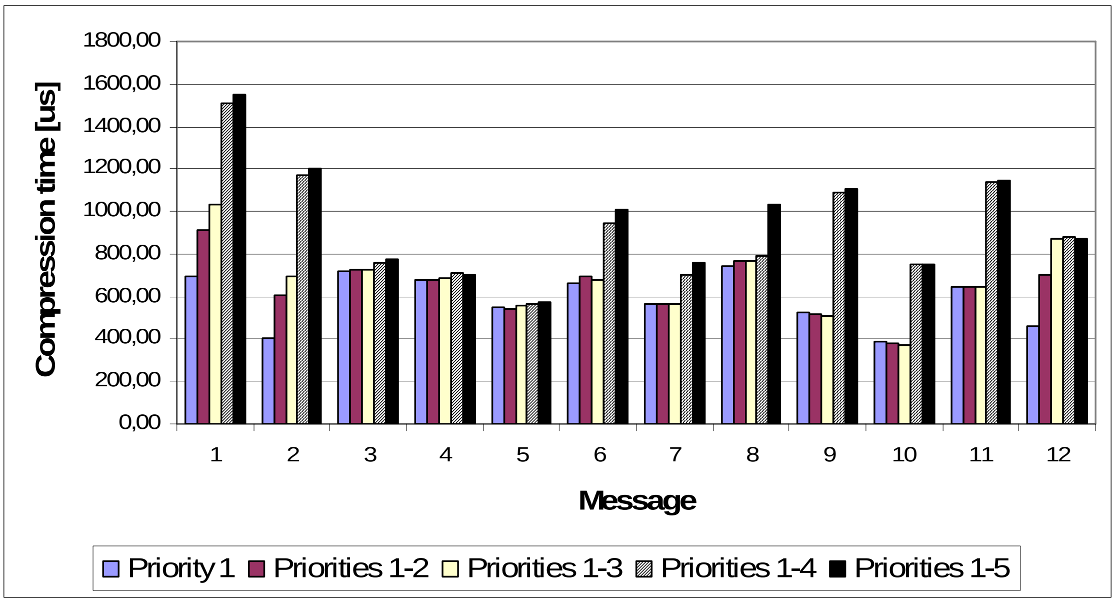


Figure 41 - Effects of different priorities on messages

9.5 Secure Hash Algorithm

The use of both dynamic and shared compressions necessitates the calculation of SHA-1 hashes both at the compressing and decompressing endpoints. In dynamic compression, an SHA-1 hash is calculated over the contents of UDVM's memory and in shared compression, another SHA-1 hash is calculated over the SIP message. The purpose of this measurement is to analyse the performance of the SHA-1 algorithm, which was implemented as part of the SigComp prototype.

The results of the measurement are shown in Figure 42. In the measurement, SHA-1 hashes were calculated over three UDVM memory snapshots of lengths 4096, 8192 and 16384 bytes and six SIP messages having lengths 260, 458, 717, 904, 1133 and 1440 bytes. The values shown in Figure 42 are averages calculated over ten measurements. From the figure, we can observe that for a 1440-byte message and a DMS of 8192 bytes, the combined overhead added by the calculation of an SHA-1 hash over the message and over the UDVM's memory is 580 microseconds. This is roughly 25 percent of the total compression time. For the other messages, the calculation of the SHA-1 hashes requires 30-41 percent of the total compression time. A similar impact is experienced on the decompressing side. Thus, we can conclude that especially in the case of shared compression, a considerable part of the total compression time is spent in calculating the SHA-1 hash values. We can expect that because of the calculation of SHA-1 hash values, dynamic and shared compressions are likely to use more CPU time than basic and static compressions.

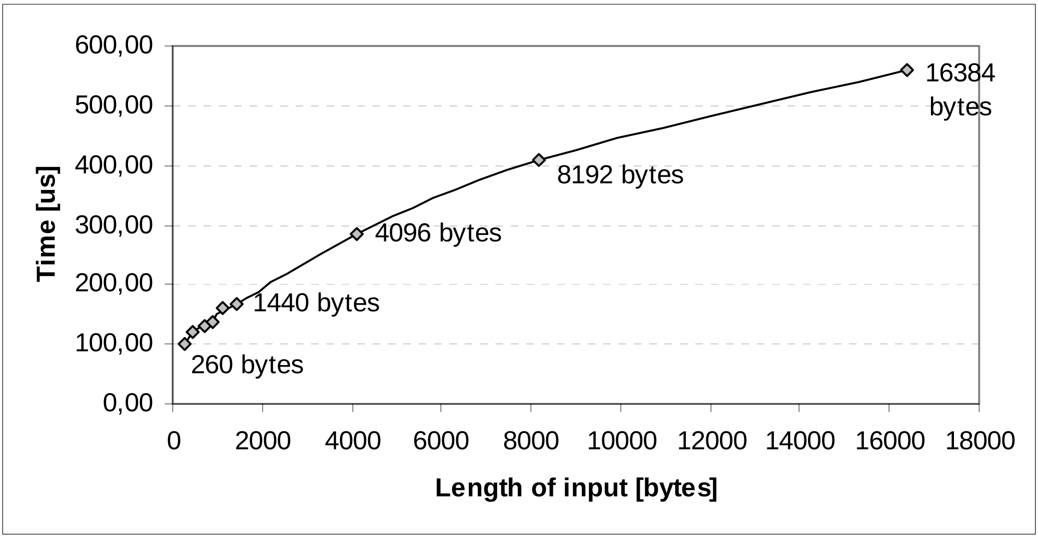


Figure 42 - Calculation time of SHA-1 hash

9.6 SigComp Mechanisms

The SigComp mechanisms studied in this section include stateless and stateful basic compressions, stateless and stateful static compressions, dynamic compression and shared compression. In stateless basic compression, only basic, i.e. message-by-message compression is applied. State information is not stored, meaning that bytecode must be provided in the header of each SigComp message. In stateful basic compression, the only state information that is stored is the bytecode, which therefore needs to be provided only in the header of the first messages. In stateless static compression, basic compression is applied together with the SIP/SDP static dictionary. Bytecode is provided in each message. In stateful static compression, the bytecode is stored and provided only in the first messages. In dynamic compression, the SIP/SDP static dictionary and dynamic compression are used. Finally, in shared compression, the SIP/SDP static dictionary, dynamic compression and shared compression are used.

The compression ratio that each mechanism achieves for the 3GPP session initiation sequence is shown in Figure 43 and Table 14. The SigComp overhead like the bytecode is included in the values shown in the figure and the table. Since the purpose here is to compare various SigComp mechanisms, maximum values were used for the factors affecting the compression efficiency of these mechanisms. In particular, the messages were compressed against the entire SIP/SDP static dictionary (i.e., priorities 1-5) and the shared state length was set to 1500 bytes. Three different values were used for the DMS: 4096, 8192 and 16384 bytes. In addition, it was assumed that the underlying transport layer protocol is unreliable. The session initiation sequence was taken from [3GPP TS 24.228] and it contains the messages exchanged between a UE and the P-CSCF.

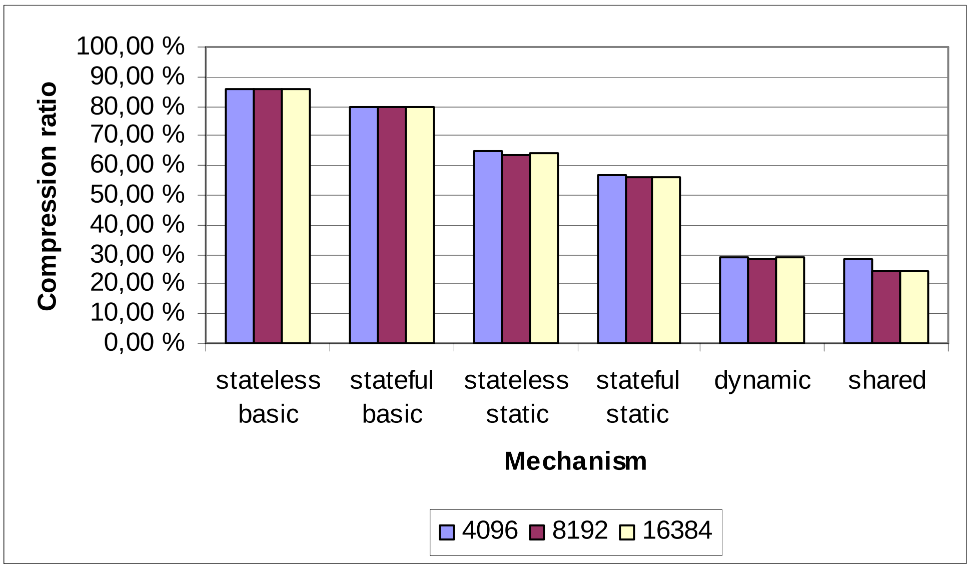


Figure 43 - Compression ratios, UDP, SD priorities 1-5 and shared state length 1500 bytes

From Figure 43, we can observe that both the stateless and stateful versions of the basic compression scheme offer very poor compression ratios and are thus practically useless. Stateless and stateful static compressions offer a clear improvement over the basic compression scheme. The best compression ratio achieved by the stateful static compression is 0.56, while that of stateful basic compression is 0.80. Thanks to static compression, we can reduce the size of the messages being compressed by referring to strings in the SIP/SDP static dictionary.

Table 14 - Compression ratios, UDP, SD priorities 1-5, shared state length 1500 bytes

Compression mechanism	Compression ratio [%]		
	DMS 4096 bytes	DMS 8192 bytes	DMS 16384 bytes
Stateless basic compression	85,75	85,75	85,86
Stateful basic compression	79,92	79,92	79,94
Stateless static compression	65,02	63,83	63,93
Stateful static compression	57,03	55,84	55,85
Dynamic compression	29,16	28,69	28,81
Shared compression	28,21	24,04	24,09

The next mechanism shown in Figure 43 is dynamic compression, which is able to achieve a compression ratio of 0.29. This is enabled by the ability of dynamic compression to use previously sent messages in the decompression process; it is possible to substitute portions of the message being compressed with pointers to previously sent messages. The final improvement is offered by shared compression, which achieves a compression ratio of 0.24 by using also received messages in the compression process. Shared compression is clearly the most efficient compression scheme with regard to achievable compression ratios.

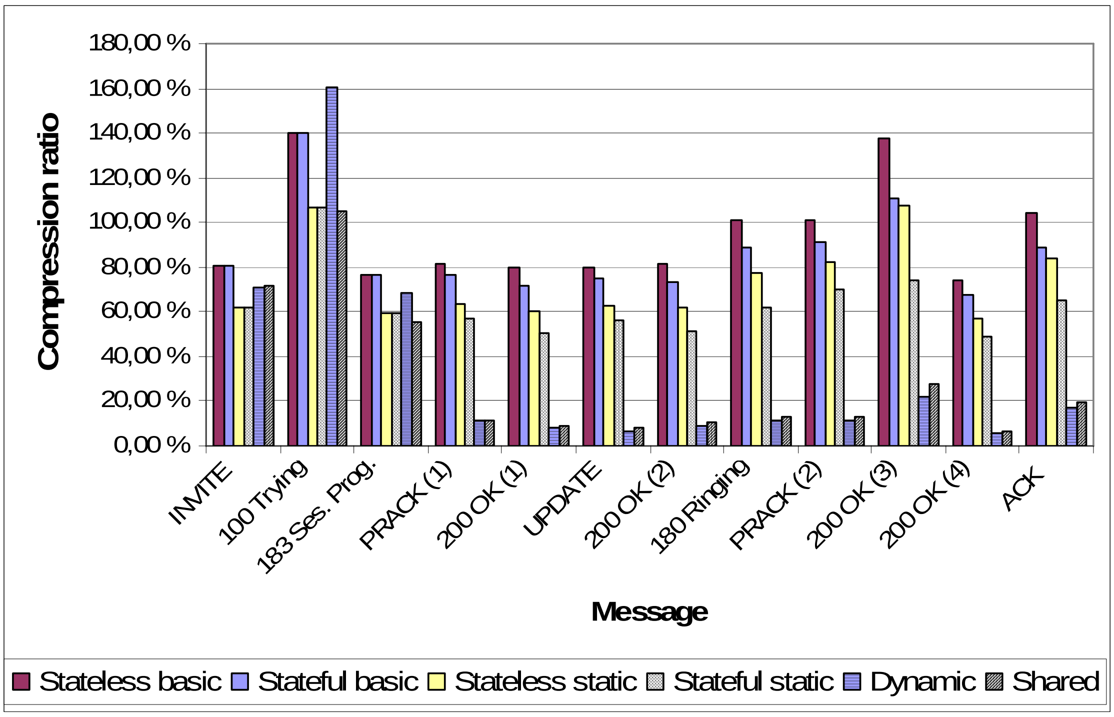


Figure 44 - Compressibility of SIP messages

The compressibility of each SIP message in the 3GPP session initiation sequence is shown in Figure 44. The picture clearly indicates the superiority of dynamic and shared compressions, especially when PRACK (1) and the later messages in the sequence are compressed. The first three messages in the sequence, INVITE, 100 Trying and 183 Session Progress have rather modest compression ratios despite of the mechanism used. This is firstly because the bytecode has to be sent in the header of the first three SigComp messages, assuming that unreliable transport is used. The bytecode introduces an overhead of 69-220 bytes depending on the mechanism and decompression memory size used, as shown in Table 15. Secondly, there is not much state information available when the first three messages are compressed. When the INVITE is processed, only the static dictionary can be used in the compression process. Dynamic compression does not have an effect until the fourth message in the sequence. However, shared compression can be applied already to the second and the third messages. The second message, 100 Trying is the most difficult message for all of the mechanisms, because it is the shortest message in the sequence and the first message sent from endpoint B to endpoint A. The size of the 100 Trying is 245 bytes while the size of the SigComp overhead is 79-249 bytes depending on the mechanism used. This means that in the worst case, the size of the SigComp header of the compressed 100 Trying message is bigger than the entire uncompressed 100 Trying message.

Table 15 - Bytecode lengths

Mechanism	Length of bytecode [bytes]		
	DMS 4096 bytes	DMS 8192 bytes	DMS 16384 bytes
Stateful and stateless basic compressions	69	69	70
Stateful and stateless static compressions	86	86	87
Dynamic compression	216	216	220
Shared compression	216	216	220

We can observe from Figure 44 the poor performance of stateless and stateful basic compressions. When the stateless basic compression is applied, there are five SigComp messages that have almost the same size or are longer than the uncompressed SIP message. Static compression offers a clear improvement over the basic compression scheme, but still achieves only modest compression ratios. Another interesting finding we can make from Figure 44 is that in the case of messages 4-12, dynamic compression performs actually better than shared compression. This is because shared compression uses much more buffer space than dynamic compression and has to overwrite the static dictionary and the oldest messages in the buffer during its second iteration over the circular buffer. When a DMS of 8192 bytes is used, shared compression uses 15129 bytes of buffer space at endpoint B, while dynamic compression consumes only 9129. In practice, this means that dynamic compression can keep almost its entire dictionary in the circular buffer at once.

The improved compression ratios of the more advanced compression mechanisms do not come without a cost. This can be observed from Figure 45, which shows the average compression times of basic, static, dynamic and shared compressions. The average compression times were calculated over ten measurements. When the size of the decompression memory is 8192 bytes, shared compression requires twice as much time as basic compression, whereas dynamic compression is 1.4 times slower than basic compression. Static compression consumes more time than basic compression because the static dictionary needs to be hashed in the search buffer. The difference between the compression times of these two mechanisms is small because even though the hashing of the static dictionary consumes some time, the dictionary also helps to save time because longer substitutions can be used. Encoding one long match is in most cases faster than encoding many short matches because less searching is required. The compression time of dynamic compression is clearly longer than that of static or basic compressions, because dynamic compression requires additional time for the calculation of SHA-1 hashes over the UDVM memory images. Shared compression requires even more time than dynamic compression because one additional SHA-1 message digest needs to be calculated for the shared state and because the shared state needs to be hashed in the search buffer. Dynamic and shared compressions consume also time when they delete content from the search buffer after it has become full.

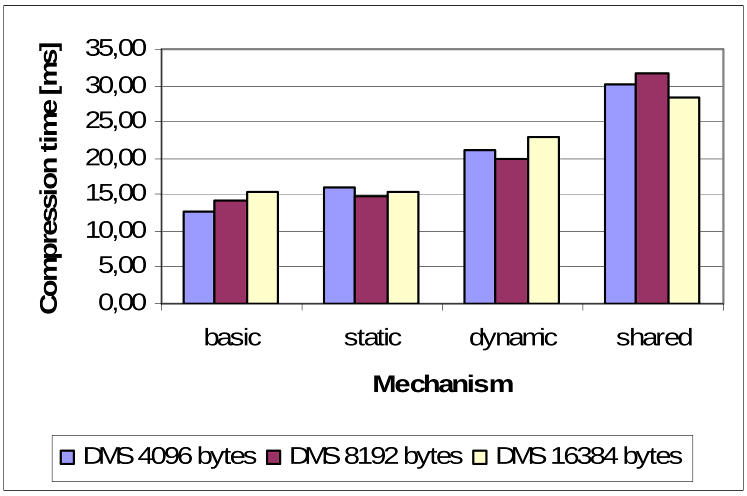


Figure 45 - Compression time

The average decompression time of the compressed 3GPP session initiation sequence is shown in Figure 46 for basic, static, dynamic and shared compressions. The averages were calculated over ten measurements. The results imply that in the case of decompression memory sizes of 4096 and 8192 bytes, the decompression of the output of dynamic compression consumes the least time. This is because the output of dynamic compression is better compressed than the output of basic and static compressions. The higher the compression ratio, the more pointers the compressed sequence contains and the longer are the strings that these pointers substitute. Because longer matches can be fetched from the dictionary during a single fetch operation, less UDVM cycles are required to decompress the entire message. Even though the UDVM must calculate an SHA-1 hash over its memory when dynamic compression is applied, the performance improvement enabled by longer matches is more than enough to cover this additional cost. However, when the shared compression scheme is used, the UDVM needs to calculate another SHA-1 hash for the shared state. Therefore the decompression consumes more time than in the case of dynamic compression.

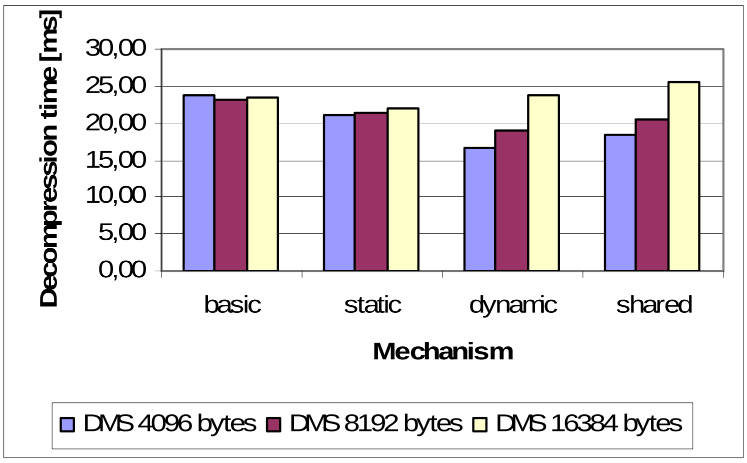


Figure 46 - Decompression time

The state memory usage of dynamic compression and shared compression is shown in Figure 47 for the case that the 3GPP session initiation sequence is being compressed.

The state memory consumption of stateless and stateful basic and static compressions is not shown in the figure, because it is negligible. Since the only state information these mechanisms save is the bytecode, the state memory usage of both of them is below 100 bytes. The size of UDVM memory images constructed by the compressor are not included in the state memory usage. The maximum amount of state memory was limited to 131 kilobytes per compartment. However, neither dynamic nor shared compression reached this limit. We can observe from the figure that for instance with a decompression memory size of 4096 bytes, shared compression uses roughly 1.4 times more state memory than dynamic compression. This is because when shared compression is used, shared state items need to be created and stored. Since the shared state items consist of uncompressed messages, the difference in the state memory usage between dynamic compression and shared compression equals to the size of the SIP message sequence for all decompression memory sizes.

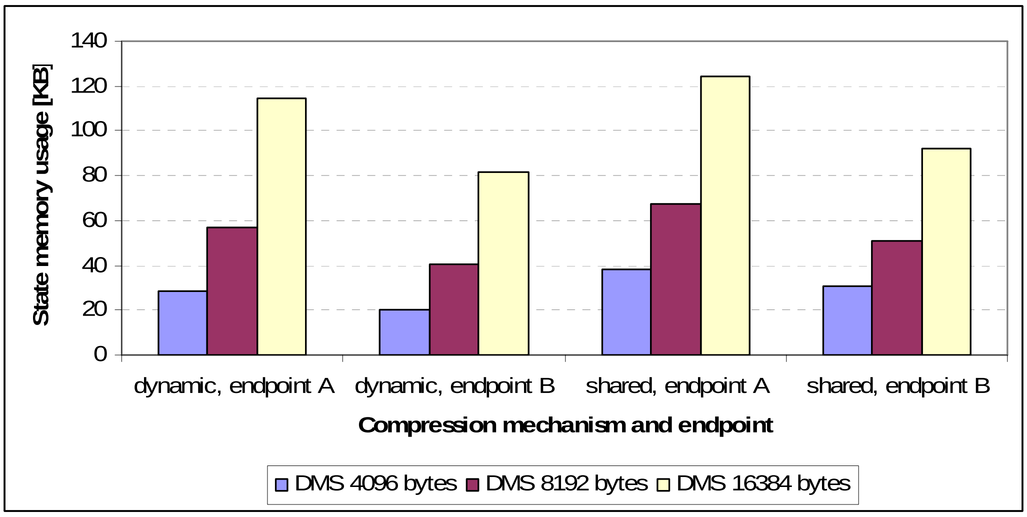


Figure 47 - State memory usage

9.7 Decompression Memory Size

When basic compression is used, all decompression memory sizes produce the same compression ratio, as indicated by Figure 43. This is because all the content fits into the search buffer and nothing needs to be replaced. However, when static compression and a DMS of 4096 bytes are used, the static dictionary occupies the first 3468 bytes of the buffer. If the message is long enough, it does not fit into the free space at the end of the buffer and must replace some existing content. However, this has only a minor effect on the compression ratio because the content being replaced consists of the static dictionary’s lowest-priority strings: a DMS of 4096 bytes results in a compression ratio of 0.65, while the use of a larger decompression memory results in a compression ratio of 0.64. Replacement of content is also the reason static compression with a DMS of 4096 is slower than with the larger decompression memory sizes, as indicated by Figure 45.

We can observe from Figure 45 that the size of the decompression memory does not have a significant impact on the compression ratio dynamic compression achieves. The reason for this is the same as in the case of static compression: although previous content needs to be overwritten in the circular buffer when the smallest buffer size is used, the deleted content consists of the lowest-priority strings of the static dictionary.

On the other hand, a DMS of 8192 bytes is large enough to store all the dictionaries required by dynamic compression. This means that dynamic compression can use a larger dictionary and does not have to spend time replacing strings in the dictionary. Therefore, the compression requires one millisecond less time when the DMS is 8192 bytes than when it is only 4096 bytes. However, the use of a larger decompression memory than 8192 bytes does not help to improve the compression ratio of dynamic compression: a DMS of 16384 bytes only slows down the compression process, because the calculation of the SHA-1 hash over the UDVM memory image requires more time.

Much like in the case of the other mechanisms, also when shared compression is applied, decompression memory sizes of 8192 and 16384 bytes tend to produce almost identical results. With shared compression and a DMS of 8192 bytes, the size of the compressed sequence is 1361 bytes and the aggregate size of SigComp messages is 2558 bytes. On the other hand, when a DMS of 16384 bytes is used, the values are 1354 and 2563 bytes, respectively. Thus, with a DMS of 16384 bytes, the size of the compressed sequence not including SigComp overhead is only 7 bytes less than when the DMS is 8192 bytes. However, if we include the SigComp overhead and compare the aggregate sizes of the SigComp messages, the DMS of 16384 bytes performs worse. The reason behind this is that when a DMS of 8192 bytes is used, the bytecode that is included in the header of the first three messages is four bytes shorter. We can also observe from Figure 45 that shared compression is the only mechanism that clearly benefits from the use of a decompression memory larger than 4096 bytes. This is because shared compression requires so much buffer space that it overwrites the content of a 4096-byte buffer two times during the compression process. When a DMS of 16384 bytes is used, nothing needs to be deleted from the buffer. This is the reason shared compression is fastest with a 16384-byte decompression memory.

When the sequence is decompressed, decompression memory size has no significant impact on the decompression time of the output of basic or static compression, as can be observed from Figure 46. This is because these two mechanisms do not calculate SHA-1 hashes over the UDVM's memory. The situation is different when dynamic and shared compressions are applied: the larger the decompression memory the more time is required to calculate the hash. Because the calculation of the hash requires more time, also the amount of decompression time required increases when larger decompression memory sizes are used.

Figure 47 indicates that the state memory usage of dynamic and shared compressions depends highly on the size of the decompression memory. This is because the length of the state items containing UDVM memory snapshots equals to the decompression memory size. Thus, doubling the decompression memory size has the effect of doubling the size of the state items containing UDVM memory snapshots. When for instance dynamic compression is applied, this means that the entire state memory usage is doubled.

Figure 48 shows the results of using another signalling flow for the session establishment, namely a basic voice session initiation sequence. It is rather different from the 3GPP Release 5 sequence because it consists of only 5 messages, the combined size of which is less than one fourth from that of the 12 messages of the 3GPP sequence. The figure implies that for a small sequence, a DMS as small as 2048 produces only slightly worse compression rations than the larger decompression memory sizes. In addition, the compression and decompression times were always at least as good for the DMS of size 2048 bytes than for the larger decompression memories, 4096 and 8192 bytes. Therefore, we can conclude that the combined size of the messages in the sequence is an important factor when selecting the most appropriate decompression memory size.

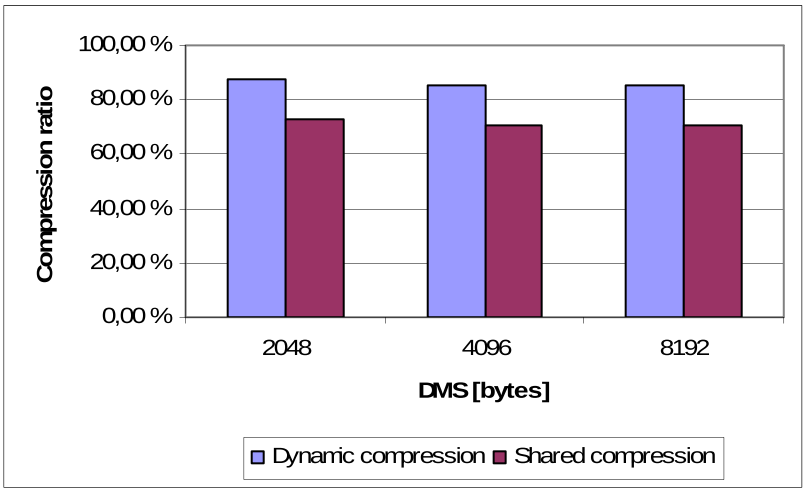


Figure 48 - Compression ratio for a basic voice session establishment sequence

9.8 Unreliable versus Reliable Transport

The purpose of the measurement presented in this chapter is to study the effects of using reliable instead of unreliable transport. TCP is used as the reliable transport layer protocol. All the results presented so far were collected using an unreliable transport layer protocol, namely UDP. The first advantage of using TCP is that if stateful compression is applied, the bytecode needs to be sent only once in each direction. An example of this are the first two messages of the 3GPP session initiation sequence received by endpoint A: 100 Trying and 183 Session Progress. When unreliable transport is used, the P-CSCF must send bytecode together with each of these messages. This is because after sending the 100 Trying, the P-CSCF does not have the information whether the message was received successfully. In contrast, the use of reliable transport eliminates the need to send the bytecode in the header of the compressed 183 Session Progress message, because the P-CSCF can be certain that the SigComp message carrying the 100 Trying message was received.

The second advantage of reliable transport also has to do with messages that are sent consecutively. The 100 Trying and 183 Session Progress messages serve again as an example. When an unreliable transport layer protocol is used, explicit state announcements must be used. This means that the P-CSCF has to acknowledge the reception of the INVITE message in the headers of both the compressed 100 Trying and the compressed 183 Session Progress messages. In addition, the P-CSCF cannot apply dynamic compression to compress the 183 Session progress, because it has not received an acknowledgement indicating the reception of the 100 Trying message. When the transport is reliable, dynamic compression can be applied already to the 183 Session Progress message. In addition, explicit state announcements are not required, meaning that the identifiers of these states do not need to be carried in the SigComp message headers. Both the possibility to use dynamic compression earlier and the absence of explicit state announcements help to reduce the size of SigComp messages.

The performance of UDP and TCP is compared in Figure 49. The sequence used in the measurements is the 3GPP session initiation sequence consisting of 12 SIP messages. A DMS of 8192 bytes and the entire static dictionary were used and the size of shared states was not restricted. Figure 49 implies that TCP clearly achieves better compression ratios. In the case of dynamic compression, TCP achieves a compression ratio of 24.6% and UDP a compression ratio of 28.7%. When shared compression is applied, the compression ratios are 22.3% for TCP and 24.0% for UDP. The compression and decompression times are nearly identical for both TCP and UDP based transports, the only exception being compression time of shared compression, which is 32.8 ms with TCP and 31.7 ms with UDP.

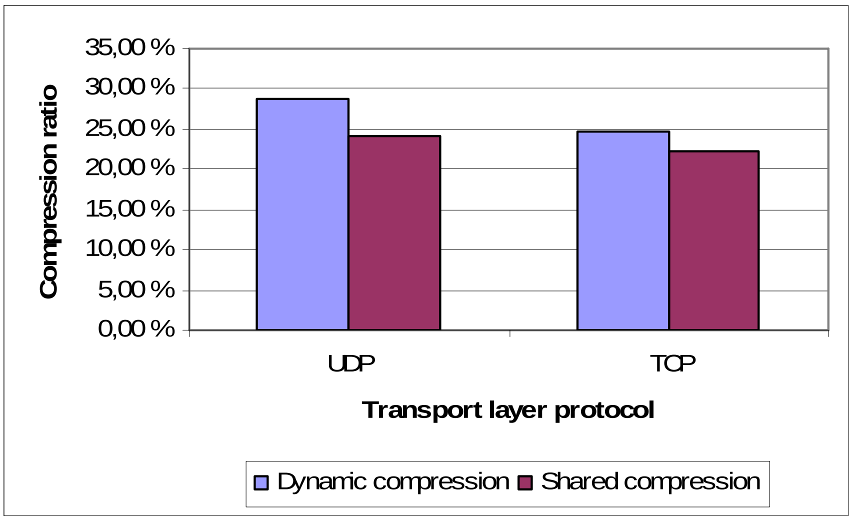


Figure 49 - Reliable versus unreliable transport, 3GPP session initiation sequence

Figure 50 shows the performance of UDP and TCP based transports when SigComp is used to compress the SIP messages of a basic voice session initiation sequence. A DMS of 8192 bytes and static dictionary priorities from one to three were used and the size of shared states was restricted to 500 bytes. In the case of dynamic compression, the use of TCP has the effect of reducing the size of the compressed sequence by 914 bytes compared to the size with UDP. The compression ratios of TCP and UDP are 47.8% and 85.4% for dynamic compression, respectively. In the case of shared compression, the compression ratio is 43.1% with TCP and 70.6% with UDP. With both dynamic and shared compressions, the compression and decompression is slightly faster when TCP based transport is used. The significant performance improvement enabled by TCP is due to the more efficient compression of the first three messages sent by the P-CSCF. The compression benefits from the use of TCP because these messages are sent consecutively without receiving anything from the UE between them; when TCP is used, the bytecode needs to be included only in the first message sent in each direction and dynamic compression can be applied earlier.

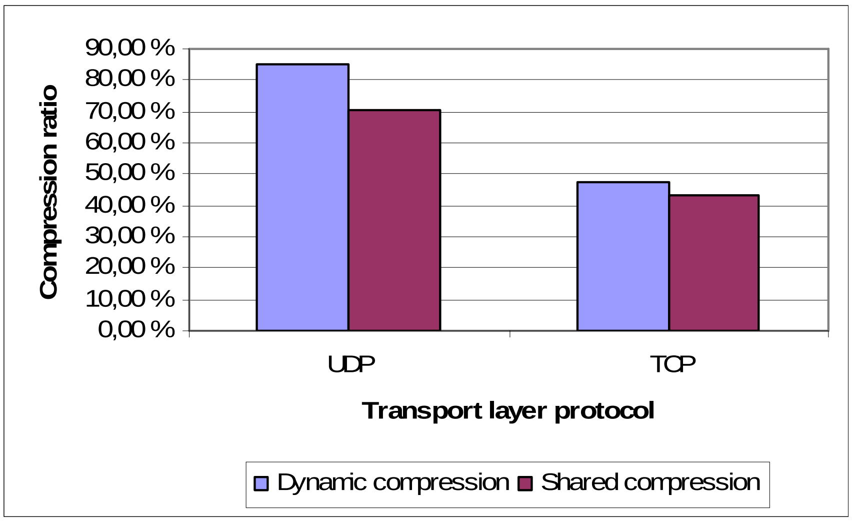


Figure 50 – Reliable versus unreliable transport, basic voice call

9.9 Central Processor Unit

In this measurement, the compression and decompression times of two SIP message sequences are studied on five different CPUs: Pentium 4 Hyper-Threading 3.0 GHz, Pentium 4 2.66 GHz, Pentium 4 1.8 GHz, Pentium M 1.6 GHz and Pentium III 600 MHz. The SIP signalling sequences used are the establishment of a video call in a release 5 network and the establishment and release of a basic voice call.

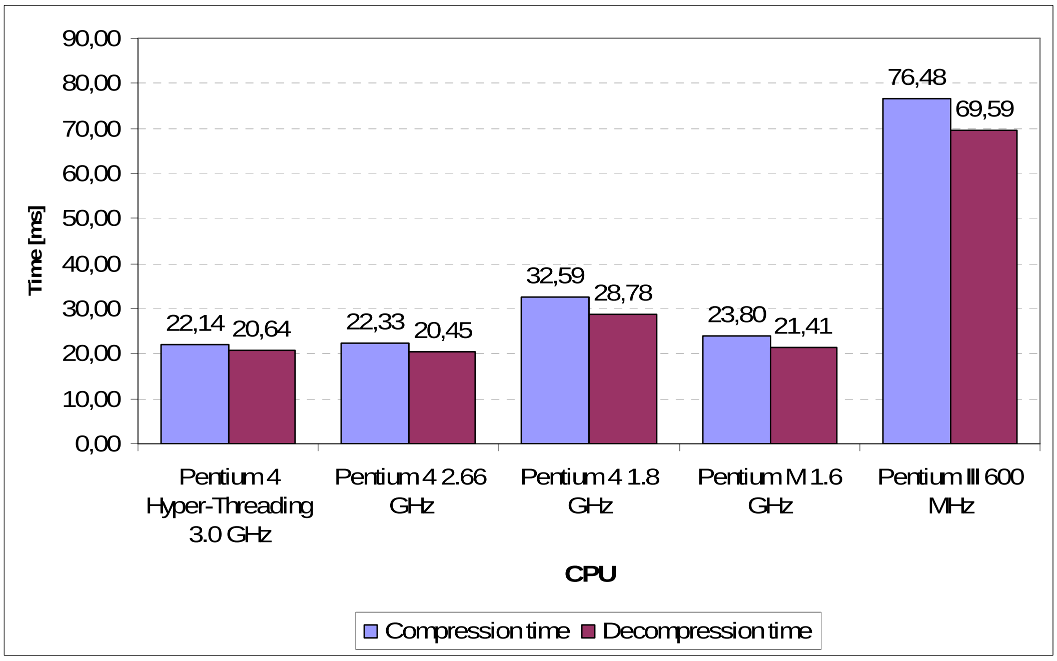


Figure 51 - Compression and decompression times on different CPUs, video sequence

The combined compression and decompression times of the messages of the video call establishment sequence are presented in Figure 51. We can observe from the figure that the performance of the Pentium 4 processors having clock rates of 3.0 and 2.66 GHz is almost identical. The Hyper-Threading technology of the 3.0 GHz CPU has no effect in this measurement, because the messages were not processed concurrently. The Pentium

M 1.6 GHz CPU is only slightly slower than the Pentium 4 processors having clock rates of 3.0 and 2.66 GHz; the compression takes 1.5 milliseconds longer and the decompression one millisecond longer than in the case of the Pentium 4 2.66 GHz CPU. The Pentium 4 1.8 GHz is over 1.4 times slower than the Pentium 4 2.66 GHz, while the Pentium III processor is nearly 3.5 times slower than the other CPUs.

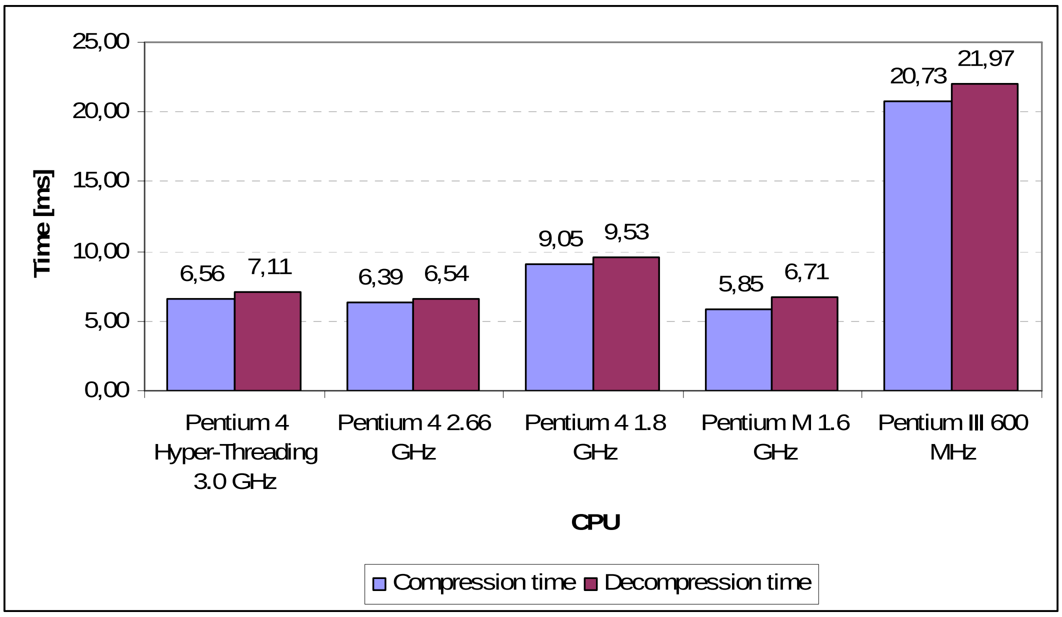


Figure 52 - Compression and decompression times on different CPUs, voice sequence

The compression and decompression times of the basic voice call establishment and release signalling flow are shown in Figure 52. We can observe that the Pentium III and Pentium 4 1.8 GHz are again clearly slower than the other CPUs. The Pentium 4 processor having the 1.8 GHz clock rate is over 1.4 times slower than the Pentium 4 processor having the 2.66 GHz clock rate with regard to both compression and decompression performance. The Pentium M 1.6 GHz is faster than any of the Pentium 4 processors regarding compression performance, although the two fastest Pentium 4 processors outperformed it in the previous measurement, in which the video session establishment sequence was used. One reason for the high performance of the Pentium M processor is that it has twice as much L2 cache memory as the Pentium 4 3.0 GHz and four times more L2 cache memory than the Pentium 4 2.66 GHz processor. The Pentium M 1.6 GHz CPU is the second fastest with regard to decompression performance. The Pentium 4 Hyper-Threading 3.0 GHz is slightly slower than the Pentium M and Pentium 4 2.66 GHz CPUs although it has the highest clock rate.

It should be taken into consideration that the differences between the computers used in this measurement are not limited to the CPU. The computers had also different amounts of main memory and cache memory, different main memory and fronts side bus speeds, and different operating system and Linux kernel versions. Although the clock rate of the CPU has the greatest impact on the compression and decompression times, also the other factors are likely to have some effect.

9.10 Impact of Signalling Compression on Radio Access Network Delay

The one-way RAN delay experienced by the 12 messages of the SIP session initiation signalling flow taken from [3GPP TS 24.228] was estimated in Section 2.1. In this section, these calculations are repeated, this time assuming that the messages are compressed using SigComp. The same signalling link bit rates are used as in Section 2.1, namely 9.6, 12.2, 16, 32, 64, 128 and 256 kbps. It is assumed that the underlying transport protocol is reliable, that the size of the decompression memory is 8192 bytes, that the entire static dictionary is used, and that the size of the shared states is limited to 1500 bytes. The overhead added by the layers below SigComp is not included in the calculations. The results are depicted in Figure 53, which also includes the RAN delay of the uncompressed SIP sequence.

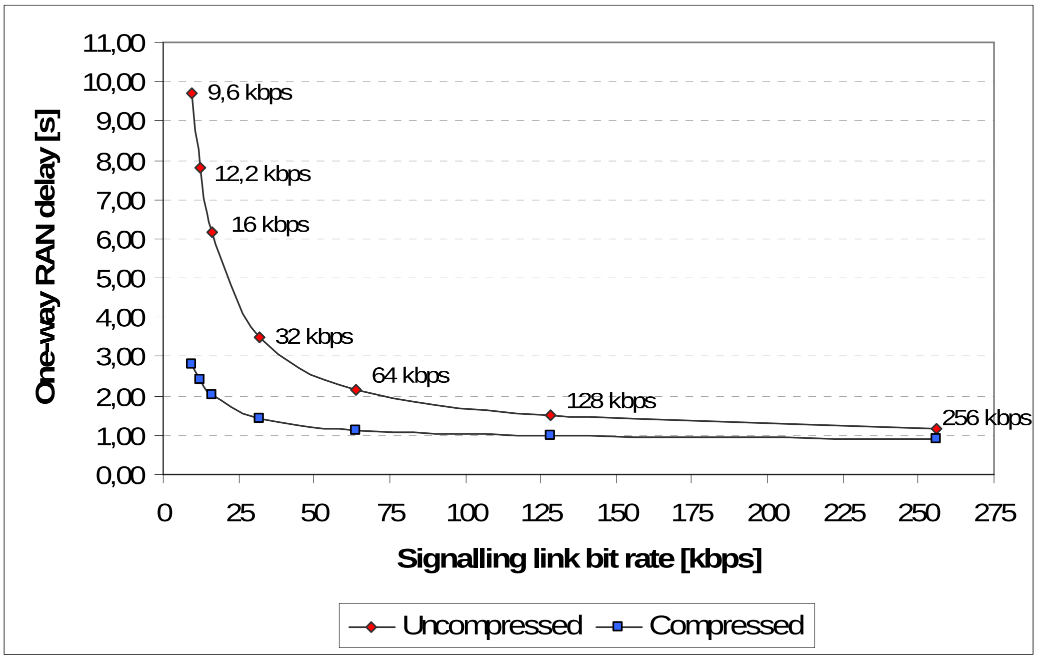


Figure 53 - Impact of SigComp on one-way RAN delay

We can observe from Figure 53 that the improvement SigComp offers decreases as the bit rate of the signalling link increases. The improvement in RAN delay is 6.9 seconds for a bit rate of 9.6 kbps. With a bit rate of 64 kbps the improvement is 1.0 seconds and with the bit rate of 256 kbps it is only 0.26 seconds. These values suggest that when the bit rate of the signalling link is more than 64 kbps, the performance improvement offered by SigComp may not be great enough to justify the use of the protocol.

10 Phase Two – Compression of Different Message Sequences

The goal of the measurements presented in this chapter is to study the compressibility, compression time and decompression time of different SIP message sequences. All the measurements were carried out on the Intel Pentium 4 3.0 GHz platform. The seven SIP signalling sequences listed in Section 8.7 are examined.

Figure 54 shows the SIP messages exchanged between a UAC and a proxy when a basic voice or video session is established. The SIP messages of both the basic voice and basic video session establishment sequences were taken from real SIP clients. Only the information required by SigComp was inserted to the messages. A PoC session establishment does not include the 180 Ringing message, but is otherwise identical to the sequence presented in Figure 54. The PoC messages were taken from [OMA-TS-POC]. The session establishment sequence for a video call in a 3GPP release 5 network was taken from [3GPP TS 24.228] and is illustrated in Figure 1.

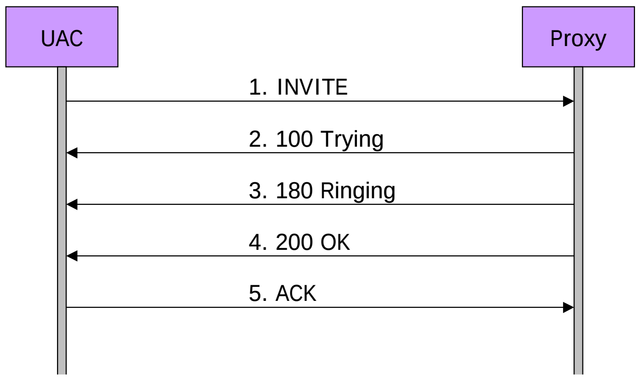


Figure 54 - Messages of a basic voice or video session establishment

A registration signalling flow in a 3GPP Release 5 network [3GPP TS 24.228] is illustrated in Figure 55. The figure shows the exchange of messages between a UE and a P-CSCF. The P-CSCF should not create a SigComp compartment until the user registration has completed successfully. Therefore, the 401 unauthorized response is sent to the UE uncompressed without involving SigComp, and the compression of the second REGISTER request can make use of neither shared nor dynamic compression.

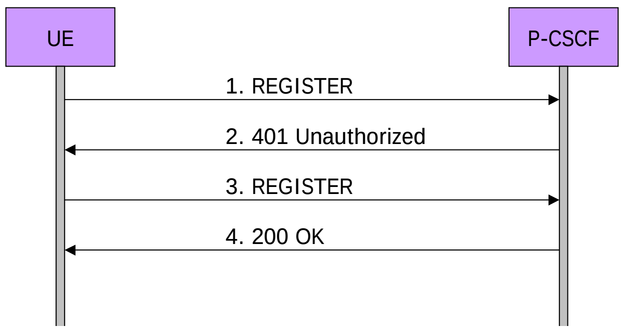


Figure 55 - Messages of a 3GPP Release 5 registration sequence

The last two SIP message sequences in the list of signalling flows studied in this measurement are alternatives to the 3GPP Release 5 session establishment sequence. The contents of the messages were taken from [3GPP TS 24.228]. The goal is to study whether the use of the alternative signalling flows would result in a better performance when SigComp compression is applied. The first alternative is depicted in Figure 56. It uses a RE-INVITE request instead of an UPDATE request and does not offer reliable delivery of provisional responses. The second alternative uses a RE-INVITE request and reliable delivery of provisional responses. The signalling flow is similar to the one in Figure 56 with the exception that a PRACK message and a 200 OK response to the PRACK are exchanged after the 180 Ringing message.

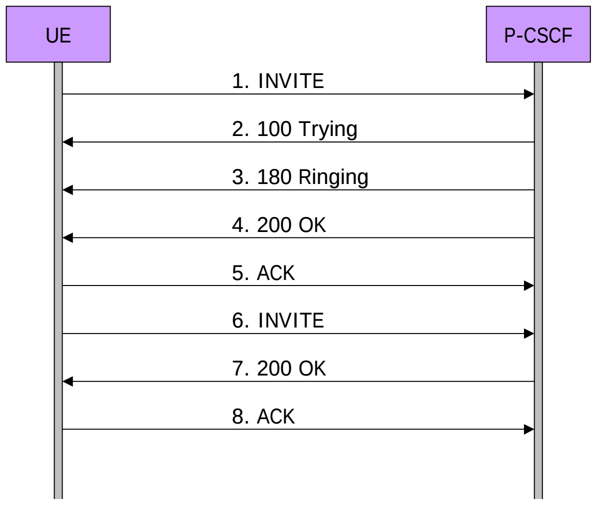


Figure 56 - Session establishment signalling flow with RE-INVITE

The values of various factors that are used in this measurement are discussed below. The results presented in Section 9.3 imply that compression is fastest when short shared states are used. In order to minimize compression time while still being able to achieve reasonable compression ratios, the size of shared states was restricted to 500 bytes. Static dictionary priorities from one to three are used, because in Section 9.4, it was concluded that the use of these priorities seems to result in good overall performance. Size 4096 is selected for the decompression memory for the reason that the aggregate sizes of the messages in different sequences range from 2766 to 10640 bytes. For the shortest sequences, the use of a DMS of 8192 would slow down compression and decompression. On the other hand, the cost of using a small decompression memory is bearable in the case of the longest sequences. The shared compression scheme is used because it is the one that will most likely be used in various SigComp implementations. TCP is used as the transport layer protocol.

Compressed and uncompressed sizes of the different sequences are shown in Figure 57. The compression ratios of the sequences are reported in Table 16. The results imply that the signalling flows having the largest combined message sizes benefit the most from SigComp compression. A high number of messages results in a good compression ratio for the entire sequence, because the compressibility of the last messages is very high. When the signalling flow consists of only four messages, as is the case with PoC and registration signalling, the compressibility of the messages is rather low. The registration sequence is, of course, a special case, since its second message is sent uncompressed. However, even if we did not include the second message in the calculations, the compression ratio of the registration sequence would still be only 0.875. Therefore, one could question whether it is worth the effort to compress the registration sequence at all.

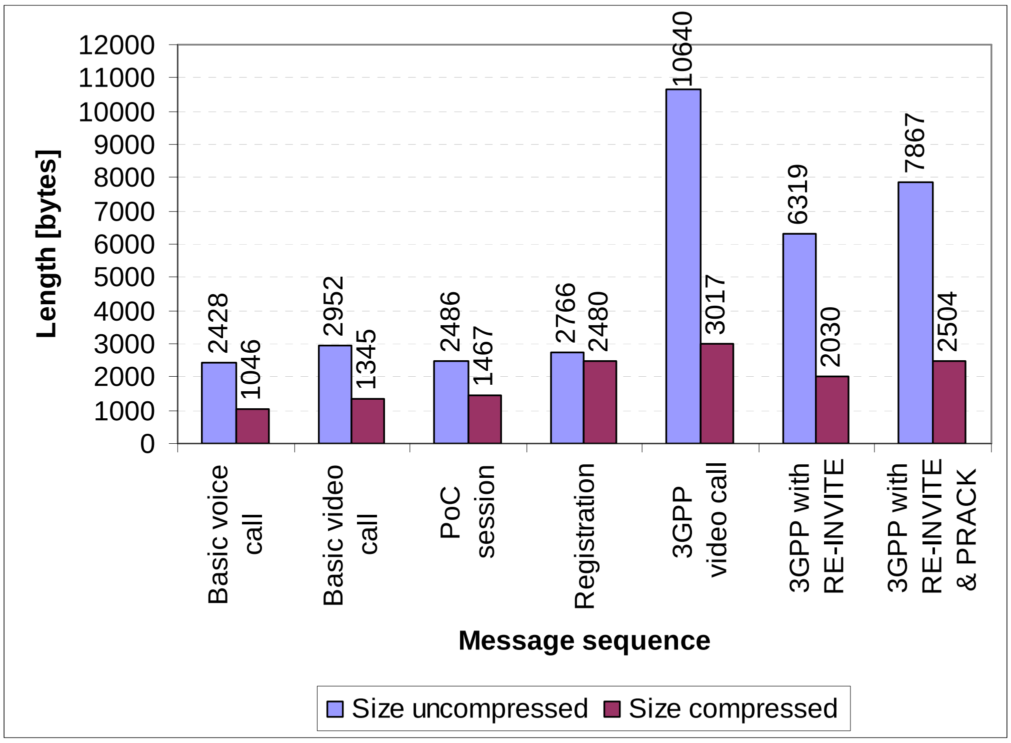


Figure 57 - Sizes of uncompressed and compressed sequences, TCP

From Table 16, we can observe that the 3GPP video call achieves the best compression ratio. However, the compressed size of the alternative sequence using RE-INVITE and reliable delivery of provisional responses is 19.4% less and the size of the sequence using RE-INVITE and unreliable delivery of provisional responses 34.6% less than the compressed size of the original 3GPP sequence. The sequences making use of the RE-INVITE request would thus be faster to transmit over the air interface. In addition, from Figure 58 we can observe that these sequences also consume less CPU time during compression and decompression. Therefore, if CPU time and the transmission time over the air interface are an issue, the use of the alternative sequences is more efficient.

Table 16 - Compression ratios

Message sequence	Number of messages	Compression ratio, TCP [%]	Compression ratio, UDP [%]
Basic voice call	5	43,08	70,63
Basic video call	5	45,56	61,82
PoC session establishment	4	59,01	68,91
Registration	4	89,66	89,66
3GPP video call	12	28,36	28,24
3GPP video call with RE-INVITE	8	32,13	43,99
3GPP video call with RE-INVITE and reliable delivery of provisional responses	10	31,83	33,6

The average compression and decompression times of the various sequences are shown in Figure 58. The averages were calculated over ten measurements. We can observe that the registration sequence consumes less time than the other sequences consisting of five or less messages, although its aggregate message size is the biggest. The reason is that the second message in the registration sequence is sent uncompressed and thus its compression and decompression times are both zero. Otherwise the results offer no big surprises; the larger the content of the sequence is and the more messages it has, the longer are the compression and decompression times. For example, the use of reliable delivery of provisional responses can be directly seen in the increased compression and decompression times. Another similar issue is the exchange of SDP content. If the SDP content needs to be transferred in multiple SIP messages before the endpoints are able to agree upon the set of codecs they will use, compression and decompression times of the entire sequence increase. Although the compression of the SDP content becomes more efficient with each new message exchanged, compression and decompression of the extra content still require additional time.

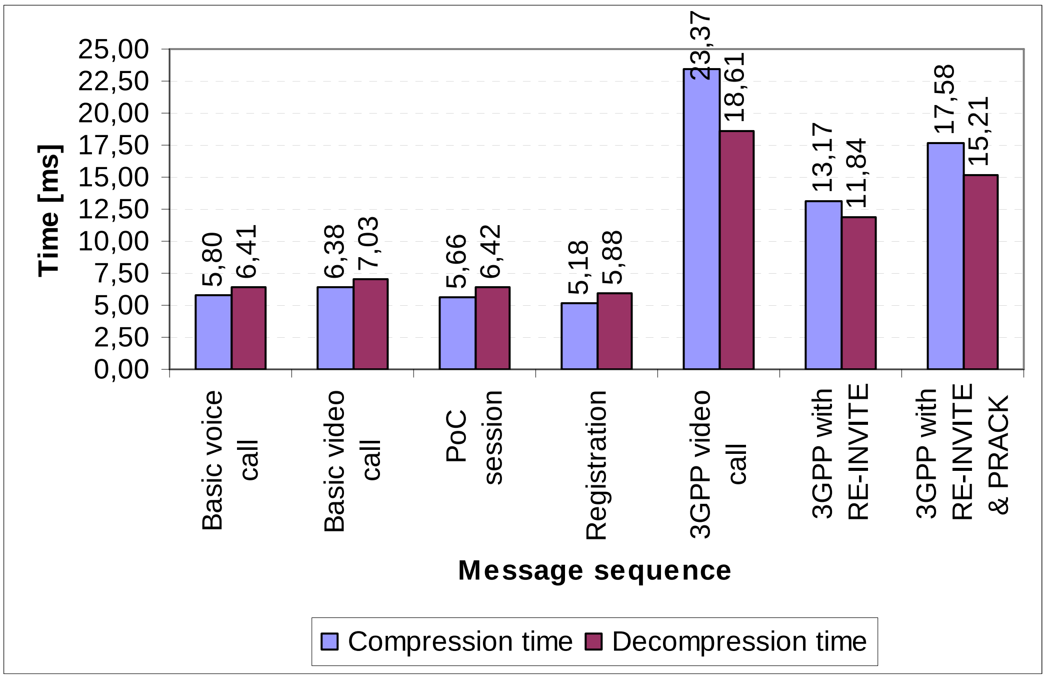


Figure 58 - Compression and decompression times, TCP

11 Phase Three – Measurements on the SigComp Prototype

In this chapter, the results of measurements carried out on the multithreaded SigComp prototype are presented and analysed. The chapter begins with a study of the effects of using different number of threads on the time messages stay in the system. Also the throughput of the system and the average time a message stays in the system are examined using realistic workloads. Also the memory consumption of the SigComp prototype is studied. The chapter ends with a simulation of a Denial-of-Service (DoS) attack, in which looping code is sent to the SigComp prototype acting as a P-CSCF.

11.1 Parameters Used in the Measurements

The SigComp prototype acts as a P-CSCF in the measurements presented in this chapter. Two SIP message sequences are used: a basic voice session establishment sequence and a video session establishment sequence in a 3GPP release 5 network. A decompression memory of size 4096 bytes is used for the former sequence and a DMS of 8192 bytes for the latter sequence. UDP based transport is used. The following SigComp mechanisms are used: static dictionary, dynamic compression and shared compression. The length of shared states is restricted to 500 bytes and strings with priorities ranging from one to three are used from the static dictionary. It is assumed that the duration, i.e. holding time, of voice calls is 180 seconds and the duration of video calls 300 seconds. The duration of a call is equal to the time between the 200 OK message sent in response to the initial INVITE request, and the BYE request. Knowing the duration of a call and the amount of simultaneous calls in the system, we can calculate the average call intensity λ using Little's theorem [Iversen 2005]:

$$L = \lambda W , \qquad (11.1)$$

where L is the average number of calls in the system and W is the mean holding time in the system per call. The interval between successive calls is $1/\lambda$. The call answer delay of each call, i.e. the time between the 180 Ringing message and the 200 OK response to INVITE, is five seconds. The measurement period is 10 minutes, starting from the moment the first call arrives to the system. In the measurements, the number of simultaneous ongoing calls is varied, which has a direct impact on the load of the system. The minimum number of simultaneous calls used is 50 for video calls and 250 for voice calls. The maximum number of simultaneous calls that can be used depends on the amount of physical memory available on the computer. If it was observed that the computer under study started to run out of physical memory and had to use virtual memory, experiments with higher number of simultaneous ongoing calls were not continued. The computer with the Intel Pentium 4 2.66 GHz processor had only 512 megabytes of main memory, of which 270 was available for the use of SigComp prototype process and the network protocol analyser program used. With this amount of free memory available, the SigComp prototype was able to process the signalling of 1500 simultaneous voice calls and 500 simultaneous video calls. On the other hand, the computer with the Intel Pentium 4 3.0 GHz Hyper-Threading CPU had 1024 megabytes of main memory and was therefore able to support a higher number of calls in the system.

11.2 Number of Workers

The aim of this measurement is to study the effects of increasing the amount of concurrency in the system. A limit can be placed on the maximum number of messages that can be processed concurrently by restricting the number of threads in the thread pool of the SigComp prototype's thread dispatcher. In the measurement, the time a single message stays in the system is monitored. This time consists of two parts: the time the message stays in the buffer waiting for service, and the time the message is being processed. The signalling of 5000 simultaneous ongoing voice calls is used as the workload.

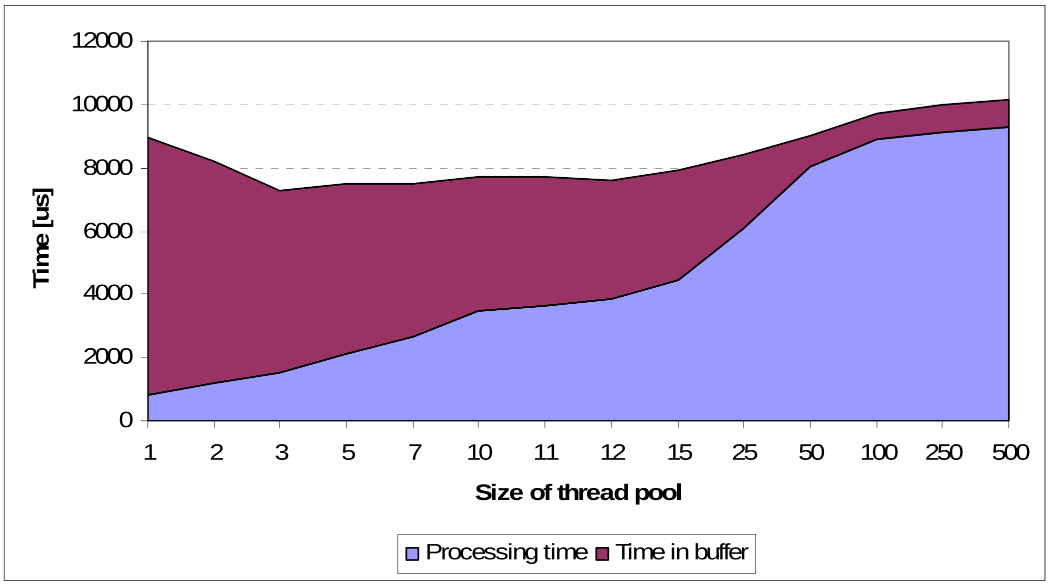


Figure 59 - Time in system for different thread pool sizes, 5000 simultaneous voice calls

The results of the measurements are presented in Figure 59. We can observe that as more threads are added to the thread pool, the time the messages stay in the buffer decreases and the time the messages are being processed increases. When the size of the thread pool is for instance one hundred threads, all of the one hundred threads might be active during large traffic bursts. In this case, the available CPU time is divided among the threads and each of them gets one percent of the speed of the CPU. In addition, the threads compete for access to the shared resources. When one thread holds exclusive access to a shared resource, other threads willing to access the resource have to wait. Waiting increases the average processing time of messages: if there are a large number of threads willing to write data to the state handler at the same instant, only one thread at a time can access the state handler and the rest of the threads have to wait. The last thread in the queue has to wait until all the other threads have completed their write operations. Unfortunately, some operations requiring exclusive access to the state handler take a lot of time. An example of this is the deletion of the state items a compartment has created, which has to be done each time a compartment is closed. In case of a video call, this typically means that 28 state items have to be located and deleted. Also the memory allocated to the state items has to be freed.

When there is only one worker, it can execute on the CPU without other workers interrupting it, and it never needs to wait for access to the shared resources. However, the average time a message stays in the buffer is substantial, because only one message at a time can be processed and the other messages in the buffer have to wait. Therefore, we clearly want the system to have more than one worker.

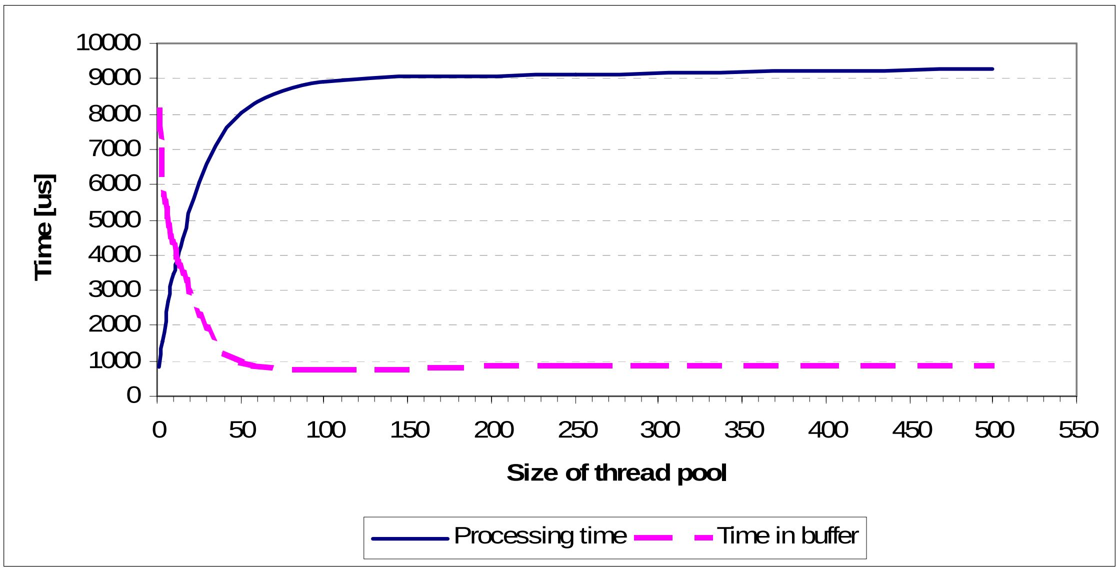


Figure 60 - Processing time and time in buffer for different thread pool sizes

The development of the processing time of the messages and the time the messages stay in the buffer is shown in Figure 60. The processing time increases steeply between thread pool sizes 1 and 50, until it begins to gradually level off. Both the processing time and the time in buffer start levelling off after the pool size 50, because the effects of adding more threads to the system decrease as the size of the thread pool gets closer to the maximum traffic burst size. There are only a few cases in which the number of messages being processed concurrently is 50 or larger. When thread pool sizes larger than 100 are used, the behaviour of the system does not change with the traffic load that was used; the average number of active threads in the system remains the same although the size of the thread pool is increased.

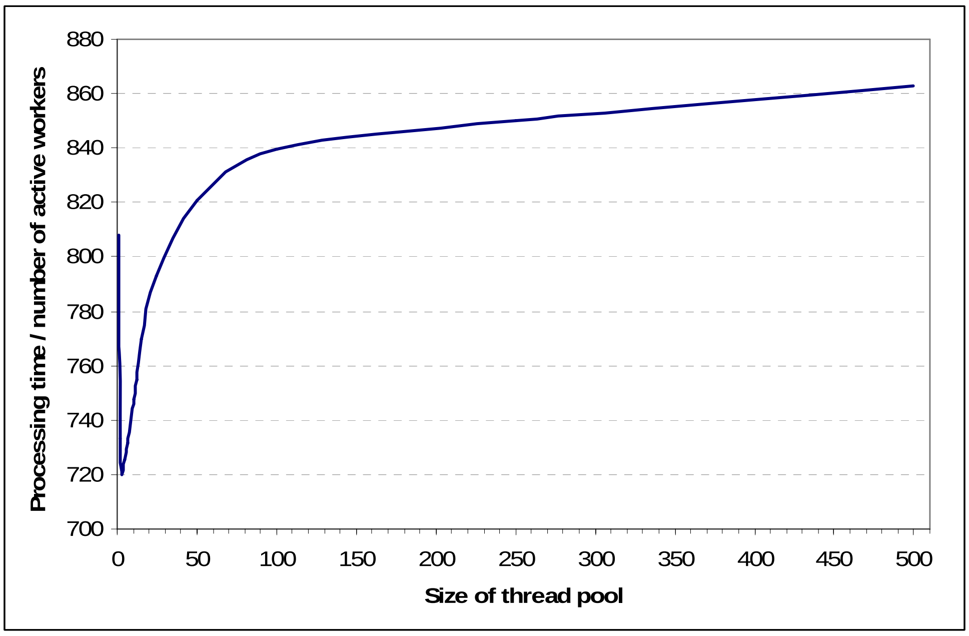


Figure 61 - Ratio of average processing time and average number of threads in system

The ratio of the average processing time and average number of threads is depicted for different thread pool sizes in Figure 61. In an ideal system, the ratio would remain constant regardless of the average number of active workers the system has, meaning that thread switches and queuing times to shared resources would not add any overhead. If the average processing time was 1 millisecond when the system has one worker thread, the average processing time with ten worker threads would be 10 milliseconds in an ideal single-processor system. In reality, the average processing time with ten workers is slightly more because of the overhead. This can also be observed from Figure 61: the ratio is at minimum when the size of the thread pool is three, and starts to grow from the value three onwards. When the size of the thread pool is grown from the value three to value five, the average number of active workers increases from 2.13 to 2.96 and the average per-message overhead caused by thread switches and queuing times to shared resources increases by 10 microseconds. When the average number of active workers increases from 2.13 to 9.80, i.e. the size of the thread pool is changed from 3 to 50, the average per-message overhead increases by 100 microseconds. With 9.80 active workers, the overhead caused by thread switches and queuing for access to the shared resources constitutes 12 percent of the processing time.

The average number of active threads, i.e. the average number of messages being processed concurrently, is depicted in Figure 62 for different thread pool sizes. We can observe that starting from the thread pool size 100, the average number of messages being processed concurrently remains at 10.7 messages. However, the momentary number of concurrently processed messages varies greatly; for example with 500 available workers the momentary number of active workers is between 1 and 140. The reason for the large variance is the bursty nature of the traffic, which is illustrated in Figure 66. The Hyper-Threading CPU used in the measurements can execute two threads in parallel on its logical processors. Therefore, we can expect that the average time messages stay in the system is at minimum when the average number of messages being processed simultaneously is near the value two. From Figure 62, we can observe that when the size of the thread pool is three, the average number of active workers is

2.14. Figure 59 indicates that the time in system decreases until the thread pool size three is reached and starts to grow from the size three onwards, meaning that the time in system is indeed at minimum when there are on average two active workers. However, with this amount of concurrency in the system, the waiting time constitutes 80 percent of the time in system. In addition, it has a great variance. The waiting time and its variance can be reduced dramatically by adding more threads to the system. Because we want to keep the waiting times low, a thread pool that has enough threads to serve even the largest traffic bursts is used in the rest of the measurements of this chapter.

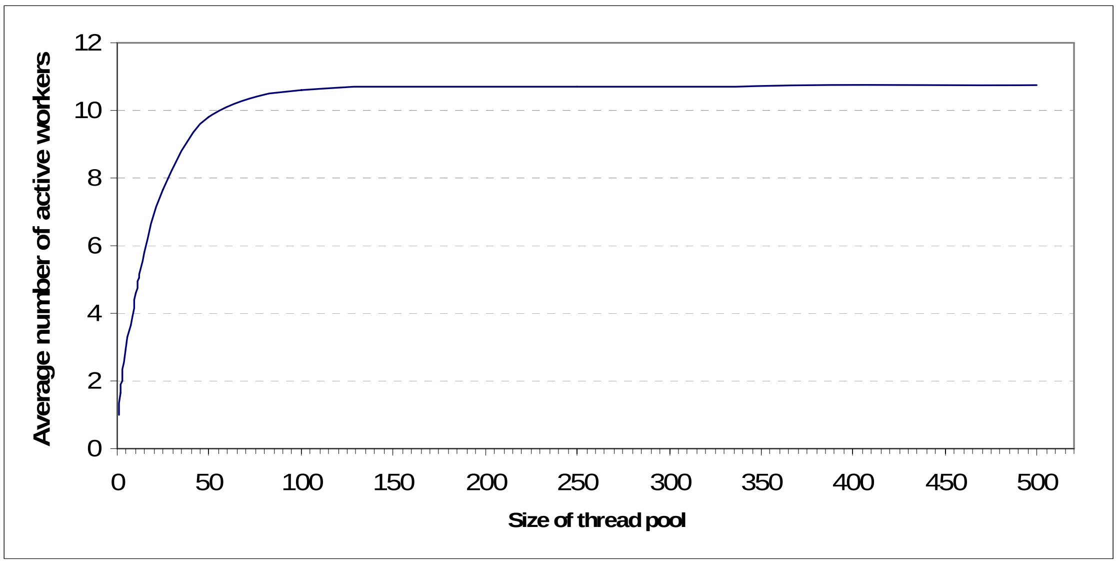


Figure 62 - Average number of active workers, 5000 calls in system

11.3 Time in System

In the measurements presented in this section, the time a single message stays in the SigComp prototype is monitored. The measurements are performed on the Intel Pentium 4 3.0 GHz Hyper-Threading platform.

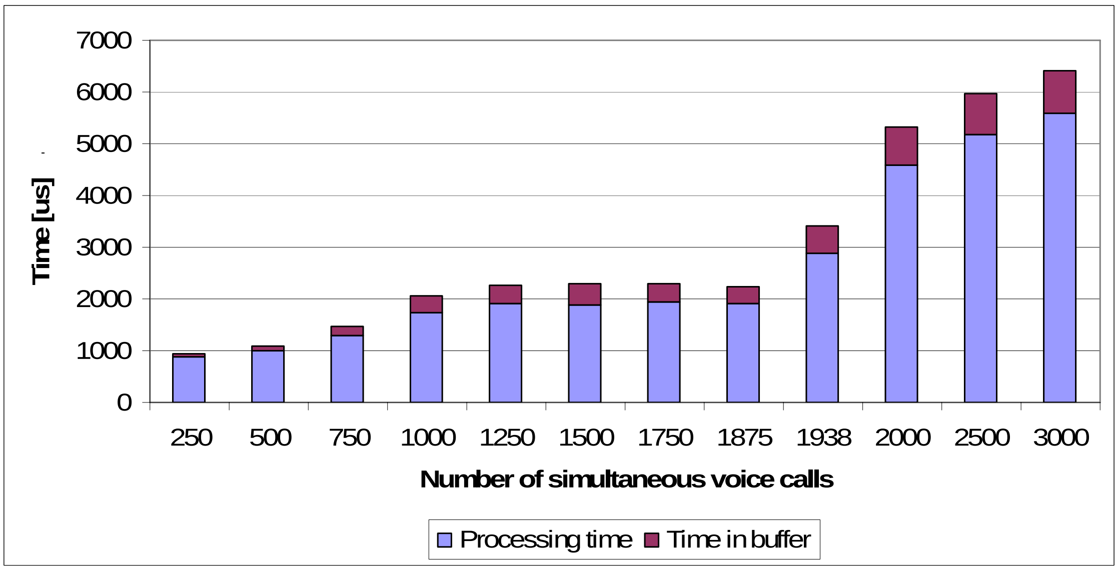


Figure 63 – Average time in system, voice calls, Intel Pentium 4 Hyper-Threading 3.0 GHz

The average time a single message stays in the system under different loads is shown in Figure 63 for voice calls and in Figure 64 for video calls. We can observe that as the number of simultaneous calls increases, also the time a single message stays in the system increases. As the load of the system increases, both the time a single message stays in the buffer and the time the message is being processed increase. When the number of simultaneous calls increases, a higher number of SIP and SigComp messages arrive to the system during each second. Therefore, there are also more active threads in the system. If there are for example ten active threads, each thread gets one tenth of the CPU time. The more active threads the system has, the smaller part of the CPU time is allocated to each of them, which increases the delay a single message experiences. Additional time is also required because the operating system has to perform thread switches. One further reason for the messages staying longer in the system with higher number of active threads is that each thread needs to get exclusive access to the state handler at some point during the compression and decompression of a message. This occurs when the thread needs to carry out an operation that modifies a shared data structure. In addition, exclusive access to compressor array is required when a compartment is created and destroyed. It is of vital importance to minimise the amount of time each thread holds exclusive access to a shared resource, and to use right scheduling policies to control access to the shared resources. As an example, the initial implementation of the state handler used a regular array to hold the state items created by SigComp compartments. This was found to be a poor approach, because if there are thousands of simultaneous ongoing sessions, the state handler's state item table must store tens of thousands of state items. The use of a regular array meant that for example find and delete operations required a linear time to complete. Therefore, the state handler was modified to use a hash table. This resulted in a significant performance improvement, because the use of a hash table allows insert, find and remove operations to be performed in a constant average time. We observed that the heaviest operation requiring exclusive access takes place when the state items a compartment has created are removed from the state handler. This happens whenever a compartment is deleted.

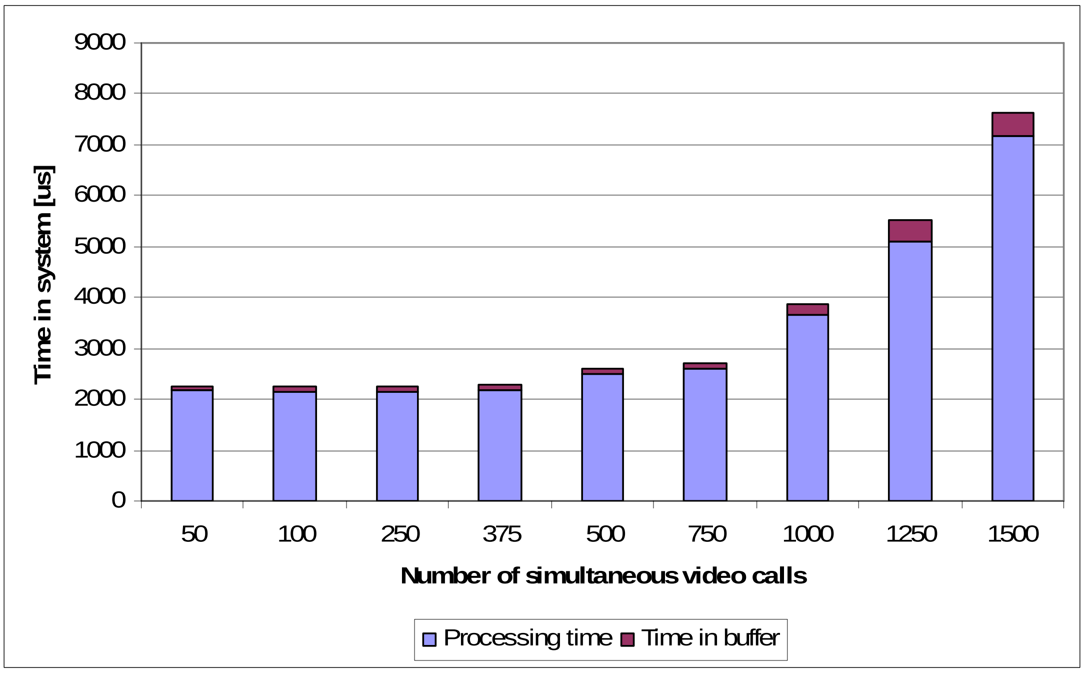


Figure 64 - Average time in system, video calls, Intel Pentium 4 Hyper-Threading 3.0 GHz

We can observe from Figure 63, which shows the average time in system for video calls that the time a single message stays in the system remains constant for traffic loads ranging from 1250 to 1875 calls in the system. In addition, when the traffic load increases from 1875 to 2000 simultaneous calls, the average time in system more than doubles. This behaviour is explained by an increase in the number of active workers in the system, as illustrated in Figure 65. The average number of active worker threads remains constant between 1250 and 1875 calls in the system, but starts to increase from the value 1938 onwards. With 2000 simultaneous calls, there are on average almost two times more active threads in the system than with 1875 calls. This results in longer processing times for the messages, because the time a thread processes a message depends directly on the number of other threads being executed at the same instant. The less active threads the system has, i.e. the lighter the load of the system is, the less time each message stays in the system. From Figure 65, we can also see that the maximum number of active threads increases rapidly as the traffic load grows. The increase in the number of active threads is because there is more traffic to process and because increased traffic results in bigger traffic bursts.

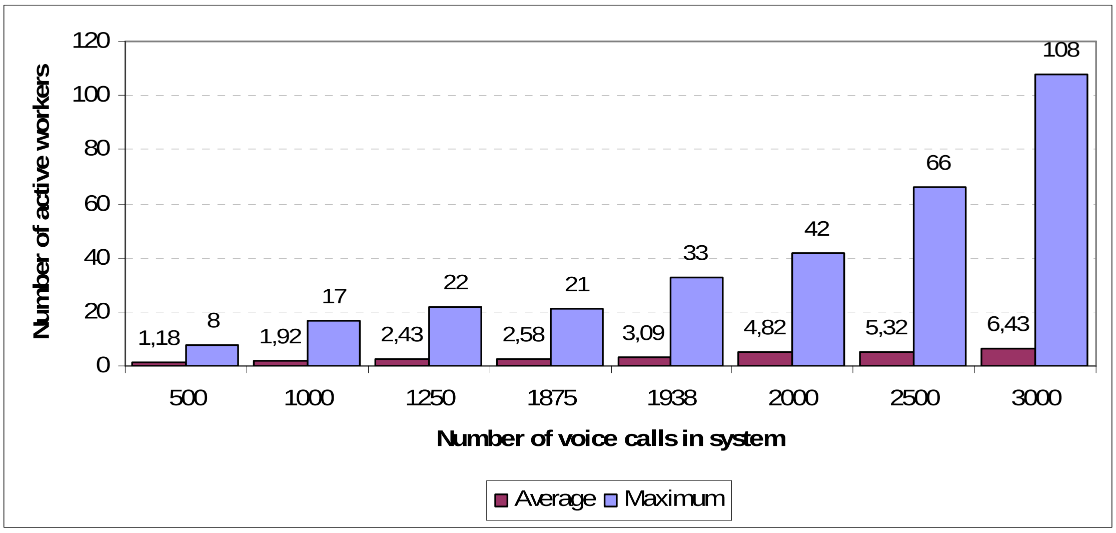


Figure 65 - Number of active workers

Figure 64 indicates that the system behaves in a similar way with the signalling of video calls as it did in the case of the voice calls: the time in system stays flat for traffic loads between 50 and 375 and 500 and 750 calls in system, and starts to increase rapidly when the load becomes higher than 750 calls in the system. This behaviour can again be explained by the increase in the number of active threads.

Figure 66 shows the delays consecutive signalling messages experience in a system serving 1000 simultaneous calls. In the figure, delays of 175 consecutive messages, starting from the 10289th message arriving to the system, are shown. We can observe that the delay of the most of the messages is below 1000 microseconds; these are messages processed by a thread that has gotten the CPU entirely to itself, i.e. there are no other messages being processed at the same time. However, certain messages experience delays much longer than 1000 microseconds. Such messages appear as peaks in the figure. We can observe that the wider the peaks are, the longer are the delays experienced by the messages constituting the peak. The peaks are formed by messages that are being processed concurrently. These messages experience longer delays, because they get only a fraction of the CPU time to themselves. The more messages are being processed simultaneously, the longer is the delay that each of the messages experiences. As an example, it seems that messages 64, 65, 66, 67 were processed concurrently, meaning that each of the workers processing the messages got only one fourth of the CPU time.

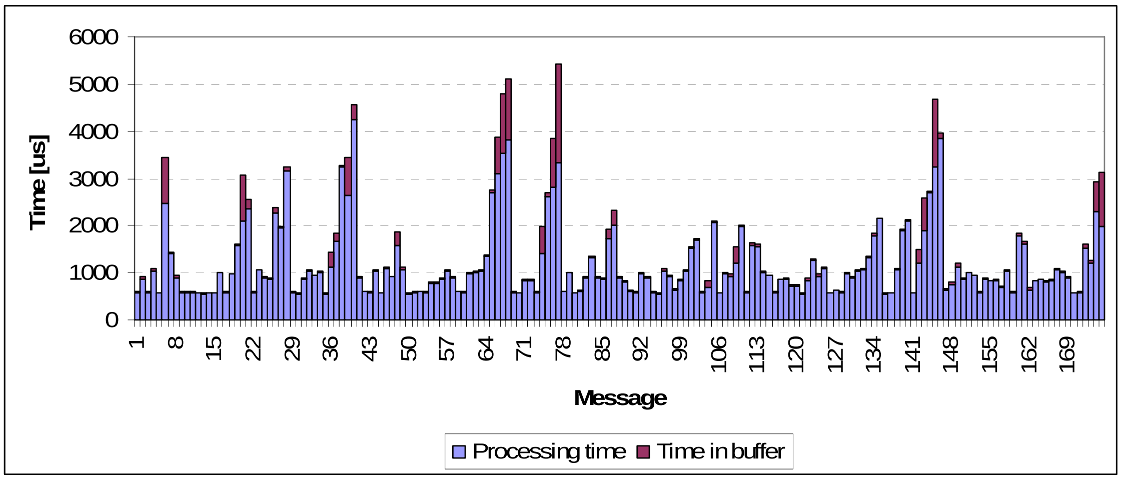


Figure 66 - Time in system, 1000 simultaneous voice calls

11.4 Hyper-Threading Processor versus a Regular Processor

Because the SigComp prototype uses multiple threads, it is supposed to benefit from the use of a CPU supporting the Hyper-Threading technology. In this measurement, the performance of two CPUs, Intel Pentium 4 Hyper-Threading 3.0 GHz and Intel Pentium 4 2.66 GHz is compared. Different traffic loads ranging from 50 to 1500 simultaneous calls in the system are used. The low amount of free main memory available on the computer having the 2.66 GHz CPU limited the number of simultaneous calls that could be used.

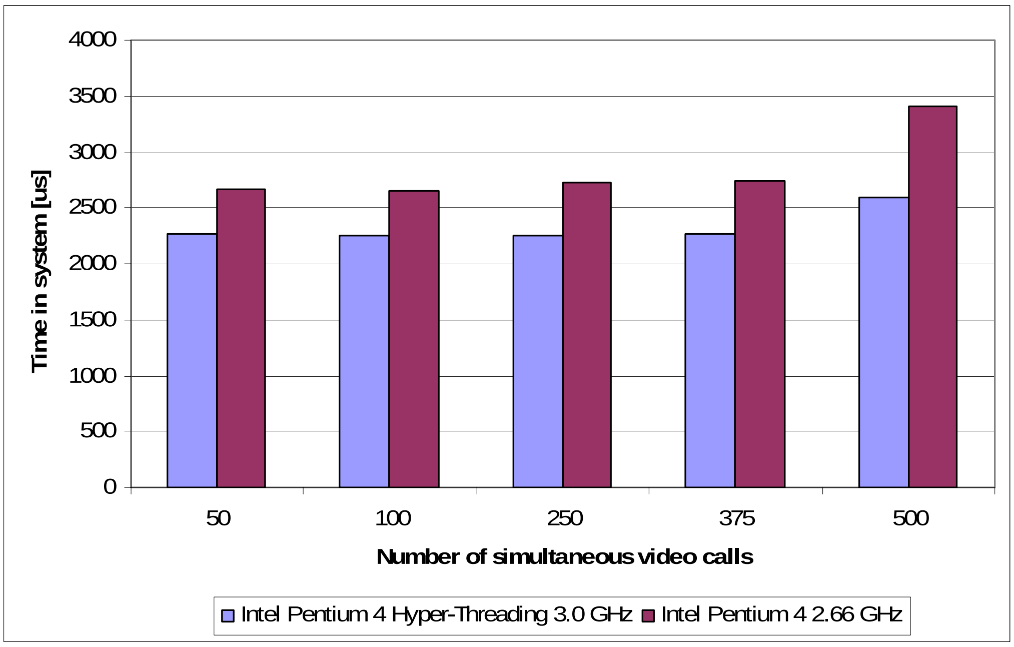


Figure 67 - Hyper-threading Pentium 4 versus a regular Pentium 4, video calls

The time the signalling messages stay in the system on average is presented in Figure 67 for video calls and in Figure 68 for voice calls. We can observe from the figures that the Hyper-Threading processor performs always better than the regular Pentium 4 processor. In addition, the difference between the processors seems to increase as the traffic load increases. As the load increases, the number of active threads the system has increases as well, meaning that a processor that can execute threads in parallel becomes more and more efficient compared to a processor executing only one thread at a time. For example, when the traffic load consists of 1500 simultaneous voice calls, the average number of messages being processed concurrently is roughly 2.5, and the Hyper-Threading CPU benefits from its ability to execute two threads in parallel on separate logical processors. On the other hand, when the traffic load is only 500 calls in system, the average number of messages being processed concurrently is close to 1.2. In this case, the gains of using a Hyper-Threading CPU are smaller, because there are on average only 1.2 threads being executed concurrently.

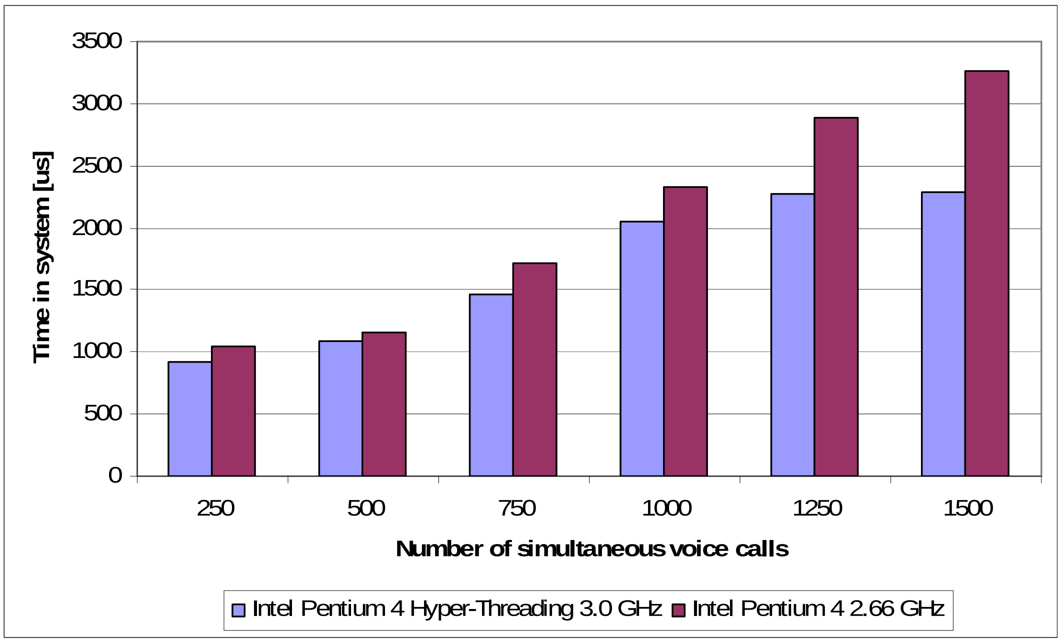


Figure 68 - Hyper-threading Pentium 4 versus a regular Pentium 4, voice calls

In Section 9.9, we saw that if messages are not processed concurrently, the performance of the regular Pentium 4 and the Pentium 4 with the Hyper-Threading technology is very close to being identical. On the other hand, Figure 68 indicates that if there are for instance 1500 simultaneous voice calls, the average time in system per message is 1.43 times longer on the computer having the regular Pentium 4 processor than on the computer with the Hyper-Threading Pentium 4 processor. Therefore, we can conclude that the use of a CPU applying the Hyper-Threading technology seems to have a positive impact on performance. However, it should be taken into consideration that the performance gap between the two computers is not necessarily explained by the use of the Hyper-Threading technology alone; the computer with the Pentium 4 3.0 GHz processor also had other advantages: a faster front side bus, larger L2 cache memory, higher clock rate and larger and faster main memory.

11.5 Throughput

The throughput of the system under different loads is shown in Figure 69 for video calls and in Figure 70 for voice calls. The throughput is presented in terms of average-size SIP messages. The throughput values were collected using the network protocol analyser software Ethereal.

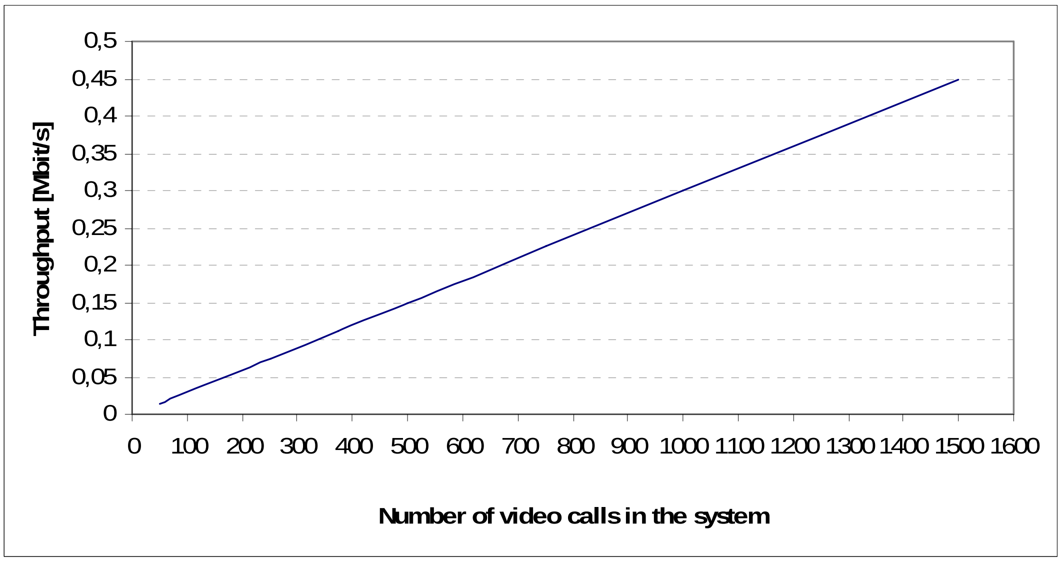


Figure 69 - Throughput of the system, video calls

We can observe from Figure 69 and Figure 70 that the throughput of the system grows in a linear fashion as the number of calls in the system increases. The throughput does not begin to level off as the traffic load increases, meaning that the maximum capacity of the system has not been reached. The greatest throughput, 0.448 Mbit/s, is achieved when there are 1500 video calls in the system. With this traffic load, the maximum load level of the two logical processors of the Intel Pentium 4 Hyper-Threading 3.0 GHz CPU was 43.1% for CPU 1 and 31.2 for CPU 2. The maximum CPU loads of the Hyper-Threading and regular Pentium 4 CPUs are presented in Table 17 for different traffic loads.

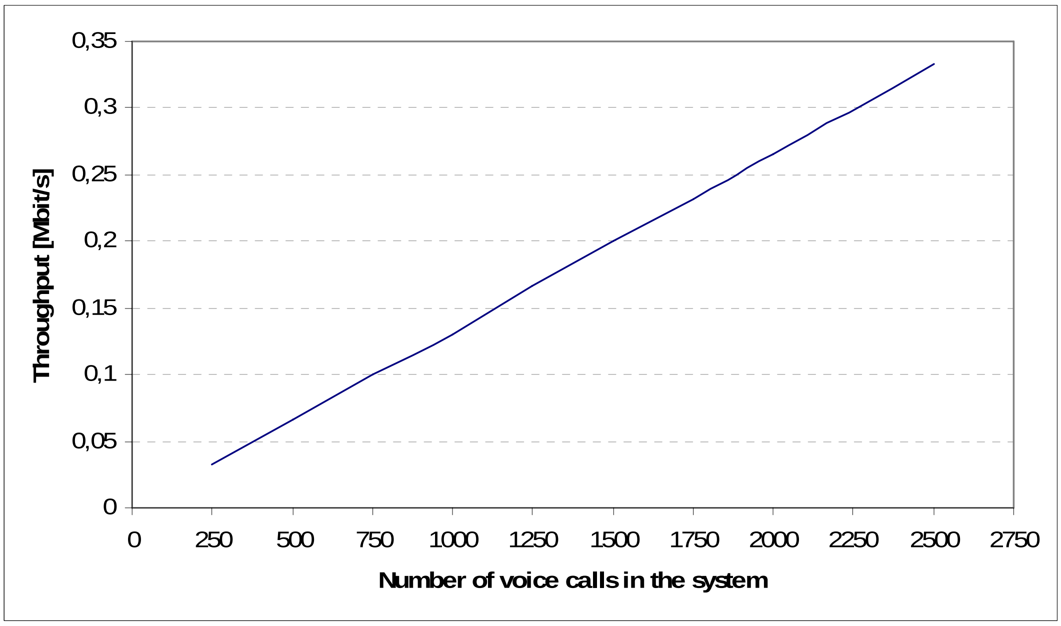


Figure 70 - Throughput of the system, voice calls

Table 17 indicates that for example the maximum load caused by 1000 simultaneous video sessions is significantly larger than the load caused by 1000 voice calls in the system. This is true also for all other cases in which the number of voice and video calls in the system is equal. The higher load caused by the video calls is because the SIP messages of the video calls are longer and the total number of messages per dialog is greater. A minor contributing factor is that a larger decompression memory size was used for the video calls. Finally, we can also observe from Table 17 that the maximum load of the Pentium 4 2.66 GHz CPU is always substantially larger than the maximum load of the Hyper-Threading CPU with the same number of calls in the system. This suggests that the Hyper-Threading CPU benefits from the ability to schedule threads on different logical processors.

Table 17 - Maximum CPU load

Number of calls in the system	Call type	Maximum CPU load [%], Intel Pentium 4 2.66 GHz	Maximum CPU load [%], Intel Pentium 4 Hyper-Threading 3.0 GHz	
			CPU 1	CPU 2
50	video	17.7	1.0	1.0
100	video	21.1	1.0	1.7
250	video	30.7	2.7	3.4
500	video	44.9	4.4	6.1
750	video	-	7.1	7.5
1000	video	-	10.5	12.5
1250	video	-	34.1	14.6
1500	video	-	43.1	31.2
250	voice	13.3	1	1.4
500	voice	17.4	1.4	2.7
1000	voice	27	3.7	4.1
1500	voice	32.9	4.4	5.7
2500	voice	-	9.2	10.2
3000	voice	-	15.6	16.6

11.6 Memory Consumption

In the measurements the results of which are presented in this section, the memory usage of the SigComp prototype is studied under different loads for both video and voice calls. The memory usage of voice calls is shown in Figure 71 and the memory usage of video calls in Figure 72.

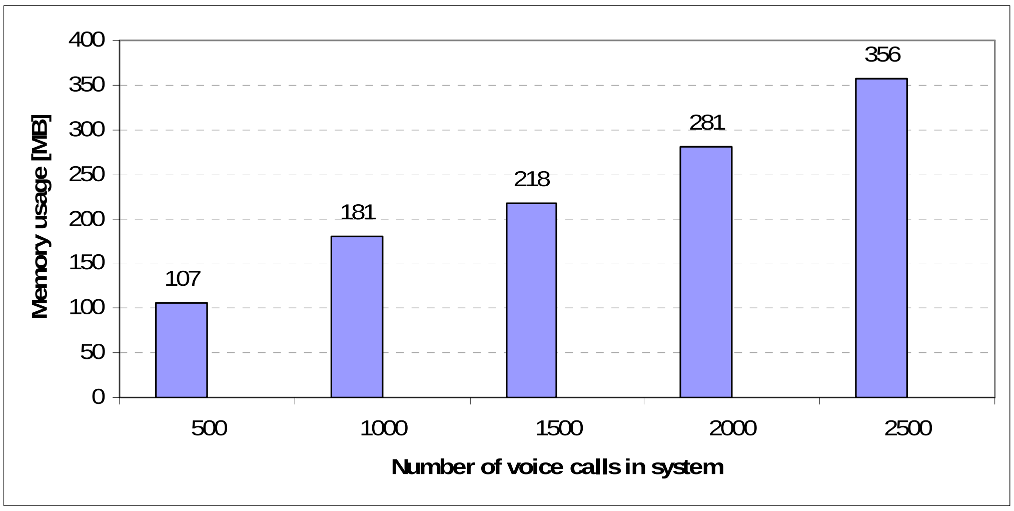


Figure 71 - Memory consumption of voice calls

We can see from Figure 71 that when there are 1500 voice calls in the system, the SigComp prototype uses 218 megabytes of memory. However, if the system serves the same number of video calls, considerably larger amount of memory, 781 megabytes, is required. The reasons for the difference between video and voice calls include that the video sessions have twice as much SIP messages as voice calls, the average size of the messages of the video sequence is twice the average size of the messages of the voice sequence, and that the decompression memory size of the video calls is two times larger than that of the voice calls. By comparing the values of Figure 71 and Figure 72, we can observe that video calls use 2.8 - 3.6 times more memory than voice calls.

We can conclude that the memory requirement of the SigComp prototype is relatively large. This is because for each decompressed message, a UDVM memory snapshot and a shared state need to be stored. For each compressed message, a UDVM memory image and a shared state need to be stored. The data structures of the state handler, shared buffer and the compressor array require storage space. In addition, a compressor object together with the hash table it uses and a feedback object must be stored for each compartment. A state memory of size 131 kilobytes and a decompression memory size of 8192 or 4096 bytes were used. The use of smaller values for decompression memory size and state memory size would help in reducing the memory consumption. However, this would also result in worse compression ratios. One additional way to reduce memory usage at the cost of achievable compression ratios would be to switch off shared compression, dynamic compression or both of them. This way UDVM memory snapshots, UDVM memory images and shared states would not need to be stored.

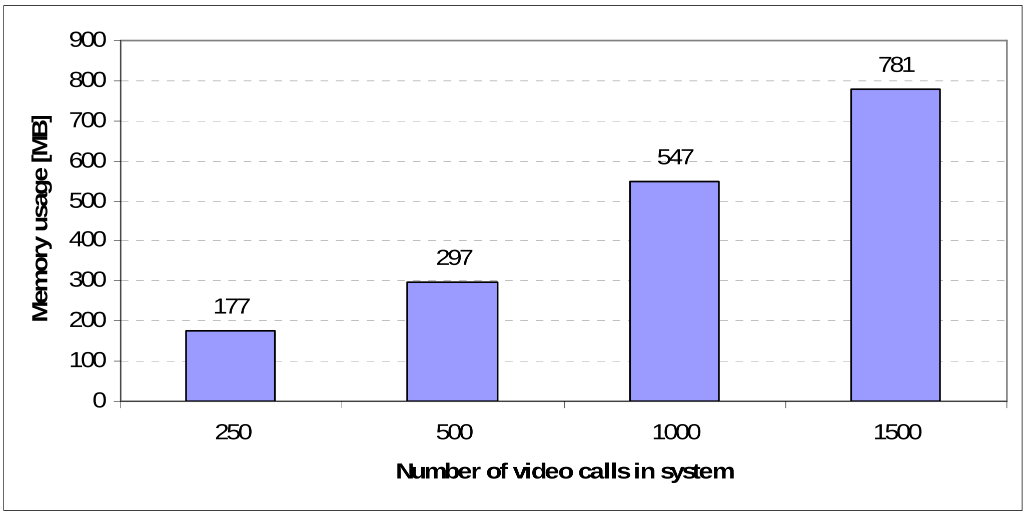


Figure 72 – Memory consumption of video calls

The way the memory usage of the SigComp prototype develops is illustrated in Figure 73, which shows the memory consumption of 1500 simultaneous voice calls during a ten-minute measurement period. The holding time of each call is 180 seconds. During the first 180 seconds of the measurement, no calls exit the system, which is the reason the memory consumption grows steeply. When calls start exiting the system the memory usage levels off. The reason the memory usage does not stay completely flat, but grows slightly towards the end of the measurement period is that new memory has to be allocated to hold for example the delay and memory usage values; the SigComp prototype stores some information for each message arriving and leaving the system. In addition, the time difference between the first and last signalling messages of a call is longer than the duration of a call, meaning that the average number of calls in the system grows slightly towards the end of the measurement period.

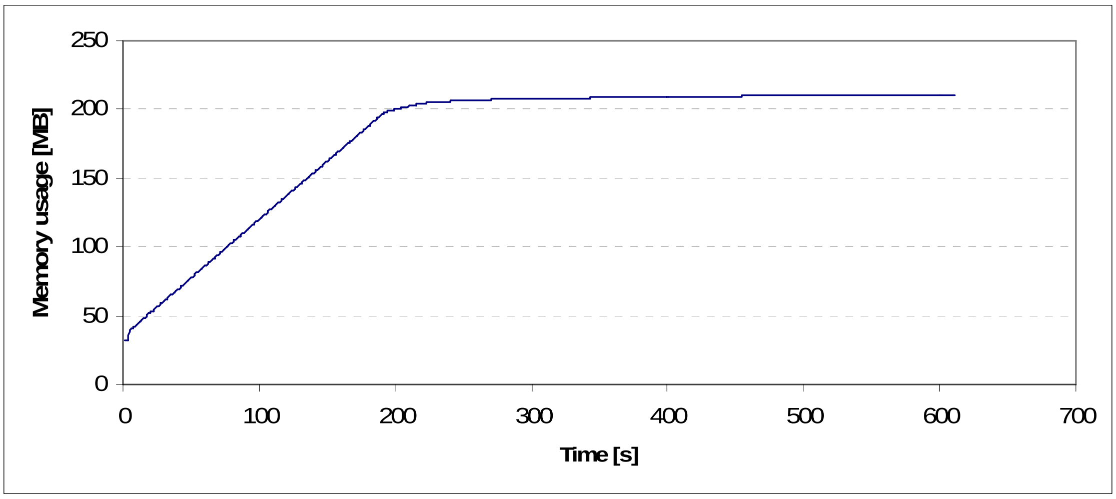


Figure 73 - Memory usage, 1500 voice calls in the system

11.7 Performance under Denial-of-service Attack

In the final measurement of this thesis, a Denial-of-Service (DoS) attack is simulated by sending looping code to the SigComp prototype acting as a P-CSCF. The aim of the measurement is to test whether the protection offered by the UDVM cycle limit is sufficient. The bytecode corresponding to the following program written in the UDVM assembly language is used:

```
:start
INPUT-BITS(16, 1024, loop)
JUMP(start)
:loop
SHA-1(0, 1, 1024)
JUMP(loop)
```

The assembly contains two loops formed by the JUMP instructions. During each iteration of the first loop, 16 bits are read from the compressed message using the INPUT-BITS instruction. [RFC 3320] specifies that if the UDVM reads successfully *n* bits of compressed data, the number of available UDVM cycles is increased by *n* times *cycles_per_bit*. The *cycles_per_bit* parameter specifies the number of UDVM cycles available to decompress each bit in a SigComp message, and its value can be 16, 32, 64 or 128. In this measurement, the minimum value, 16, is used, meaning that each successful INPUT-BITS instruction increments the number of available UDVM cycles by 256. The first loop is over when the last complete 16-bit block has been read. In the second loop, SHA-1 message digest values are calculated over one-byte sequences until all the available UDVM cycles have been used. The cost of each SHA-1 instruction is 1 + *length*, i.e. two UDVM cycles in this case.

In the measurement, SigComp messages carried the bytecode of the assembly presented above in the message header and an INVITE message in the payload of the message.

The length of the INVITE message used was 1437 bytes. The measurement was carried out on the Pentium 4 3.0 GHz Hyper-Threading platform. It was observed that the looping code managed to use on average 274 milliseconds of CPU time per message before running out of UDVM cycles. In addition, a rate of at least eight messages per second was sufficient to place a CPU load of one hundred percent, i.e. consume all capacity of the P-CSCF.

The only protection the SigComp specifications offer against DoS attacks is the UDVM cycle limit. However, this protection is clearly not enough. It can only limit the amount of damage that can be caused, but does not remove the problem. If our DoS attack was repeated in a system using a larger value for the parameter *cycles_per_bit* than the minimum, and using a longer SIP message in the payload of the SigComp message, the attack would have been even more successful. What SigComp needs is additional protection mechanisms. One such mechanism is a bytecode verifier; even the simplest kind of verifier would be able to detect the malicious nature of our bytecode.

12 Conclusion

The main goal of this thesis was to examine the performance of the SigComp protocol through measurements performed on the SigComp prototype implemented as a part of the thesis work. The secondary goals were (1) to describe the way SigComp functionality can be implemented and (2) to examine the way to minimise the load SigComp places on the network node performing compression and decompression.

In this thesis, a SigComp prototype was implemented and its performance was evaluated. The performance measurements were divided into three phases. In the first and second phases of the measurements, the performance of the SigComp protocol was studied. In the third phase of the measurements, the focus was on the performance of the SigComp prototype. In the measurements, the prototype acted as a P-CSCF, decompressing SIP signalling traffic initiated from the RAN side and compressing SIP traffic terminating to the RAN side.

A modified version of the LZSS compression algorithm was implemented and a decompression algorithm written in the UDVM assembly language to process the output of the modified LZSS algorithm. Also a UDVM interpreter, which compiles UDVM assembly language programs into UDVM bytecode, was implemented. The SigComp prototype was implemented as a multithreaded application. Also another program was designed to generate SIP signalling traffic for the SigComp prototype.

To the author’s best knowledge, SigComp performance on the core network side has not been evaluated and techniques for optimising SigComp performance have not been studied in detail before. In addition, this thesis is the first to describe how SigComp functionality can be implemented. We implemented a SigComp prototype and evaluated its performance through measurements. The results obtained in the first phase of the measurements carried out for this thesis show how SigComp should be configured to optimise its performance and minimise the load SigComp places. The results of the second phase of the measurements demonstrate SigComp performance in various contexts. Finally, the results obtained in the third phase of the measurements demonstrate the performance of a network node performing compression and decompression of SIP messages.

12.1 Advantages and Limitations of Signalling Compression

SigComp has numerous advantages. It is generic: besides SIP, it can be used to compress any text-based protocol, for example the Real Time Streaming Protocol (RTSP). The UDVM approach allows SigComp to be flexible; since the decompression algorithm is supplied together with the first message, any compression algorithm can be used. SigComp can run on a variety of platforms including mobile terminals, because of the possibility to select the amount of memory, compression algorithm and compression mechanisms used at the sending endpoint. Also other services than SIP-based call setup can benefit from the use of SigComp. It can be used to decrease the delay of SIP-based services like push-to-talk over cellular, instant messaging and presence information. Finally, SigComp is able to coexist with other compression mechanisms such as robust header compression.

SigComp also has a number of limitations. Although the UDVM brings flexibility to SigComp, it also introduces some challenges. Firstly, the ability to upload the decompression algorithm to the network node performing SIP message decompression makes SigComp vulnerable to Denial-of-Service attacks. The use of UDVM cycle limits does not remove this problem, although it limits the amount of damage that a malicious user can cause. Some additional mechanism is likely to be required. An example of such a mechanism is the bytecode verifier used in the Java Virtual Machine approach. Secondly, a virtual machine like the UDVM is rather complex. It is challenging to design a UDVM that is robust and fast enough. In addition to the UDVM being complex, also the entire protocol is getting more and more complex with each new Internet Draft that is published to fix the problems that have been found in SigComp.

Because SigComp builds compression dictionaries dynamically and also uses the static SIP/SDP dictionary, most existing compression algorithms cannot be used with SigComp without modifying or even redesigning them. Many compression algorithms are also proprietary and there may be patent issues restricting their usage. Most well-known compression algorithms have countless different variations. A UDVM bytecode written for one variation will not work with a different version of the same algorithm. In addition to the modifications required to existing compression algorithms, SigComp also requires changes to existing SIP implementations.

SigComp increases the amount of state information that has to be stored for each ongoing session. In addition, the system must be able to access this information rapidly in order to keep the compression and decompression times as low as possible. The maximum amount of state memory that can be used by a single session is 131 kilobytes. It was observed that if shared compression and a decompression memory of sufficient size are used, this limit can be reached without difficulties.

It remains to be seen whether the use of SigComp alone will be sufficient. It can only affect the RAN delay; the core network delay, bearer establishment and the overhead added by lower protocol layers will not be affected. Compression ratios that would reduce the size of SIP signalling messages to the same level as in the case of GSM seem unachievable.

12.2 Considerations

12.2.1 Performance of SigComp Protocol

The following considerations were made:

1. The compression time of the compressor and the decompression time of the UDVM depend highly on the length of the uncompressed message, SigComp mechanisms used, the amount of previous state information available, search technique used by the compression algorithm, size of the decompression memory, amount of bytes used from the static SIP/SDP dictionary, length of the compressor’s look-ahead buffer etc. For instance, on an Intel Pentium 4 3.0 GHz platform, the compression time of a single message varied between 669 and 3730 microseconds, and the decompression time of a single message between 843 and 3329 microseconds. Depending on the type of the message sequence and the issues listed above, the compression time of the entire sequence varied

between 5091 and 32865 microseconds, while the decompression time of the entire sequence was between 5875 and 25410 microseconds.

2. The overhead added by the UDVM approach is considerable compared to the use of a fixed decompression algorithm. It is not unusual that the decompression time of a message is longer than its compression time.
3. The use of hashing in the compressor of the SigComp prototype reduced the amount of compression time consumed by 64% compared to linear searching. When hashing is used, most of the compression time is spent in organizing the hash map. It is beneficial to store the hash tables between messages instead of re-generating the hash table each time a new message is compressed.
4. It was observed that compression is most efficient when the maximum size of substituted strings is 258 bytes. Although as much as eight bits are required to encode length values, this additional overhead is minor compared to the savings achieved because longer matches can be encoded using a single offset/length pair. The use of long matches also allows reductions in compression and decompression times.
5. Most of the content in the static SIP/SDP dictionary is likely to be useless when compressing any SIP sequence. If too large part of the static dictionary is inserted to the search buffer, the penalty in compression time is significant. It was found out that the best results are achieved by using static dictionary priorities from one to two or from one to three.
6. When dynamic compression, shared compression, or both of them are applied, the calculation of SHA-1 hashes constitutes a considerable part of the compression and decompression times. It was observed that the calculation of the hash constitutes 25-40 percent of the compression time, depending on the message. It has an effect of the same magnitude on decompression times.
7. Basic, i.e. message-by-message compression is practically useless, because it offers very poor compression ratios. The use of static compression with the ability to save the bytecode clearly improves the efficiency of the compression. However, only dynamic compression and shared compression offer satisfactory compression ratios. The best compression ratio, 22.3%, was achieved by shared compression.
8. The better performance of dynamic and shared compressions regarding the achievable compression ratios does not come without a cost: when compressing, shared compression can be two times slower and dynamic compression 1.4 times slower than basic compression. However, when decompressing, dynamic and shared compressions perform slightly better than the less advanced compression mechanisms.
9. The use of shared compression has a negative impact on compression time, especially when a technique like hashing is used. It was observed that in the worst case, the compression time of shared compression is 1.58 times the compression time of dynamic compression, while the improvement in compression ratio is only 8.4%. However, restricting the length of the shared states can reduce the negative impact of shared compression on compression time.
10. Because shared compression uses considerably more buffer space than dynamic compression, dynamic compression can achieve higher compression ratios in the case of the last messages of a large sequence. Therefore, it might be beneficial to switch off shared compression after the first few messages of the sequence have been sent.

11. The combined size of the messages in a sequence and the mechanisms used are important factors when selecting the most appropriate decompression memory size. For instance, when a message sequence with a size of 10640 bytes was being compressed, shared compression was the only mechanism that was found to benefit from a decompression memory larger than 4096 bytes. A decompression memory of size 16384 is too large for most SIP signalling flows.
12. SigComp benefits greatly from the use of reliable transport instead of unreliable transport, especially in the case of message sequences with a low number of messages. It was found out that when shared compression is applied, the combined size of compressed messages of a basic voice session initiation sequence transmitted over TCP was only 61% of the size achieved with UDP based transport.
13. SigComp has been designed especially for narrowband links. When the bit rate of the signalling link is more than 64 kbps, the performance improvement offered by SigComp may not be great enough to justify the use of compression. With a signalling link bit rate of 9.6 kbps, SigComp is able to reduce the RAN delay by approximately 70%. In contrast, with a bit rate of 256 kbps the improvement is only about 20%.
14. It was observed that the more messages a signalling flow has and the larger is the combined size of the messages, the better compression ratios can be achieved.
15. The compression ratio achieved for a registration sequence in a 3GPP release 5 network was only 0.88. Therefore, it might not be worth the effort to compress the registration sequence at all.
16. The use of alternative signalling flows for a video call in a 3GPP release 5 network resulted in lower compression and decompression times and smaller compressed size for the entire session initiation sequence. This suggests that considerable improvements can be achieved by redesigning inefficient signalling flows.

12.2.2 SigComp Prototype

1. The shared resources, i.e. the state handler and the compressor array can become the bottlenecks of the SigComp architecture unless they are not carefully designed. This is because threads require exclusive access when they modify the content of the shared resources. It is of vital importance to minimise the amount of time each thread holds exclusive access to a shared resource, and to use right scheduling policies to control access to the shared resources. It is also important to use data structures that minimise the cost of insert, delete and find operations.
2. Overhead caused by thread switches and especially queuing for access to the shared resources increases considerably as more concurrency is allowed in the system. However, a high number of threads are required to keep the time messages wait for service low. Under the maximum traffic load and with the maximum amount of concurrency in the system used, the per-message overhead added by thread switches and queuing time to the shared resources formed almost 17 percent of the total time messages stay in the system.
3. The SigComp prototype benefits from the use of a CPU supporting Intel’s Hyper-Threading technology. The gains of using a Hyper-Threading CPU are the bigger the more simultaneously active threads the system has. It was observed that two CPUs, of which only one supported the Hyper-Threading technology, had equal performance when messages were not processed

concurrently. However, when a workload of 1500 simultaneous voice calls was used and messages were processed concurrently, the time in system per message was 1.43 times less for the Hyper-Threading CPU.

4. The memory requirement of SigComp is relatively large: the signalling of 1500 simultaneous voice calls uses 218 megabytes of memory, and the signalling of the same number of video calls consumes 781 megabytes of memory. However, the memory requirement can be reduced at the cost of achievable compression ratios by using less advanced compression mechanisms and smaller decompression memory size.
5. The time in system, CPU load and memory usage depend highly on the type of the SIP signalling traffic being compressed and decompressed. The video call establishment and release sequence in a 3GPP release 5 network that was used placed a higher load on the CPU, had longer delays and used more memory than a basic voice call establishment and release sequence.
6. A Denial-of-Service attack, in which SigComp messages containing looping code were send to the P-CSCF, was simulated. It was observed that a constant stream of eight messages per second was enough to consume all CPU power of the P-CSCF with the parameters and messages that were used.

12.3 Future Research

This thesis focused on SigComp performance on the core network side. Another equally interesting topic is the performance of SigComp in the 3G mobile terminals supporting SIP. It would be interesting to see whether the same algorithms that perform well in the core network nodes performing compression and decompression are optimal for use in the mobile terminals. After all, since the performance of the terminals keeps increasing all the time, there might actually be more resources available on the UE side than in the core network element which may have to take care of thousands of simultaneous sessions.

SigComp security risks are another issue that has not yet been studied in detail. Much is to be learned from for instance the Java programming language, since both the UDVM and Java Virtual Machine use uploadable bytecodes. The design of a UDVM bytecode verifier might become a necessity.

The use of highly optimised data structures in SigComp-aware compression algorithms is another issue worth studying. Besides hash tables, there are a number of other data structures that can be used in longest-match string searching. These include for example linked lists, search tries and binary search trees. Also the hash table approach can be improved by storing keys of different lengths. The performance of algorithms relying on linear searching can be improved significantly by organizing carefully their search buffers.

Since SigComp makes heavy use of shared resources, access to these resources is worth optimising. One way of achieving performance improvements is through the use of highly optimised scheduling policies.

Yet another topic for further work is the parallel use of payload compression like SigComp and header compression like ROHC. In the measurements of this thesis, header compression was not applied.

13 References

[3GPP TS 23.002] 3GPP TS 23.002 V6.5.0, Technical Specification Group Services and Systems Aspects; Network Architecture (Release 5), June 2004

[3GPP TS 23.228] 3GPP TS 23.228 V6.6.0, Technical Specification Group Services and System Aspects, IP Multimedia Subsystem (IMS); Stage 2 (Release 6), June 2004.

[3GPP TS 24.228 2004] 3GPP TS 24.228 V5.9.0, Technical Specification Group Core Network; Signalling flows for the IP multimedia call control based on SIP and SDP; Stage 3 (Release 5), June 2004.

[Andreadis 2003] Andreadis et al. 2003. Protocols for High-Efficiency Wireless Networks. Kluwer Academic Publishers.

[Andrews 2000] Andrews, G. R. 2002. Foundations of Multithreaded, Parallel and Distributed Programming. Addison Wesley.

[Balazs 2004] Balazs, A. 2004. Push-to-talk Performance over GPRS.

[Camarillo 2002] Camarillo, G. 2002. SIP demystified. McGraw-Hill.

[Camarillo 2004] Camarillo, G. & Garcia-Martin, M. 2004. The 3G IP Multimedia Subsystem (IMS) – Merging the Internet and the Cellular Worlds. Wiley.

[Doldi 2003] Doldi, L. 2003. UML 2 illustrated : developing real-time and communications systems. Toulouse: TransMeth Sud-Ouest.

[Draft Price] Price, R. et al. 2003. SigComp User Guide. <draft-price-rohc-sigcomp-user-guide-02.txt>. Internet-Draft. Work in progress.

[Draft Roach] Roach, A. 2004. A Negative Acknowledgement Mechanism for Signalling Compression. <draft-ietf-rohc-sigcomp-nack-01>. Internet-Draft. Work in progress.

[Draft Surtees] Surtees, A. et al. 2004. Implementer's Guide for SigComp. <draft-ietf-rohc-sigcomp-impl-guide-04.txt>. Internet-Draft. Work in progress.

[Fidrich 2003] Fidrich, M. et al. 2003. SIP compression.

[FIPS 180] Secure Hash Standard. Federal Information Processing Standards Publication 180-1. April 1995.

[Foster 2002] Foster, G. et al. 2002. Performance Estimation of Efficient UMTS Packet Voice Call Control. Motorola & Cork Institute of Technology.

[Gamma 1995] Gamma, E. et al. 1995. Design patterns. Elements of reusable object-oriented software. Addison-Wesley.

[Iversen 2005] Iversen, V. B. 2005. Teletraffic Engineering Handbook. ITU-D.

[Jain 1991] Jain, R. 1991. The art of computer systems performance analysis. Techniques for experimental design, measurement, simulation and modeling. John Wiley & Sons.

[Lindholm 1997] Lindholm, T. et Yellin, F. 1997. The Java Virtual Machine Specification. Addison-Wesley.

[Marr 2002] Marr, D. T. et al. 2002. Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal.Vol 6. p4-15.

[Nordberg 2003] Nordberg, M. et al. 2003. Improving SigComp performance through extended operations. IEEE Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003. IEEE 58th 5: 3425-3428.

[Nortel 2000] Nortel Networks 2000. A Comparison Between GERAN Packed-Switched Call Setup Using SIP and GSM Circuit-Switched Call Setup Using RIL3-CC, RIL3-MM, RIL3-RR and DTAP Rev. 0.3.

[OMA-TS- POC] Open Mobile Alliance 2005. OMA PoC Control Plane. Candidate Version 1.0 – 17 March 2005.

[Patterson 1998] Patterson, D. A. et Hennessy, J. L. 1998. Computer organization and design: the hardware/software interface. 2nd Edition. Morgan Kaufmann Publishers.

[Preiss 1999] Preiss, B. R. 1999. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. Wiley.

[RFC 959] Postel, J. et Reynolds, J. 1985. File Transfer Protocol (FTP). RFC 959.

[RFC 3095] Bormann, C. et al. 2001. Robust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. RFC 3095.

[RFC 3173] Shacham, A. et al. 2001. IP Payload Compression Protocol (IPComp). RFC 3173.

[RFC 3174] Eastlake, D. 2001. US Secure Hash Algorithm 1 (SHA1). RFC 3174.

[RFC 3261]	Rosenberg, J. et al. 2002. SIP: Session Initiation Protocol. RFC 3261.
[RFC 3320]	Price, R. et al. 2003. Signaling Compression (SigComp). RFC 3320.
[RFC 3321]	Hannu, H. et al. 2003. Signaling Compression (SigComp) – Extended Operations. RFC 3321.
[RFC 3322]	Hannu H. 2003. Signaling Compression (SigComp) Requirements & Assumptions. RFC 3322.
[RFC 3485]	Garcia-Martin, M. et al. 2003. The Session Initiation Protocol (SIP) and Session Description Protocol (SDP) Static Dictionary for Signaling Compression (SigComp). RFC 3485.
[RFC 3486]	Camarillo, G. 2003. Compressing the Session Initiation Protocol (SIP). RFC 3486.
[Sayood 2006]	Sayood, K. 1996. Introduction to data compression. Morgan Kaufmann Publishers.
[Tanenbaum 2001]	Tanenbaum, A. S. 2001. Modern Operating Systems. 2nd Edition. Prentice Hall.
[Weiss 1999]	Weiss, M. A. 1999. Data structures & algorithm analysis in Java. Addison Wesley.
[West 2002]	West, M. et al. 2002. IP header and signaling compression for 3G systems. Siemens/Roke Manor Research Limited, UK.
[Ziv 1977]	Ziv, J. & Lempel, A. 1977. A Universal Algorithm for Data Compression. IEEE Transactions on Information Theory.

14 Appendices

14.1 Appendix A – The UDVM Instruction Set

Instruction	Bytecode value	Cost in UDVM cycles
DECOMPRESSION-FAILURE	0	1
AND	1	1
OR	2	1
NOT	3	1
LSHIFT	4	1
RSHIFT	5	1
ADD	6	1
SUBTRACT	7	1
MULTIPLY	8	1
DIVIDE	9	1
REMAINDER	10	1
SORT-ASCENDING	11	$1+k\times(ceiling(\log_2(k))+n)$
SORT-DESCENDING	12	$1+k\times(ceiling(\log_2(k))+n)$
SHA-1	13	$1+length$
LOAD	14	1
MULTILOAD	15	$1+n$
PUSH	16	1
POP	17	1
COPY	18	$1+length$
COPY-LITERAL	19	$1+length$
COPY-OFFSET	20	$1+length$
MEMSET	21	$1+length$
JUMP	22	1
COMPARE	23	1
CALL	24	1
RETURN	25	1
SWITCH	26	$1+n$
CRC	27	$1+length$
INPUT-BYTES	28	$1+length$
INPUT-BITS	29	1
INPUT-HUFFMAN	30	$1+n$
STATE-ACCESS	31	$1+state\_length$
STATE-CREATE	32	$1+state\_length$
STATE-FREE	33	1

OUTPUT	34	$1 + output\_length$
END-MESSAGE	35	$1 + state\_length$

14.2 Appendix B – LZSS Assembly

```
at (32)

:index                                                          pad (2)
:length_value                                                   pad (2)
:old_pointer                                                    pad (2)

at (42)

:requested_feedback_location                                    pad (1)
:requested_feedback_length                                      pad (1)
:requested_feedback_field                                       pad(12)
:hash_start                                                     pad(8)

at (64)

:byte_copy_left                                                 pad (2)
:byte_copy_right                                                pad (2)
:input_bit_order                                                pad (2)
:decompressed_pointer                                           pad (2)

:returned_parameters_location                                   pad (1)
:returned_sigcomp_version                                       pad (1)
:length_of_partial_state_id_a                                   pad (1)
:partial_state_identifier_a                                     pad (6)
:length_of_partial_state_id_b                                   pad (1)
:partial_state_identifier_b                                     pad (20)
:extended_flags                                                 pad (2)
:shared_state_id                                                pad (6)
:padding                                                        pad (6)
:minimum_access_length                                          pad (2)
:announcement_location                                          pad (2)
:decompressed_start                                             pad (2)
:decompressed_length                                            pad (2)
:shared_hash_length                                             pad (2)

align (64)

:initialize_memory

STATE-ACCESS (dictionary_id, 6, 0, 0, 1024, 0)

set (udvm_memory_size, 8192)
set (state_length, (udvm_memory_size - 64))

MULTILOAD (64, 4, circular_buffer, udvm_memory_size, 0, circular_buffer)
```

```

:decompress_sigcomp_message

INPUT-BYTES (1, extended_flags, !)
COMPARE ($extended_flags, 32768, initialize_state_announcement,
access_shared_state, access_shared_state)

:access_shared_state

INPUT-BYTES (6, shared_state_id, !)
STATE-ACCESS (shared_state_id, 6, 0, 0, $decompressed_start, 0)

:initialize_state_announcement

MULTILOAD (minimum_access_length, 4, 6, length_of_partial_state_id_a,
$decompressed_pointer, 5120)
COPY-LITERAL (padding, 8, $decompressed_pointer)

LSHIFT ($extended_flags, 1)
COMPARE ($extended_flags, 32768, algorithm_start, announce_acked_state_id,
announce_acked_state_id)

:announce_acked_state_id

LOAD (length_of_partial_state_id_a, 1536)
INPUT-BYTES (6, partial_state_identifier_a, !)
LOAD (announcement_location, length_of_partial_state_id_b)

:algorithm_start

:next_character

INPUT-HUFFMAN (index, end_of_message, 2, 9, 0, 255, 16384, 4, 4096, 8191, 1)

COMPARE ($index, 8192, length, end_of_message, literal)

:literal

set (index_lsb, (index + 1))

OUTPUT (index_lsb, 1)
COPY-LITERAL (index_lsb, 1, $decompressed_pointer)
JUMP (next_character)

:length

INPUT-BITS (4, length_value, !)
ADD ($length_value, 3)
LOAD (old_pointer, $decompressed_pointer)
COPY-OFFSET ($index, $length_value, $decompressed_pointer)
OUTPUT ($old_pointer, $length_value)
JUMP (next_character)
```

```

:end_of_message

LSHIFT ($extended_flags, 1)
COMPARE ($extended_flags, 32768, end, announce_shared_state,
announce_shared_state)

:announce_shared_state

COPY-LITERAL (decompressed_length, 1, $announcement_location)

set (buffer_size, (udvm_memory_size - circular_buffer))

MULTILOAD (decompressed_length, 2, 65528, $decompressed_pointer)
SUBTRACT ($shared_hash_length, $decompressed_start)
REMAINDER ($shared_hash_length, buffer_size)
ADD ($decompressed_length, $shared_hash_length)

LOAD ($decompressed_start, $decompressed_length)
SHA-1 ($decompressed_start, $shared_hash_length, $announcement_location)


:end

set (hash_length, (state_length + 8))

LOAD (requested_feedback_location, 1158)
MULTILOAD (hash_start, 4, state_length, 64, decompress_sigcomp_message, 6)

SHA-1 (hash_start, hash_length, requested_feedback_field)

END-MESSAGE (requested_feedback_location, returned_parameters_location,
state_length, 64, decompress_sigcomp_message, 6, 0)

:dictionary_id

byte (0xfb, 0xe5, 0x07, 0xdf, 0xe5, 0xe6)

:circular_buffer
```

14.3 Appendix C – SIP Message Sequences

14.3.1 Basic Voice Call

14.3.1.1 INVITE

INVITE sip:7040005004@192.168.55.54 SIP/2.0
Via: SIP/2.0/UDP 192.168.55.61;comp=sigcomp
Route: <sip:131.160.31.19;comp=sigcomp>
Call-ID: b0957730@192.168.55.61-1000
CSeq: 48733 INVITE
Contact: <sip:8881003@192.168.55.61;comp=sigcomp>

```
User-Agent: 3Com ICD 1.0.1.2.7
From: <sip:8881003@192.168.55.61>;tag=f3d981df
To: <sip:7040005004@192.168.55.54>
Proxy-Authorization: Digest
username="8881003",realm="192.168.55.61",nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093
",opaque="",uri="sip:7040005004@192.168.55.54",response="6629fae49393a0539745097850
7c4ef1"
Content-Length: 105
Content-Type: application/sdp

v=0
o=username 0 48732 IN IP4 192.168.55.61
s=
c=IN IP4 192.168.55.61
t=0 0
m=audio 7206 RTP/AVP 0
```

14.3.1.2 100 Trying

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 192.168.55.61:5060;comp=sigcomp;received=192.168.55.61
From: <sip:8881003@192.168.55.61>;tag=f3d981df
To: <sip:7040005004@192.168.55.54>;tag=124871409
Call-ID: b0957730@192.168.55.61-1000
CSeq: 48733 INVITE
Content-Length: 0
```

14.3.1.3 180 Ringing

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 192.168.55.61:5060;comp=sigcomp;received=192.168.55.61
Record-Route: 131.160.31.19;comp=sigcomp
From: <sip:8881003@192.168.55.61>;tag=f3d981df
To: <sip:7040005004@192.168.55.54>;tag=124871409
Call-ID: b0957730@192.168.55.61-1000
CSeq: 48733 INVITE
Contact: <sip:192.168.55.54:5060>
Allow: INVITE, ACK, CANCEL, BYE, OPTIONS, PRACK
Accept: application/sdp
Content-Length: 161
Content-Type: application/sdp

v=0
o=username 0 48732 IN IP4 192.168.55.61
s=Basic Session
c=IN IP4 192.168.55.15
t=0 0
m=audio 20000 RTP/AVP 0 103 19
a=rtpmap:103 telephone-event/8000
```

14.3.1.4 200 OK to INVITE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.55.61:5060;comp=sigcomp;received=192.168.55.61
Record-Route: 131.160.31.19;comp=sigcomp
From: <sip:8881003@192.168.55.61>;tag=f3d981df
```

```
To: <sip:7040005004@192.168.55.54>;tag=124871409
Call-ID: b0957730@192.168.55.61-1000
CSeq: 48733 INVITE
Contact: <sip:192.168.55.54:5060>
Allow: INVITE, ACK, CANCEL, BYE, OPTIONS, PRACK
Accept: application/sdp
Content-Length: 161
Content-Type: application/sdp

v=0
o=username 0 48732 IN IP4 192.168.55.61
s=Basic Session
c=IN IP4 192.168.55.15
t=0 0
m=audio 20000 RTP/AVP 0 103 19
a=rtpmap:103 telephone-event/8000
```

14.3.1.5 ACK

```
ACK sip:192.168.55.54:5060 SIP/2.0
Via: SIP/2.0/UDP 192.168.55.61;comp=sigcomp
Route: <sip:131.160.31.19;comp=sigcomp>
Call-ID: b0957730@192.168.55.61-1000
CSeq: 48733 ACK
User-Agent: 3Com ICD 1.0.1.2.7
From: <sip:8881003@192.168.55.61>;tag=f3d981df
To: <sip:7040005004@192.168.55.54>;tag=124871409
Proxy-Authorization: Digest
username="8881003",realm="192.168.55.61",nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093
",opaque="",uri="sip:192.168.55.54:5060",response="6629fae49393a05397450978507c4ef1"
Content-Length: 0
```

14.3.1.6 BYE

```
BYE sip:192.168.55.54:5060 SIP/2.0
Via: SIP/2.0/UDP 192.168.55.61;comp=sigcomp
Route: <sip:131.160.31.19;comp=sigcomp>
Call-ID: b0957730@192.168.55.61-1000
CSeq: 48734 BYE
User-Agent: 3Com ICD 1.0.1.2.7
From: <sip:8881003@192.168.55.61>;tag=f3d981df
To: <sip:7040005004@192.168.55.54>;tag=124871409
Proxy-Authorization: Digest
username="8881003",realm="192.168.55.61",nonce="1cec4341ae6cbe5a359ea9c8e88df84f",op
aque="",uri="sip:192.168.55.54:5060",response="4767ead078938ad80e7b3a49defdcd64"
Content-Length: 0
```

14.3.1.7 200 OK to BYE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.55.61:5060;comp=sigcomp;received=192.168.55.61
From: <sip:8881003@192.168.55.61>;tag=f3d981df
To: <sip:7040005004@192.168.55.54>;tag=124871409
Call-ID: b0957730@192.168.55.61-1000
CSeq: 48734 BYE
Content-Length: 0
```

14.3.2 Basic Video Call

14.3.2.1 INVITE

```
INVITE sip:888000@192.168.57.80:5061;transport=UDP;user=phone SIP/2.0
Via: SIP/2.0/UDP 192.168.57.71:5060;comp=sigcomp;branch=z9hG4bK11006044821
Route: <sip:131.160.31.19;comp=sigcomp>
From: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>;tag=229614673
To: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>
Call-ID: 11100604432@192.168.57.71
CSeq: 1 INVITE
Contact: <sip:192.168.57.71:5060;comp=sigcomp;transport=UDP>
Allow: INVITE, ACK, CANCEL, BYE, OPTIONS, PRACK
Accept: application/sdp
Supported: 100rel
Max-Forwards: 70
Privacy: none
P-Asserted-Identity: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>
Content-Length:  355
Content-Type: application/sdp

v=0
o=- 1100604476 1100604476 IN IP4 192.168.57.111
s=Basic Session
c=IN IP4 192.168.57.111
t=0 0
a=sendrecv
m=audio 30000 RTP/AVP 96 4 8 0
a=rtpmap:96 AMR/8000
m=video 30002 RTP/AVP 103 104 34 105
a=rtpmap:103 H263-2000/8000
a=fmtp:103 profile=0;level=10
a=rtpmap:104 H263-1998/8000
a=rtpmap:105 MP4V-ES/90000
a=fmtp:105 profile-level-id=8
```

14.3.2.2 100 Trying

```
SIP/2.0 100 Trying
From: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>;tag=229614673
To: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>
Call-ID: 11100604432@192.168.57.71
CSeq: 1 INVITE
Via: SIP/2.0/UDP 192.168.57.71:5060;comp=sigcomp;branch=z9hG4bK11006044821
Content-Length: 0
```

14.3.2.3 180 Ringing

```
SIP/2.0 180 Ringing
From: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>;tag=229614673
To: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>;tag=1234
Call-ID: 11100604432@192.168.57.71
CSeq: 1 INVITE
Via: SIP/2.0/UDP 192.168.57.71:5060;comp=sigcomp;branch=z9hG4bK11006044821
Record-Route: 131.160.31.19;comp=sigcomp
Contact: <sip:192.168.57.80:5061>
Allow: INVITE,ACK,CANCEL,BYE,OPTIONS,INFO
```

```
Accept: application/SDP
Content-Length: 0
```

14.3.2.4 200 OK to INVITE

```
SIP/2.0 200 OK
From: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>;tag=229614673
To: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>;tag=1234
Call-ID: 11100604432@192.168.57.71
CSeq: 1 INVITE
Content-Type: APPLICATION/SDP
Content-Length: 163
Contact: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>;tag=1234
Via: SIP/2.0/UDP 192.168.57.71:5060;comp=sigcomp;branch=z9hG4bK11006044821
Record-Route: 131.160.31.19;comp=sigcomp
User-Agent: Pingtel/0.4.0

v=0
o=Pingtel 5 0 IN IP4 131.160.21.57
s=phone-call
c=IN IP4 131.160.21.57
t=0 0
m=audio 8766 RTP/AVP 0
m=video 8767 RTP/AVP 96
a=rtpmap:96 H263-1998/8000
```

14.3.2.5 ACK

```
ACK sip:888000@192.168.57.80:5061;transport=UDP;user=phone SIP/2.0
Via: SIP/2.0/UDP 192.168.57.71:5060;comp=sigcomp;branch=z9hG4bK11006044821
Route: <sip:131.160.31.19;comp=sigcomp>
From: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>;tag=229614673
To: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>;tag=1234
Call-ID: 11100604432@192.168.57.71
CSeq: 1 ACK
Max-Forwards: 70
Content-Length:  0
```

14.3.2.6 BYE

```
BYE sip:888000@192.168.57.80:5061 SIP/2.0
Via: SIP/2.0/UDP 192.168.57.71:5060;comp=sigcomp;branch=z9hG4bK11006044821
Route: <sip:131.160.31.19;comp=sigcomp>
From: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>;tag=229614673
To: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>;tag=1234
Call-ID: 11100604432@192.168.57.71
CSeq: 2 BYE
Content-Length: 0
User-Agent: Pingtel/0.6.1
```

14.3.2.7 200 OK to BYE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.57.71:5060;comp=sigcomp;branch=z9hG4bK11006044821
From: <sip:888000@192.168.57.80:5061;transport=UDP;user=phone>;tag=1234
To: <sip:+603002391@192.168.57.71:5060;transport=UDP;user=phone>;tag=229614673
Call-ID: 11100604432@192.168.57.71
CSeq: 2 BYE
```

```
Content-Length:  0
```

14.3.3 Push-to-talk over Cellular Session Establishment

14.3.3.1 INVITE

```
INVITE sip:PoCConferenceFactoryURI.networkA.net SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp
Route: <sip:SIPcoreA.networkA.net:7531;comp=sigcomp>
Call-ID: b0957730@networkA.net-1000
CSeq: 48733 INVITE
Contact: <sip:PoC-ClientA@networkA.net;comp=sigcomp>;+g.poc.talkburst
User-Agent: PoC-client/OMA1.0 Acme-Talk5000/v1.01
From: <sip:PoC-ClientA@networkA.net>;tag=f3d981df
To: <sip:PoC_ServerA@networkA.net>
P-Preferred-Identity: "PoC User A" <sip:PoC-UserA@networkA.net>
Accept-Contact: *;+g.poc.talkburst; require;explicit
Privacy: Id
Supported: Timer
Session-Expires: 1800;refresher=uac
Allow: INVITE,ACK,CANCEL,BYE,REFER,MESSAGE, SUBSCRIBE,NOTIFY, PUBLISH
Content-Length: 194
Content-Type: application/sdp

v=0
o=username 0 48732 IN IP6 5555::aaa:bbb:ccc:ddd
s=
t=0 0
c=IN IP6 5555::aaa:bbb:ccc:ddd
m=audio 3456 RTP/AVP 97
a=rtpmap:97 AMR
a=rtcp:5560
m=application 2000 udp TBCP
a=fmtp:TBCP queuing=1; tb_priority=2; timestamp=1
```

14.3.3.2 100 Trying

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp
From: <sip:PoC-ClientA@networkA.net>;tag=f3d981df
To: <sip:PoC_ServerA@networkA.net>
Call-ID: b0957730@networkA.net-1000
CSeq: 48733 INVITE
Content-Length: 0
```

14.3.3.3 200 OK to INVITE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp
Record-Route: sip:SIPcoreA.networkA.net:7531;comp=sigcomp
From: <sip:PoC-ClientA@networkA.net>;tag=f3d981df
To: <sip:PoC_ServerA@networkA.net>
Call-ID: b0957730@networkA.net-1000
CSeq: 48733 INVITE
Contact: <sip:Pre-establishedSessionIdentityA@PoC-ServerA.networkA.net>;+g.poc.talkburst
Accept: application/sdp
P-Asserted-Identity: <sip:PoC-ServerA@networkA.net>
```

```
Server: PoC-serv/OMA1.0
Require: Timer
Session-Expires: 1800;refresher=uac
Allow: INVITE,ACK,CANCEL,BYE,REFER,MESSAGE, SUBSCRIBE,NOTIFY, PUBLISH
Content-Length: 197
Content-Type: application/sdp

v=0
o=username 0 48732 IN IP6 57777::eee:fff:aaa:bbb
t=0 0
c=IN IP6 57777::eee:fff:aaa:bbb
m=audio 57787 RTP/AVP 97
a=rtpmap:97 AMR
a=rtcp:57000
m=application 57790 udp TBCP
a=fmtp:TBCP queuing=1; tb_priority=2; timestamp=1
```

14.3.3.4 ACK

```
ACK sip:PoCConferenceFactoryURI.networkA.net SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp
Route: <sip:SIPcoreA.networkA.net:7531;comp=sigcomp>
Call-ID: b0957730@networkA.net-1000
CSeq: 48733 ACK
User-Agent: PoC-client/OMA1.0 Acme-Talk5000/v1.01
From: <sip:PoC-ClientA@networkA.net>;tag=f3d981df
To: <sip:PoC_ServerA@networkA.net>
Content-Length: 0
```

14.3.3.5 BYE

```
BYE sip:PoCConferenceFactoryURI.networkA.net SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp
Route: <sip:SIPcoreA.networkA.net:7531;comp=sigcomp>
Call-ID: b0957730@networkA.net-1000
CSeq: 48734 BYE
User-Agent: PoC-client/OMA1.0 Acme-Talk5000/v1.01
From: <sip:PoC-ClientA@networkA.net>;tag=f3d981df
To: <sip:PoC_ServerA@networkA.net>
Content-Length: 0
```

14.3.3.6 200 OK to BYE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp
From: <sip:PoC-ClientA@networkA.net>;tag=f3d981df
To: <sip:PoC_ServerA@networkA.net>
Call-ID: b0957730@networkA.net-1000
CSeq: 48734 BYE
Content-Length: 0
```

14.3.4 3GPP Video Call

14.3.4.1 INVITE

```
INVITE tel:+1-212-555-2222 SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
```

```
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>
P-Preferred-Identity: "John Doe" <sip:user1_public1@home1.net>
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
Privacy: none
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 127 INVITE
Require: precondition, sec-agree
Proxy-Require: sec-agree
Supported: 100rel
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321;
port-c=8642; port-s=7531
Contact: <sip:[5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp>
Allow: INVITE, ACK, CANCEL, BYE, PRACK, UPDATE, REFER, MESSAGE
Content-Type: application/sdp
Content-Length: 523

v=0
o=- 2987933615 2987933615 IN IP6 5555::aaa:bbb:ccc:ddd
s=-
c=IN IP6 5555::aaa:bbb:ccc:ddd
t=0 0
m=video 3400 RTP/AVP 98 99
b=AS:75
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos none remote sendrecv
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
a=rtpmap:99 MP4V-ES
m=audio 3456 RTP/AVP 97 96
b=AS:25.4
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos none remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.4.2 100 Trying

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 127 INVITE
Content-Length: 0
```

14.3.4.3 183 Session Progress

```
SIP/2.0 183 Session Progress
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
```

```
Record-Route: <sip:pcscf2.visited2.net;lr>, <sip:scscf2.home2.net;lr>,
<sip:scscf1.home1.net;lr>, <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>
P-Asserted-Identity: "John Smith" <sip:user2_public1@home2.net>, <tel:+1-212-555-2222>
Privacy: none
P-Media-Authorization:
0020000100100101706466322e76697369746564322e6e6574000c020139425633303732
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 127 INVITE
Require: 100rel
Contact: <sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp>
Allow: INVITE, ACK, CANCEL, BYE, PRACK, UPDATE, REFER, MESSAGE
RSeq: 9021
Content-Type: application/sdp
Content-Length: 584

v=0
o=- 2987933623 2987933623 IN IP6 5555::eee:fff:aaa:bbb
s=-
c=IN IP6 5555::eee:fff:aaa:bbb
t=0 0
m=video 10001 RTP/AVP 98 99
b=AS:75
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=conf:qos remote sendrecv
a=rtpmap:98 H263
a=rtpmap:99 MP4V-ES
a=fmtp:98 profile-level-id=0
m=audio 6544 RTP/AVP 97 96
b=AS:25.4
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=conf:qos remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.4.4 PRACK to 183

```
PRACK sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>,
<sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 128 PRACK
Require: precondition, sec-agree
```

```
Proxy-Require: sec-agree
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321;
port-c=8642; port-s=7531
RAck: 9021 127 INVITE
Content-Type: application/sdp
Content-Length: 509

v=0
o=- 2987933615 2987933616 IN IP6 5555::aaa:bbb:ccc:ddd
s=-
c=IN IP6 5555::aaa:bbb:ccc:ddd
t=0 0
m=video 3400 RTP/AVP 98
b=AS:75
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
m=audio 3456 RTP/AVP 97 96
b=AS:25.4
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.4.5 200 OK to PRACK

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
CSeq: 128 PRACK
Content-Type: application/sdp
Content-Length: 562

v=0
o=- 2987933623 2987933624 IN IP6 5555::eee:fff:aaa:bbb
s=-
c=IN IP6 5555::eee:fff:aaa:bbb
t=0 0
m=video 10001 RTP/AVP 98
b=AS:75
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=conf:qos remote sendrecv
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
m=audio 6544 RTP/AVP 97 96
```

```
b=AS:25.4
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=conf:qos remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.4.6 UPDATE

```
UPDATE sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>,
<sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 129 UPDATE
Require: sec-agree
Proxy-Require: sec-agree
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321;
port-c=8642; port-s=7531
Content-Type: application/sdp
Content-Length: 517

v=0
o=- 2987933615 2987933617 IN IP6 5555::aaa:bbb:ccc:ddd
s=-
c=IN IP6 5555::aaa:bbb:ccc:ddd
t=0 0
m=video 3400 RTP/AVP 98
b=AS:75
a=curr:qos local sendrecv
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
m=audio 3456 RTP/AVP 97 96
b=AS:25.4
a=curr:qos local sendrecv
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.4.7 200 OK to UPDATE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
```

```
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 129 UPDATE
Content-Type: application/sdp
Content-Length: 526

v=0
o=- 2987933623 2987933625 IN IP6 5555::eee:fff:aaa:bbb
s=-
c=IN IP6 5555::eee:fff:aaa:bbb
t=0 0
m=video 10001 RTP/AVP 98
b=AS:75
a=curr:qos local sendrecv
a=curr:qos remote sendrecv
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
m=audio 6544 RTP/AVP 97 96
b=AS:25.4
a=curr:qos local sendrecv
a=curr:qos remote sendrecv
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.4.8 180 Ringing

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Record-Route: <sip:pcscf2.visited2.net;lr>, <sip:scscf2.home2.net;lr>,
<sip:scscf1.home1.net;lr>, <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 129 UPDATE
Require: 100rel
Contact: <sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp>
Allow: INVITE, ACK, CANCEL, BYE, PRACK, UPDATE, REFER, MESSAGE
RSeq: 9022
Content-Length: 0
```

14.3.4.9 PRACK

```
PRACK sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>,
<sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
```

```
Call-ID: cb03a0s09a2sdfglkj490333
Require: sec-agree
Proxy-Require: sec-agree
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321;
port-c=8642; port-s=7531
Cseq: 130 PRACK
RAck: 9022 127 INVITE
Content-Length: 0
```

14.3.4.10 200 OK to PRACK

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 130 PRACK
Content-Length: 0
```

14.3.4.11 200 OK to INVITE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Record-Route: <sip:pcscf2.visited2.net;lr>, <sip:scscf2.home2.net;lr>,
<sip:scscf1.home1.net;lr>, <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
CSeq: 127 INVITE
Contact: <sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp>
Content-Type: application/sdp
Content-Length: 584

v=0
o=- 2987933623 2987933623 IN IP6 5555::eee:fff:aaa:bbb
s=-
c=IN IP6 5555::eee:fff:aaa:bbb
t=0 0
m=video 10001 RTP/AVP 98 99
b=AS:75
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=conf:qos remote sendrecv
a=rtpmap:98 H263
a=rtpmap:99 MP4V-ES
a=fmtp:98 profile-level-id=0
m=audio 6544 RTP/AVP 97 96
b=AS:25.4
a=curr:qos local none
a=curr:qos remote none
a=des:qos mandatory local sendrecv
a=des:qos mandatory remote sendrecv
a=conf:qos remote sendrecv
a=rtpmap:97 AMR
```

a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event

14.3.4.12 ACK

ACK sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>, <sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 127 ACK
Content-Length: 0

14.3.4.13 BYE

BYE sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>, <sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Require: sec-agree
Proxy-Require: sec-agree
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321; port-c=8642; port-s=7531
CSeq: 153 BYE
Content-Length: 0

14.3.4.14 200 OK to BYE

SIP/2.0 200 OK
Via: SIP/2.0/UDP pcscf2.visited2.net:5088;branch=z9hG4bK361k21.1, SIP/2.0/UDP scscf2.home2.net;branch=z9hG4bK764z87.1, SIP/2.0/UDP scscf1.home1.net;branch=z9hG4bK332b23.1, SIP/2.0/UDP pcscf1.visited1.net;branch=z9hG4bK240f34.1, SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
CSeq: 153 BYE
Content-Length: 0

14.3.5 3GPP Video Call with RE-INVITE and Unreliable Delivery of Provisional Responses

14.3.5.1 INVITE

INVITE tel:+1-212-555-2222 SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70

```
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>
P-Preferred-Identity: "John Doe" <sip:user1_public1@home1.net>
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
Privacy: none
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 127 INVITE
Require: sec-agree
Proxy-Require: sec-agree
Supported: 100rel
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321;
port-c=8642; port-s=7531
Contact: <sip:[5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp>
Allow: INVITE, ACK, CANCEL, BYE, PRACK, UPDATE, REFER, MESSAGE
Content-Type: application/sdp
Content-Length: 319

v=0
o=- 2987933615 2987933615 IN IP6 5555::aaa:bbb:ccc:ddd
s=-
c=IN IP6 5555::aaa:bbb:ccc:ddd
t=0 0
m=video 3400 RTP/AVP 98 99
b=AS:75
a=inactive
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
a=rtpmap:99 MP4V-ES
m=audio 3456 RTP/AVP 97 96
b=AS:25.4
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.5.2 100 Trying

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 127 INVITE
Content-Length: 0
```

14.3.5.3 180 Ringing

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Record-Route: <sip:pcscf2.visited2.net;lr>, <sip:scscf2.home2.net;lr>,
<sip:scscf1.home1.net;lr>, <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 129 UPDATE
Require: 100rel
```

```
Contact: <sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp>
Allow: INVITE, ACK, CANCEL, BYE, PRACK, UPDATE, REFER, MESSAGE
RSeq: 9022
Content-Type: application/sdp
Content-Length: 360

v=0
o=- 2987933623 2987933623 IN IP6 5555::eee:fff:aaa:bbb
s=-
c=IN IP6 5555::eee:fff:aaa:bbb
t=0 0
m=video 10001 RTP/AVP 98 99
b=AS:75
a=conf:qos remote sendrecv
a=rtpmap:98 H263
a=rtpmap:99 MP4V-ES
a=fmtp:98 profile-level-id=0
m=audio 6544 RTP/AVP 97 96
b=AS:25.4
a=conf:qos remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
a=inactive
```

14.3.5.4 200 OK to INVITE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Record-Route: <sip:pcscf2.visited2.net;lr>, <sip:scscf2.home2.net;lr>,
<sip:scscf1.home1.net;lr>, <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
CSeq: 127 INVITE
Contact: <sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp>
Content-Type: application/sdp
Content-Length: 360

v=0
o=- 2987933623 2987933623 IN IP6 5555::eee:fff:aaa:bbb
s=-
c=IN IP6 5555::eee:fff:aaa:bbb
t=0 0
m=video 10001 RTP/AVP 98 99
b=AS:75
a=conf:qos remote sendrecv
a=rtpmap:98 H263
a=rtpmap:99 MP4V-ES
a=fmtp:98 profile-level-id=0
m=audio 6544 RTP/AVP 97 96
b=AS:25.4
a=conf:qos remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

```
a=inactive
```

14.3.5.5 ACK

```
ACK sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>,
<sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 127 ACK
Content-Length: 0
```

14.3.5.6 INVITE

```
INVITE tel:+1-212-555-2222 SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>
P-Preferred-Identity: "John Doe" <sip:user1_public1@home1.net>
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
Privacy: none
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 128 INVITE
Require: sec-agree
Proxy-Require: sec-agree
Supported: 100rel
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321;
port-c=8642; port-s=7531
Contact: <sip:[5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp>
Allow: INVITE, ACK, CANCEL, BYE, PRACK, UPDATE, REFER, MESSAGE
Content-Type: application/sdp
Content-Length: 280

v=0
o=- 2987933615 2987933616 IN IP6 5555::aaa:bbb:ccc:ddd
s=-
c=IN IP6 5555::aaa:bbb:ccc:ddd
t=0 0
m=video 3400 RTP/AVP 98
b=AS:75
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
m=audio 3456 RTP/AVP 97 96
b=AS:25.4
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
a=sendrecv
```

14.3.5.7 200 OK to INVITE

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
```

```
Record-Route: <sip:pcscf2.visited2.net;lr>, <sip:scscf2.home2.net;lr>,
<sip:scscf1.home1.net;lr>, <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
CSeq: 128 INVITE
Contact: <sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp>
Content-Type: application/sdp
Content-Length: 338

v=0
o=- 2987933623 2987933624 IN IP6 5555::eee:fff:aaa:bbb
s=-
c=IN IP6 5555::eee:fff:aaa:bbb
t=0 0
m=video 10001 RTP/AVP 98
b=AS:75
a=conf:qos remote sendrecv
a=rtpmap:98 H263
a=fmtp:98 profile-level-id=0
m=audio 6544 RTP/AVP 97 96
b=AS:25.4
a=conf:qos remote sendrecv
a=rtpmap:97 AMR
a=fmtp:97 mode-set=0,2,5,7; maxframes=2
a=rtpmap:96 telephone-event
```

14.3.5.8 ACK

```
ACK sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>,
<sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 128 ACK
Content-Length: 0
```

14.3.6 3GPP Video Call with RE-INVITE and Reliable Delivery of Provisional Responses

14.3.6.1 INVITE

Same as the INVITE message presented in Section 14.3.5.1.

14.3.6.2 100 Trying

Same as the 100 Trying message presented in Section 14.3.5.2.

14.3.6.3 180 Ringing

Same as the 180 Ringing message presented in Section 14.3.5.3.

14.3.6.4 PRACK to 180 Ringing

```
PRACK sip:[5555::eee:fff:aaa:bbb]:8805;comp=sigcomp SIP/2.0
```

```
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
Route: <sip:pcscf1.visited1.net:7531;lr;comp=sigcomp>, <sip:scscf1.home1.net;lr>,
<sip:scscf2.home2.net;lr>, <sip:pcscf2.visited2.net;lr>
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Require: sec-agree
Proxy-Require: sec-agree
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321;
port-c=8642; port-s=7531
Cseq: 128 PRACK
RAck: 9022 127 INVITE
Content-Length: 0
```

14.3.6.5 200 OK to PRACK

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
From: <sip:user1_public1@home1.net>;tag=171828
To: <tel:+1-212-555-2222>;tag=314159
Call-ID: cb03a0s09a2sdfglkj490333
Cseq: 128 PRACK
Content-Length: 0
```

14.3.6.6 200 OK to INVITE

Same as the 200 OK message presented in Section 14.3.5.4.

14.3.6.7 ACK

Same as the ACK message presented in Section 14.3.5.5.

14.3.6.8 INVITE

Same as the INVITE message presented in Section 14.3.5.6.

14.3.6.9 200 OK to INVITE

Same as the 200 OK message presented in Section 14.3.5.7.

14.3.6.10 ACK

Same as the ACK message presented in Section 14.3.5.8.

14.3.7 3GPP Registration Sequence

14.3.7.1 REGISTER

```
REGISTER sip:registrar.home1.net SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd];comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
From: <sip:user1_public1@home1.net>;tag=4fa3
To: <sip:user1_public1@home1.net>
Contact: <sip:[5555::aaa:bbb:ccc:ddd];comp=sigcomp>;expires=600000
Call-ID: apb03a0s09dkjdfglkj49111
```

Authorization: Digest username="user1_private@home1.net", realm="registrar.home1.net", nonce="", uri="sip:registrar.home1.net", response=""
Security-Client: ipsec-3gpp; alg=hmac-sha-1-96; spi-c=23456789; spi-s=12345678; port-c=2468; port-s=1357
Require: sec-agree
Proxy-Require: sec-agree
CSeq: 1 REGISTER
Supported: path
Content-Length: 0

14.3.7.2 401 Unauthorized

SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd];comp=sigcomp;branch=z9hG4bKnashds7
From: <sip:user1_public1@home1.net>;tag=4fa3
To: <sip:user1_public1@home1.net>
Call-ID: apb03a0s09dkjdfglkj49111
WWW-Authenticate: Digest realm="registrar.home1.net", nonce=base64(RAND + AUTN + server specific data), algorithm=AKAv1-MD5
Security-Server: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321; port-c=8642; port-s=7531
CSeq: 1 REGISTER
Content-Length: 0

14.3.7.3 REGISTER

REGISTER sip:registrar.home1.net SIP/2.0
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Max-Forwards: 70
P-Access-Network-Info: 3GPP-UTRAN-TDD; utran-cell-id-3gpp=234151D0FCE11
From: <sip:user1_public1@home1.net>;tag=4fa3
To: <sip:user1_public1@home1.net>
Contact: <sip:[5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp>;expires=600000
Call-ID: apb03a0s09dkjdfglkj49111
Authorization: Digest username="user1_private@home1.net", realm="registrar.home1.net", nonce=base64(RAND + AUTN + server specific data), algorithm=AKAv1-MD5, uri="sip:registrar.home1.net", response="6629fae49393a05397450978507c4ef1"
Security-Client: ipsec-3gpp; alg=hmac-sha-1-96; spi-c=23456789; spi-s=12345678; port-c=2468; port-s=1357
Security-Verify: ipsec-3gpp; q=0.1; alg=hmac-sha-1-96; spi-c=98765432; spi-s=87654321; port-c=8642; port-s=7531
Require: sec-agree
Proxy-Require: sec-agree
CSeq: 2 REGISTER
Supported: path
Content-Length: 0

14.3.7.4 200 OK

SIP/2.0 200 OK
Via: SIP/2.0/UDP [5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp;branch=z9hG4bKnashds7
Path: <sip:term@pcscf1.visited1.net;lr>
Service-Route: <sip:orig@scscf1.home1.net;lr>
From: <sip:user1_public1@home1.net>;tag=4fa3
To: <sip:user1_public1@home1.net>
Call-ID: apb03a0s09dkjdfglkj49111
Contact: <sip:[5555::aaa:bbb:ccc:ddd]:1357;comp=sigcomp>;expires=600000

CSeq: 2 REGISTER
Date: Wed, 11 July 2001 08:49:37 GMT
P-Associated-URI: <sip:user1_public2@home1.net>, <sip:user1_public3@home1.net>, <sip:+1-212-555-1111@home1.net;user=phone>
Content-Length: 0

14.4 Appendix D – Measurement Results: Linear Search versus Hashing

14.4.1 Linear Search

Message	Length uncompressed	Length compressed	Length of SigComp message	Compression ratio (compr/uncompr)
INVITE	1437	750	983	0,522
100 Trying	254	21	266	0,083
183 Session Progress	1440	520	765	0,361
PRACK (1)	1318	110	151	0,083
200 OK (1)	904	44	85	0,049
UPDATE	1291	51	99	0,040
200 OK (2)	865	47	95	0,054
180 Ringing	563	29	77	0,052
PRACK (2)	717	34	96	0,047
200 OK (3)	260	14	69	0,054
200 OK (4)	1133	23	78	0,020
ACK	458	15	91	0,033
TOTAL	10640	1658	2855	0,156

Message	User CPU time [ms], compression	User CPU time [ms], decompression	Wall-clock time [us], compression	Wall-clock time [us], decompression
INVITE	3,86	2,00	4456,29	2229,86
100 Trying	0,14	0,29	315,86	658,71
183 Session Progress	3,00	1,86	3209,86	1884,14
PRACK (1)	0,71	0,57	1025,14	902,71
200 OK (1)	0,43	0,57	634,29	676,29
UPDATE	8,00	0,57	8085,00	777,43
200 OK (2)	6,00	0,43	6153,14	815,43
180 Ringing	6,14	0,43	6239,71	684,71
PRACK (2)	0,71	0,57	608,57	659,86
200 OK (3)	0,29	0,43	365,00	559,43
200 OK (4)	0,71	0,43	646,71	783,43
ACK	0,00	0,43	380,14	595,14
TOTAL	30,00	8,57	32119,71	11227,14

14.4.2 Hashing

Message	Length uncompressed	Length compressed	Length of SigComp message	Compression ratio
INVITE	1437	739	972	0,514
100 Trying	254	20	265	0,079
183 Session Progress	1440	516	761	0,358
PRACK (1)	1318	109	150	0,083
200 OK (1)	904	43	84	0,048
UPDATE	1291	54	102	0,042
200 OK (2)	865	50	98	0,058
180 Ringing	563	30	78	0,053
PRACK (2)	717	34	96	0,047
200 OK (3)	260	16	71	0,062
200 OK (4)	1133	23	78	0,020
ACK	458	15	91	0,033
TOTAL	10640	1649	2846	0,155

Message	User CPU time [ms], compression	User CPU time [ms], decompression	Wall-clock time [us], compression	Wall-clock time [us], decompression
INVITE	1,29	2,14	1550,00	2234,29
100 Trying	0,86	0,43	1204,43	644,29
183 Session Progress	0,43	1,57	772,71	2013,14
PRACK (1)	0,43	0,86	699,71	948,29
200 OK (1)	0,43	0,29	569,29	725,14
UPDATE	1,00	0,43	1005,14	749,57
200 OK (2)	0,86	0,71	761,71	693,86
180 Ringing	1,00	0,43	1036,29	657,57
PRACK (2)	1,00	0,29	1103,00	708,43
200 OK (3)	0,71	0,43	748,71	672,57
200 OK (4)	1,00	0,43	1144,71	642,43
ACK	0,86	0,57	875,14	575,71
TOTAL	9,86	8,57	11470,86	11265,29

14.4.3 Time Requirement of Hash Map Updates

Message	Time, hash map update [us]	Time, compression [us]	Percentage of time spent updating the hash map
INVITE	1007	690	59,34 %
100 Trying	1096	254	81,19 %
183 Session Progress	272	677	28,66 %
PRACK (1)	303	573	34,59 %
200 OK (1)	302	355	45,97 %
UPDATE	669	525	56,03 %
200 OK (2)	515	386	57,16 %
180 Ringing	874	332	72,47 %
PRACK (2)	883	321	73,34 %
200 OK (3)	657	246	72,76 %
200 OK (4)	829	442	65,22 %
ACK	745	256	74,43 %
AVG	679,33	421,42	60,10 %

14.5 Appendix E – Measurement Results: Length of Look-ahead Buffer

14.5.1 Buffer Length 18 Bytes, 4 Bits Used to Encode Length Values

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Compression ratio (compr/uncompr)
INVITE	1437	694	927	0,483
100 Trying	254	42	287	0,165
183 Session Progress	1440	515	760	0,358
PRACK (1)	1318	227	268	0,172
200 OK (1)	904	137	178	0,152
UPDATE	1291	188	236	0,146
200 OK (2)	865	127	175	0,147
180 Ringing	563	87	135	0,155
PRACK (2)	717	107	169	0,149
200 OK (3)	260	40	95	0,154
200 OK (4)	1133	155	210	0,137
ACK	458	65	141	0,142
TOTAL	10640	2384	3581	0,224

Message	User CPU time [ms], compression	User CPU time [ms], decompression	Wall-clock time [us], compression	Wall-clock time [us], decompression
INVITE	1,43	2,14	1543,00	2282,86
100 Trying	1,00	0,00	1192,71	692,71
183 Session Progress	1,00	2,00	797,29	2064,86
PRACK (1)	0,86	1,14	761,57	1386,43
200 OK (1)	0,43	0,86	618,00	1019,29
UPDATE	0,86	1,00	1088,00	1310,29
200 OK (2)	0,86	0,43	820,14	959,71
180 Ringing	1,00	1,00	1056,29	885,43
PRACK (2)	1,00	0,86	1144,00	1122,57
200 OK (3)	0,29	0,29	758,71	701,43
200 OK (4)	1,57	1,00	1264,14	1235,29
ACK	0,71	0,57	928,14	811,14
TOTAL	11,00	11,29	11972,00	14472,00

14.5.2 Buffer Length 66 Bytes, 6 Bits Used to Encode Length Values

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Compression ratio (compr/uncompr)
INVITE	1437	706	939	0,491
100 Trying	254	24	269	0,094
183 Session Progress	1440	497	742	0,345
PRACK (1)	1318	125	166	0,095
200 OK (1)	904	61	102	0,067
UPDATE	1291	78	126	0,060
200 OK (2)	865	62	110	0,072
180 Ringing	563	44	92	0,078
PRACK (2)	717	50	112	0,070
200 OK (3)	260	20	75	0,077
200 OK (4)	1133	50	105	0,044
ACK	458	24	100	0,052
TOTAL	10640	1741	2938	0,164

Message	User CPU time [ms], compression	User CPU time [ms], decompression	Wall-clock time [us], compression	Wall-clock time [us], decompression
INVITE	1,43	2,00	1538,29	2183,14
100 Trying	1,00	0,29	1181,86	652,14
183 Session Progress	0,71	1,57	781,29	1984,29
PRACK (1)	0,71	0,71	711,86	1019,71
200 OK (1)	0,57	0,57	587,86	775,00
UPDATE	0,86	0,57	1028,43	838,86
200 OK (2)	0,57	0,29	773,29	723,14
180 Ringing	1,00	0,57	1039,14	729,29
PRACK (2)	1,00	0,71	1111,57	785,43
200 OK (3)	0,57	0,43	752,71	688,29
200 OK (4)	1,29	0,71	1184,71	756,14
ACK	0,71	0,29	891,43	614,57
TOTAL	10,43	8,71	11582,43	11750,00

14.5.3 Buffer Length 258 Bytes, 8 Bits Used to Encode Length Values

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Compression ratio (compr/uncompr)
INVITE	1437	739	972	0,514
100 Trying	254	20	265	0,079
183 Session Progress	1440	516	761	0,358
PRACK (1)	1318	109	150	0,083
200 OK (1)	904	43	84	0,048
UPDATE	1291	54	102	0,042
200 OK (2)	865	50	98	0,058
180 Ringing	563	30	78	0,053
PRACK (2)	717	34	96	0,047
200 OK (3)	260	16	71	0,062
200 OK (4)	1133	23	78	0,020
ACK	458	15	91	0,033
TOTAL	10640	1649	2846	0,155

Message	User CPU time [ms], compression	User CPU time [ms], decompression	Wall-clock time [us], compression	Wall-clock time [us], decompression
INVITE	1,29	2,14	1550,00	2234,29
100 Trying	0,86	0,43	1204,43	644,29
183 Session Progress	0,43	1,57	772,71	2013,14
PRACK (1)	0,43	0,86	699,71	948,29
200 OK (1)	0,43	0,29	569,29	725,14
UPDATE	1,00	0,43	1005,14	749,57
200 OK (2)	0,86	0,71	761,71	693,86
180 Ringing	1,00	0,43	1036,29	657,57
PRACK (2)	1,00	0,29	1103,00	708,43
200 OK (3)	0,71	0,43	748,71	672,57
200 OK (4)	1,00	0,43	1144,71	642,43
ACK	0,86	0,57	875,14	575,71
TOTAL	9,86	8,57	11470,86	11265,29

14.6 Appendix F – Measurement Results: Length of Shared States

14.6.1 Shared State Length 500 Bytes

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Compression ratio (compr/uncompr)	Avg time [us], compression	Avg time [us], decompression
INVITE	1437	739	972	0,514	1550,00	2234,29
100 Trying	254	20	265	0,079	1204,43	644,29
183 Progress	1440	516	761	0,358	772,71	2013,14
PRACK (1)	1318	109	150	0,083	699,71	948,29
200 OK (1)	904	43	84	0,048	569,29	725,14
UPDATE	1291	54	102	0,042	1005,14	749,57
200 OK (2)	865	50	98	0,058	761,71	693,86
180 Ringing	563	30	78	0,053	1036,29	657,57
PRACK (2)	717	34	96	0,047	1103,00	708,43
200 OK (3)	260	16	71	0,062	748,71	672,57
200 OK (4)	1133	23	78	0,020	1144,71	642,43
ACK	458	15	91	0,033	875,14	575,71
TOTAL	10640	1649	2846	0,155	11470,86	11265,29

14.6.2 Shared State Length 750 Bytes

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Compression ratio (compr/uncompr)	Avg time [us], compression	Avg time [us], decompression
INVITE	1437	739	973	0,514	1592,57	2171,71
100 Trying	254	21	267	0,083	1321,14	651,00
183 Progress	1440	507	753	0,352	834,71	1993,29
PRACK (1)	1318	74	115	0,056	836,57	802,86
200 OK (1)	904	40	81	0,044	701,57	703,71
UPDATE	1291	54	102	0,042	1272,14	763,43
200 OK (2)	865	40	88	0,046	1164,00	637,00
180 Ringing	563	26	74	0,046	1061,29	631,14
PRACK (2)	717	27	89	0,038	1294,14	662,86
200 OK (3)	260	9	64	0,035	989,14	516,43
200 OK (4)	1133	23	78	0,020	1152,86	651,86
ACK	458	15	91	0,033	1175,57	574,71
TOTAL	10640	1575	2775	0,148	13395,71	10760,00

14.6.3 Shared State Length 1000 Bytes

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Compression ratio (compr/uncompr)	Avg time [us], compression	Avg time [us], decompression
INVITE	1437	739	972	0,514	1545,00	2245,29
100 Trying	254	21	266	0,083	1327,86	654,71
183 Progress	1440	436	681	0,303	758,71	1778,14
PRACK (1)	1318	73	114	0,055	830,86	872,14
200 OK (1)	904	43	84	0,048	724,29	669,29
UPDATE	1291	52	100	0,040	1597,71	772,00
200 OK (2)	865	38	86	0,044	1532,71	660,71
180 Ringing	563	23	71	0,041	1096,71	643,00
PRACK (2)	717	29	91	0,040	1482,00	662,57
200 OK (3)	260	9	64	0,035	1094,71	526,71
200 OK (4)	1133	23	78	0,020	1215,29	660,71
ACK	458	15	91	0,033	1472,57	590,57
TOTAL	10640	1501	2698	0,141	14678,43	10735,86

14.6.4 Shared State Length 1500 Bytes

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Compression ratio (compr/uncompr)	Avg time [us], compression	Avg time [us], decompression
INVITE	1437	739	972	0,514	1564,57	2243,29
100 Trying	254	21	266	0,083	1456,14	667,71
183 Progress	1440	295	540	0,205	731,86	1538,14
PRACK (1)	1318	70	111	0,053	949,43	797,71
200 OK (1)	904	40	81	0,044	1267,86	762,57
UPDATE	1291	52	100	0,040	2140,00	806,29
200 OK (2)	865	38	86	0,044	1907,00	692,00
180 Ringing	563	29	77	0,052	1169,57	656,29
PRACK (2)	717	27	89	0,038	1940,71	668,43
200 OK (3)	260	9	64	0,035	1686,00	533,29
200 OK (4)	1133	26	81	0,023	1199,43	670,86
ACK	458	15	91	0,033	1840,29	582,29
TOTAL	10640	1361	2558	0,128	17852,86	10618,86

14.7 Appendix G – Measurement Results: Secure Hash Algorithm

Input	Length [bytes]	SHA-1 hash of the input, avg time [us]	SHA-1 hash of the input, standard deviation [us]
200 OK (3)	260	99,70	5,03
ACK	458	120,90	30,46
PRACK (2)	717	130,50	30,15
200 OK (1)	904	136,10	19,72
200 OK (4)	1133	161,50	25,33
183 Session Progress	1440	167,60	21,37
UDVM memory snapshot 4096	4096	283,42	6,30
UDVM memory snapshot 8192	8192	408,25	26,59
UDVM memory snapshot 16384	16384	559,17	36,81

14.8 Appendix H – Measurement Results: SigComp Mechanisms

14.8.1 Basic Compression

14.8.1.1 DMS 4096 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Length of SigComp message, bytecode saved	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/uncompr)	Compr ratio (SigComp/uncompr), bytecode saved
INVITE	1437	1085	1164	1164	0	0,810	0,810
100 Trying	254	276	355	355	0	1,398	1,398
183 Progress	1440	1027	1106	1106	0	0,768	0,768
PRACK (1)	1318	996	1075	1006	0	0,816	0,763
200 OK (1)	904	641	720	651	0	0,796	0,720
UPDATE	1291	956	1035	966	0	0,802	0,748
200 OK (2)	865	625	704	635	0	0,814	0,734
180 Ringing	563	490	569	500	0	1,011	0,888
PRACK (2)	717	645	724	655	0	1,010	0,914
200 OK (3)	260	279	358	289	0	1,377	1,112
200 OK (4)	1133	759	838	769	0	0,740	0,679
ACK	458	397	476	407	0	1,039	0,889
TOTAL	10640	8176	9124	8503		0,858	0,799

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	1541	38,65	2799	71,16
100 Trying	1272	31,05	1167	27,60
183 Progress	1020	27,07	2585	50,71
PRACK (1)	1272	32,48	2579	44,59
200 OK (1)	1151	27,13	1873	22,70
UPDATE	1257	42,21	2502	79,26
200 OK (2)	977	25,57	1845	25,61
180 Ringing	892	44,26	1628	60,31
PRACK (2)	1060	24,40	1919	56,65
200 OK (3)	681	33,21	1231	45,58
200 OK (4)	880	70,55	2114	101,55
ACK	810	36,03	1410	38,46
TOTAL	12814		23650	

14.8.1.2 DMS 8192 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Length of SigComp message, bytecode saved	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)	Compr ratio (SigComp/ uncompr), bytecode saved
INVITE	1437	1085	1164	1164	0	0,810	0,810
100 Trying	254	276	355	355	0	1,398	1,398
183 Session Prog.	1440	1027	1106	1106	0	0,768	0,768
PRACK (1)	1318	996	1075	1006	0	0,816	0,763
200 OK (1)	904	641	720	651	0	0,796	0,720
UPDATE	1291	956	1035	966	0	0,802	0,748
200 OK (2)	865	625	704	635	0	0,814	0,734
180 Ringing	563	490	569	500	0	1,011	0,888
PRACK (2)	717	645	724	655	0	1,010	0,914
200 OK (3)	260	279	358	289	0	1,377	1,112
200 OK (4)	1133	759	838	769	0	0,740	0,679
ACK	458	397	476	407	0	1,039	0,889
TOTAL	10640	8176	9124	8503		0,858	0,799

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	1658	30,69	2744	67,57
100 Trying	1336	20,59	1193	26,87
183 Session Prog.	1069	36,44	2574	48,49
PRACK (1)	1402	36,42	2500	20,36
200 OK (1)	1304	29,01	1868	36,06
UPDATE	1406	33,73	2440	32,44
200 OK (2)	1113	35,29	1849	42,56
180 Ringing	1039	27,66	1558	32,67
PRACK (2)	1210	44,19	1878	73,89
200 OK (3)	762	26,76	1178	33,33
200 OK (4)	925	30,41	2088	26,80
ACK	919	44,25	1377	22,59
TOTAL	14145		23249	

14.8.1.3 DMS 16384 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Length of SigComp message, bytecode saved	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)	Compr ratio (SigComp/ uncompr), bytecode saved
INVITE	1437	1085	1165	1165	0	0,811	0,811
100 Trying	254	276	356	356	0	1,402	1,402
183 Session Prog.	1440	1027	1107	1107	0	0,769	0,769
PRACK (1)	1318	996	1076	1006	0	0,816	0,763
200 OK (1)	904	641	721	651	0	0,798	0,720
UPDATE	1291	956	1036	966	0	0,802	0,748
200 OK (2)	865	625	705	635	0	0,815	0,734
180 Ringing	563	490	570	500	0	1,012	0,888
PRACK (2)	717	645	725	655	0	1,011	0,914
200 OK (3)	260	279	359	289	0	1,381	1,112
200 OK (4)	1133	759	839	769	0	0,741	0,679
ACK	458	397	477	407	0	1,041	0,889
TOTAL	10640	8176	9136	8506		0,859	0,799

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	1719	26,36	2768	33,69
100 Trying	1350	42,37	1235	14,48
183 Session Prog.	1175	40,63	2546	52,17
PRACK (1)	1527	37,95	2530	74,34
200 OK (1)	1412	32,44	1898	28,60
UPDATE	1544	42,70	2445	28,60
200 OK (2)	1205	22,07	1877	32,86
180 Ringing	1152	20,47	1594	23,35
PRACK (2)	1301	41,84	1916	44,33
200 OK (3)	843	41,86	1228	38,43
200 OK (4)	1025	46,34	2132	92,16
ACK	1049	27,61	1444	51,38
TOTAL	15302		23613	

14.8.2 Static Compression

14.8.2.1 DMS 4096 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Length of SigComp message, bytecode saved	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)	Compr ratio (SigComp/ uncompr), bytecode saved
INVITE	1437	791	887	887	0	0,617	0,617
100 Trying	254	174	270	270	0	1,063	1,063
183 Session Prog.	1440	756	852	852	0	0,592	0,592
PRACK (1)	1318	739	835	749	0	0,634	0,568
200 OK (1)	904	448	544	458	0	0,602	0,507
UPDATE	1291	718	814	728	0	0,631	0,564
200 OK (2)	865	436	532	446	0	0,615	0,516
180 Ringing	563	339	435	349	0	0,773	0,620
PRACK (2)	717	491	587	501	0	0,819	0,699
200 OK (3)	260	183	279	193	0	1,073	0,742
200 OK (4)	1133	546	642	556	0	0,567	0,491
ACK	458	287	383	297	0	0,836	0,648
TOTAL	10640	5908	7060	6286		0,664	0,591

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	1497	25,96	2455	64,91
100 Trying	937	31,18	1087	13,98
183 Session Prog.	987	22,40	2252	57,79
PRACK (1)	1337	23,44	2215	63,09
200 OK (1)	1183	29,98	1607	50,51
UPDATE	1272	38,28	2156	34,69
200 OK (2)	1002	44,21	1586	32,73
180 Ringing	880	31,02	1412	47,69
PRACK (2)	1088	30,13	1675	44,99
200 OK (3)	669	31,15	1089	42,05
200 OK (4)	891	46,13	1780	50,85
ACK	793	31,94	1271	29,80
TOTAL	12536		20586	

14.8.2.2 DMS 8192 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Length of SigComp message, bytecode saved	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)	Compr ratio (SigComp/ uncompr), bytecode saved
INVITE	1437	791	887	887	0	0,617	0,617
100 Trying	254	174	270	270	0	1,063	1,063
183 Session Prog.	1440	756	852	852	0	0,592	0,592
PRACK (1)	1318	739	835	749	0	0,634	0,568
200 OK (1)	904	448	544	458	0	0,602	0,507
UPDATE	1291	718	814	728	0	0,631	0,564
200 OK (2)	865	436	532	446	0	0,615	0,516
180 Ringing	563	339	435	349	0	0,773	0,620
PRACK (2)	717	491	587	501	0	0,819	0,699
200 OK (3)	260	183	279	193	0	1,073	0,742
200 OK (4)	1133	546	642	556	0	0,567	0,491
ACK	458	287	383	297	0	0,836	0,648
TOTAL	10640	5908	7060	6286		0,664	0,591

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	1574	35,40	2399	31,18
100 Trying	1030	45,49	1090	33,37
183 Session Prog.	1048	34,08	2295	74,28
PRACK (1)	1377	33,15	2259	71,28
200 OK (1)	1231	39,21	1647	36,81
UPDATE	1315	30,12	2236	101,25
200 OK (2)	1019	22,49	1613	25,35
180 Ringing	915	26,73	1390	28,55
PRACK (2)	1114	22,76	1708	51,99
200 OK (3)	709	29,61	1108	30,88
200 OK (4)	900	36,42	1837	53,08
ACK	830	34,87	1298	55,37
TOTAL	13060		20879	

14.8.2.3 DMS 16384 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Length of SigComp message, bytecode saved	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)	Compr ratio (SigComp/ uncompr), bytecode saved
INVITE	1437	791	888	888	0	0,618	0,618
100 Trying	254	174	271	271	0	1,067	1,067
183 Session Prog.	1440	756	853	853	0	0,592	0,592
PRACK (1)	1318	739	836	749	0	0,634	0,568
200 OK (1)	904	448	545	458	0	0,603	0,507
UPDATE	1291	718	815	728	0	0,631	0,564
200 OK (2)	865	436	533	446	0	0,616	0,516
180 Ringing	563	339	436	349	0	0,774	0,620
PRACK (2)	717	491	588	501	0	0,820	0,699
200 OK (3)	260	183	280	193	0	1,077	0,742
200 OK (4)	1133	546	643	556	0	0,568	0,491
ACK	458	287	384	297	0	0,838	0,648
TOTAL	10640	5908	7072	6289		0,665	0,591

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	1685	24,61	2531	55,03
100 Trying	1129	45,98	1160	32,84
183 Session Prog.	1067	43,84	2293	28,00
PRACK (1)	1527	41,67	2311	84,17
200 OK (1)	1297	33,68	1696	26,16
UPDATE	1379	43,11	2256	56,43
200 OK (2)	1103	30,52	1702	44,93
180 Ringing	1009	41,15	1417	31,48
PRACK (2)	1219	40,68	1711	17,41
200 OK (3)	746	22,39	1135	17,09
200 OK (4)	920	15,78	1845	45,80
ACK	956	32,54	1294	16,29
TOTAL	14039		21352	

14.8.3 Dynamic Compression

14.8.3.1 DMS 4096 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)
INVITE	1437	791	1018	4032	0,708
100 Trying	254	174	407	4032	1,602
183 Session Prog.	1440	756	989	8064	0,687
PRACK (1)	1318	126	154	8064	0,117
200 OK (1)	904	43	71	12096	0,079
UPDATE	1291	52	87	12096	0,067
200 OK (2)	865	44	79	16128	0,091
180 Ringing	563	34	69	20160	0,123
PRACK (2)	717	30	79	16128	0,110
200 OK (3)	260	14	56	24192	0,215
200 OK (4)	1133	23	65	28224	0,057
ACK	458	18	81	20160	0,177
TOTAL	10640	2105	3155		0,297

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	2456	38,66	2774	136,30
100 Trying	1063	23,06	1288	34,95
183 Session Prog.	1953	31,61	2695	135,42
PRACK (1)	1911	38,09	1294	26,77
200 OK (1)	1203	42,88	1112	32,05
UPDATE	2040	30,59	1118	21,30
200 OK (2)	1419	33,75	1109	32,63
180 Ringing	1413	28,49	1063	55,69
PRACK (2)	1282	30,86	1099	48,42
200 OK (3)	792	40,92	972	27,67
200 OK (4)	1884	30,29	1080	36,07
ACK	1007	30,34	1016	30,47
TOTAL	18423		16620	

14.8.3.2 DMS 8192 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)
INVITE	1437	791	1018	8128	0,708
100 Trying	254	174	407	8128	1,602
183 Session Prog.	1440	756	989	16256	0,687
PRACK (1)	1318	123	151	16256	0,115
200 OK (1)	904	43	71	24384	0,079
UPDATE	1291	52	87	24384	0,067
200 OK (2)	865	44	79	32512	0,091
180 Ringing	563	27	62	40640	0,110
PRACK (2)	717	31	80	32512	0,112
200 OK (3)	260	16	58	48768	0,223
200 OK (4)	1133	23	65	56896	0,057
ACK	458	15	78	40640	0,170
TOTAL	10640	2095	3145		0,296

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	2687	29,07	2877	64,41
100 Trying	1330	34,51	1424	32,42
183 Session Prog.	2187	31,03	2620	80,67
PRACK (1)	1885	33,62	1494	23,87
200 OK (1)	1411	38,50	1294	43,06
UPDATE	1735	32,99	1371	45,24
200 OK (2)	1262	28,09	1386	34,20
180 Ringing	1383	20,97	1248	22,53
PRACK (2)	1256	29,80	1294	23,27
200 OK (3)	819	35,67	1246	27,10
200 OK (4)	1632	25,39	1364	39,69
ACK	947	26,22	1241	46,99
TOTAL	18534		18859	

14.8.3.3 DMS 16384 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)
INVITE	1437	791	1022	16320	0,711
100 Trying	254	174	411	16320	1,618
183 Session Prog.	1440	756	993	32640	0,690
PRACK (1)	1318	123	151	32640	0,115
200 OK (1)	904	43	71	48960	0,079
UPDATE	1291	52	87	48960	0,067
200 OK (2)	865	44	79	65280	0,091
180 Ringing	563	27	62	81600	0,110
PRACK (2)	717	31	80	65280	0,112
200 OK (3)	260	16	58	97920	0,223
200 OK (4)	1133	23	65	114240	0,057
ACK	458	15	78	81600	0,170
TOTAL	10640	2095	3157		0,297

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	2946	34,75	3084	45,18
100 Trying	1631	44,93	1824	37,21
183 Session Prog.	2338	39,99	2954	47,37
PRACK (1)	2040	19,26	1926	27,95
200 OK (1)	1573	19,93	1714	33,02
UPDATE	2030	24,67	1766	15,61
200 OK (2)	1550	25,48	1730	36,01
180 Ringing	1674	34,71	1733	48,57
PRACK (2)	1412	30,97	1802	37,29
200 OK (3)	1103	37,71	1666	25,64
200 OK (4)	1936	28,39	1697	27,26
ACK	1229	36,03	1645	26,88
TOTAL	21461		23541	

14.8.4 Shared Compression

14.8.4.1 DMS 4096 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)
INVITE	1437	791	1025	5469	0,713
100 Trying	254	20	266	5723	1,047
183 Session Prog.	1440	546	792	11195	0,550
PRACK (1)	1318	111	152	12513	0,115
200 OK (1)	904	43	84	17449	0,093
UPDATE	1291	59	107	18740	0,083
200 OK (2)	865	44	92	23637	0,106
180 Ringing	563	62	110	28232	0,195
PRACK (2)	717	41	103	24917	0,144
200 OK (3)	260	14	69	33241	0,265
200 OK (4)	1133	56	111	38406	0,098
ACK	458	18	94	30800	0,205
TOTAL	10640	1805	3005		0,282

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	2600	65,38	2910	143,78
100 Trying	1373	36,35	1176	40,60
183 Session Prog.	2182	45,66	2421	120,22
PRACK (1)	2349	37,65	1481	37,28
200 OK (1)	1866	43,80	1309	42,30
UPDATE	2609	49,60	1332	27,78
200 OK (2)	1984	35,40	1316	31,67
180 Ringing	1597	23,93	1295	30,28
PRACK (2)	1790	35,48	1283	25,64
200 OK (3)	1269	45,88	1148	35,93
200 OK (4)	2199	26,14	1384	43,86
ACK	1505	36,12	1192	27,50
TOTAL	23324		18247	

14.8.4.2 DMS 8192 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)
INVITE	1437	791	1025	9565	0,713
100 Trying	254	20	266	9819	1,047
183 Session Prog.	1440	546	792	19387	0,550
PRACK (1)	1318	109	150	20705	0,114
200 OK (1)	904	43	84	29737	0,093
UPDATE	1291	52	100	31028	0,077
200 OK (2)	865	44	92	40021	0,106
180 Ringing	563	27	75	48712	0,133
PRACK (2)	717	31	93	41301	0,130
200 OK (3)	260	16	71	57817	0,273
200 OK (4)	1133	23	78	67078	0,069
ACK	458	15	91	51280	0,199
TOTAL	10640	1717	2917		0,274

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	2822	35,25	2941	57,28
100 Trying	1581	39,00	1332	48,31
183 Session Prog.	2301	35,14	2606	47,64
PRACK (1)	2125	41,49	1761	26,55
200 OK (1)	1609	35,23	1549	38,00
UPDATE	2088	34,30	1564	25,34
200 OK (2)	1681	36,35	1591	20,20
180 Ringing	1625	30,42	1443	15,80
PRACK (2)	1525	36,42	1540	23,40
200 OK (3)	1229	54,64	1401	51,39
200 OK (4)	1874	27,00	1520	63,13
ACK	1679	24,63	1396	27,45
TOTAL	22139		20644	

14.8.4.3 DMS 16384 Bytes

Message	Length uncompr	Length compr	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compr ratio (SigComp/ uncompr)
INVITE	1437	791	1029	17757	0,716
100 Trying	254	20	270	18011	1,063
183 Session Prog.	1440	546	796	35771	0,553
PRACK (1)	1318	109	150	37089	0,114
200 OK (1)	904	43	84	54313	0,093
UPDATE	1291	52	100	55604	0,077
200 OK (2)	865	44	92	72789	0,106
180 Ringing	563	27	75	89672	0,133
PRACK (2)	717	31	93	74069	0,130
200 OK (3)	260	16	71	106969	0,273
200 OK (4)	1133	23	78	124422	0,069
ACK	458	15	91	92240	0,199
TOTAL	10640	1717	2929		0,275

Message	Avg time [us], compression and memory image	Standard deviation, compr time [us]	Avg time [us], decompr	Standard deviation, decompr time [us]
INVITE	3086	37,71	3217	34,28
100 Trying	1976	306,93	1724	251,34
183 Session Prog.	2561	30,40	2861	23,44
PRACK (1)	2381	48,02	2123	61,47
200 OK (1)	1896	54,87	1931	38,17
UPDATE	2356	42,38	2016	32,74
200 OK (2)	1873	42,41	1958	37,12
180 Ringing	1909	40,83	1920	46,50
PRACK (2)	1748	54,78	1999	63,98
200 OK (3)	1389	34,52	1833	24,77
200 OK (4)	2174	36,71	1896	34,63
ACK	1522	41,98	1887	49,12
TOTAL	24872		25365	

14.9 Appendix I – Measurement Results: Decompression Memory Size

For the results of the measurements in which the 3GPP video call session establishment sequence was used, see Appendix H.

14.9.1 Dynamic Compression

14.9.1.1 Basic Voice Call, DMS 2048 Bytes

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	248	475	1984	0,924
100 Trying	311	170	403	1984	1,296
180 Ringing	615	352	585	3968	0,951
200 OK (1)	610	352	585	5952	0,959
ACK	378	46	81	3968	0,214
TOTAL	2428	1168	2129		0,877

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1330	42,87	1429	33,80
100 Trying	1003	36,82	1197	20,30
180 Ringing	1080	25,08	1535	30,04
200 OK (1)	1107	44,67	1585	51,88
ACK	747	16,83	925	14,88
TOTAL	5267		6670	

14.9.1.2 Basic Voice Call, DMS 4096 Bytes

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	245	472	4032	0,918
100 Trying	311	170	403	4032	1,296
180 Ringing	615	327	560	8064	0,911
200 OK (1)	610	328	561	12096	0,920
ACK	378	43	78	8064	0,206
TOTAL	2428	1113	2074		0,854

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1494	42,47	1514	31,37
100 Trying	1196	203,08	1264	33,97
180 Ringing	1027	30,09	1592	59,53
200 OK (1)	1086	33,24	1646	121,34
ACK	710	28,92	1030	37,25
TOTAL	5513		7046	

14.9.1.3 Basic Voice Call, DMS 8192 Bytes

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	245	472	8128	0,918
100 Trying	311	170	403	8128	1,296
180 Ringing	615	327	560	16256	0,911
200 OK (1)	610	328	561	24384	0,920
ACK	378	43	78	16256	0,206
TOTAL	2428	1113	2074		0,854

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1610	25,52	1605	24,67
100 Trying	1311	30,01	1454	132,52
180 Ringing	1184	43,82	1720	27,69
200 OK (1)	1270	39,07	1751	38,28
ACK	858	27,34	1352	23,69
TOTAL	6233		7882	

14.9.2 Shared Compression

14.9.2.1 Basic Voice Call, DMS 2048 Bytes

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	248	482	2498	0,938
100 Trying	311	68	311	2809	1,000
180 Ringing	615	208	454	5408	0,738
200 OK (1)	610	208	454	8002	0,744
ACK	378	27	75	6396	0,198
TOTAL	2428	759	1776		0,731

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1442	47,04	1508	31,76
100 Trying	1181	52,77	1176	20,56
180 Ringing	1278	27,40	1413	20,12
200 OK (1)	1350	28,79	1466	35,28
ACK	1235	36,87	1052	26,76
TOTAL	6487		6615	

14.9.2.2 Basic Voice Call, DMS 4096 Bytes

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	245	479	4546	0,932
100 Trying	311	64	310	4857	0,997
180 Ringing	615	181	427	9504	0,694
200 OK (1)	610	180	426	14146	0,698
ACK	378	25	73	10492	0,193
TOTAL	2428	695	1715		0,706

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1615	26,86	1599	33,31
100 Trying	1399	29,08	1251	18,11
180 Ringing	1202	27,71	1494	20,01
200 OK (1)	1281	24,05	1513	25,57
ACK	984	15,32	1157	25,58
TOTAL	6481		7013	

14.9.2.3 Basic Voice Call, DMS 8192 Bytes

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	245	479	8642	0,932
100 Trying	311	64	310	8953	0,997
180 Ringing	615	181	427	17696	0,694
200 OK (1)	610	180	426	26434	0,698
ACK	378	25	73	18684	0,193
TOTAL	2428	695	1715		0,706

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1718	32,37	1704	33,38
100 Trying	1603	40,90	1420	53,91
180 Ringing	1350	28,77	1642	26,35
200 OK (1)	1510	34,62	1644	45,91
ACK	1148	29,33	1408	36,30
TOTAL	7330		7817	

14.10 Appendix J – Measurement Results: Unreliable versus Reliable Transport

14.10.1 Unreliable Transport

14.10.1.1 Dynamic Compression, DMS 8192 Bytes, 3GPP Video Call

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1437	739	965	8128	0,672
100 Trying	254	173	405	8128	1,594
183 Session Prog.	1440	719	951	16256	0,660
PRACK (1)	1318	124	152	16256	0,115
200 OK (1)	904	43	71	24384	0,079
UPDATE	1291	52	87	24384	0,067
200 OK (2)	865	44	79	32512	0,091
180 Ringing	563	27	62	40640	0,110
PRACK (2)	717	31	80	32512	0,112
200 OK (3)	260	16	58	48768	0,223
200 OK (4)	1133	23	65	56896	0,057
ACK	458	15	78	40640	0,170
TOTAL	10640	2006	3053		0,287

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	3322	48,96	2883	64,41
100 Trying	1852	41,94	1527	28,68
183 Session Prog.	2117	37,83	2779	44,61
PRACK (1)	1807	75,40	1612	25,29
200 OK (1)	1284	20,35	1271	20,65
UPDATE	1800	40,27	1351	21,85
200 OK (2)	1303	34,65	1270	32,97
180 Ringing	1426	29,60	1291	43,22
PRACK (2)	1371	36,19	1312	32,91
200 OK (3)	879	14,45	1219	39,01
200 OK (4)	1682	43,93	1310	60,09
ACK	1213	118,74	1208	25,85
TOTAL	20054		19031	

14.10.1.2 Shared Compression, DMS 8192 Bytes, 3GPP Video Call

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1437	739	972	9565	0,676
100 Trying	254	21	266	9819	1,047
183 Session Prog.	1440	295	540	19387	0,375
PRACK (1)	1318	70	111	20705	0,084
200 OK (1)	904	40	81	29737	0,090
UPDATE	1291	52	100	31028	0,077
200 OK (2)	865	38	86	40021	0,099
180 Ringing	563	29	77	48712	0,137
PRACK (2)	717	27	89	41301	0,124
200 OK (3)	260	9	64	57817	0,246
200 OK (4)	1133	26	81	67078	0,071
ACK	458	15	91	51280	0,199
TOTAL	10640	1361	2558		0,240

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	3395	29,34	3062	47,92
100 Trying	2389	37,02	1364	37,98
183 Session Prog.	2287	34,76	2145	49,93
PRACK (1)	2405	31,58	1693	55,84
200 OK (1)	2313	49,72	1630	32,64
UPDATE	3499	42,60	1646	24,82
200 OK (2)	3000	36,54	1506	45,12
180 Ringing	2006	19,49	1499	28,98
PRACK (2)	2856	73,62	1495	24,85
200 OK (3)	2431	34,87	1339	15,95
200 OK (4)	2432	31,80	1547	31,10
ACK	2664	157,62	1488	35,95
TOTAL	31677		20413	

14.10.1.3 Dynamic Compression, DMS 8192 Bytes, Basic Voice Call

See Section 14.9.1.3

14.10.1.4 Shared Compression, DMS 8192 Bytes, Basic Voice Call

See Section 14.9.2.3

14.10.2 Reliable Transport

14.10.2.1 Dynamic Compression, DMS 8192 Bytes, 3GPP Video Call

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1437	739	959	8128	0,667
100 Trying	254	173	399	8128	1,571
183 Session Prog.	1440	570	585	16256	0,406
PRACK (1)	1318	124	146	16256	0,111
200 OK (1)	904	43	65	24384	0,072
UPDATE	1291	52	81	24384	0,063
200 OK (2)	865	44	73	32512	0,084
180 Ringing	563	21	50	40640	0,089
PRACK (2)	717	31	74	32512	0,103
200 OK (3)	260	14	50	48768	0,192
200 OK (4)	1133	23	59	56896	0,052
ACK	458	15	72	40640	0,157
TOTAL	10640	1849	2613		0,246

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	3246	33,56	2826	73,87
100 Trying	1857	20,32	1521	26,72
183 Session Prog.	2039	44,57	2526	53,33
PRACK (1)	1789	31,00	1517	27,95
200 OK (1)	1354	54,41	1274	29,06
UPDATE	1859	39,48	1342	39,08
200 OK (2)	1279	24,86	1308	25,97
180 Ringing	991	17,22	1352	21,90
PRACK (2)	1362	34,34	1323	30,69
200 OK (3)	962	32,29	1206	30,89
200 OK (4)	2059	32,95	1296	32,10
ACK	1183	35,71	1195	27,07
TOTAL	19978		18685	

14.10.2.2 Shared Compression, DMS 8192 Bytes, 3GPP Video Call

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1437	739	972	9565	0,676
100 Trying	254	21	260	9819	1,024
183 Session Prog.	1440	292	314	19387	0,218
PRACK (1)	1318	124	159	20705	0,121
200 OK (1)	904	40	75	29737	0,083
UPDATE	1291	52	94	31028	0,073
200 OK (2)	865	38	80	40021	0,092
180 Ringing	563	49	91	48712	0,162
PRACK (2)	717	24	80	41301	0,112
200 OK (3)	260	9	58	57817	0,223
200 OK (4)	1133	55	104	67078	0,092
ACK	458	15	85	51280	0,186
TOTAL	10640	1458	2372		0,223

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	3328	43,90	3017	56,85
100 Trying	2393	43,15	1348	38,17
183 Session Prog.	2006	29,97	2104	52,04
PRACK (1)	2288	29,14	1867	43,12
200 OK (1)	2431	32,30	1514	28,02
UPDATE	3520	57,09	1638	29,95
200 OK (2)	3003	45,82	1547	39,11
180 Ringing	2766	58,60	1526	25,09
PRACK (2)	2856	58,27	1481	19,74
200 OK (3)	2487	40,35	1392	33,67
200 OK (4)	3807	67,32	1585	32,18
ACK	1978	34,00	1423	24,26
TOTAL	32865		20441	

14.10.2.3 Dynamic Compression, DMS 8192 Bytes, Basic Voice Call

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	245	472	8642	0,918
100 Trying	311	170	397	8953	1,277
180 Ringing	615	174	189	17696	0,307
200 OK (1)	610	15	30	26434	0,049
ACK	378	43	72	18684	0,190
TOTAL	2428	647	1160		0,478

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1614	55,26	1603	19,81
100 Trying	1305	15,86	1390	23,52
180 Ringing	1079	48,38	1542	29,96
200 OK (1)	1018	43,80	1221	44,51
ACK	895	42,46	1233	35,16
TOTAL	5911		6989	

14.10.2.4 Shared Compression, DMS 8192, Basic Voice Call

Message	Length uncompressed	Length compressed	Length of SigComp message	Cumulative amount of state memory used [bytes]	Compression ratio (uncompr/Sigcomp)
INVITE	514	245	479	8642	0,932
100 Trying	311	64	304	8953	0,977
180 Ringing	615	136	158	17696	0,257
200 OK (1)	610	15	37	26434	0,061
ACK	378	26	68	18684	0,180
TOTAL	2428	486	1046		0,431

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1719	36,51	1726	46,99
100 Trying	1604	30,52	1421	52,31
180 Ringing	1049	27,39	1611	24,71
200 OK (1)	1007	28,94	1204	37,94
ACK	1192	37,00	1383	32,21
TOTAL	6571		7346	

14.11 Appendix K – Measurement Results: Central Processor Unit

14.11.1 Pentium 4 Hyper-Threading 3.0 GHz

14.11.1.1 Video Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	2822	35,25	2941	57,28
100 Trying	1581	39,00	1332	48,31
183 Session Prog.	2301	35,14	2606	47,64
PRACK (1)	2125	41,49	1761	26,55
200 OK (1)	1609	35,23	1549	38,00
UPDATE	2088	34,30	1564	25,34
200 OK (2)	1681	36,35	1591	20,20
180 Ringing	1625	30,42	1443	15,80
PRACK (2)	1525	36,42	1540	23,40
200 OK (3)	1229	54,64	1401	51,39
200 OK (4)	1874	27,00	1520	63,13
ACK	1679	24,63	1396	27,45
TOTAL	22139		20644	

14.11.1.2 Voice Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1532	17,28	1594	47,55
100 Trying	1436	35,91	1260	38,76
180 Ringing	1232	48,52	1542	111,69
200 OK (1)	1312	34,57	1540	41,21
ACK	1049	39,54	1172	33,01
BYE	1181	29,49	1169	28,78
200 OK (2)	959	37,21	1167	43,12
TOTAL	6561		7108	

14.11.2 Pentium 4 2.66 GHz

14.11.2.1 Video Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	2820	37,12	3281	30,42
100 Trying	1635	20,05	1095	16,38
183 Session Prog.	2238	25,80	2988	424,67
PRACK (1)	2102	36,22	1745	396,56
200 OK (1)	1711	209,94	1482	381,07
UPDATE	2052	26,94	1462	250,98
200 OK (2)	1644	23,51	1537	407,28
180 Ringing	1769	19,33	1391	387,99
PRACK (2)	1454	23,65	1353	5,33
200 OK (3)	1178	194,39	1495	621,11
200 OK (4)	1922	17,76	1312	32,39
ACK	1805	22,50	1313	383,84
TOTAL	22329		20453	

14.11.2.2 Voice Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1499	20,80	1572	126,01
100 Trying	1441	20,79	1046	24,98
180 Ringing	1163	19,39	1510	406,89
200 OK (1)	1274	13,52	1442	15,28
ACK	1012	14,80	967	5,56
BYE	1105	14,06	962	10,73
200 OK (2)	946	11,79	981	8,29
TOTAL	6390		6537	

14.11.3 Pentium 4 1.8 GHz

14.11.3.1 Video Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	4315	90,15	5157	76,76
100 Trying	2253	63,98	1564	53,72
183 Session Prog.	3502	481,59	4636	470,70
PRACK (1)	3268	209,01	2513	428,80
200 OK (1)	2382	330,43	2041	337,47
UPDATE	3098	135,58	1909	46,26
200 OK (2)	2335	83,12	1868	78,48
180 Ringing	2548	75,17	1722	52,65
PRACK (2)	2069	195,82	1919	83,33
200 OK (3)	1458	60,27	1959	125,58
200 OK (4)	2777	67,19	1788	102,04
ACK	2582	88,47	1704	116,89
TOTAL	32587		28780	

14.11.3.2 Voice Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	2215	113,48	2396	163,85
100 Trying	2017	111,97	1569	105,43
180 Ringing	1647	16,27	2052	60,67
200 OK (1)	1846	36,87	2148	51,33
ACK	1327	17,98	1366	60,56
BYE	1676	13,48	1323	25,36
200 OK (2)	1277	48,01	1426	275,75
TOTAL	9051		9531	

14.11.4 Pentium M 1.6 GHz

14.11.4.1 Video Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	3097	7,02	3348	14,54
100 Trying	1382	27,85	1318	7,01
183 Session Prog.	2765	33,18	2970	10,81
PRACK (1)	2497	16,89	1822	2,50
200 OK (1)	1744	35,17	1531	10,03
UPDATE	2456	30,79	1620	30,23
200 OK (2)	1768	34,84	1594	25,31
180 Ringing	1618	22,67	1454	8,63
PRACK (2)	1546	20,31	1559	4,60
200 OK (3)	1114	20,53	1354	6,43
200 OK (4)	2165	27,38	1492	39,72
ACK	1650	17,92	1345	7,66
TOTAL	23801		21406	

14.11.4.2 Voice Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1322	13,76	1537	12,46
100 Trying	1170	19,60	1115	20,39
180 Ringing	1177	29,75	1493	16,67
200 OK (1)	1262	25,79	1518	17,63
ACK	918	23,27	1043	18,95
BYE	1040	36,43	1053	20,46
200 OK (2)	842	12,82	1011	18,60
TOTAL	5847		6707	

14.11.5 Pentium III 600 MHz

14.11.5.1 Video Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	10071	230,58	10931	156,94
100 Trying	4790	67,28	4086	219,57
183 Session Prog.	8588	189,88	9525	20,57
PRACK (1)	7757	167,27	5880	37,42
200 OK (1)	5611	17,57	5059	195,74
UPDATE	7539	18,30	5271	184,27
200 OK (2)	5559	15,38	4951	17,71
180 Ringing	5752	16,18	4738	91,19
PRACK (2)	4833	15,35	5070	6,70
200 OK (3)	3472	13,64	4836	251,70
200 OK (4)	6861	17,98	4789	73,15
ACK	5649	25,30	4451	196,82
TOTAL	76481		69588	

14.11.5.2 Voice Call

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	5058	1009,05	5085	138,35
100 Trying	4339	469,75	3644	98,11
180 Ringing	3981	486,56	4746	85,85
200 OK (1)	4210	77,54	5061	529,15
ACK	3143	255,61	3431	36,34
BYE	3404	18,80	3384	10,82
200 OK (2)	2923	142,18	3380	17,86
TOTAL	20731		21967	

14.12 Appendix L – Measurement Results: Different Sequences

14.12.1 Basic Voice Session Establishment

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	514	245	479	4546	0,932
100 Trying	311	64	304	4857	0,977
180 Ringing	615	136	158	9504	0,257
200 OK (1)	610	15	37	14146	0,061
ACK	378	26	68	10492	0,180
TOTAL	2428	486	1046		0,431

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	1539	30,86	1623	65,07
100 Trying	1436	49,11	1288	48,28
180 Ringing	915	47,94	1290	26,29
200 OK (1)	885	22,37	1035	41,36
ACK	1023	46,28	1171	38,13
TOTAL	5797		6407	

14.12.2 Basic Video Session Establishment

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1069	511	745	5101	0,697
100 Trying	317	20	260	5418	0,820
180 Ringing	472	67	89	9922	0,189
200 OK (1)	663	156	178	14617	0,268
ACK	431	31	73	11016	0,169
TOTAL	2952	785	1345		0,456

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	2099	23,40	2134	64,19
100 Trying	1457	67,47	1170	52,06
180 Ringing	808	52,87	1118	37,64
200 OK (1)	967	42,10	1401	36,78
ACK	1052	45,63	1203	28,17
TOTAL	6383		7026	

14.12.3 Push-to-talk Session Establishment

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	980	569	797	5012	0,813
100 Trying	244	23	263	5256	1,078
200 OK (1)	882	335	357	10170	0,405
ACK	380	15	50	10550	0,132
BYE	380	13	48	14962	0,126
200 OK (2)	237	12	54	15199	0,228
TOTAL	3103	967	1569		0,506
TOTAL without BYE and 200 OK	2486	942	1467		0,590

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompre ssion	Standard deviation, decompress ion time [us]
INVITE	2011	44,83	2326	28,58
100 Trying	1417	55,17	1152	34,91
200 OK (1)	1206	45,06	1779	35,98
ACK	1023	19,08	1160	32,76
BYE	1205	35,04	1206	31,31
200 OK (2)	1006	36,36	1145	31,75
TOTAL	7868		8768	
TOTAL without BYE and 200 OK	5657		6417	

14.12.4 Registration

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
REGISTER	747	423	657	0	0,880
401 Unauthorized	476	476	476	0	1,000
REGISTER	963	565	799	4995	0,830
200 OK (1)	580	314	548	4612	0,945
TOTAL	2766	1778	2480		0,897

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
REGISTER	1756	29,84	1980	34,31
401 Unauthorized	0	0,00	0	0,00
REGISTER	1900	49,26	2201	61,21
200 OK (1)	1527	76,08	1695	46,06
TOTAL	5183		5875	

14.12.5 3GPP Video Session Establishment

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1437	791	1025	5469	0,713
100 Trying	254	20	260	5723	1,024
183 Session Prog.	1440	539	561	11195	0,390
PRACK (1)	1318	126	161	12513	0,122
200 OK (1)	904	46	81	17449	0,090
UPDATE	1291	59	101	18740	0,078
200 OK (2)	865	44	86	23637	0,099
180 Ringing	563	151	193	28232	0,343
PRACK (2)	717	41	97	24917	0,135
200 OK (3)	260	11	60	33241	0,231
200 OK (4)	1133	255	304	38406	0,268
ACK	458	18	88	30800	0,192
TOTAL	10640	2101	3017		0,284

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	2594	38,70	2793	84,81
100 Trying	1374	53,33	1175	39,06
183 Session Prog.	1910	37,89	2248	80,70
PRACK (1)	2332	35,06	1523	43,25
200 OK (1)	1950	43,10	1303	41,47
UPDATE	2598	42,24	1316	30,71
200 OK (2)	2010	46,74	1299	24,53
180 Ringing	1684	37,18	1503	27,12
PRACK (2)	1681	27,84	1329	28,81
200 OK (3)	1288	21,68	1135	31,13
200 OK (4)	2456	13,02	1793	20,33
ACK	1495	34,19	1196	34,06
TOTAL	23373		18613	

14.12.6 3GPP Video Session Establishment with RE-INVITE Request and Unreliable Delivery of Provisional Responses

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1205	739	973	5237	0,807
100 Trying	255	21	261	5492	1,024
180 Ringing	1003	382	404	10527	0,403
200 OK (1)	905	29	51	15464	0,056
ACK	459	72	114	11891	0,248
RE-INVITE	1164	53	95	17087	0,082
200 OK (2)	869	31	73	21988	0,084
ACK	459	10	59	22447	0,129
TOTAL	6319	1337	2030		0,321

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	2268	36,38	2680	53,94
100 Trying	1395	60,49	1204	70,48
180 Ringing	1311	41,43	1854	49,62
200 OK (1)	1427	42,55	1059	49,43
ACK	1075	32,99	1312	20,44
RE-INVITE	2293	32,88	1316	25,80
200 OK (2)	1922	35,81	1264	37,33
ACK	1483	47,90	1151	43,86
TOTAL	13173		11839	

14.12.7 3GPP Video Session Establishment with RE-INVITE and Reliable Delivery of Provisional Responses

Message	Length uncompressed [bytes]	Length compressed [bytes]	Length of SigComp message [bytes]	Cumulative amount of state memory used [bytes]	Compression ratio (SigComp/uncompr)
INVITE	1205	739	973	5237	0,807
100 Trying	255	21	261	5492	1,024
180 Ringing	1003	378	400	10527	0,399
PRACK	1318	201	236	11845	0,179
200 OK (1)	261	19	54	16138	0,207
200 OK (2)	905	23	58	21075	0,064
ACK (1)	459	15	64	17502	0,139
RE-INVITE	1164	263	312	22698	0,268
200 OK (3)	838	31	80	27568	0,095
ACK (2)	459	10	66	28027	0,144
TOTAL	7867	1700	2504		0,318

Message	Avg time [us], compression and memory image	Standard deviation, compression time [us]	Avg time [us], decompression	Standard deviation, decompression time [us]
INVITE	2264	26,94	2679	56,05
100 Trying	1373	29,22	1189	40,73
180 Ringing	1306	28,83	1821	65,98
PRACK	2279	25,28	1660	37,75
200 OK (1)	1087	35,77	1223	40,63
200 OK (2)	1975	27,13	1226	37,16
ACK (1)	1472	31,93	1217	33,99
RE-INVITE	2345	25,92	1787	28,87
200 OK (3)	2005	42,52	1252	26,32
ACK (2)	1475	28,93	1158	30,62
TOTAL	17578		15211	

14.13 Appendix M – Measurement Results: Number of Workers

1 worker	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	9217	8412	805	1	1
	8648	7839	809	1	1
	9357	8549	808	1	1
	8621	7812	809	1	1
AVG	8960,75	8153,00	807,75	1,00	1,00

2 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	8446	7244	1202	1,64	2
	7852	6655	1197	1,63	2
	8304	7104	1200	1,64	2
AVG	8200,67	7001,00	1199,67	1,64	2,00

3 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	7021	5491	1530	2,13	3
	7834	6283	1551	2,15	3
	6911	5391	1520	2,11	3
AVG	7255,33	5721,67	1533,67	2,13	3,00

5 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	7748	5588	2160	2,98	5
	7384	5216	2168	2,99	5
	7489	5363	2126	2,92	5
	7357	5237	2120	2,93	5
AVG	7494,50	5351,00	2143,50	2,96	5,00

7 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	7604	4936	2668	3,66	7
	7163	4495	2668	3,63	7
	7674	4991	2683	3,67	7
AVG	7480,33	4807,33	2673,00	3,65	7,00

10 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	7810	4373	3437	4,59	10
	7922	4365	3557	4,75	10
	8151	4691	3460	4,65	10
	6958	3600	3358	4,49	10
AVG	7710,25	4257,25	3453,00	4,62	10,00

11 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	7691	4078	3613	4,83	11
	7832	4073	3759	4,97	11
	7779	4168	3611	4,8	11
	7593	4005	3588	4,8	11
AVG	7723,75	4081,00	3642,75	4,85	11,00

12 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	7622	3760	3862	5,1	12
	7850	3942	3908	5,17	12
	7781	3969	3812	5,07	12
	7115	3382	3733	4,95	12
AVG	7592,00	3763,25	3828,75	5,07	12,00

15 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	7664	3317	4347	5,69	15
	7926	3421	4505	5,8	15
	8224	3663	4561	5,94	15
AVG	7938,00	3467,00	4471,00	5,81	15,00

25 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	8574	2417	6157	7,76	25
	8891	2486	6405	8,05	25
	7705	2013	5692	7,2	25
AVG	8390,00	2305,33	6084,67	7,67	25,00

50 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	8897	995	7902	9,66	50
	9398	1051	8347	10,14	50
	8709	834	7875	9,59	50
AVG	9001,33	960,00	8041,33	9,80	50,00

100 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	9764	774	8990	10,64	100
	9787	749	9038	10,7	100
	9569	880	8689	10,48	100
AVG	9706,67	801,00	8905,67	10,61	100,00

250 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	9665	858	8807	10,37	107
	10692	885	9807	11,50	110
	9562	854	8708	10,26	116
AVG	9973,00	865,67	9107,33	10,71	111,00

500 workers	Time in system [us]	Time in buffer [us]	Processing time [us]	Avg active workers	Max active workers
	10024	878	9146	10,62	142
	10492	886	9606	11,1	100
	9932	883	9049	10,5	121
AVG	10149,33	882,33	9267,00	10,74	121,00

14.14 Appendix N – Measurement Results: Time in System and Throughput

14.14.1 Pentium 4 Hyper-Threading 3.0 GHz

14.14.1.1 Voice Calls

Number of simultaneous calls	Session initiation interval [s]	Avg time in system [us]	Avg time being processed [us]	Avg time in buffer [us]	Max CPU load [%], CPU 1	Max CPU load [%], CPU 2
250	0,760	927	879	48	1	1,4
500	0,380	1093	998	95	1,4	2,7
750	0,253	1470	1283	187	1,7	3,1
1000	0,190	2056	1735	321	3,7	4,1
1250	0,152	2277	1907	370	3,4	4,8
1500	0,127	2285	1895	390	4,4	5,7
1750	0,109	2307	1953	354	5,4	6,8
1875	0,101	2240	1899	341	5,4	6,5
1938	0,098	3421	2888	533	6,1	6,5
2000	0,095	5311	4583	728	6,1	7,1
2500	0,076	5983	5183	800	8,2	10,2
3000	0,063	7166	6209	957	15,6	16,6

Number of simultaneous calls	Avg packets/s	Avg packet size [bytes]	Total traffic [Mbit/s]	SigComp traffic [Mbit/s]	SIP traffic [Mbit/s]	Number of messages	Measurement time [s]
250	16,94	408,90	0,055	0,022	0,033	10250	605
500	33,71	410,78	0,111	0,044	0,067	20378	604
750	50,39	411,60	0,166	0,066	0,100	30584	607
1000	67,10	412,24	0,221	0,088	0,130	40703	607
1250	83,81	412,34	0,276	0,110	0,167	50801	606
1500	100,21	412,70	0,331	0,131	0,200	60708	606
1750	116,52	412,86	0,385	0,153	0,232	70812	608
1875	126,30	412,72	0,417	0,165	0,252	77369	613
1938	129,97	412,79	0,429	0,170	0,259	79204	609
2000	133,77	412,90	0,442	0,175	0,266	81497	609
2500	167,04	413,25	0,552	0,219	0,333	101491	608
3000	184,02	420,72	0,619	0,249	0,370	111598	606

14.14.1.2 Video Calls

Number of simultaneous calls	Session initiation interval [s]	Avg time in system [us]	Avg time being processed [us]	Avg time in buffer [us]	Max CPU load [%], CPU 1	Max CPU load [%], CPU 2
50	6,2	2261	2175	86	1	1
100	3,100	2249	2154	95	1	1,7
250	1,240	2251	2158	93	2,7	3,4
375	0,827	2272	2176	96	2,7	4,1
500	0,620	2597	2484	113	4,4	6,1
750	0,413	2721	2600	121	7,1	7,5
1000	0,310	3869	3640	229	10,5	12,5
1250	0,248	5533	5096	437	34,1	14,6
1500	0,207	7635	7166	469	43,1	31,2

Number of simultaneous calls	Avg packets/s	Avg packet size [bytes]	Total traffic [Mbit/s]	SigComp traffic [Mbit/s]	SIP traffic [Mbit/s]	Number of packets	Measurement time [s]
50	4,405	561,251	0,02	0,005	0,015	2659	604
100	8,563	573,803	0,039	0,009	0,03	5195	607
250	21,127	579,738	0,098	0,023	0,075	12811	606
375	31,494	582,381	0,147	0,034	0,112	19062	605
500	41,966	582,535	0,196	0,046	0,15	25429	606
750	62,842	584,044	0,294	0,069	0,225	38139	607
1000	83,564	584,772	0,391	0,091	0,299	50489	604
1250	104,487	585,158	0,489	0,114	0,375	63168	605
1500	124,842	585,622	0,585	0,137	0,448	75780	607

14.14.2 Pentium 4 2.66 GHz

14.14.2.1 Voice Calls

Number of simultaneous calls	Session initiation interval [s]	Avg time in system [us]	Avg time being processed [us]	Avg time in buffer [us]	Max CPU load [%]	Avg packets/s
250	0,76	1046	987	59	13,3	16,954
500	0,38	1158	1023	135	17,4	33,678
750	0,253	1717	1325	392	26,6	50,404
1000	0,19	2333	1613	720	27	67,096
1250	0,152	2884	1866	1018	32,1	83,726
1500	0,127	3257	2254	1003	32,9	100,323

Number of simultaneous calls	Avg packet size [bytes]	Total traffic [Mbit/s]	Number of messages	Measurement time [s]
250	407,996	0,055	10267	606
500	410,907	0,111	20380	605
750	411,812	0,166	30554	606
1000	412,31	0,221	40761	608
1250	412,695	0,276	50759	606
1500	412,819	0,331	60642	605

14.14.2.2 Video Calls

Number of simultaneous calls	Session initiation interval [s]	Avg time in system [us]	Avg time being processed [us]	Avg time in buffer [us]	Max CPU load [%]	Avg packets/s
50	6,2	2660	2549	111	17,7	4,4
100	3,100	2652	2552	100	21,1	8,599
250	1,240	2721	2594	127	30,7	21,116
375	0,827	2736	2614	122	36,7	31,441
500	0,620	3405	3100	305	44,9	41,935

Number of simultaneous calls	Avg packet size [bytes]	Total traffic [Mbit/s]	Number of messages	Measurement time [s]
50	559,266	0,02	2647	602
100	571,335	0,039	5197	604
250	579,659	0,098	12755	604
375	582,053	0,146	19044	606
500	583,115	0,196	25370	605