INTEGRATION OF ROHC USER SPACE LIBRARY WITH LINUX KERNEL USING NETLINK SOCKET

Submitted by:

Javed Raza

Supervised by:

Assoc Prof Dr Athar Mahboob

Thesis

Submitted to:

Department of Electronics and Power Engineering

Pakistan Navy Engineering College

National University of Sciences and Technology, Islamabad

In fulfillment of requirements for the award of the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

With Specialization in Communications

March 2012

IN THE NAME OF ALLAH,

THE MOST BENEFICENT, THE MOST MERCIFUL

ABSTRACT

The low bandwidth cellular link from the cellular handset to the cell base station when used for VoIP suffers from a problem. There is a disproportionately high header overhead as IP telephony speech data will be transferred encapsulated in RTP/UDP/IP. A voice packet along with link framing layer will be appended with IPv4 header (20 bytes), a UDP header (8 bytes) and an RTP header (12 bytes) which makes a total header of 40 bytes. If IPv6 header is used instead of IPv4 then IP header would be of 40 bytes making a total of 60 bytes of header. Payload size depends on frame sizes and speech coding being used and it can be as low as 15-20 bytes for certain audio codecs. Hence for VoIP header, overhead can be as high as 60% to 80%. To alleviate this problem the IETF has standardized header compression techniques under the umbrella of Robust Header Compression (ROHC).

This project involves the integration of Robust Header Compression with Linux Kernel using Netlink Socket which is the interface between ROHC user space library and kernel module. The netfilter module passes all specified incoming packets to ROHC library for decompression and all specified outgoing packets for compression. Robust Header Compression Library is built to compress the headers of internet packets using official Internet standards specified in various relevant RFCs. We have created a ROHC daemon which runs an infinite loop listening on a Netlink Socket and calling ROHC compression and decompression functions on each packet received from the Linux kernel. The daemon maintains state for all outgoing packets and sends them out using RAW sockets. The incoming packets use a packet type identifier for ROHC and are passed on to the ROHC decompressor which generates packets with full UDP/IP headers. The decompressed packets are re-injected into the Linux kernel networking stack and are passed on to the recipient application. A VOIP application is tested over a low bandwidth link and the utility of the Robust Header Compression is demonstrated.

ACKNOWLEDGEMENT

Today, I am very much thankful to Almighty Allah, the All Knowing and Everlasting for His sympathies and blessings that He has always shower upon all His creatures and to cherish me with this great success. Indeed, He is the Guide without Whose blessings; I would not have had this moment of achievement.

I would also like to thank my parents and my family members for their all-out love and continuous support throughout my period of studies. It is their prayers which have made me whatever I am today and rescued me at times when I could easily stumble.

I am unable to find suitably fitting words to literally express my profound gratitude to my Thesis Advisor Assoc Prof Dr. Athar Mahboob, who is not only an ocean of knowledge within himself, but also a thorough gentleman. His patience and guidance, besides keeping me motivated to achieve the desired outcome have remained the sole contributors in timely and effective completion of the work. I would also like to thank Mr. Faraz Haider for his guidance in clarification of many concepts related to my work.

Last but definitely not the least; I would convey my appreciation to all the honorable members of the Guidance Committee, whose timely direction on various aspects of this work has kept me on track without getting astray.

TABLE OF CONTENTS

1. INTRODUCTION 11

1.1 Overview 11

1.2 Thesis Scope 14

1.3 Thesis Organization 15

2. ROBUST HEADER COMPRESSION 16

2.1 Introduction 16

2.2 Compression and Decompression States 17

2.2.1 Compressor States 18

2.2.2 Decompressor States 20

2.2.3 Modes of Operation 21

2.3 Encoding Techniques 24

2.4 CRC Techniques 24

2.5 ROHC Benefits 25

3. LINUX KERNEL 27

3.1 Introduction 27

3.2 Kernel Space 29

3.2.1 Basic Facilities 31

3.2.2 Process Management 32

3.2.3 Memory Management 32

3.2.4 Device Management 33

3.3 User Space 33

3.4 Communication between Kernel and User Space 34

4. NETLINK SOCKETS 35

4.1 Background 35

4.2 Introduction 35

4.3 Netlink Sockets vs Other IPCs 38

4.3.1 Asynchronous 38
4.3.2 No Compilation Time Dependency 38
4.3.3 Multicasting 38
4.3.4 Full Duplex 39
4.3.5 Provides BSD Style Socket 40

5. OUR IMPLEMENTATION 41
5.1 Overview 41
5.2 Creating Development Environment for Project 43
5.3 Building of ROHC Library 43
5.4 ROHC Compression 44
5.5 ROHC Decompression 45
5.6 Testing Scenario 46

6. RESULTS 48
6.1 Achieved Results 50
6.2 Comparison of Obtained Results 56

7. CONCLUSIONS 62
7.1 Concluding Remarks 62
7.2 Future Work 63

REFERENCES 65

APPENDIX 67

LIST OF TABLES

Table 1.1 IETF Header Compression Standards Comparative Analysis ….. 13

Table 2.1 Header Compression Gains ……………………………………….. 25

Table 2.2 Advantages of ROHC ………………………………………………. 26

Table 6.1 Header Compression in Various Profiles ………………………... 56

LIST OF FIGURES

Figure 1.1 VOIP Transmission using SIP and RTP .....……………............ 12

Figure 2.1 IP/UDP/RTP Headers ……….…………………………………. 17

Figure 2.2 Compressor State Diagram …….……..……………………….. 19

Figure 2.3 Decompressor State Diagram ……………………..…………….. 21

Figure 2.4 State Diagram for U mode ……….....………………………….. 22

Figure 2.5 State Diagram for O mode ……..…………………….………... 23

Figure 2.6 State Diagram for R mode ….…….………………………......... 24

Figure 3.1 Linux Architecture Overview ….……….………….……………. 28

Figure 3.2 Layers of Software Running on Linux ……………......……… 29

Figure 3.3 Linux Kernel Space Components ………………..…………… 30

Figure 4.1 Netlink Socket Purpose ..……..…………………...................... 36

Figure 4.2 Multicasting Technique of Netlink Socket ……………………... 39

Figure 5.1 Main Process Diagram of the Project ……............................. 42

Figure 5.2 ROHC Sender Machine ……………………......................... 45

Figure 5.3 ROHC Receiver Machine ....………………………… 46

Figure 5.4 Generating Stream of Packet …...………………...................... 47

Figure 6.1 Result of Uncompressed Packet using Wireshark 49

Figure 6.2 Runtime Process of Compressed IP Packet ....…...…….. 50

Figure 6.3 Result of IP Compressed Packet using Wireshark ...………… 51

Figure 6.4 Runtime Process of Compressed IP/UDP Packet ……...……. 52

Figure 6.5 Result of IP/UDP Compressed Packet using Wireshark ……. 53

Figure 6.6 Runtime Process of Compressed IP/UDP/RTP Packet ……... 54

Figure 6.7 Result of IP/UDP/RTP Compressed Packet using Wireshark .. 55

Figure 6.8 Percentage Reduction in Header Sizes 57

Figure 6.9 Runtime Process of Decompressed IP Packet 58

Figure 6.10 Runtime Process of Decompressed IP/UDP Packet 59

Figure 6.11 Runtime Process of Decompressed IP/UDP/RTP Packet …... 60

Figure 7.1 ROHC Compression of VOIP Packet …………………………. 63

GLOSSARY

2.5G	2.5 Generation wireless networks
3G	3rd Generation wireless networks
CRTP	Compressed Real-Time Transport Protocol
CTCP	Compressed Transport Control Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPCP	The PPP Internet Control Protocol (RFC1332)
IPHC	Internet Protocol Header Compression (RFC 2507)
IPC	Inter Process Communication
MAC	Media Access Control
RFC	Request For Comments
ROHC	Robust Header Compression (RFC3095)
RTP	Real-Time Transport Protocol
SIP	Session Initiation Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VoIP	Voice over Internet Protocol
BER	Bit Error Rate
API	Application Programming Interface
RTT	Round Trip Time
PSTN	Public Switched Telephone Network
AMR	Adaptive Multi Rate
BSD	Berkeley Software Distribution

Chapter 1

INTRODUCTION

1.1 Overview

During the last decade, cellular telephony and Internet have become part and parcel of our everyday life. The Internet protocol based telephony also known as Voice over IP (VOIP) using SIP and RTP was launched back in 1990's and is improving day by day since then. SIP or Session Initiation Protocol is an application layer protocol that may run over TCP or UDP transport layers and is used for signaling messages which are used to set up voice and video calls and associated value added services. The actual transmission of voice and video data takes place using RTP, the Real time Transport Protocol. RTP runs over UDP. VOIP has come to replace the circuit switched PSTN and ISDN [1]. Cellular telephony is also set to adopt VOIP in the 4G cellular networks. VOIP based telephony adds significant header to each unit of sampled packetized voice in the form of RTP, UDP and IP headers. For low bandwidth links such as the cellular link between handset and base station a significant portion of the link bandwidth is wasted due to headers. Most of the header is in fact redundant and can be compressed for the low bandwidth link to create more room for voice data. The Internet Engineering Task Force (IETF) has defined a number of standard

methods to compress headers under the umbrella of ROHC.

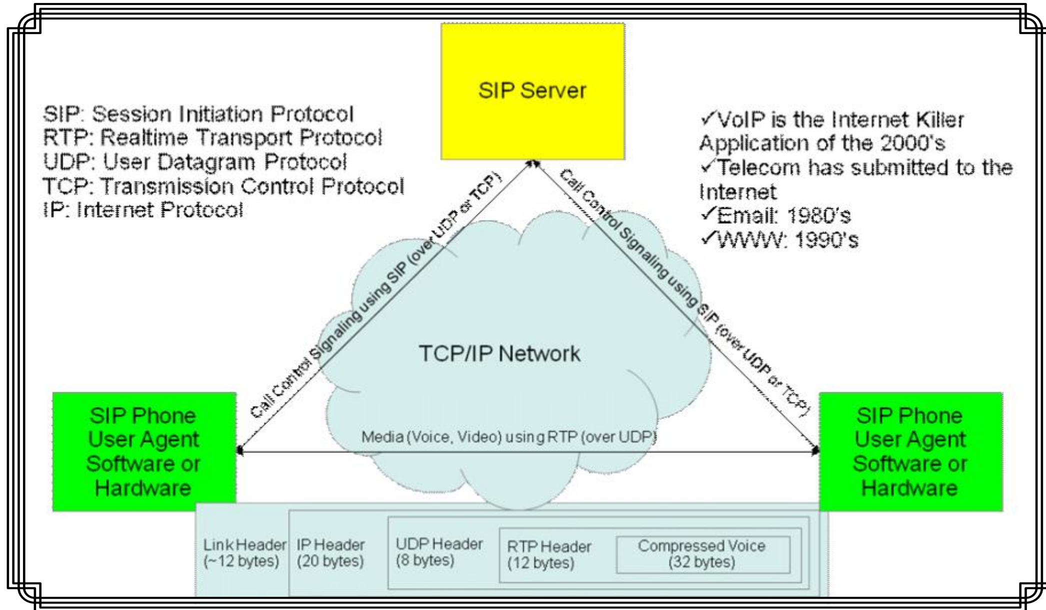

Figure 1.1: VOIP Transmission using SIP and RTP

Historically, there have been a number of attempts to compress protocol headers to increase link efficiency on the Internet starting way back in the 1980s. Some of the earlier approaches and their short comings are discussed below.

For low bandwidth links such as PSTN in order to increase IP/TCP flows performance Van Jacobson [6] compression scheme was introduced. Compression of IP/UDP was not supported by this scheme as UDP use was not common then. Compression of 40 bytes to an average 4 bytes is obtained via this scheme. State of TCP connection is saved at both link ends, and sending header field differences which changes. Van Jacobson compression is not suitable for wireless links and applications containing multimedia data which are predominantly UDP based.

Later, UDP as well as RTP traffic compression was achieved via IPHC and the CRTP schemes. Like Van Jacobson they use delta compression technique. They have their own feedback mechanism to recover from error conditions rather than depending on the TCP recovery mechanisms. They achieve compression of

header up to 2 bytes. For wireless links it is suitable with checksum link layer that is not much robust to deal high bit error rates, high RTT times and greater losses.

On 2.5G and 3G links high BER and long RTT are common; a proficient and strong compression scheme was required. In order to meet these criteria's ROHC scheme was developed. This framework can be expanded to various packet stream profiles like IP/UDP/RTP, IP/ESP, IP/UDP and Uncompressed. New profile like IP/TCP can be easily added to this. A comparison of various header compression scheme developed on the Internet is given in Table 1.1 [18].

Table 1.1: IETF Header Compression Standards Comparative Analysis

IETF standard	**VJ, CTCP (RFC 1144)**	**IPHC (RFC 2507)**	**CRTP (RFC 2508)**	**ROHC (RFC 3095)**
Headers	IPv4/TCP	IPv4 (options & fragments included), IPv6 (extension headers included), AH, Minimal Encap-sulation header, Tunnelled IP headers, TCP (options included),UDP	IPv4, IPv6 (extension headers included), AH, Minimal Encapsulation header, Tunnelled IP headers, UDP, RTP	IPv4 (options and fragments included), IPv6 (extension headers included), AH, Minimal Encapsulation headers, GRE, Tunnelled IP headers, UDP, RTP
Minimum Header Compressed	Two bytes	Two bytes	Two bytes	One byte
Link Type (BER/RTT)	Dial up (Low/Short)	Dial up & wireless (Low to medium/ Short to medium)	Dial up & wireless(Low to medium/ Short to medium)	Wireless (High/ Long)
Encoding	Differential	Differential	Differential	Window-based Least Significant Bit
Error recovery (Feedback)	TCP based (No)	Twice (Yes)	Twice (Yes)	Local repair (Yes)
Recommended in (standards)	--	UMTS Release99 onward CDMA2000 Release B onwards	--	UMTS Release4 onward CDMA2000 Release B onward

1.2 Thesis Scope

The low bandwidth cellular link, when used for VOIP, from the cellular hand to the cell base station suffers from a problem. The disproportionately high header overhead IP telephony speech data will be transferred by RTP mostly. A packet along with link framing layer will be appended with IPv4 header of 20 bytes, a UDP header of 8 bytes and an RTP header of 12 bytes. This makes a total of 40 bytes of header. If IPv6 header is used instead of IPv4 then IP header would be of 40 bytes making a total of 60 bytes. Payload size depends on frame sizes and speech coding being used as it can be as low as 15-20 bytes for certain audio codecs.

As the main problem is disproportionately large header sizes, the reduction in header size is essential to ensure efficient link utilization. The performance of header compression is fairly low in case of IP Header Compression and Compressed Real Time Protocol. In addition a bit error rate of around the value of 0.01 to 0.001 is acceptable, as long as the resources to be utilized efficiently; furthermore, it can take around 100 to 200 milliseconds round trip time. The high BER increases the chance of erroneous data transmission without even detecting of error by receiver. The header compressor system for the cellular links should cater for the losses incurring in the pre compression and also in the compression and decompression stages. Cost wise analysis of resources established the fact that the bandwidth being the highest and processing is relatively low cost option. Hence ROHC is an efficient technique that works in a bandwidth limited environment.

ROHC achieves robustness by feedback mechanism. 1 byte compression of header can be achieved by this. It is complex when analyzed with other schemes. It is beneficial in wireless links where the radio spectrum resource is very expensive. To quote from RFC3095 [9], “Bandwidth is the most costly resource in cellular links. Processing power is very cheap in comparison. Implementation or computational simplicity of a header compression scheme is

therefore of less importance than its compression ratio and robustness.”

1.3 Thesis Organization

This thesis is organized in terms of chapters. Chapter 1 with the title of "Introduction" intends to familiarize with the background knowledge and overview of the topic. It guides with the thesis organization for the upcoming chapters.

Chapter 2 with the title of "Robust Header Compression" contains the details of various protocols. In this chapter we have discussed about the Robust Header Compression technique that how the packet header is compressed and decompressed. It also discuss about the states and operation of ROHC. Some benefits and Encoding Techniques of ROHC are also covered in the chapter to show the efficiency of Robust Header Compression.

Following is the chapter 3 with the title "Linux Kernel". This chapter deals with the Kernel space and User space and the inter process communication between the Kernel space and User space. Kernel being the core component of the Linux is responsible for the major communication functionalities.

Chapter 4 relates to "Netlink Sockets" which explains this special IPC for transferring data between processes running in Kernel and User space. The chapter also discusses the advantages that Netlink Socket offers as compared to the other IPC methods.

Chapter 5 explains the "Implementation" of the thesis in a detailed manner. In this chapter there is a main diagram for the implementation. It tells about the techniques and how the coding technique works for ROHC.

Chapter 6 with the title of "Results" is comprised of the results achieved from the implementation of this project.

Finally, in chapter 7 we provide conclusions and some recommendations for future work.

Chapter 2

ROBUST HEADER COMPRESSION

2.1 Introduction

In this chapter we discuss the operational model used by IETF Robust Header Compression. This discussion provides a clear conceptual understanding of ROHC on our part.

Robust Header Compression (ROHC) [2,3], a method for IP, UDP, RTP, and TCP headers compression method. Figure 2.1 shows how each protocol adds up to form a 40 byte header for each packet. Some of the fields of these headers are static while others are not.

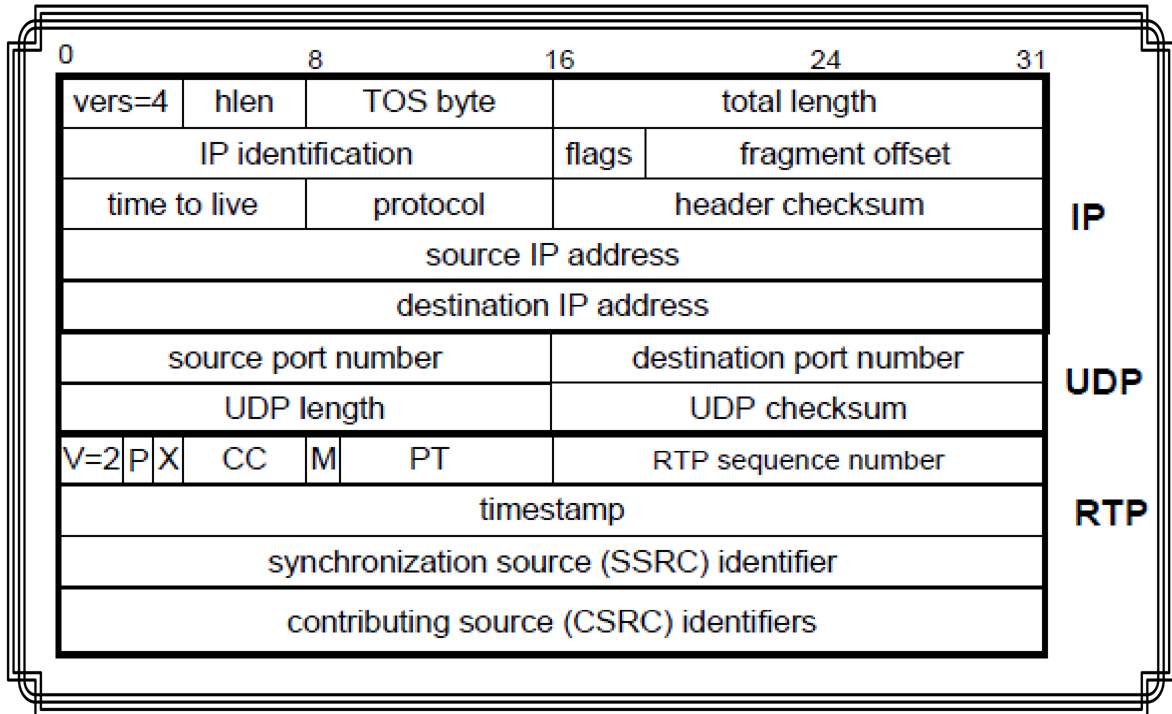

Figure 2.1: IP/UDP/RTP Headers

In applications which involve streaming, the overhead 40 bytes is added on every packet for IPv4 or 60 bytes for IPv6. It makes 60% of total sent data especially for VOIP [5]. They can be tolerated with the wired links as there capacity is not a problem but it definitely an issue for wireless links as bandwidth is not that much.

40 bytes or 60 bytes of overhead compression into 1 or 3 bytes can be done if we place compressor before the sending link that has narrow capacity and after link decompressor can be used. Larger overhead is compressed to only a few bytes, while decompressor creates original packet by reversing process at compressor.

2.2 Compression and Decompression States

The two state machines interact with each other, a compressor machine and a decompressor machine, each of them is used in a context just one time, this is as specified by ROHC Header compression. There are three states in each of the

compressor and the decompressor and they are linked with each other, these remains valid even if the meaning of the states between two is slightly different for compressor and decompressor. Starting of state machines is done at the compression state that is lowest then it slowly goes to the higher states.

The compressor and decompressor transitions do not necessarily synchronize. Normally, lower states are where the compressor comes back while if context damage is identified then decompressor would transit back.

2.2.1 Compressor States

The three compressor states for ROHC compression are: Initialization and Refresh (IR), First Order (FO), and Second Order (SO) states [2,3]. Operation of compressor starts in the level of IR which further on increases to higher state. Function carried out in the highest compression state that it can achieve; under constraint that compressor is sure enough that decompressor has relevant information for header compression and decompression required in that state.

The transitions between the compression states are made by the compressor based on the following considerations:

- variation in packet headers
- positive feedback from the decompressor(Acknowledgments - ACKs)
- negative feedback from decompressor (Negative ACKs - NACKs)
- Unidirectional mode operation such as timeouts occurring periodically like occurring over simplex channels or where feedback mechanism is not enabled.

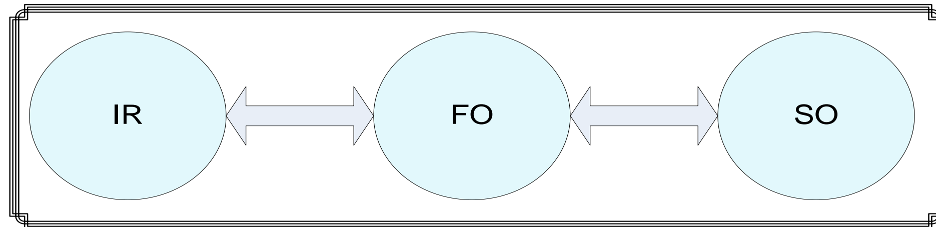

Figure 2.2: Compressor State Diagram

2.2.1.1 Initialization and Refresh (IR) State

Initialization and Refresh state function is to initialize the static parts of the context in the decompressor or recovery after failure. The compressor sends entire header information in this state, which comprises that field which is static and non-static when uncompressed with additional information. Waiting is done by the compressor when it is in IR state up to the point where the correct static information has been achieved by decompressor.

2.2.1.2 First Order (FO) State

The First Order state function is to communicate irregularities in the packet stream, in this state the compressor infrequently sends information about all the dynamic fields, and information transmitted is compressed minimally to some extent. The updating can be done for the field that is static.

The compressor goes from the Initialization and Refresh state into the First Order state and from the Second Order state whenever the headers of the packet stream do not conform to the previous pattern, the compressor stays in the First Order state till the time it is certain that the decompressor has obtained all the new pattern parameters. Changes in fields which are irregular are communicated in all packets and are part of a uniform pattern.

Detecting the corruption occurred in packets that are sent in the First order state having context updating information is must to avoid from invalid updates and inconsistency in the context.

2.2.1.3 Second Order (SO) State

The maximum compression can be achieved in the Second Order state. When header which is to be reduced is expected given the Real Time Transport Protocol Sequence Number (SN) and the compressor is determined that decompressor has gained all function parameters like Sequence Number to other fields, the compressor enters the Second Order state. Correct decompression of packets settle on correct decompression of the Sequence Number when they are sent in the Second Order state, however, correct information is required to be delivered at the decompressor for its success when sent in the preceding FO state packets.

When the headers do not conform to the uniform pattern and cannot be compressed individually on the basis of previous context information, the compressor goes back to the First Order state and leaves this state.

2.2.2 Decompressor States

The decompressor initializes from "No Context" state which is the state at the lowest level of compression and then transfer to states that are higher than this. Once the decompressor has entered the “Full Context” state, it never leaves this state normally.

The decompressor, initially in "No Context" state has not decompressed a packet. The decompressor transfers to the "Full Context" state when the packet has been decompressed perfectly by the reception of static and dynamic information in the initialization packet, and transmit back to lower states upon repeated failures. However, when the failure occurs, initially it transmit again to the "Static Context" state where packet reception is capable to allow transfer to the "Full Context" state again. The decompressors go back to the "No Context" state in the case when the FO state packets sent decompression doesn't get success in "Static Context” state.

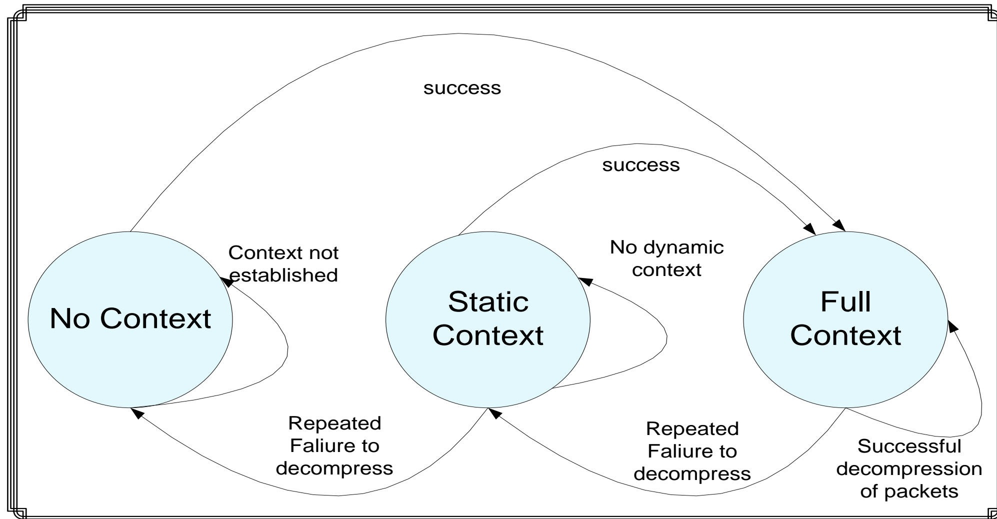

Figure 2.3: Decompressor State Diagram

2.2.3 Modes of operation

ROHC scheme consists of three modes named Unidirectional mode, Bidirectional Optimistic mode and Bidirectional Reliable mode. All implementations of ROHC must support these three modes of operation. These modes are described briefly as follows:

2.2.3.1 Unidirectional mode (U-mode)

A path from compressor to decompressor in one dimension is followed by packets in the U-mode so when the return path from decompressor to compressor is not present then ROHC is used over links through this mode.

In Unidirectional mode, only due to intermission and irregularity in header modify pattern in stream of compressed package, shift between compressor states are accomplished. Compression in the U-mode will be less efficient because of the timely refresh and lack of feedback for beginning of error recovery.

Compression with ROHC should begin in the U-mode. When the packet arrives at decompressor and a feedback is generated as an indication of change of

mode is required, transition to one of the modes that are bidirectional can be executed.

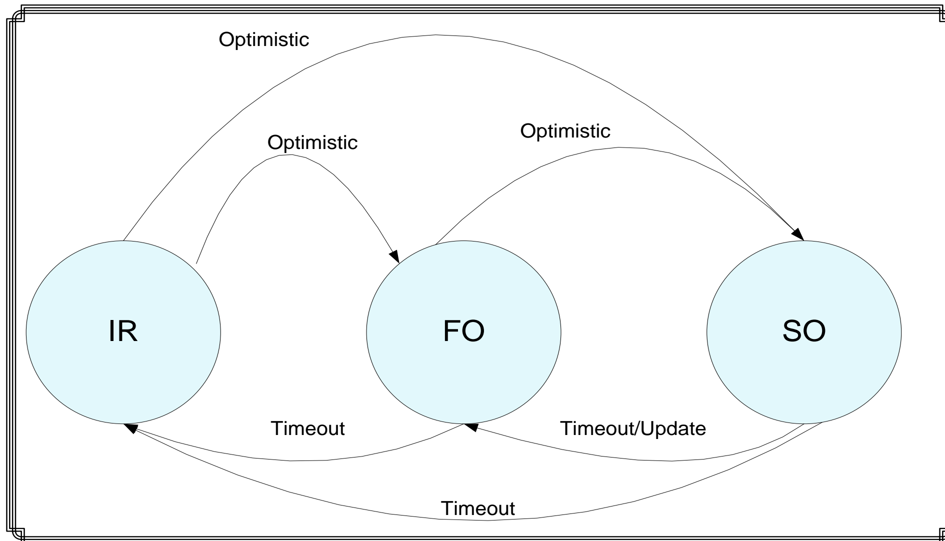

Figure 2.4: State Diagram for U mode

2.2.3.2 Bidirectional Optimistic mode (O-mode)

The O-mode the difference to unidirectional is that error recovery requests are sent through feedback channel and important context updates are acknowledged when received at the decompressor to the compressors and not only for sequence number modifications, however, refreshes that are according to a specified period are not applied in O-mode.

The objective of bidirectional optimistic mode is to compression efficiency maximization and feedback channel minimization. Due to residual errors or invalid context the Bidirectional Optimistic mode reduces the damaged headers being transmitted to the upper layers. When long error bursts occur, the recurrence of invalid context can be higher than the Bidirectional Reliable mode.

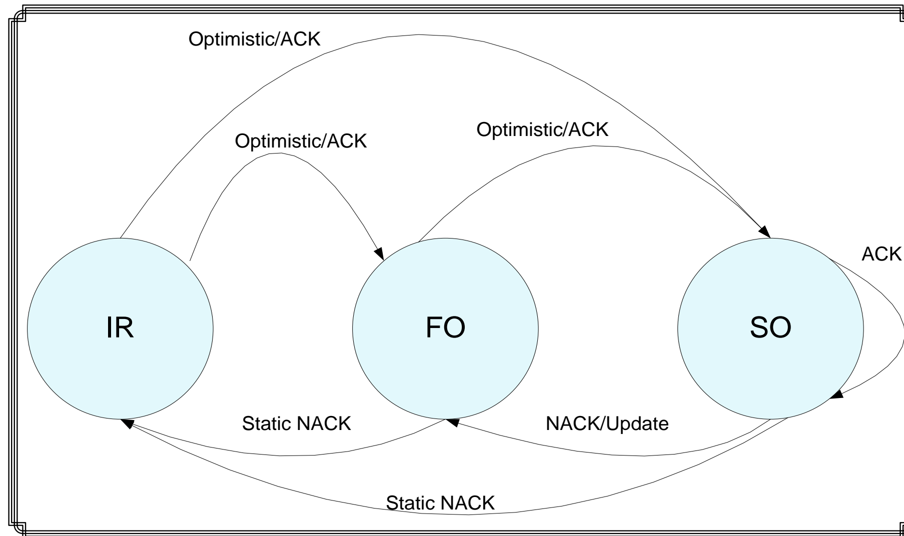

Figure 2.5: State Diagram for O mode

2.2.3.3 Bidirectional Reliable mode (R-mode)

The R- mode differences from unidirectional mode and Bidirectional Optimistic mode are the use of feedback channel more comprehensively and a strict logic which prevent synchronization loss of context between compressor and decompressor with the exception of excessive residual bit error rates. All context updates as well as sequence number field updates are acknowledged through feedback but context updates is not done by every packet.

Bidirectional Reliable mode's objective is to increase robustness for the prevention of damage and loss; it reduces invalidation under header loss or conditions of error stream. O-mode has a higher possibility of context invalidation than R-mode; consequently a greater quantity of corrupted headers can be transmitted by R-mode when context invalidation occurs.

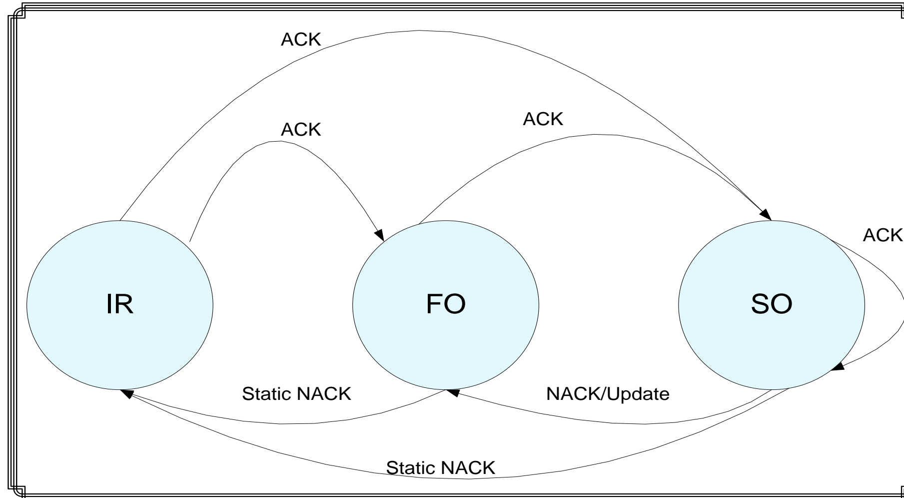

Figure 2.6: State Diagram for R mode

2.3 Encoding Techniques

There are several techniques that is used for the compression of ROHC. The purpose of applying this technique is significant byte saving and the idea is, it is not necessary to send full value with every packet. One of these techniques is Least Significant Bit (LSB) encoding which involves transmission of k least significant bit where k is a positive integer. With this value and the value received earlier the decompressor derives the reference value. Another method Window based Least Significant Encoding (WLSB) is used. In this method compressor may not be able to determine reference value of decompressor. It provides robustness to the LSB technique. Hence window of all possible values is used.

2.4 CRC Techniques

For initial packets an 8 bit CRC is appended by ROHC on all packet fields excluding the payload data. This is applied according a polynomial which is in Eq.1 [8]

$$1 + x + x^2 + x^8 \qquad (1)$$

In Compressed headers CRC is calculated on the original packet overhead. Some fields of header are changing and some are not so CRC is calculated in two parts CRC static and CRC dynamic. Transmission bit errors and context id related issues. In order to uniquely identify CRC for context updates two CRC widths are used. Compressed headers polynomials are in Eq.2 and Eq.3 [8]:

$$1 + x + x^3 \tag{2}$$

$$1 + x + x^2 + x^3 + x^6 + x^7 \tag{3}$$

2.5 ROHC Benefits

The ROHC compression can be achieved in most of common links, network and transport layer protocols as shown in Table 2.1 taken from [19].

Table 2.1: Header Compression Gains

Protocol Header	Total header size (bytes)	Min. compressed header size (bytes)	Compression gain (%)
IP4/UDP	28	1	96.4
IP4/UDP/RTP	40	1	97.5
IP6/UDP	48	3	93.75
IP6/UDP/RTP	60	3	95

Table 2.2 shows the advantages of using ROHC in wireless networks. The table is taken from [19]. It simulates a VoIP environment in which flow of packet is done using IPv6/UDP/RTP profiles. Flow of 31 bytes after every 20 milliseconds (ms) over a simulated wireless channel with BER level of 10^{-3} and used an uncorrelated BER model.

Table 2.2: Advantages of ROHC

	Without ROHC	With ROHC
Total Packets Transmitted	6000	6000
Packets Lost	3125 (52%)	1448 (24%)
Call Bandwidth (Kbps)	35.5	12.9
Average Header Size	60	3.1
Packets Lost Due to Error in Header	2309 (38%)	188 (3%)

Codecs that are able to handle payload bit errors ROHC are apparently able to benefit more from ROHC. ROHC has the capability to reduce packet loss by approximately 50% even without using such a codec. The compressed packets can be sent on certain wireless links in a single link frame. By this way the radio resources can be used effectively. These features make ROHC use in wireless networks possible to incorporate high-quality VOIP [4] services.

Chapter 3

LINUX KERNEL

3.1 Introduction

Linux is an open source operating system used in all sorts of computing and communications platforms. A large number of cell phones, PDA's, Tablet PCs, laptops, desktop PCs, servers, clusters, routers, and firewalls are now running the Linux operating system. Linux operating system is designed using the philosophy and principles of UNIX operating system. This design approach separates the system software into the kernel and user space.

The kernel is the core component of the operating system and serves as a link between the hardware level data processing and the end user applications. Managing resources which includes communication between hardware and software is one of the major responsibilities of a kernel. Serving basic component of a operating system, kernel provides for resources lowest layer such as processors and I/O devices that must be controlled by the application software to perform its duties well. These facilities are available via system calls and interprocess communication mechanisms.

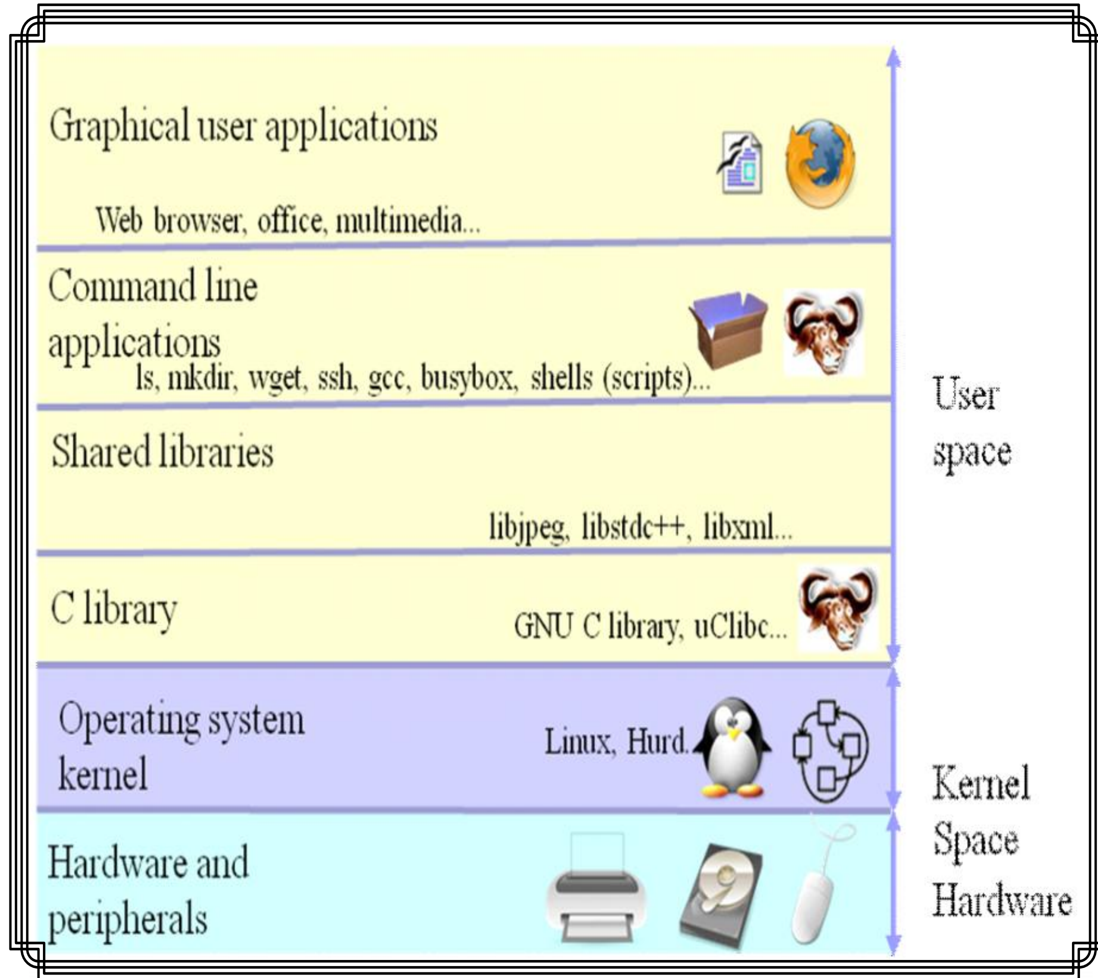

Figure 3.1: Linux Architecture Overview [22]

Operating system such as Linux which is an open source system gives good environment for application development as it is used in various platforms such as small embedded devices to large scale devices. Basic architecture or Linux is shown in figure 3.2 using kernel and user spaces. Different kernels do operating systems tasks differently. This is because every kernel is designed and implemented differently. As for example most of the operating system are run by microkernel which are in user space and act as servers which intends operating system maintainability improvement and monolithic kernels achieve goals by executing operating system code in the same address space aiming to increase system service.

Concept of kernel [10] is adopted in various operating systems. Kernel existence is result of computer system design with abstraction layers series, where each of them relying on layers beneath functions. Viewing kernel from this point it can be seen that it is software lowest level of abstraction. Avoiding kernel means designing software on system instead of the abstraction layer. By this design complexity will be increased. However this will definitely increase the design complexity to such an extent that implementation of simplest system would only be possible.

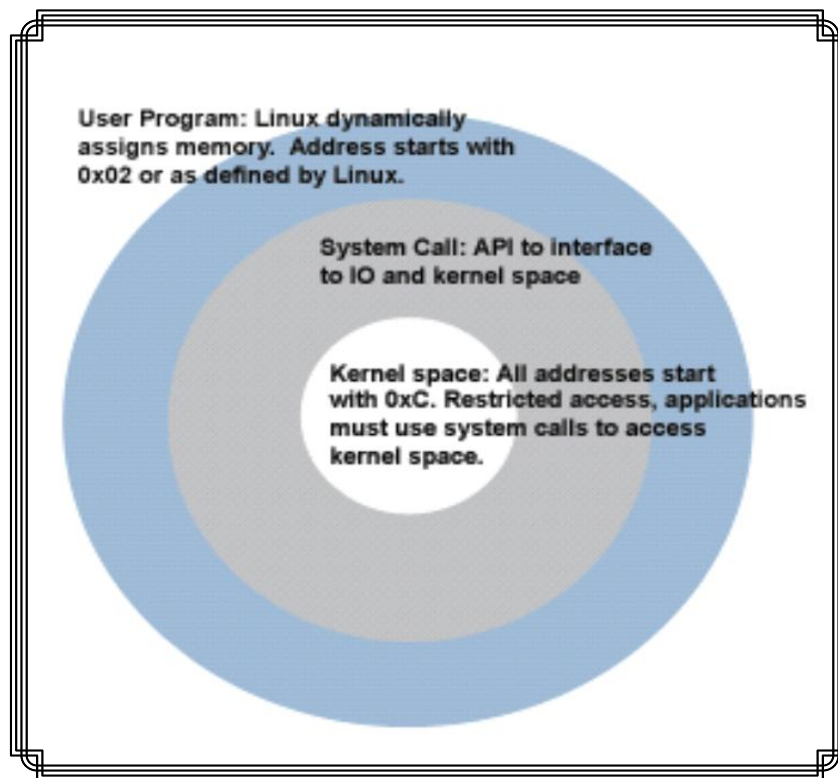

Figure 3.2: Layers of Software Running on Linux

3.2 Kernel Space

Kernel space runs with the help of kernel. Kernel was also referred to as the core or nucleus. This terminology for kernel was adopted due to the fact that early computers used memory form called core memory. This is called core because it contains important operating system support characteristics.

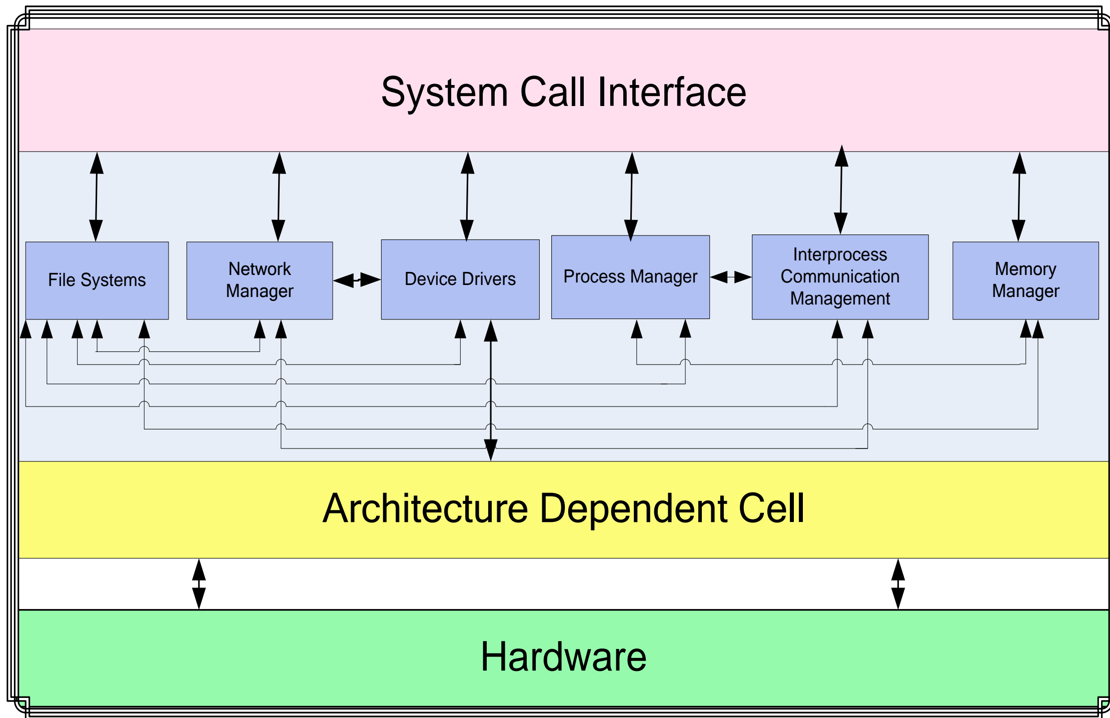

Figure 3.3: Linux Kernel Space Components

Execution of kernel starts in the supervisory mode via the boot loader. After kernel initialization the first process starts, this generally is the init process. After doing this kernel doesn't execute directly normally, it executes only in response to external events. These external events can be via system calls to kernel in order to service requests made by applications through hardware interrupts to notify kernel of real world events. Kernel provides loop when processes are not available to execute; called idle process.

The crucial and complex programming task in a kernel space is the kernel development. For good performance its central position in operating system is necessary which demands its efficient design and implementation. It may be possible sometimes that a kernel will not use the abstraction mechanism. Reason being the memory management concerns and one reason for further difficulty of its difficulty in development is lack of reentrancy.

Kernel Space provides following features:

1) Processes low level scheduling (dispatching)

2) Communication between processes

3) Synchronization in process,

4) Process control blocks manipulation,

5) Context switching,

6) Handling of Interrupts,

7) Creation and destruction of process,

8) Suspension and resumption of process.

3.2.1 Basic Facilities

Basic function of a kernel is management of computer resources and allowing the other programs to use and run on it. Resources are:

- The Central Processing Unit. Central part of a computer system, responsible for execution or running programs on it. Kernel has the decision power to decide at which time how many programs should run and how they are allocated to the processor or processors.

- The computer's memory. Memory stores both program instructions and data. For program execution they both should be present. Different programs access memory frequently may want to demands more than available memory in computer. Kernel decides which memory each process can use and what to do when enough of it is not available.

- Any Input/output (I/O) devices present in the computer, such as keyboard, mouse etc. Kernel allocates requests from applications to perform I/O request to a suitable device and provides well-suited methods for using the device.

These features can be implemented by kernel or on by relying upon other processes. Although if it is relying on other process some IPC means is required for allowing the processes to access those services that is provided by each

other.

Kernel must be providing running programs with a method to make requests to access these facilities.

3.2.2 Process Management

Kernel [11] major task is to allowing application execution and facilitating them with features like hardware abstractions. Memory applications can access is called as a process. Equipment built in for protection of memory must be taken in account by the kernel process manager.

Kernel sets up an address space for application in order to run an application. It loads file in memory that contain application code may be via demand paging. A queue for the program is set up by this and refers to location that is given in a program hence execution starts. These types of kernels are multitasking kernels. They can give a false impression that running process is higher than maximum processes that can simultaneously run physically.

Processes a system can run concurrently is equivalent to number of CPUs installed (if processors support simultaneous multithreading then it can change).

3.2.3 Memory Management

Full access to system memory allowing processes to access memory safely when they require it. First step for this is usually virtual addressing, achieved via paging and/or segmentation. Virtual address spaces are different for different processes. The memory that one process access at a particular address may be different from what another process access at that address [21].

On many systems, virtual address of program refers to data not in memory. Indirection layer provided by virtual addressing allows operating system to use other data stores, like hard drive, to store what would otherwise have to remain in

main memory (RAM). As a result, operating systems can allow programs to use more memory than the system has physically available. A scheme known as demand paging is when program needs data not in RAM currently. In that case CPU signals happening of the event and kernel gives response by writing inactive memory chunk on disk and replacement by the data requested by program. Program can be resumed from the point where it was stopped.

3.2.4 Device Management

In order the processes access the peripherals that are attached to a computer that are controlled via kernel with the help of device drivers. For example showing something on screen requires application request to kernel which would display request to device driver which is responsible for character/pixel plotting.

A list of available devices is at the kernel. This list may be known in advance, configured by user (from the earlier PC and systems not for personal use) or detected at run time by the operating system (known as plug and play).

3.3 User Space

All application software runs in user space. User space refers to libraries and programs that operating system use for kernel interaction. Each user space program has a virtual memory of its own. Unless requested other program memory cannot be used. In today`s world memory protection is main stream of today`s operating system. If in case of debugger and depending upon privileges kernel can be requested by the processes to take another process memory. Shared memory regions between other processes can also be requested [7].

Another approach taken in experimental operating systems is to have a single address space for all software, and rely on the programming language's virtual machine to make sure that arbitrary memory cannot be accessed - applications simply cannot acquire any references to the objects that they are not allowed to access

3.4 Communication between Kernel and User Space

Performance of a useful work depends upon the kernel provided services access. Each kernel implements it differently but mostly a C library or API that invokes kernel related functions is invoked.

Kernel function invoking method varies in every kernel. For instance it is not possible for a process running in user space to call kernel directly [14], as in this case processor access control rules violation. A few possibilities for kernel invoking methods:

- Interrupt simulated by software:

  It is a very common method and is available on most of the hardware.

- Call gate

  It is a special address that kernel stores in memory of a kernel at such a location which is known to the processor in the form of a list. On detection of a call to that address by processor it instead redirects without causing an access violation to targeted location. Hardware support is required which is available normally.

- Special systems call instruction

  Special hardware support is required which may be lacked in common architectures (notably, x86) however they are added in the recent models of x86 processors. It is worth to mention that not all PCs operating systems when provided with this use this.

- Memory-based queue

  This method generates large number requests but does not wait for each one result. Request details can be added to memory area that is scanned after a specified time period by kernel to find requests.

Chapter 4

NETLINK SOCKETS

4.1 Background

During the development of Linux 1.3 kernel Alan Cox added netlink sockets. It was added like a interface that aims to provide user and kernel multiple bidirectional links. It was later extended by Alexey Kuznetsov [20] during Linux 1.2 kernel development intends to extend this messaging interface with flexibility to transform it with the infrastructures required advanced routing. Since then Netlink Sockets became main interface of Linux for user and kernel space communication. Netlink design formulation is the same as Linux. Quoting Linus Torvalds [11] "Linux is evolution, not intelligent design". Neither its design document nor its specification is available. The only thing we are left with is the source code.

4.2 Introduction

The mechanism of Netlink is socket based for kernel and user space, between user space and different kernel and user space processes communication. These

type of sockets cannot go out of host edges as their processes addresses are from (inherently local) PIDs.

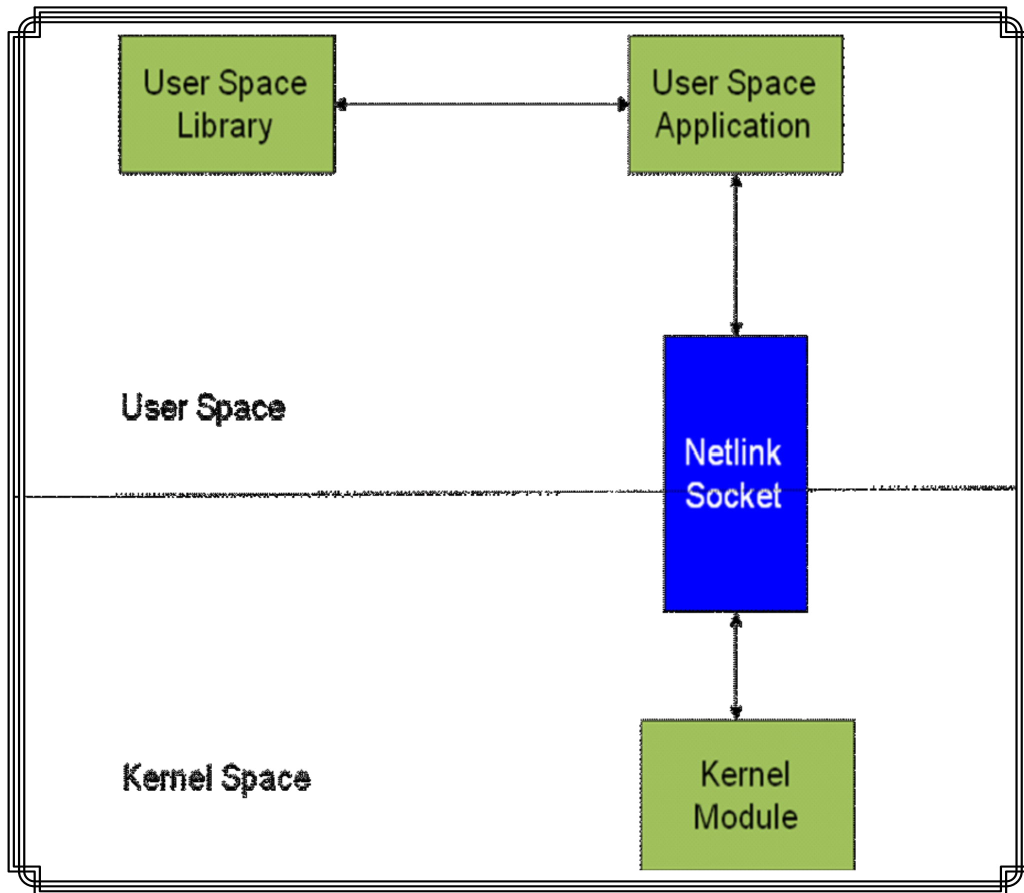

Figure 4.1: Netlink Socket Purpose

Netlink [12] as described is a messaging service that can be compared with a datagram type service for Interprocess Communication (IPC) system. Its design method is same as that of the BSD type and it uses methods like socket(), bind(), sendmsg() and recvmsg() similar to other methods of socket polling.

Jamal Hadi Salim [15,16] in 2001 at the ForCES IETF group tried to make a standard protocol between Forwarding Engine Component and a Control Plane Component. But this was not completed and other than this a protocol that is domain specific came in to existence. The Forwarding Engine Component is the router part responsible for forwarding while the other one is responsible for forwarding engine management and configuration.

Like iproute2, to communicate with user space to Linux kernel many networking utilities use netlink. A socket-based interface required for user space processes and kernel module internal API is what a Netlink comprised of. Instead of ioctl it is made more flexible.

Full duplex communication is provided in netlink. Compared to AF_INET address family which TCP/IP socket, netlink socket uses the family AF_NETLINK address. include/linux/netlink.h is the kernel header file. Each netlink socket features explains protocol type of its own.

The following is a subset of protocol types and their features of Netlink [13]:

- NETLINK_ROUTE: BGP, OSPF, RIP user-space routing daemons, between and kernel packet forwarding module communication channel. It is responsible for kernel routing table updation by user-space.
- NETLINK_FIREWALL: Packets by IPv4 firewall code is received by this.
- NETLINK_NFLOG: netfilter module in kernel-space and user-space iptable management tool communication channel.
- NETLINK_ARPD: User space arp table management is its responsibility.

These above features were missing in system calls, proc file systems and ioctls. Adding new features is a non trivial task when communication method between kernel and user space is other than netlink socket. In that case there is a fear of damaging system stability by polluting kernel. Simplicity is added feature of netlink socket, for the type of protocol a constant is added to header file. The application can talk to kernel module using socket style APIs immediately. However the netlink socket in comparison to other methods is described in the next section.

4.3 Netlink Sockets vs. Other IPCs

The various kernel and user space IPC methods [7], like ioctl, system call, proc file system or netlink socket. Comparisons of these are as under.

4.3.1 Asynchronous

Providing a socket queue for smoothing the message burst, netlink is asynchronous. For sending the message, system call The system call for sending a netlink message make a queue of message to the queue of receiver after which the reception handler receiver is invoked. In reference to the context of reception handler, receiver may choose either immediately process the message or leave it for processing in different context later on. Synchronous processing is what the system calls require unlike netlink. Therefore, to pass a message from kernel to the user space if system call is used, Kernel scheduling refinement can be affected if message processing is long.

4.3.2 No Compilation Time Dependency

System call if included in loadable module, which is in device drivers mostly, is not that appropriate. In compilation time code implementing a kernel`s system call is linked with the kernel statically. No time dependency in compilation exists in netlink socket, between application in loadable kernel modules in Linux kernel netlink core and netlink.

4.3.3 Multicasting

A near perfect event distribution mechanism from kernel to user space is provided in netlink sockets. It supports multicast and hence it is an add on benefit over ioctls, proc and system calls. The multicasting of message can be done by a process to the address of netlink group and any process can hear to that particular group address. These processes can be any in number.

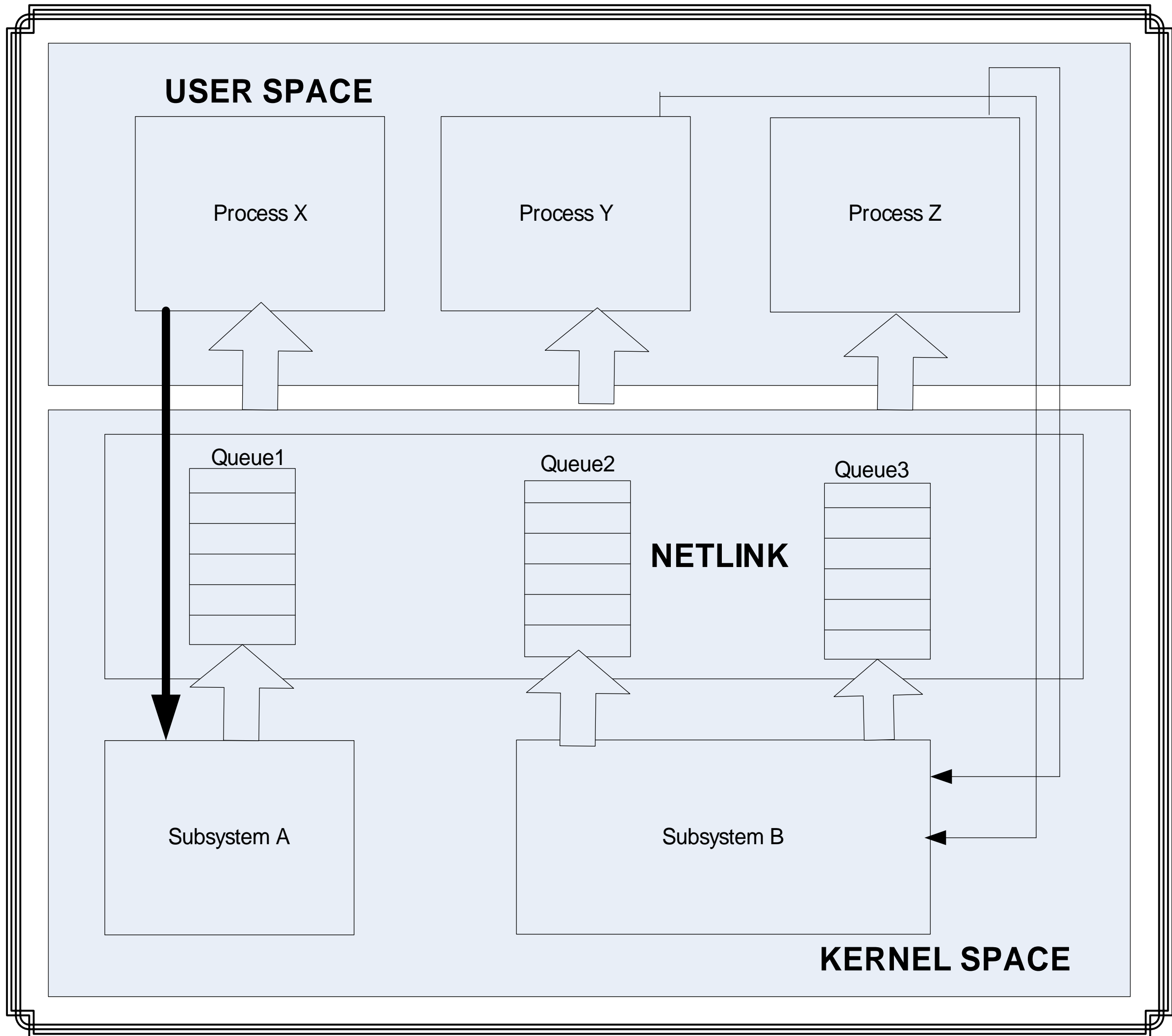

Figure 4.2: Multicasting Technique of Netlink Socket

4.3.4 Full Duplex

In a user-space application if there is an urgent message for kernel module then using system call and ioctl we cannot do the communication as they are simplex IPCs and only be initiated by user space program. Despite the fact that intensive polling is expensive, normally kernel is polled by applications to achieve changes in state. Netlink allows kernel to initiate sessions too and this is called full duplex characteristic of netlink socket.

4.3.5 Provides BSD style socket

BSD stands for Berkeley Software Distribution (BSD), it comprises on application programming interface (API) which is a library for developing applications in the C programming language to perform inter-process communication, most commonly for communications across a computer network. Finally, a BSD socket-style API that is provided by netlink socket is understandable by the software development community, so the cost of training is minimum with respect to other APIs.

.

Chapter 5

OUR IMPLEMENTATION

5.1 Overview

The complete architecture of our implementation is shown in figure 5.1. It involves the integration of ROHC with Linux kernel using Netlink Socket which is the interface between ROHC user space library and kernel module. The kernel passes all incoming packets to ROHC library for decompression and all outgoing packets for compression.

For making this project we have taken different steps to achieve our target. We have made project development environment to initiate the project. We have built Robust Header Compression Library to compress the headers of internet packets and run test programs. We have built Netlink socket examples and we have studied how user space and kernel space communicate using Netlink sockets [23]. We have created a ROHC Compressor within ROHC daemon which creates an infinite loop listening on the Netlink Socket and calling ROHC compression functions on each packet received from a kernel module. Similarly, we have created ROHC decompressor within ROHC daemon which calls ROHC

decompressor functions. To get the packets to the ROHC daemon we have used the facilities of the Linux library netfilter and used the target QUEUE.

For capturing the packets we have used Wireshark and some screen shots have been taken to show the compression and decompression results. We have also tested with a VoIP application over a Null-Modem link.

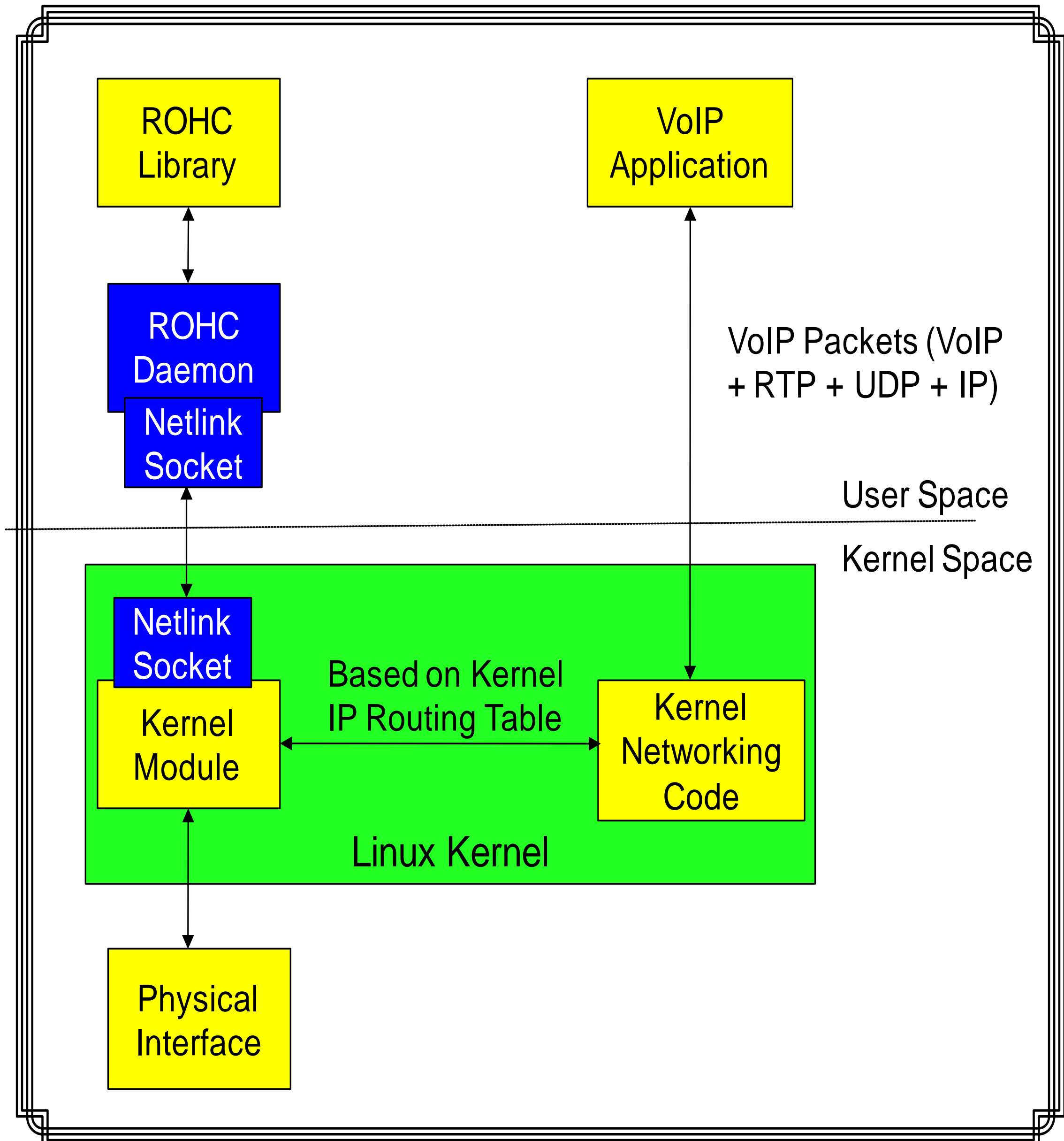

Figure 5.1: Main Process Diagram of the Project

5.2 Creating Development Environment for Project

Firstly we build the kernel code and user code and run it separately and then send message from kernel space to user space to check whether they are sending or receiving messages and can communicate with each other or not. The result shows that the kernel says hello and the user-space receives the message. After that various steps have been taken to build the ROHC library. Command "sudo dpkg-configure -a" was run. Security updates were applied using command "sudo aptitude update" and "sudo aptitude safe-upgrade".

5.3 Building the ROHC Library

Building the ROHC library requires some pre-requisite software. We need the C-compiler and a number of other libraries.

Rohc-1.3.1 was installed and rohc library was configured using ./autogen.sh command. Then autotools were installed and libtools were installed in which autoconf was run and then netfilter is built which is a library for applications dealing with netlink sockets. We added the following command to build ROHC library.

$ sudo apt-get install automake autoconf libtool build-essential

Then for building and running ROHC tests the libpcap, libpcap-dev were installed using

$ sudo apt-get install libpcap libpcap-dev

Then we downloaded the source code and extracted it using command

$ tar xvjf rohc-1.3.1.tar.bz2

For running library tests, we extracted the traffic captures

$ tar xvjf rohc-test-1.3.1.tar.bz2

Inside the source directory $ cd rohc-1.3.1, we configured the library

$ ./configure --prefix=/usr

Then we built the code by using $ make all

Then we installed the library using make install command.

5.4 ROHC Compression

ROHC Compression and Decompression is achieved by doing following process. The implementation was done with the help of a code obtained from [17].

For the compression, first an RTP packet is generated. This packet is then sent to the ROHC engine for the packet compression. As ROHC compression process runs in the user space, the packet is sent to the IP table routine whose basic function includes picking packet coming from a specified IP and sending it to NFQUEUE. NFQUEUE is a routine that lies in the stream compressor code and its basic purpose is to redirect the incoming packets. Nfqueue is the library of netlink socket which sends packet pulled from the kernel and sends it to arpping where it retrieves the MAC address using libnet library and then we prepare the ethernet header and header is compressed using ROHC library then we encapsulate the packet with ethernet header and send to the network. There is a source MAC address and destination MAC address and eth type. Then we send the packet to link layer which forwards it. We finalize the packet which consists of the ROHC header packet and then send it to a raw socket. A raw socket sends or receives the raw datagram excluding link level headers. It is the kind of socket in which no standard protocol is required instead it is being developed by the user. Then the packet is sent to the network.

The implementation flow is shown in figure 5.2. For the implementation purposes the programs are implemented in a set of files and run and the result is obtained which is captured with Wireshark.

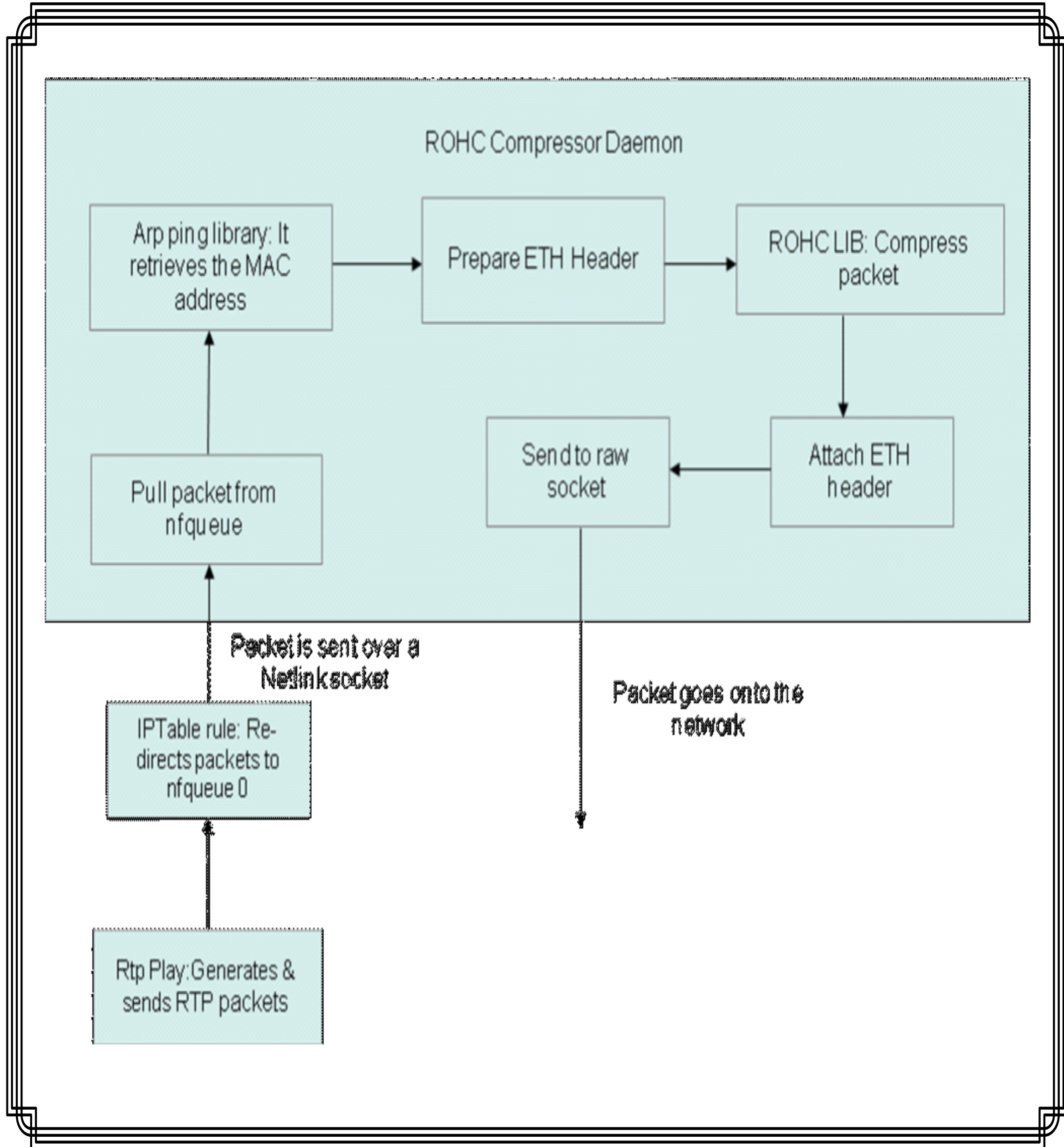

Figure 5.2: ROHC Sender Machine

5.5 ROHC Decompression

The packet is then received for decompression in the ROHC decompressor daemon which reads from raw socket. It then checks the Ethernet type and forwards the ROHC packets to remove the Ethernet header. Then the packet is

decompressed and the packet is pushed through socket to local host. The packet is read at port by rtp utility. The implementation flow is shown in figure 5.3.

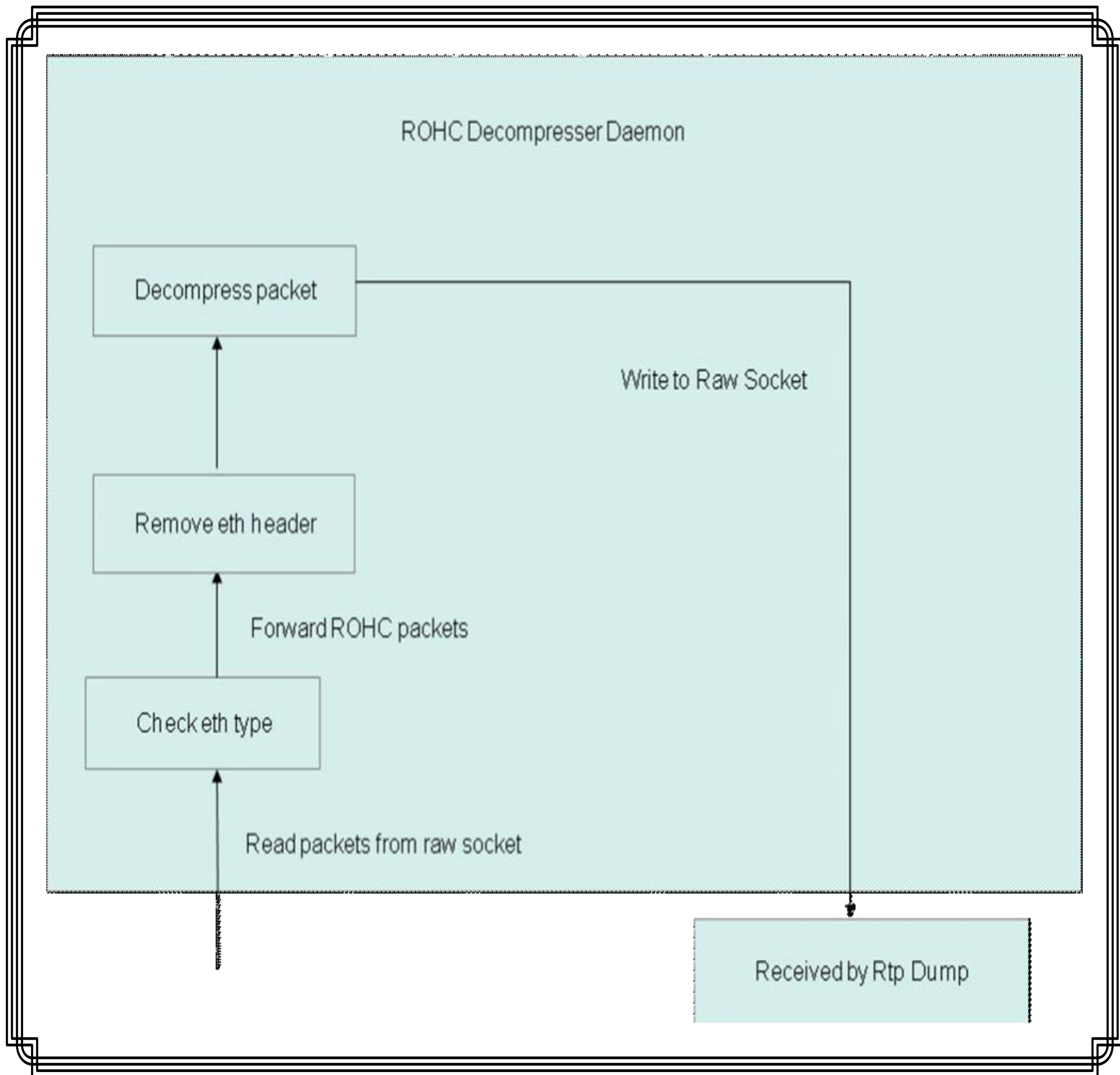

Figure 5.3: ROHC Receiver Machine

5.6 Testing Scenario

For testing the ROHC daemon operations a testing scenario is created via code routines such as RTP_Play() and RTP_dump(). The RTP play code sends the data on the network. It sends a packet and provides the following information related to a RTP packet.

Typically a voice packet size is of 30 bytes. When RTP_Play() program is run headers are added with each voice chunk. The screen shot in figure 5.4 shows the mp3 file which is taken as a VOIP application for testing and is converted to rtp stream using wav2rtp software and then the headers of packet is compressed.

```
send rtp stream
streamCompressor | streamDecompr... | send rtp stream | receieve rtp stream | me
/GstPipeline:pipeline0/GstSpeexEnc:speexenc1.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstSpeexEnc:speexenc1.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstCapsFilter:capsfilter0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstCapsFilter:capsfilter0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstAudioResample:audioresample0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstAudioResample:audioresample0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstAudioConvert:audioconvert1.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstAudioConvert:audioconvert1.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0.GstGhostPad:src0: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstSpeexDec:speexdec0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstSpeexDec:speexdec0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstQueue:queue0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstQueue:queue0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstOggDemux:oggdemux0.GstOggPad:serial_2fcaca9d: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstOggDemux:oggdemux0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstTypeFindElement:typefind.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstTypeFindElement:typefind.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin:decodebin0.GstGhostPad:sink: caps = NULL
/GstPipeline:pipeline0/GstOggMux:oggmux0.GstPad:sink_801819293: caps = NULL
/GstPipeline:pipeline0/GstOggMux:oggmux0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstSpeexEnc:speexenc0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstSpeexEnc:speexenc0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstAudioConvert:audioconvert0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstAudioConvert:audioconvert0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20.GstDecodePad:src0: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20/GstMad:mad0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20/GstMad:mad0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20/GstMPEGAudioParse:mpegaudioparse0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20/GstMPEGAudioParse:mpegaudioparse0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20/GstID3Demux:id3demux0.GstPad:src: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20/GstID3Demux:id3demux0.GstPad:sink: caps = NULL
/GstPipeline:pipeline0/GstDecodeBin2:decodebin20/GstTypeFindElement:typefind.GstPad:src: caps = NULL
Setting pipeline to NULL ...
Freeing pipeline ...
root@ROHCSender1:/home/iks/workspace/streamCompressor/bin# gst-launch-0.10 -v filesrc location=./Comfort_Eagle-Lov
e_You_Madly.mp3 ! decodebin2 !  audioconvert ! speexenc ! oggmux ! decodebin ! audioconvert ! audioresample ! 'aud
io/x-raw-int,rate=16000,width=16,channels=1' ! speexenc ! rtpspeexpay ! udpsink host=192.168.56.3 port=1234
```

Figure 5.4: Generating Stream of Packet

In addition to the RTP_Play() program there is a supportive code of RTP_dump() that is used for testing the ROHC implementation. RTP_play() is at the sender`s side whereas RTP_dump() is at the Receiver`s side. The function of RTP_dump() is to capture and analyze RTP packets.

Chapter 6

RESULTS

The code has been simulated for header compression of different packets. This chapter summarizes the achieved results and shows the runtime process of various profiles, compression and decompression of packets using wireshark and compression gain of various header compression profiles.

The screenshot in Figure 6.1 shows the uncompressed packets obtained via wireshark. In this figure the header size is of 40 bytes. It is worth mentioning that the 40 bytes of header is obtained when the packet is passed through the transport and the IP layer. On this we didn't apply the implemented compressed scheme. The data is routed to the network. The total length of the packet is 124 bytes including 14 bytes of ethernet header. So the data is of 70 bytes and header size is of 40 bytes making a total of 110 bytes.

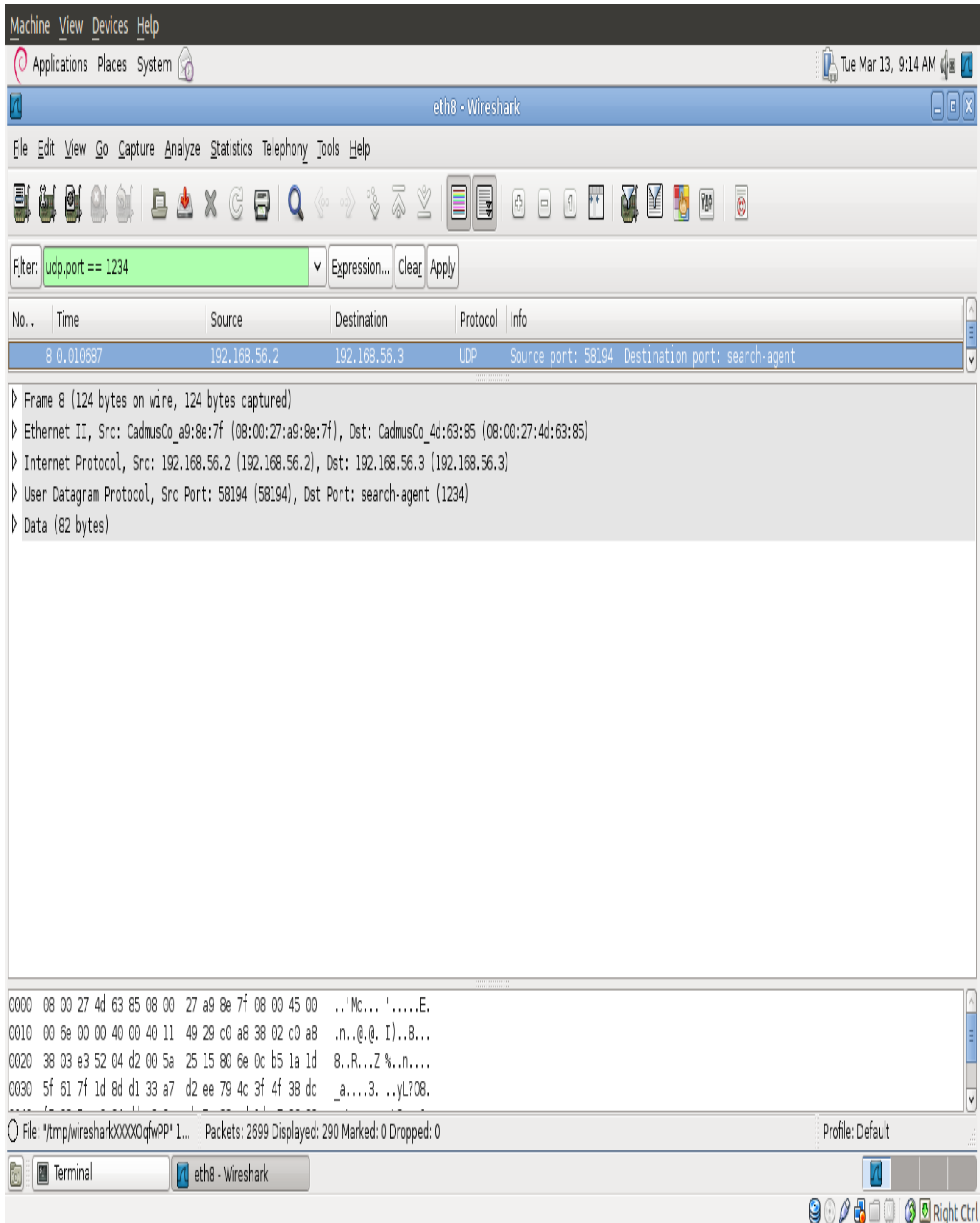

Figure 6.1: Result of Uncompressed Packet using Wireshark

6.1 Achieved Results

Figure 6.2 shows the runtime process of ROHC compression using IP profile in which the size of packet is 110 bytes. It shows that 17 bytes of header is compressed out of 20 bytes, so now header size in IP profile is 3 bytes. In this figure you can see that the packet at the IP level is compressed. When the IP header is compressed the packet length is now compressed to 93 bytes, from which payload size is 90 and header size is 3 bytes.

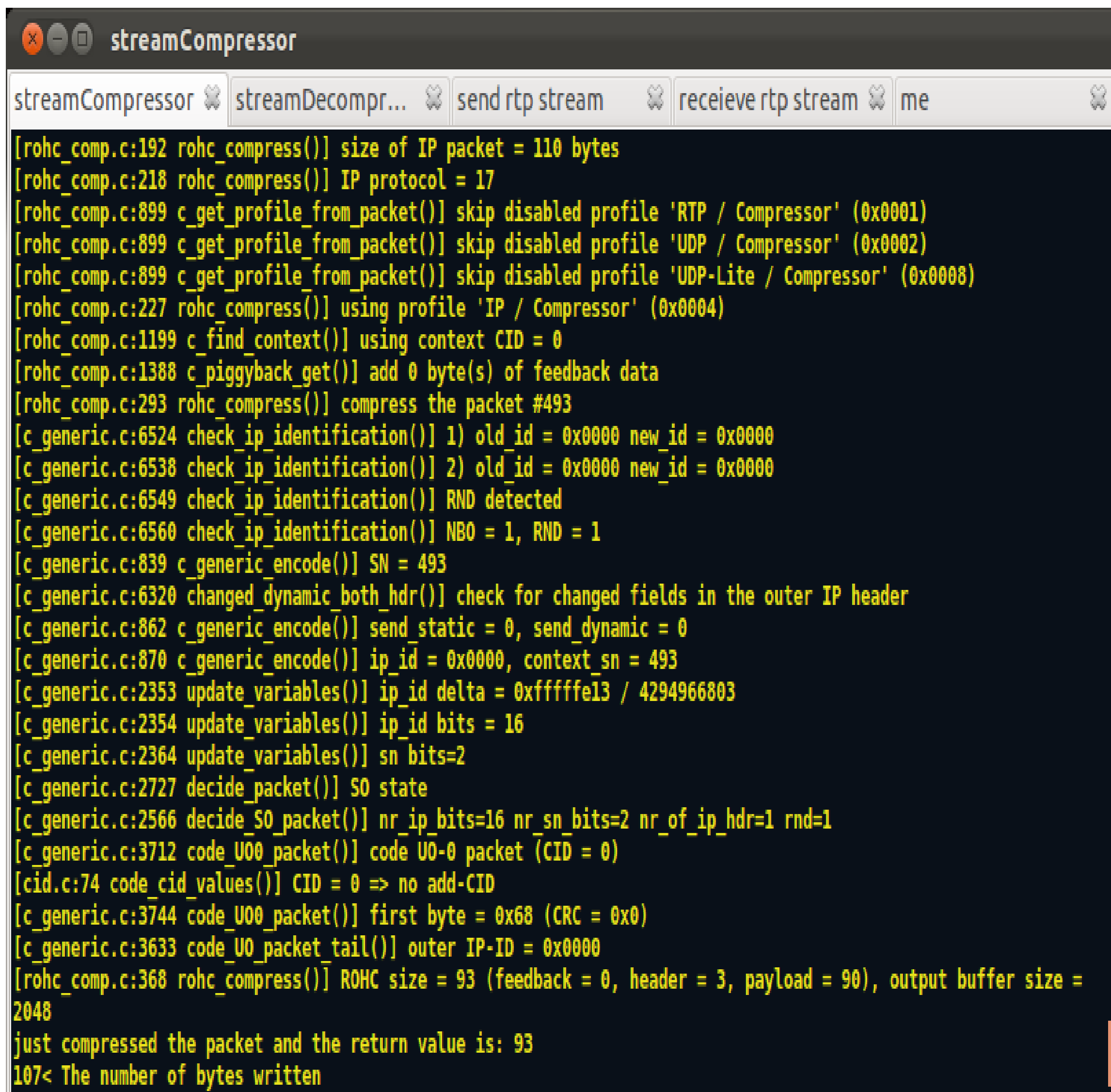

Figure 6.2: Runtime Process of Compressed IP Packet

The result of compression shown in figure 6.3 is captured using Wireshark. The figure shows IP compressed packets in which the header size is compressed to 3 bytes. In actual the header size was 20 bytes of IP. The total size of packet was of 110 bytes and when the IP header is compressed the length of packet now becomes 93 bytes.

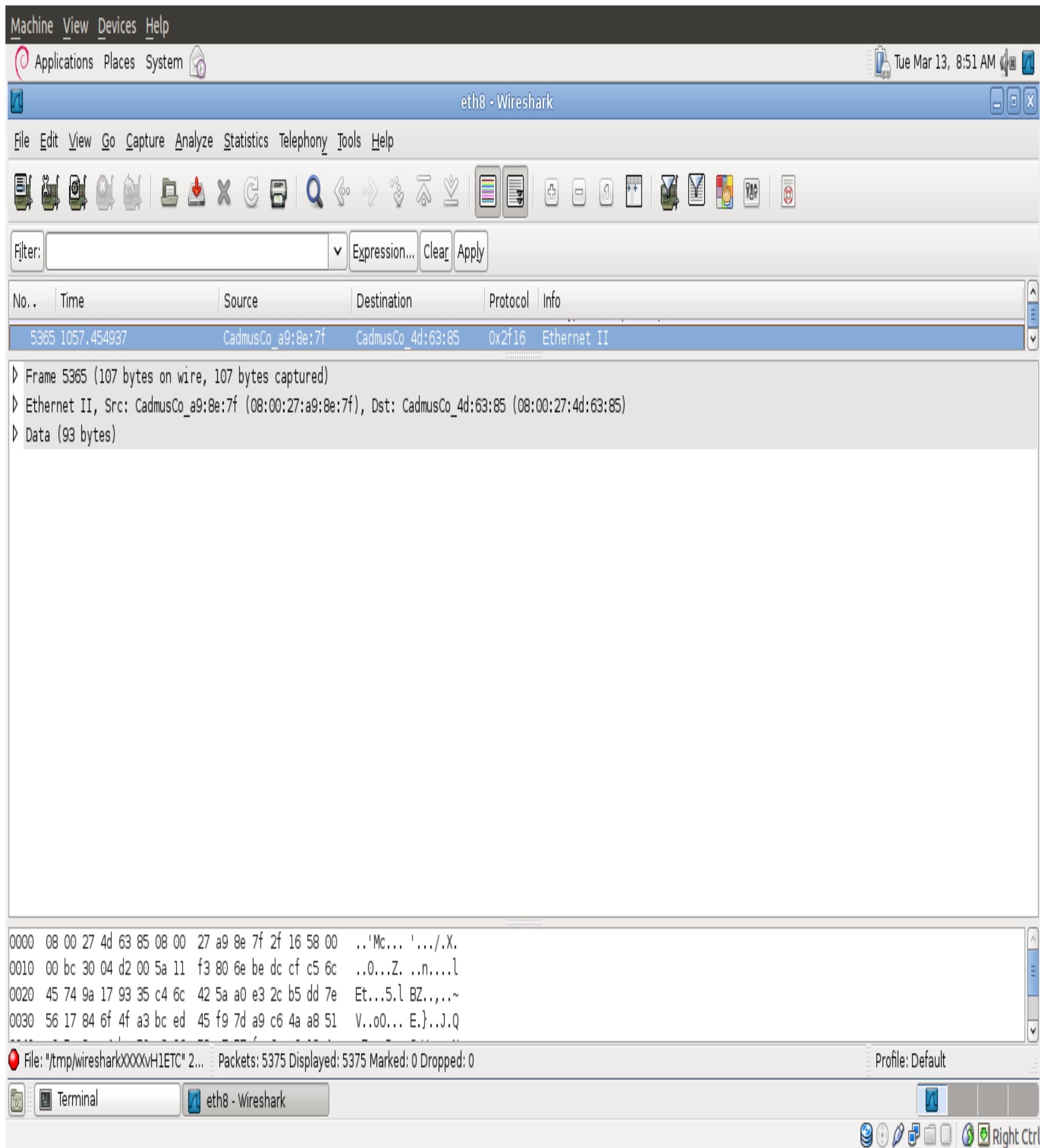

Figure 6.3: Result of IP Compressed Packet using Wireshark

Figure 6.4 shows the runtime process of ROHC compression using IP/UDP profile in which the size of packet is same of 110 bytes. It shows that 23 bytes of header is compressed out of 28 bytes (20 bytes of IP and 8 bytes of UDP), so now header size in IP/UDP profile is 5 bytes. In this figure you can see that the packet at the UDP level is compressed. When the IP/UDP header is compressed the packet length is now compressed to 87 bytes, from which payload size is 82 and header size is 5 bytes.

```
streamCompressor
streamCompressor | streamDecompr... | send rtp stream | receieve rtp stream | me
[rohc_comp.c:192 rohc_compress()] size of IP packet = 110 bytes
[rohc_comp.c:218 rohc_compress()] IP protocol = 17
[rohc_comp.c:899 c_get_profile_from_packet()] skip disabled profile 'RTP / Compressor' (0x0001)
[rohc_comp.c:227 rohc_compress()] using profile 'UDP / Compressor' (0x0002)
[rohc_comp.c:1199 c_find_context()] using context CID = 0
[rohc_comp.c:1388 c_piggyback_get()] add 0 byte(s) of feedback data
[rohc_comp.c:293 rohc_compress()] compress the packet #349
[c_generic.c:6524 check_ip_identification()] 1) old_id = 0x0000 new_id = 0x0000
[c_generic.c:6538 check_ip_identification()] 2) old_id = 0x0000 new_id = 0x0000
[c_generic.c:6549 check_ip_identification()] RND detected
[c_generic.c:6560 check_ip_identification()] NBO = 1, RND = 1
[c_generic.c:839 c_generic_encode()] SN = 349
[c_generic.c:6320 changed_dynamic_both_hdr()] check for changed fields in the outer IP header
[c_generic.c:862 c_generic_encode()] send_static = 0, send_dynamic = 0
[c_generic.c:870 c_generic_encode()] ip_id = 0x0000, context_sn = 349
[c_generic.c:2353 update_variables()] ip_id delta = 0xfffffea3 / 4294966947
[c_generic.c:2354 update_variables()] ip_id bits = 16
[c_generic.c:2364 update_variables()] sn bits=2
[c_generic.c:2727 decide_packet()] SO state
[c_generic.c:2566 decide_SO_packet()] nr_ip_bits=16 nr_sn_bits=2 nr_of_ip_hdr=1 rnd=1
[c_generic.c:3712 code_UO0_packet()] code UO-0 packet (CID = 0)
[cid.c:74 code_cid_values()] CID = 0 => no add-CID
[c_generic.c:3744 code_UO0_packet()] first byte = 0x6e (CRC = 0x6)
[c_generic.c:3633 code_UO_packet_tail()] outer IP-ID = 0x0000
[c_udp.c:431 udp_code_UO_packet_tail()] UDP checksum = 0x66f2
[rohc_comp.c:368 rohc_compress()] ROHC size = 87 (feedback = 0, header = 5, payload = 82), output buffer size = 20
48
just compressed the packet and the return value is: 87
101< The number of bytes written
```

Figure 6.4: Runtime Process of Compressed IP/UDP Packet

The screenshot in Figure 6.5 shows IP/UDP compressed packets obtained via

wireshark. In this figure the header size is compressed to 5 bytes when IP header is also included. In actual the header size was 20 bytes of IP + 8 bytes of UDP. The total size of packet was of 110 bytes and when the IP and UDP header is compressed the length of packet is now 87 bytes.

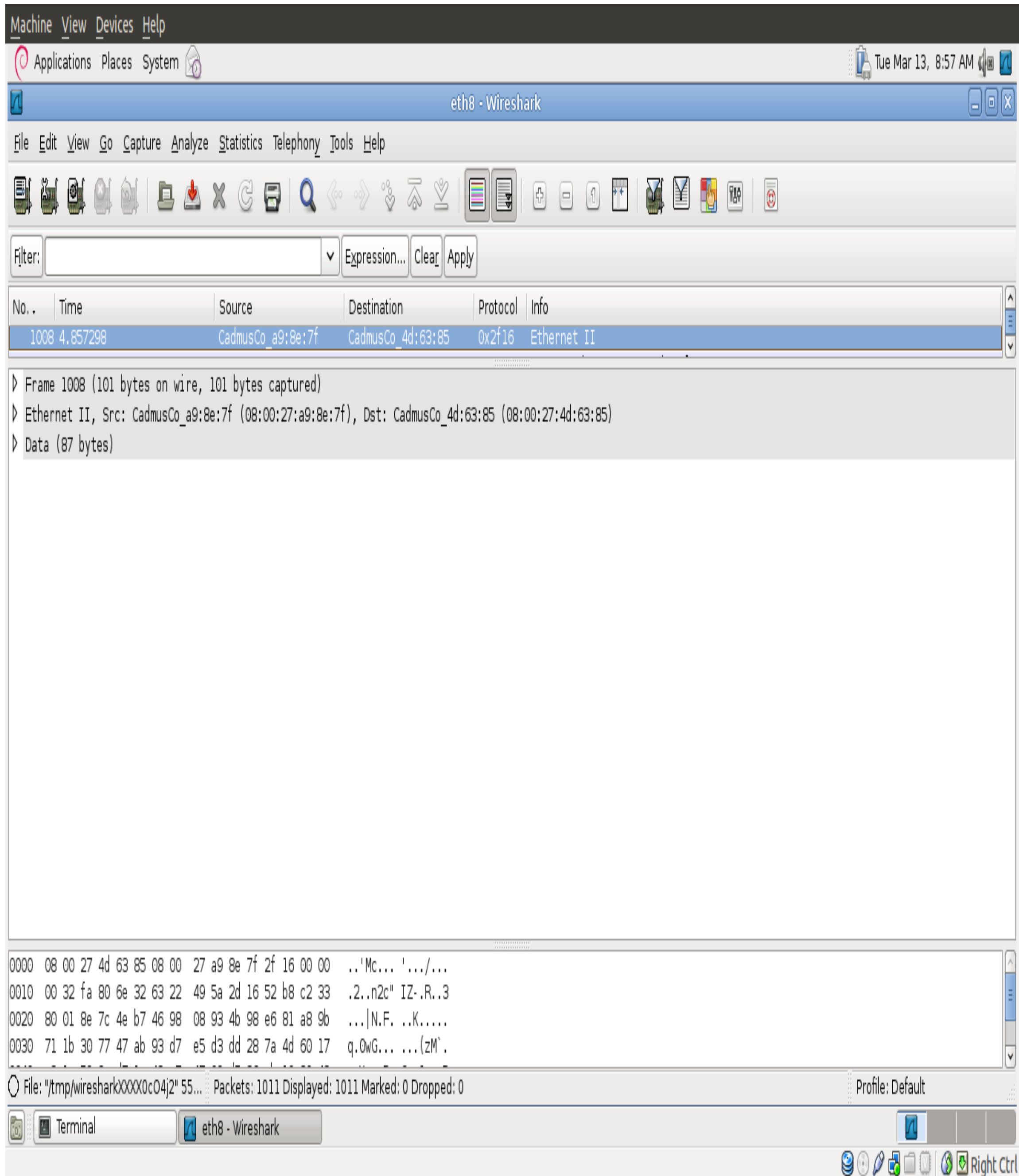

Figure 6.5: Result of IP/UDP Compressed Packet using Wireshark

Figure 6.6 shows the runtime process of ROHC compression using IP/UDP/RTP profile in which the size of packet is same of 110 bytes. It shows that 35 bytes of header is compressed out of 40 bytes (20 bytes of IP, 8 bytes of UDP and 12 bytes of RTP), so now header size in IP/UDP/RTP profile is 5 bytes. In this figure you can see that the packet at the RTP level is compressed. When the IP/UDP/RTP header is compressed the packet length is now compressed to 75 bytes, from which payload size is 70 and header size remains 5 bytes.

```
streamCompressor
streamCompressor | streamDecompr... | send rtp stream | receieve rtp stream | me
[rohc_comp.c:192 rohc_compress()] size of IP packet = 110 bytes
[rohc_comp.c:218 rohc_compress()] IP protocol = 17
[rohc_comp.c:979 c_get_profile_from_packet()] UDP port = 0x4d2 (1234)
[rohc_comp.c:227 rohc_compress()] using profile 'RTP / Compressor' (0x0001)
[rohc_comp.c:1199 c_find_context()] using context CID = 0
[rohc_comp.c:1388 c_piggyback_get()] add 0 byte(s) of feedback data
[rohc_comp.c:293 rohc_compress()] compress the packet #329
[c_generic.c:6524 check_ip_identification()] 1) old_id = 0x0000 new_id = 0x0000
[c_generic.c:6538 check_ip_identification()] 2) old_id = 0x0000 new_id = 0x0000
[c_generic.c:6549 check_ip_identification()] RND detected
[c_generic.c:6560 check_ip_identification()] NBO = 1, RND = 1
[ts_sc_comp.c:89 c_add_ts()] Timestamp = 1735352859
[ts_sc_comp.c:126 c_add_ts()] state SEND_SCALED
[ts_sc_comp.c:128 c_add_ts()] ts_stride calculated = 320
[ts_sc_comp.c:129 c_add_ts()] previous ts_stride = 320
[ts_sc_comp.c:141 c_add_ts()] ts_stride = 320
[ts_sc_comp.c:144 c_add_ts()] ts_offset = 1735352859 modulo 320 = 219
[ts_sc_comp.c:147 c_add_ts()] ts_scaled = (1735352859 - 219) / 320 = 5422977
[ts_sc_comp.c:152 c_add_ts()] TS can be deducted from SN
[c_generic.c:839 c_generic_encode()] SN = 5651
[c_generic.c:6320 changed_dynamic_both_hdr()] check for changed fields in the outer IP header
[c_generic.c:862 c_generic_encode()] send_static = 0, send_dynamic = 0
[c_generic.c:870 c_generic_encode()] ip_id = 0x0000, context_sn = 5651
[c_generic.c:2353 update_variables()] ip_id delta = 0xffffe9ed / 4294961645
[c_generic.c:2354 update_variables()] ip_id bits = 16
[c_generic.c:2364 update_variables()] sn bits=3
[c_generic.c:2435 update_variables()] ts_scaled = 5422977 on 3 bits
[c_generic.c:2727 decide_packet()] SO state
[c_generic.c:2566 decide_SO_packet()] nr_ip_bits=16 nr_sn_bits=3 nr_of_ip_hdr=1 rnd=1
[c_generic.c:3712 code_UO0_packet()] code UO-0 packet (CID = 0)
[cid.c:74 code_cid_values()] CID = 0 => no add-CID
[c_generic.c:3744 code_UO0_packet()] first byte = 0x1f (CRC = 0x7)
[c_generic.c:3633 code_UO_packet_tail()] outer IP-ID = 0x0000
[c_udp.c:431 udp_code_UO_packet_tail()] UDP checksum = 0x64b8
[rohc_comp.c:368 rohc_compress()] ROHC size = 75 (feedback = 0, header = 5, payload = 70), output buffer size = 20
48
just compressed the packet and the return value is: 75
89< The number of bytes written
```

Figure 6.6: Runtime Process of Compressed IP/UDP/RTP Packet

The screenshot in Figure 6.7 shows IP/UDP/RTP compressed packets obtained via wireshark. In this the header size is compressed to 5 bytes when IP and UDP headers are also included. In actual the header size was 20 bytes of IP + 8 bytes of UDP + 12 bytes of RTP. The total size of packet was of 110 bytes and when the IP, UDP and RTP header is compressed, the length of packet is now 75 bytes.

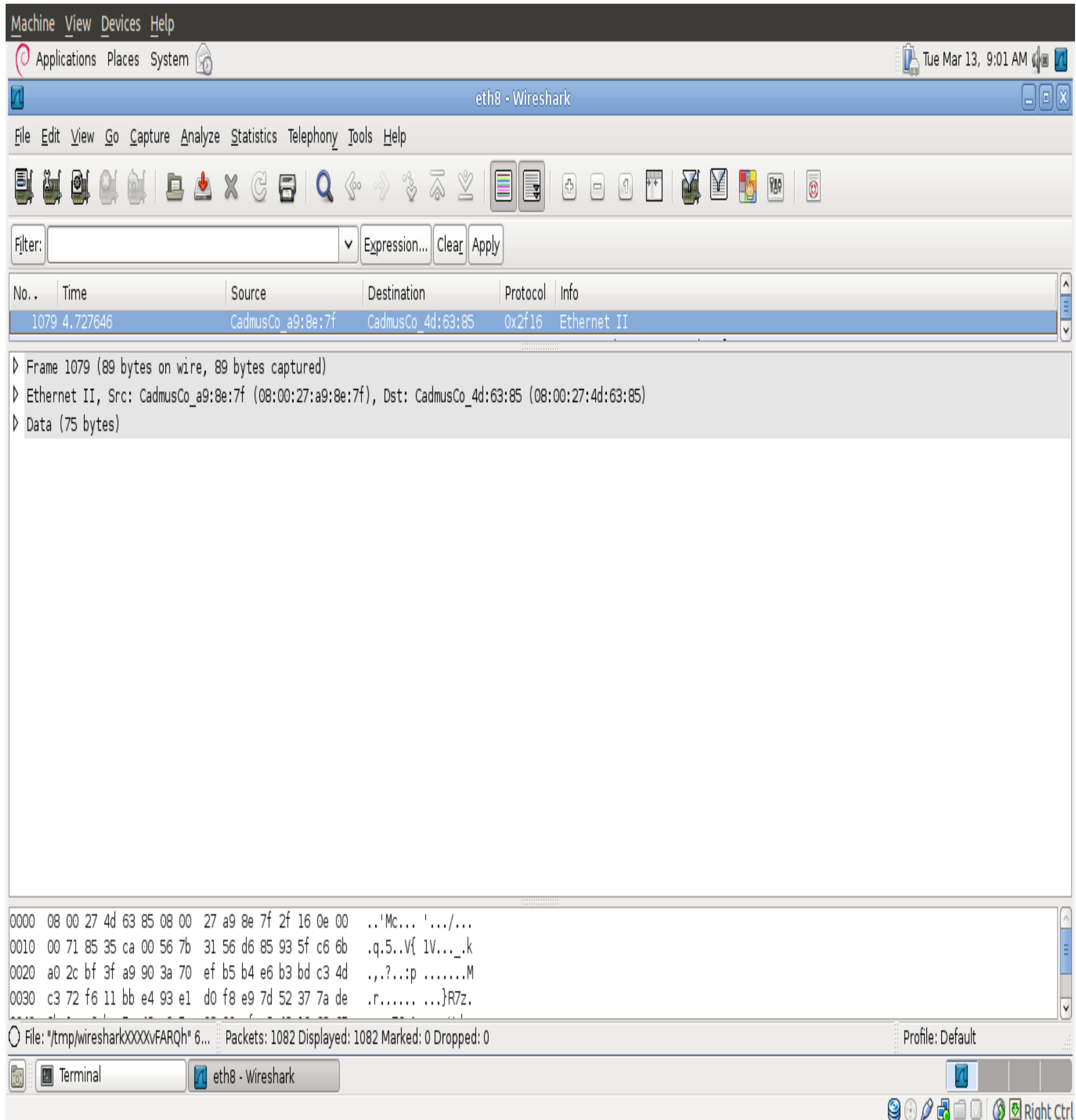

Figure 6.7: Result of IP/UDP/RTP Compressed Packet using Wireshark

6.2 Comparison of Obtained Results

Following shows the obtained results of header compression in various profiles in table 6.1. It can be seen that ROHC header compression ratio increases as the header size increases or we can say that when data is passed from more protocols serially the more headers are compressed. It shows that the total header size is of 40 bytes. When ROHC compression is applied using IP profile, the compressed headers are 17 bytes and uncompressed headers are 23 bytes. When compression is applied using IP/UDP profile, the compressed headers are 23 bytes and uncompressed headers are 17 bytes. When compression is applied using IP/UDP/RTP profile, the compressed headers are 35 bytes and uncompressed headers are only 5 bytes. This shows that 40 bytes headers are now compressed to 5 bytes.

Table 6.1: Header Compression in Various Profiles

Header Compression Profiles	Compressed Headers (bytes)	Uncompressed Headers (bytes)
Uncompressed	40	40
IP	17	23
IP/UDP	23	17
IP/UDP/RTP	35	5

The bar graph in Figure 6.8 shows the percentage reduction in header sizes in a graphical form. The Uncompressed profile is 100% as there is no compression. When ROHC compression is applied on IP profile then headers are compressed to 57.50%. Similarly, when ROHC compression is applied on IP/UDP profile then headers are compressed to 42.50% and when ROHC compression is applied on IP/UDP/RTP profile then headers are compressed 12.50%. This shows the percentage reduction in header size from 100% to 12.50%.

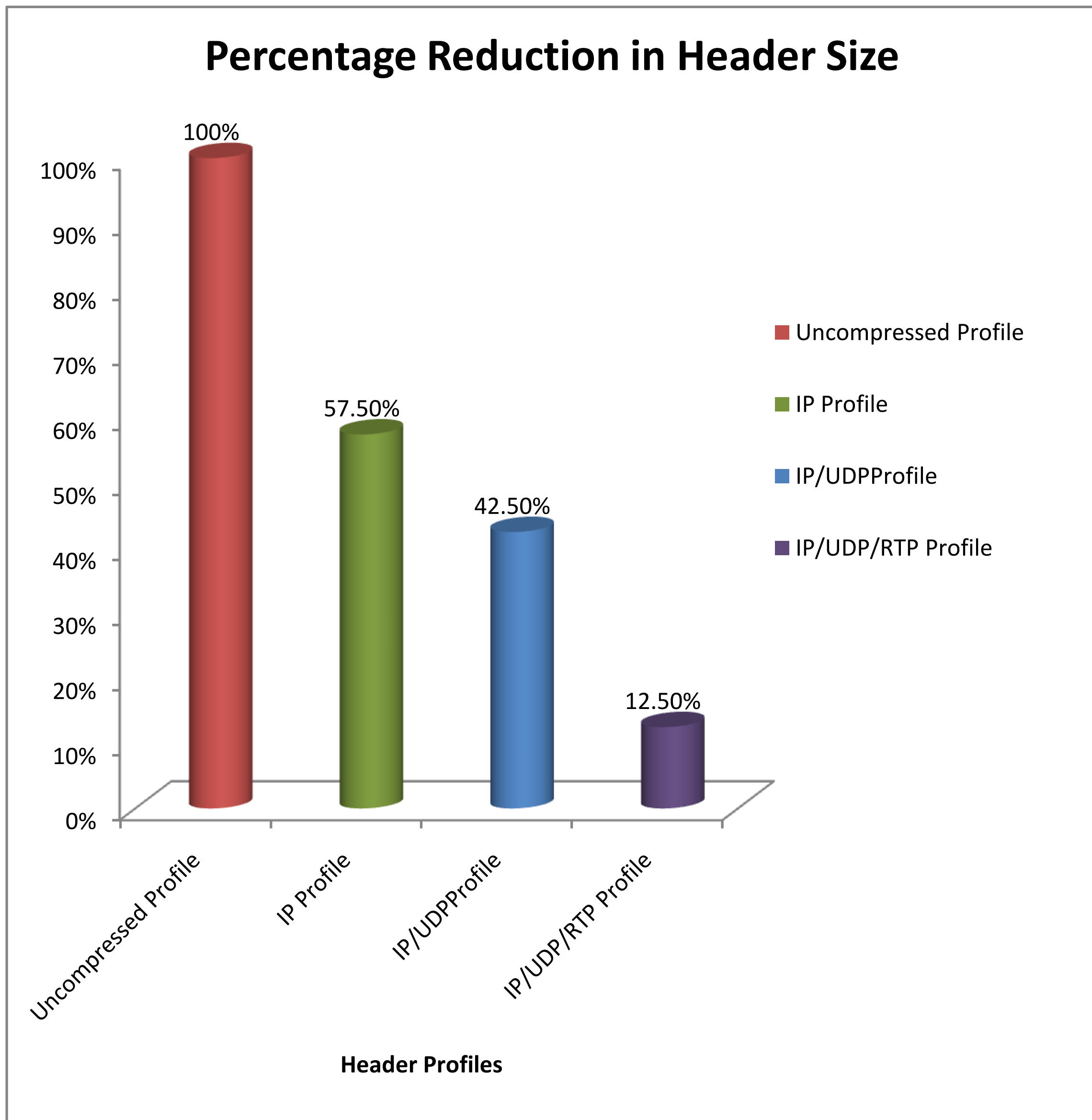

Figure 6.8: Percentage Reduction in Header Size

Figure 6.9 shows the runtime process of ROHC decompression using IP profile. It shows that the payload size is of 90 bytes and the header size is of 3 bytes making a total of 93 bytes. Now the packet of 93 bytes is decompressed using ROHC decompressor. The figure shows uncompressed packet length received after decompression.

```
streamDecompressor
streamCompressor | streamDecompr... | send rtp stream | receieve rtp stream | me
[rohc_decomp.c:432 rohc_decompress()] decompress the packet #320
[rohc_decomp.c:587 d_decode_feedback_first()] skip 0 byte(s) of padding
[rohc_decomp.c:929 rohc_decomp_decode_cid()] no add-CID found, CID defaults to 0
[rohc_decomp.c:762 d_decode_header()] ROHC packet is not an IR packet
[rohc_decomp.c:777 d_decode_header()] context with CID 0 found
[rohc_decomp.c:815 d_decode_header()] the second byte in the packet is at offset 1
[d_generic.c:2524 d_generic_decode()] nbo = 1 rnd = 1
[d_generic.c:2604 d_generic_decode()] decode the packet (type 0)
[d_generic.c:2687 decode_uo0()] first byte = 0x00 (real CRC = 0x0, SN = 0x0)
[d_generic.c:3517 do_decode_uo0_and_uo1()] SN = 320
[d_generic.c:3547 do_decode_uo0_and_uo1()] outer IP-ID = 0x0000 (rnd = 1, ID bits = 0)
[d_generic.c:5543 build_uncompressed_ip4()] IP-ID = 0x0000
[d_generic.c:5547 build_uncompressed_ip4()] IHL = 0x5
[d_generic.c:5552 build_uncompressed_ip4()] Total Length = 0x006e (IHL * 4 + 90)
[d_generic.c:5555 build_uncompressed_ip4()] IP checksum = 0x4929
[d_generic.c:3668 do_decode_uo0_and_uo1()] size = 20 => CRC = 0x0
[d_generic.c:5915 update_inter_packet()] current time = 3465888041 and last time = 3465868021
[d_generic.c:5929 update_inter_packet()] inter_arrival_time = 19550 and current arrival delta is = 20020
[d_generic.c:2785 decode_uo0()] ROHC payload (length = 90 bytes) starts at offset 3
[d_generic.c:2617 d_generic_decode()] uncompressed packet length = 110 bytes
[rohc_decomp.c:444 rohc_decompress()] state in decompressor = 3
[rohc_decomp.c:517 rohc_decompress()] feedback curr -12
[rohc_decomp.c:524 rohc_decompress()] feedback curr 0
124< The number of bytes written
```

Figure 6.9: Runtime Process of Decompressed IP Packet

Figure 6.10 shows the runtime process of ROHC decompression using IP/UDP profile. It shows that the payload size is of 82 bytes and the header size is of 5 bytes making a total of 87 bytes. Now the 87 bytes packet is decompressed using ROHC decompressor. The figure shows uncompressed packet length received after decompression.

```
streamDecompressor
streamCompressor | streamDecompr... | send rtp stream | receieve rtp stream | me
[rohc_decomp.c:432 rohc_decompress()] decompress the packet #222
[rohc_decomp.c:587 d_decode_feedback_first()] skip 0 byte(s) of padding
[rohc_decomp.c:929 rohc_decomp_decode_cid()] no add-CID found, CID defaults to 0
[rohc_decomp.c:762 d_decode_header()] ROHC packet is not an IR packet
[rohc_decomp.c:777 d_decode_header()] context with CID 0 found
[rohc_decomp.c:815 d_decode_header()] the second byte in the packet is at offset 1
[d_generic.c:2524 d_generic_decode()] nbo = 1 rnd = 1
[d_generic.c:2604 d_generic_decode()] decode the packet (type 0)
[d_generic.c:2687 decode_uo0()] first byte = 0x73 (real CRC = 0x3, SN = 0xe)
[d_generic.c:3517 do_decode_uo0_and_uo1()] SN = 222
[d_generic.c:3547 do_decode_uo0_and_uo1()] outer IP-ID = 0x0000 (rnd = 1, ID bits = 0)
[d_udp.c:507 udp_decode_uo_tail_udp()] UDP checksum = 0xae58
[d_generic.c:5543 build_uncompressed_ip4()] IP-ID = 0x0000
[d_generic.c:5547 build_uncompressed_ip4()] IHL = 0x5
[d_generic.c:5552 build_uncompressed_ip4()] Total Length = 0x006e (IHL * 4 + 90)
[d_generic.c:5555 build_uncompressed_ip4()] IP checksum = 0x4929
[d_udp.c:562 udp_build_uncompressed_udp()] UDP checksum = 0xae58
[d_udp.c:566 udp_build_uncompressed_udp()] UDP length = 0x005a
[d_generic.c:3668 do_decode_uo0_and_uo1()] size = 28 => CRC = 0x3
[d_generic.c:5915 update_inter_packet()] current time = 3726115040 and last time = 3726091184
[d_generic.c:5929 update_inter_packet()] inter_arrival_time = 20314 and current arrival delta is = 23856
[d_generic.c:2785 decode_uo0()] ROHC payload (length = 82 bytes) starts at offset 5
[d_generic.c:2617 d_generic_decode()] uncompressed packet length = 110 bytes
[rohc_decomp.c:444 rohc_decompress()] state in decompressor = 3
[rohc_decomp.c:517 rohc_decompress()] feedback curr -12
[rohc_decomp.c:524 rohc_decompress()] feedback curr 0
124< The number of bytes written
^C
```

Figure 6.10: Runtime Process of Decompressed IP/UDP Packet

Figure 6.11 shows the runtime process of ROHC decompression using IPUDP/RTP profile. It shows that the payload size is of 70 bytes and the header size is of 5 bytes making a total of 75 bytes. Now the 75 bytes packet is decompressed using ROHC decompressor. The uncompressed packet length received after decompression is of 110 bytes. The Ethernet header of 14 bytes is added to it making a total of 124 bytes which is received.

```
streamDecompressor
streamCompressor | streamDecompr... | send rtp stream | receieve rtp stream | me
we have a detected a rohc packet
[rohc_decomp.c:432 rohc_decompress()] decompress the packet #147
[rohc_decomp.c:587 d_decode_feedback_first()] skip 0 byte(s) of padding
[rohc_decomp.c:929 rohc_decomp_decode_cid()] no add-CID found, CID defaults to 0
[rohc_decomp.c:762 d_decode_header()] ROHC packet is not an IR packet
[rohc_decomp.c:777 d_decode_header()] context with CID 0 found
[rohc_decomp.c:815 d_decode_header()] the second byte in the packet is at offset 1
[d_generic.c:2524 d_generic_decode()] nbo = 1 rnd = 1
[d_generic.c:2604 d_generic_decode()] decode the packet (type 0)
[d_generic.c:2687 decode_uo0()] first byte = 0x6f (real CRC = 0x7, SN = 0xd)
[d_generic.c:3517 do_decode_uo0_and_uo1()] SN = 5469
[d_generic.c:3547 do_decode_uo0_and_uo1()] outer IP-ID = 0x0000 (rnd = 1, ID bits = 0)
[ts_sc_decomp.c:141 ts_deducted()] old ts_scaled = 5422794
[ts_sc_decomp.c:143 ts_deducted()] sn = 5469, ts_sc->sn = 5468
[ts_sc_decomp.c:144 ts_deducted()] new ts_scaled = 5422795
[d_generic.c:3563 do_decode_uo0_and_uo1()] ts deducted = 1735294619
[d_rtp.c:782 rtp_decode_uo_tail_rtp()] UDP checksum = 0x2c90
[d_generic.c:5543 build_uncompressed_ip4()] IP-ID = 0x0000
[d_generic.c:5547 build_uncompressed_ip4()] IHL = 0x5
[d_generic.c:5552 build_uncompressed_ip4()] Total Length = 0x006e (IHL * 4 + 90)
[d_generic.c:5555 build_uncompressed_ip4()] IP checksum = 0x4929
[d_rtp.c:837 rtp_build_uncompressed_rtp()] UDP checksum = 0x2c90
[d_rtp.c:842 rtp_build_uncompressed_rtp()] UDP + RTP length = 0x005a
[d_generic.c:3668 do_decode_uo0_and_uo1()] size = 40 => CRC = 0x7
[d_generic.c:5915 update_inter_packet()] current time = 4143851911 and last time = 4143831503
[d_generic.c:5929 update_inter_packet()] inter_arrival_time = 18746 and current arrival delta is = 20408
[ts_sc_decomp.c:57 d_add_ts()] new TS = 1735294619 - old TS = 1735294619
[ts_sc_decomp.c:84 update_ts_sc()] timestamp = 1735294619
[ts_sc_decomp.c:85 update_ts_sc()] ts_stride = 320
[ts_sc_decomp.c:89 update_ts_sc()] ts_offset = 1735294619 modulo 320 = 219
[ts_sc_decomp.c:93 update_ts_sc()] ts_scaled = (1735294619 - 219) / 320 = 5422795
[d_generic.c:2785 decode_uo0()] ROHC payload (length = 70 bytes) starts at offset 5
[d_generic.c:2617 d_generic_decode()] uncompressed packet length = 110 bytes
[rohc_decomp.c:444 rohc_decompress()] state in decompressor = 3
[rohc_decomp.c:517 rohc_decompress()] feedback curr -12
[rohc_decomp.c:524 rohc_decompress()] feedback curr 0
124< The number of bytes written
```

Figure 6.11: Runtime Process of Decompressed IP/UDP/RTP Packet

This result shows that the packets are decompressed using various profiles and total packets size of 75 bytes which was compressed is now decompressed and total of 110 bytes with 14 bytes of Ethernet headers are achieved and the voice of mp3 which was taken as a VOIP application has now gain its original position.

Chapter 7

CONCLUSIONS

7.1 Concluding Remarks

The low bandwidth cellular link when used for VoIP suffers from a problem of large header overhead. High header overhead IP telephony speech data will be transferred encapsulated in RTP/UDP/IP. A voice packet along with link framing layer will be attached with IPv4, UDP and RTP headers of 40 bytes. Payload size depends on frame sizes and speech coding being used and it can be as low as 15-20 bytes for certain audio codecs. Hence for VoIP header, overhead can be as high as 60% to 80%. To improve this problem the IETF has standardized header compression techniques under the umbrella of ROHC. Project comprises of Robust Header Compression integration with Linux Kernel using Netlink Socket. Robust Header Compression Library is built to compress the headers of internet packets using official Internet standards specified in various relevant RFCs. A VoIP application is tested over a low bandwidth link and the utility of the Robust Header Compression is demonstrated.

The results we have been able to achieve are that the VOIP packet header is

compressed from 40 to 5 bytes as shown in figure 7.1. This implies that the reduction of each VOIP packet is from 110 bytes to 75 bytes.

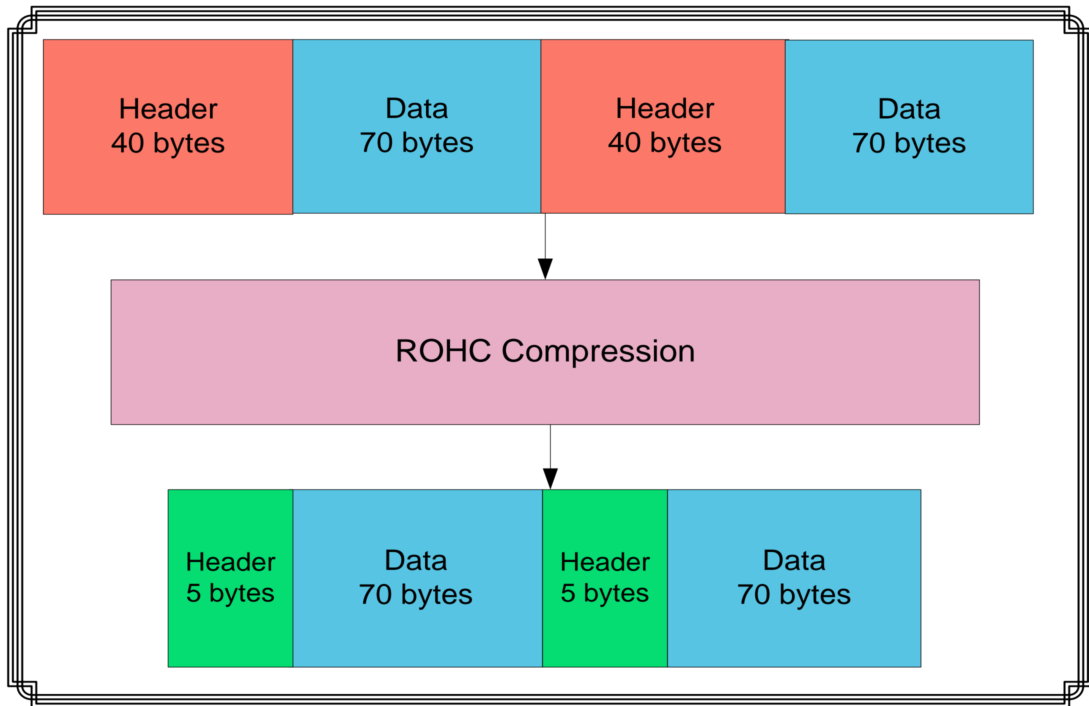

Figure 7.1: ROHC Compression of VOIP Packet

The performance of the ROHC based communication link is good. A lot of bandwidth can be saved to transmit data. The ROHC implementation is found to be most efficient when the data is encapsulated in large number of protocols, as each protocol will have a header itself. From this work we have been able to get a lot of knowledge about Linux and programming with it. This work builds a strong capability of socket programming and header compression techniques in our skill set.

7.2 Future Work

Based on the experience and progress we have made during this work, we can recommend following possibilities for future work:

- Development of a user interface for configuring and managing the ROHC capabilities by an end user.
- Porting the software we have developed to android and other PDA systems for use on wireless links.
- Enhancements to ROHC library to incorporate newer compression profiles being standardized by IETF such as TCP/IP.

REFERENCES

[1] ZENG Ye,SHEN Yi,HUANGs Jun-qiao(Department of Control Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074,China); VoIP Gateway Design based on Embedded Linux System[J];Journal of Chongqing Institute of Technology;2006-05

[2] Hong S. Jung and P. Park. Effect of Robust Header Compression (ROHC) and Packet Aggregation on Multi-hop Wireless Mesh Networks. In Proc. IEEE CIT'06, Seoul, Korea, September 2006.

[3] C. Bormann, C. Burmeister, M. Degermark, H. Fukushima, H. Hannu, L.-E.Jonsson, R. Hakenberg, T. Koren, K. Le, Z. Liu, A. Martensson, A. Miyazaki, K. Svanbro, T. Wiebke, T. Yoshimura, and H. Zheng, "RObust Header Compression (ROHC): Framework and Four Profiles: RTP, UDP, ESP, and Uncompressed," IETF RFC 3095, July 2001

[4] S. Rein, F.H.P. Fitzek, and M. Reisslein. Voice Quality Evaluation for Wireless Transmission with ROHC. In Proc. IMSA, pages 461–466, Honolulu, USA, August 2003.

[5] R. Pries, A. Mader, and D. Staehle. Performance of header compression for VoIP in wireless LANs. Technical report, University of Wurzburg, Institute of Computer Science, April 2007

[6] Casner,S. and Jacobson, V., "Compressing IP/UDP/RTP Headers for Low Speed Serial Links", February 1999.

[7] B.Wang Q.Xiong W. Res," The comparison of communication methods between user and Kernel space in embedded Linux", IEEE Conf, ICCP 2010.

[8] D. Taylor, A.Herkersdorf, A. D¨oring, G.Dittmann," Header Compression (ROHC) in Next-Generation Network Processors", Applied Research Laboratory, Department of Computer Science and Engineering ,Washington University in Saint Louis, 2002

[9] Bormann, C., Editor, "Robust Header Compression (ROHC)", RFC 3095, June 2001.

[10] Love R. Linux Kernel Development, 2nd edition, Novell Press, 2003.

[11] Torvalds L. et al. The Linux kernel. Web pages at: http://www.kernel.org [10 December 2010].

[12] Kaichuan He K. Why and How to Use Netlink Socket. Linux Journal, 2005. Web pages at: http://www.linuxjournal.com/article/7356 [10 December 2010].

[13] Dhandapani G., Sundaresan A. Netlink sockets: An overview Web pages at: http://qos.ittc.ku.edu/netlink/html/ [6 December 2009]

[14] Linux man-pages project. Netlink - Communication between kernel and userspace. Web pages at: http://www.kernel.org/doc/man-pages/online/pages/man7/netlink.7.html [6 December 2009].

[15] Salim J., Khosravi H., Kleen A., Kuznetsov A. RFC 3549 - Linux Netlink as an ip services protocol. Webpages at: http://www.faqs.org/rfcs/rfc3549.html [6 December 2009].

[16] Salim J., Haas R., Blake S. Netlink2 as ForCES Protocol (Internet-Draft), 2004. Web pages at: http://tools.ietf.org/html/draft-jhsrha-forces-netlink2-02 [6 December 2009].

[17] Robust Header Compression (ROHC) library at https://launchpad.net/rohc/ [12 September 2010].

[18] Concept of Robust Header Compression (ROHC) at http://www.effnet.com/sites/effnet/pdf/uk/Whitepaper_Robust_Header_Compression.pdf [12 September 2010].

[19] Effnet ROHC (Robust Header Compression) Performance on Intel® Core Microarchitecture - Based Processors. Webpages at www.effnet.com/19350_EFFNET_Final.pdf [12 September 2010].

[20] Kuznetsov A. iproute: advanced routing tools for Linux. Web pages at: http://linux-foundation.org [6 December 2009].

[21] Linux Forum at http://www.linuxforums.org/forum/kernel/167412-how-send-data-kernel-module-user application-netlink-sockets.html [6 September 2009].

[22] Micheal Opdenacker, GNU/Linux and Free Software at http://free-electrons.com [12 September 2010].

[23] Kernel Space - User Space Interfaces. Webpages at http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html. [6 December 2009].

APPENDIX

Nfqueue

```
#ifndef _nfqueue_header_
#define _nfqueue_header_
        class NFQueue
        {
                public:
                        NFQueue(int dataPacketSize_, int queueNumber_);
                        ~NFQueue();
                        /*
                        The return values can be one of the following:
                        0: Queue successfully initialized
                        1: error during nfq_open()
                        2: error during nfq_bind_pf()
                        3: error during nfq_create_queue()
                        4: can't set packet_copy mode
                        */
                        int initializeQueue(int (*callbackFunction)(/*void* pt2Object,*/
                                        struct nfq_q_handle *qh, struct nfgenmsg
*nfmsg,
                                         struct nfq_data *nfa, void *data));
                        //5: socket generated an error
                        int  readQueue();
                        int getQueueNumber();
                        void destroyQueue();
                private:
                        struct nfq_handle *h;
                        struct nfq_q_handle *qh;
                        struct nfnl_handle *nh;
```

```
int fd;
int rv;
char *buf;
int dataPacketSize;
int queueNumber;
};
#endif
```

ROHC Compdaemon

```
#ifndef _rohc_compd_header_
#define _rohc_compd_header_
class ROHCCompDaemon: public ROHCDaemon
{
public:
ROHCCompDaemon(int dataPacketSize, int queueNumber_);
static int callbackFunctionRTPImplementation(struct
nfq_q_handle *qh,
struct nfgenmsg *nfmsg,struct nfq_data *nfa, void *data);
static uint32_t processRTPPacket (struct nfq_data *tb);
};
#endif
```

ROHC Daemon

```
#ifndef _rohcd_header_
#define _rohcd_header_
class ROHCDaemon: public NFQueue
{
public:
ROHCDaemon(int dataPacketSize, int queueNumber_);
```

```
        static ROHCEngine* getROHCEngine();
    };
#endif
```

ROHC Engine

```
#ifndef _ROHCEngine_header_
#define _ROHCEngine_header_
using namespace std;
/* includes for using ROHC library */
extern "C" {
}
  struct flowTableEntry
  {
    uint32_t flowID;
    uint32_t ttl;
    string   src_ip;
    string   dst_ip;
    char *src_mac_address;
    char *dst_mac_address;
  };
    class ROHCEngine
    {
        public:
      ROHCEngine();\
        //General Utility Functions
        bool fileExists(const char *filename);
        void printFlowTable();
        void updateTimeToLive();
      unsigned char atoh (unsigned char data);
        //Flow Specific Utility Functions
      uint32_t static calculateFlowID(uint32_t,uint16_t);
```

```
            bool flowExists(uint32_t flowID);
            void updateFlow(uint32_t flowID);
            bool noActiveFlows();
            //Flow Management Functions
        char* getFlowSrcMacAddress(uint32_t flowID);
        char* getFlowDstMacAddress(uint32_t flowID);
            uint32_t getFlowIDAtIndexNum(uint32_t index);
            flowTableEntry * getFlowTableRowPointer(uint32_t flowID);
            int * getFlowMutex(uint32_t flowID);

            //0: Flow added successfully
            //1: Flow could not be added
        int addFlow(uint32_t flowID , string src_ip, string dst_ip, char*
src_mac_address);
        //ROHC functions
        void initializeROHCLibrary();
        void setCompressionProfile(int profileID);
        /* Profile IDs
         * 1: IP
         * 2: IP/UDP
         * 3: IP/UDP/RTP
         * 4: Uncompressed
         */
        int compressPacket(char* uncompressedPacket, char* rohcPacket,
                    int uncompressedPacketLength);
        //Packet processing functions
        unsigned char* createEthernetHeader(char *src_mac, char *dst_mac,
int protocol);
        int sendPacketOnRawSocket(unsigned char* packet, int packetSize);
    private:
```

```
        /* the ROHC compressor */
        struct rohc_comp *compressor;
    //A flowTableEntry is a structure used to manage the states of a flow.
            flowTableEntry * flowTable;
    //The rowLockArray is a set of mutexes to manage thread-safety among
flows.
            int *rowLockArray;
    };
#endif
```

Stream Compressor

```
#ifndef _rohc_header_
#define _rohc_header_
    class StreamCompressor : public PThread
    {
            public:
                    StreamCompressor();
                    ~StreamCompressor();
                    static ROHCEngine* getROHCEngine();
                    friend void *runROHCCompDaemonStream(void *t);
//                  friend void *runSIPDaemonForSBCStream(void *t);
                    friend void *runROHCEngine(void *t);
                    static ROHCEngine * rohc_engine;
                    static ROHCCompDaemon *
                    staticROHCCompressorDaemonHandle;
//                  static SIPDaemonForPhone *
staticSIPDaemonHandleForPhone;
            private:
//                  bool keepSIPDaemonForPhoneAlive;
                    bool keepROHCCompDaemonAlive;
```

```
                bool keepROHCEngineDaemonAlive;
    };
#endif
```

Stream Decompressor

```
#ifndef _rohc_header_2
#define _rohc_header_2
   class StreamDecompressor : public PThread
    {       public:
                StreamDecompressor();
               ~StreamDecompressor();
                static ROHCEngine* getROHCEngine();
                friend void *runROHCDecompDaemonStream(void *t);
                friend void *runROHCEngine(void *t);
                static ROHCEngine * rohc_engine;
                static ROHCDecompDaemon
                *staticROHCDecompressorDaemonHandle;
            private:
                 bool keepROHCDecompDaemonAlive;
                bool keepROHCEngineDaemonAlive;
    };
#endif
```

ROHC Decompressor Daemon

```
#ifndef _rohc_decompd_header_
#define _rohc_decompd_header_
class ROHCDecompDaemon
{
  public:
    static void processROHCStream();
    static ROHCEngine* getROHCEngine();
}; #endif
```