

Diploma Thesis

ROHC Implementation

Students:
Dino Mani, Markus Gaugler, Christoph Frehner

Industrial Partner:
Motorola USA

Supervisor:
Prof. Dr. Guido M. Schuster

December 2005
University of Applied Sciences Rapperswil

Abstract

Thanks the technical progress, during the last years the IP telephony became more and more common. In the future, also cellular telephony links will be based on IP technology.

In realtime communication over the Real Time Transport Protocol the header of this packets uses a lot of capacity compared to the real information content. Hence, header compression is very desirable in communication channels, where bandwidth is limited and valuably, such as wireless links. Consider that, there will exist different compression techniques which try to reduce the header overhead. One of this is the ROHC¹ scheme. The advantage of this method compared to the others is its robustness against packet loss and bit errors. This characteristic is very important for links with significant bit error rates and long round-trip times, such as wireless links are.

This diploma thesis implements a part of the standard track about the ROHC described in the RFC 3095[4]. The source code is written in C language on a Linux distribution. The implementation includes a framework with a compressor and decompressor part. The advantage of this compression method and its robustness should be shown with an appropriate test bench.

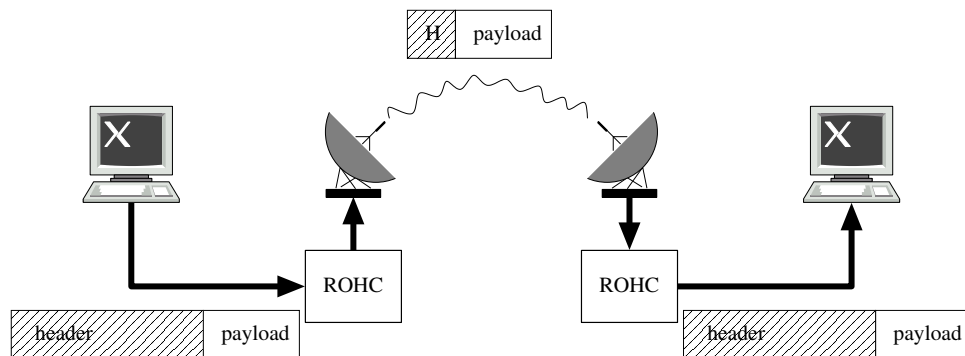


Figure 1: network with ROHC

¹RObust Header Compression

Contents

1	Introduction	1
2	Conceptual formulation	3
2.1	Kurzbeschreibung	3
2.2	Aufgabenstellung	3
2.3	Erwartete Ergebnisse	3
3	Project schedule	5
4	Theory of ROHC	7
4.1	Header fields	7
4.1.1	Ethernet header	7
4.1.2	IP header	8
4.1.3	UDP header	9
4.1.4	RTP header	9
4.2	Modes	10
4.2.1	Unidirectional mode, U-mode	10
4.2.2	Bidirectional optimistic mode, O-mode	11
4.2.3	Bidirectional reliable mode, R-mode	11
4.3	Compressor	11
4.3.1	States	11
4.4	Decompressor	12
4.4.1	States	12
4.5	Encoding methods	13
4.5.1	Least Significant Bits (LSB) encoding	13
4.5.2	Window-based LSB (W-LSB)encoding	14
4.5.3	Timer-based compression	15
4.5.4	Scaled RTP Timestamp encoding	15
4.5.5	Offset IP identifier encoding	15
4.6	CRC calculation	16
4.6.1	CRC's for IR and IR-DYN packets	16
4.6.2	CRC's in compressed headers	16
4.7	Packettypes	16

4.7.1	Context identifier byte	17
4.7.2	IR packet	18
4.7.3	IR-DYN packet	19
4.7.4	UOR-2-TS packet	19
4.7.5	UO-1-TS packet	20
4.7.6	UO-1-ID packet	20
4.7.7	UO-0 packet	21
4.7.8	FEEDBACK-1 packet	21
5	Existing ROHC-Project	22
6	Implementation	24
6.1	Compressor	24
6.1.1	Compressor framework	24
6.2	Decompressor	30
6.2.1	Decompressor framework	30
6.2.2	Handling of IR packets	30
6.2.3	Handling of IR-DYN packets	31
6.2.4	Handling of UOR-2-TS packets	31
6.2.5	Handling of UO-1-TS and -ID packets	32
6.2.6	Handling of UO-0 packets	32
6.2.7	Additional informations and problems	33
6.3	Graphical User Interface	33
6.3.1	Building GUI	33
6.3.2	The two setup windows	34
6.3.3	Statistic tool	38
6.4	Packet Stream	39
6.4.1	PCAP Library	40
6.4.2	LibNet	42
7	Test scenario	44
7.1	Scenario design	44
7.2	Functionality	45
7.3	Results	45
7.4	Configuration	45
8	Conclusion	47
8.1	Results	47
8.2	Future Challenges	47
8.3	Difficulties	47
8.4	Closing words	48
9	Glossary	49
	Bibliography	50

Chapter 1

Introduction

Motorola¹ is a Fortune 100 global communications leader that provides seamless mobility products and solutions across broadband, embedded systems and wireless networks.

Mobile Devices in the fourth generation will be IP-phones. This will have a lot of advantages and changes.

One of these changes is, that you have to pay for data quantity and not duration, as we know it today for calls. The end customer is only interested in this new technology, if it is cheaper, of course. With this packet-base method you can save bandwidth, which is the most valuable thing in wireless network environments. In common IP/UDP/RTP packets, the header information which is attached to each packet is significantly higher than the transmitted payload. In fact, the customer will pay a lot for information you don't want to send, and the network providers waste a lot of bandwidth.

For instance a IPv4 packet with UDP and RTP protocol contains at least 40 octets header informations. But the payload takes normally only 20 octets. So the packet header could introduce an overhead of more than 50% of the whole packet size. This is not in terms of the customer, who is interested to put money aside by using IP telephony over wireless links.

With ROHC it is possible to reduce this header overhead. The knowledge that there are a lot of redundant informations in this headers, makes it obvious, that you don't have to send all informations with every packet. It's only necessary to send the changed header fields. The other field parameters are either always known or they could be deduced from a value that is sent by a compression specific packet type. To do this is not a new idea. But the advantage from ROHC to other methods is the robustness. That means, when a packet is corrupted with bit errors, ROHC is able to correct this errors and the packet could be forwarded on a wired link anyway.

As we know, common wireless links possess a significant bit error rate and long round trip times. So it's necessary to be impassible against lost and

¹<http://www.motorola.com>

damaged packets. The ROHC packets are also less vulnerable against bit errors, because they are smaller. If the compressor works in the highest compression state, only 2 header octets are required to send over the link. So it could be saved more than 95% of the uncompressed packet header.

In the content of this work a basic implementation of this standard for RTP has been realised. Therefore the theoretical background of the Realtime Transport Protocol has to be acquired. A feasible possibility is to expand the existing ROHC open source implementation from sourceforge.net² with the RTP profile. This profile is not included yet at the open source project. To do this it's indispensable to achieve incorporation into the developing environment and network programming with Linux.

As final task, test results has to be recorded and analysed. To do this it's essential to find a useful network emulator, analyse the data stream and compare the performance of the compressed and the uncompressed packet stream at the receiver. To simulate a real wireless transmission network, there could be used a small wireless network or an network emulator like NISTnet, see diploma thesis [5]. Further it will be nice if the simulation could be realised in realtime on a sightly user interface.

²<http://rohc.sourceforge.net>

Chapter 2

Conceptual formulation

2.1 Kurzbeschreibung

Eines der Probleme von Voice over IP ist der grosse Overhead von 40 Bytes welcher die IP/UDP/RTP Struktur mit sich bringt. Dies ist vor allem ein Problem in der nächsten Generation des Mobilfunks, welcher aus jedem Handy ein IP Host machen wird. Da die Bits über eine Luftschnittstelle immer viel teurer sind als über ein Glasfaserkabel, ist es nicht zulässig einen Overhead von über 50% zu haben. Die IETF hat eine Lösung zu diesem Problem erarbeitet welche in RFC 3095 dokumentiert ist. Auch gibt es schon Linux Software welche einen Teil des RFC's implementiert. Ziel dieser Diplomarbeit ist es die Implementation auf <http://rohc.sourceforge.net/> so zu erweitern, dass es möglich ist, RTP/UDP/IP Pakete zu komprimieren.

2.2 Aufgabenstellung

- Einarbeitung in die Theorie und die Linux Entwicklungsumgebung.
- Entwicklung des RTP/UDP/IP Kompressionsteils
- Austesten der Implementierung mit verschiedenen Netzwerkbedingungen

2.3 Erwartete Ergebnisse

- Dokumentation der Theorie, der Entwicklung, und der Tests
- Funktionsfähige Implementation integriert in das <http://rohc.sourceforge.net/> Projekt
- Sie führen ein persönliches Laborbuch, wo Sie aufschreiben wann Sie was für wie lange machen und was die Ergebnisse sind.

- Sie schicken mir vor jeder Sitzung eine Zusammenfassung welche dokumentiert, was Sie in der letzten Woche gemacht haben.

Chapter 3

Project schedule

Project: UIN RWMC

Progression
Project RWMC Implementation

Task	November							December								
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Preparation																
- Studying RFC																
- Installation O.S & IDE																
- Installation RWMC																
Programming																
- Self analysis																
- Create Spec Files																
- Connect to																
- Decompose																
- Basic analysis																
- Test Koneksi																
- Network Emulator																
- Capture & analyze packets																
- Filtering																
- Logging & Correlating																
Documentation																

Chapter 4

Theory of ROHC

In this section the fundamental knowledge about the ROHC standard is introduced, which is needed to understand the followed chapters. More details about the parts that occur in this implementation will be found in the specific chapter. All other informations can be found in the RFC 3095 [4].

4.1 Header fields

The basic principle of header compression is, that a lot of header fields do not change their value or change only seldom during a communication session. Therefore it is useful to separate the header fields in static and dynamic fields.

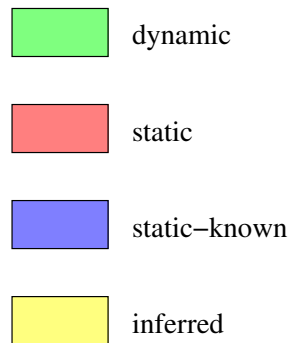


Figure 4.1: Legend for header fields

4.1.1 Ethernet header

The full content of the ethernet header is static. But it can not be leaved out because the packet has to be sent over an ethernet connection. Later in a real environment it isn't need to send this informations.

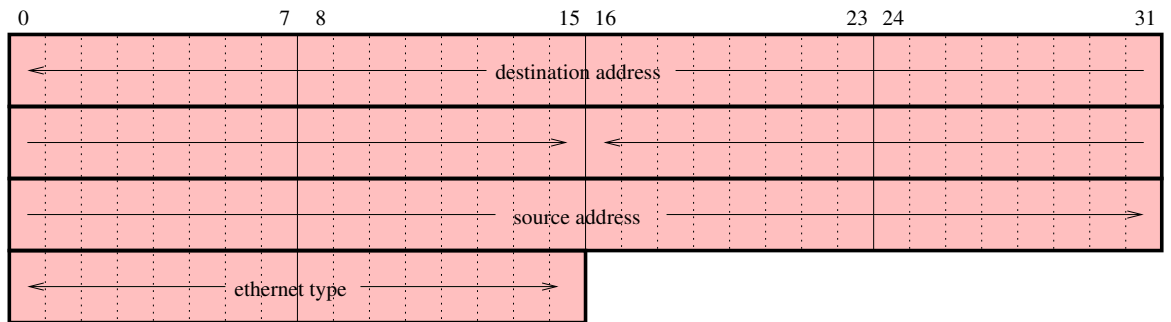


Figure 4.2: Ethernet header

4.1.2 IP header

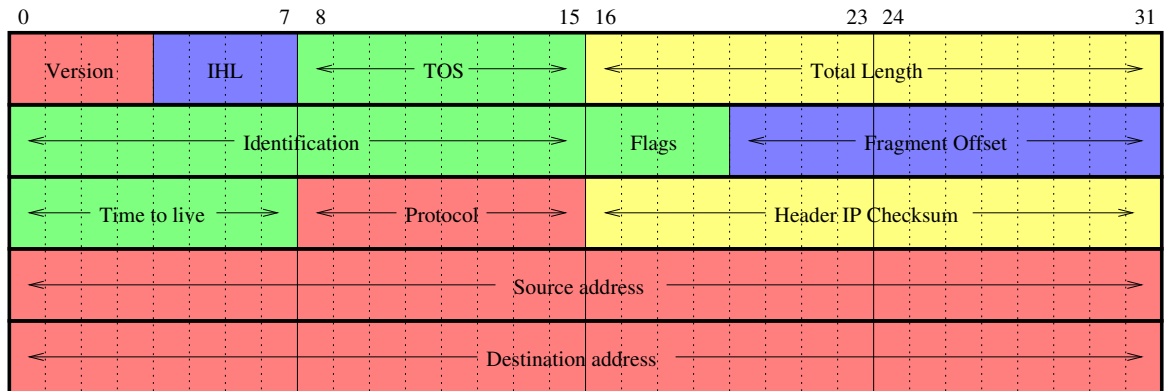


Figure 4.3: IP header

- The version field could only be four or six. In this implementation only version four is allowed.
- The length of the IP header must be 20 bytes. Additional octets are possible in the IP standard.
- The ToS is expected to change. But this will occur only seldom. It is updated with a dynamic context update.
- The total length of the IP packet could be recalculated by the decompressor, so this field will never be transmitted.
- The value of the IP identifier is changing each packet. Normally the value change with the same delta as the sequence number, but exceptions are also possible, perhaps for video transmissions. Its value is updated with every received packet.

- The flags could change, but not often. Update this will be done with dynamic context update.
- The fragment offset part is only used, when multiple packets belong to each other. This is never used in RTP stream.
- Time to live values are normally the same for all packets of a stream. But of course that the packet could pass a different number of routers before it comes to the ROHC link, it is possible that this value will change. It is updated with every dynamic context update.
- Protocol is always the same for one stream. So it is updated only with the static context update.
- Header IP checksum must be recalculated by the decompressor. This is never transmitted.
- The addresses of the destination and source are static of course. For all informations about the ip header fields, look at the RFC which describes the internet protocol[2].

4.1.3 UDP header

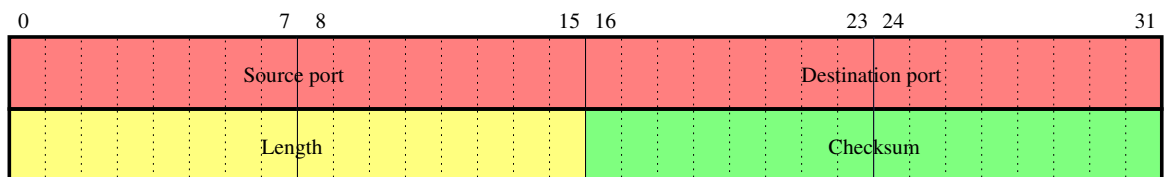


Figure 4.4: UDP header

- Both portes are static, because they are part of the context definition of a packet stream.
- The length in the UDP header is recalculated by the decompressor itself. It is never transmitted.
- The UDP checksum is part of the dynamic context, it's also updated with it.

4.1.4 RTP header

- All flags in the first octet are dynamic. They are updated with the dynamic context. Also the payload type.

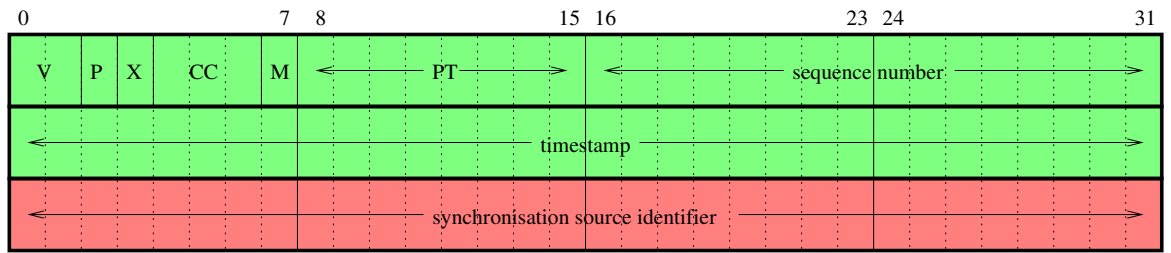


Figure 4.5: RTP header

- The sequence number as well as the timestamp are dynamic. They will be updated with every packet
- Synchronisation source identifier is the only RTP field which is static. Update of this value will only occur when the static context will be updated.

4.2 Modes

Each, the compressor and the decompressor, can be run in three modes. These are unidirectional, optimistic and reliable. The first one works in one direction only, and the two others are designed for bidirectional communication channels with feedback.

4.2.1 Unidirectional mode, U-mode

Unidirectional mode is for channels which have no feedback possibilities, or no feedback is desired. If a transmission is started, both sides, compressor and decompressor, start in this mode, because it's unknown at this time, if feedback is possible or not.

When no feedback is possible, the compressor doesn't know the decompressors condition. So it couldn't change the state depending on the knowledge of the decompressor. Consider this, the full context or probably only the dynamic, is retransmitted after predefined timeouts. Also when there are changes in the context fields, which needs to be transmitted with another packet.

In this mode only packets which contains a CRC-number are allowed. This is necessary, that the decompressor can check the correctness of the received packet.(see chapter 4.6)

Of course, this is not very efficient, because some large packets are sent when they aren't needed, and when context is lost on decompressor side, it has to wait for the next IR-packet until he can decompress more packets. The compressor could not react on the decompressor. The probability of loss is higher than having a feedback channel.

But ironically a simple feedback is intended in this mode anyway. Optionally the decompressor could send an acknowledge to the compressor, which will tell that a CRC verified packet is correctly transmitted. There are no other feedbacks intended in this mode.

4.2.2 Bidirectional optimistic mode, O-mode

This mode is similar to the unidirectional mode. But, as its name is telling, there is a feedback used. Periodic refreshes aren't known in this mode. The feedback can tell the compressor about occurred errors, to recover them. Optionally it could send acknowledges to significant context updates, which aren't usual.

The feedback channel in optimistic mode is only used sparse. But the efficiency is better and the error probability smaller than in the unidirectional mode.

4.2.3 Bidirectional reliable mode, R-mode

The R-mode differs in many ways from the other two modes. In this mode the robustness against loss and damage propagation is maximized. The probability of context invalidation is much lower than in optimistic mode. To get this, the feedback channel is used more often. A feedback is sent to acknowledge all context changes.

4.3 Compressor

4.3.1 States

The compressor knows three states. These are the initialisation and refresh state, the first order state and the second order state. In figure 4.6 all these states, and the possible changes are showed.

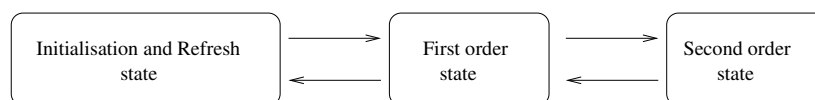


Figure 4.6: Compressor states

Initialisation and refresh state, IR state

This state is used to initialise the decompressor with the static and dynamic context. To do this it sends IR packets. It stays in this state until it will be fairly confident that the decompressor has received the context correctly. This could happen with a feedback which indicates that a IR packet could be decompressed correctly. If no feedback is possible the compressor gets

the confidence about succeeded initialising by sending consecutively some IR packets.

First order state

In this state it's possible to communicate changes in the dynamic part of the context. So informations about all the dynamic fields are sent rarely. These are only partially compressed. From the static context only few numbers can be updated.

The compressor stays in this state when after transmitting the static context. Or if changes in the dynamic context occurs during the compressor stays in the second order state. It changes to IR state when also parts of the static context changes. It will change to the second order state, when it is fairly confident that the decompressor knows the dynamic context.

Second order state

When the compressor stays in this state, compression is optimal. So the compressor tries to stay in this state as often as possible.

In this state only the sequence number and a few number of TS or ID bits could be transmitted. To reach this state the compressor must be fairly confident that the decompressor knows all context fields. Also all parameters to derivate the other header fields must be well known.

The state has to change to the FO or IR state, as soon as some other fields will change or if the timeout occurs.

4.4 Decompressor

4.4.1 States

The decompressor knows three states as well, which are completely independent from the modes. Every mode is possible in every states. The name of this states are related to the known context. The states and the possible changes are visible in figure 4.7.

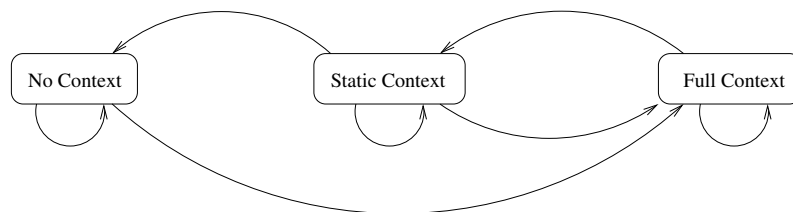


Figure 4.7: Decompressor states

No context

In the no context state, the decompressor has no context. A transmission is always started in this state of course. Being in this state, only IR-packets are allowed to receive. All other packettypes will be discarded, because the decompressor needs to know the informations of the context. When one packet is succesful decompressed, this is checked by the crc-check, the decompressor changes in the full context state.

Static context

When the decompressor stays in this state, it knows only the static context informations, but it has no dynamic informations or they are damaged. If repeated packets will fail, the state will be changed to the no context one. The allowed packets in this state are: IR, IR-DYN, UOR-2-TS.

Full context

If the decompressor stays in this state, it knows the full context. Once it entered this state it will stay here. So every packet is allowed to decompress. Only when several continous packets will fail the CRC-check, it will leave this state and change to the static context state.

4.5 Encoding methods

In this section the encoding methods which are used in the ROHC standard are described. These methods brings a significant byte saving with it, because some large values only adept small changes. So it's not reasonable to transmit the full value with every packet.

4.5.1 Least Significant Bits (LSB) encoding

With LSB encoding, only the k least significant bits are transmitted, where k is a positive integer. With this value and a previously received reference value v_{ref} the decompressor can derive the original value.

To calculate the correct value, it's necessary, that the compressor and the decompressor use the same reference. And the original value is the only value that has exact the same k least significant bits of those transmitted.

Both sides works with an interpretation interval, which can be described as a function $f(v_{ref}, k)$. Let

$$f(v_{ref}, k) = [v_{ref} - p, v_{ref} + (2^{k-1}) - p] \quad (4.1)$$

f defines for every value of k a uniquely identified value in the interpretation interval. p is an integer, and used to shift the interval with respect to v_{ref}

depending on the field characteristics. For example, if the field values are expected always to increase, $p = -1$ will be a good choice. If the field values are expected to stay the same or increase $p = 0$ is recommended. For com-

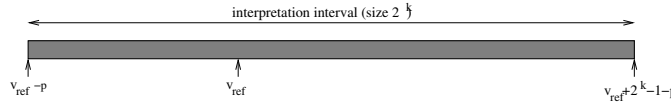


Figure 4.8: Interpretation interval

press a value v , the compressor has to find a minimum value of k , such that v falls into the interval $f(v_{ref}, k)$. The function to get k is $k = g(v_{ref}, v)$. The decompressor is able to recalculate the original value v with this transmitted k bits.

Note that the values have a finite range. If they are near to the minimum or the maximum value, a wraparound is possible.

It's easy to understand, that when the reference will be wrong, perhaps because of a transmission error or packet loss, all recalculated values will be wrong. To get sure, that this couldn't occur, only values that are verified with a CRC-check will be used as a reference on decompressor side. The compressor uses the last sent values which are secured by a CRC. So it will be several values in a window. When a packet transmission is confirmed with a feedback, the compressor has to delete the corresponding reference value from it's window. If there are more possible reference values in the window, it needs a greater value of k to identify the value uniquely, which results in sending a packet with less compression.

In unidirectional and optimistic mode, the decompressor tries, in case of a failed sequence number decompression, to decompress the same received value with the last reference again. This procedure mitigates damage propagation when a small CRC fails to detect a damaged value.

4.5.2 Window-based LSB (W-LSB)encoding

With this method, the previous one achieve robustness.

The compressor may not be able to determine the exact reference value of the decompressor. So it uses a window of all possible values.

So, initially this sliding widow is empty. After sending a compressed value v , protected by a CRC, this value is added to the sliding window. For each value being compressed, the compressor chooses the minimum of necessary bits to compress it uniquely. This is done with $k = \max(g(v_{min}, v), g(v_{max}, v))$. g is the function defined in the previous section, v_{min} and v_{max} are the minimum and maximum values in the sliding window. When the compressor is sufficiently confident that a reference value isn't much longer used by the decompressor, it deletes this and all older values from the sliding window.

4.5.3 Timer-based compression

Timer-based compression is a compression standard which can be used alternate for the compression of the RTP timestamp. Due it isn't used in this implementation. The definition of this compression method is defined in the RFC 1889 [3].

4.5.4 Scaled RTP Timestamp encoding

The timestamp isn't increased by an arbitrary number from packet to packet. Instead, the increase is a fix value depending on the codec, respectively its sample rate and frame. For example, when the sample rate of an audio codec is 8kHz and a frame covers 20ms, the increase of the timestamp is always $8000 * 0.02 = 160$ or a multiple of this. This step size is called *TS_STRIDE*. To compress this value, the timestamp is downscaled before by the factor of *TS_STRIDE*. Doing this saves $\text{floor}(\log_2(\text{TS_STRIDE}))\text{bits}$ for each value.

During initialisation the compressor transmit the actual timestamp value and the *TS_STRIDE*.

After this it's not reasonable to send the full timestamp. So, only the compressed value *TS_SCALED* is transmitted. *TS_SCALED* is calculated as followed:

$$TS_SCALED = \frac{TS - TS_OFFSET}{TS_STRIDE} \quad (4.2)$$

To compress this, it's possible to use W-LSB encoding or timer-based encoding.

To recalculate the timestamp, the decompressor use following equation:

$$TS = TS_SCALED * TS_STRIDE + TS_OFFSET \quad (4.3)$$

TS_STRIDE is transmitted during initialisation, and *TS_OFFSET* can be calculated by the decompressor with

$$TS_OFFSET = TS \% TS_STRIDE \quad (4.4)$$

Wraparounds in the timestamp are detected by the compressor. Should this happen, it's very unusual but possible, the compressor will send a initialisation packet.

4.5.5 Offset IP identifier encoding

The sequence number in RTP streams will increase by one with each packet. The IP identifier will increase by at least the same amount. Knowing this, it's more reasonable to compress the offset instead of the ip identifier itself. The compressor calculates the offset as followed:

$$Offset = IPidentifier - sequencenumber \quad (4.5)$$

Both values are taken from the actual packet, of course.

To compress this offset, W-LSB encoding is used.

When the compressor is receiving a new packet, it calculates the offset with same formula as the compressor, but it uses his reference values from context. With the W-LSB method the offset for this packet could be calculated. Note that the received value could be zero, when the offset hasn't changed. Finally, the IP identifier value could be calculated with

$$IP_{identifier} = sequencenumber + offset \quad (4.6)$$

Some stacks do not use a counter for the ip identifier, or the values are generated randomly. If this will occur, it is not possible to use a compression methode for the packet stream.

4.6 CRC calculation

4.6.1 CRC's for IR and IR-DYN packets

For these packets it's very important to have a high quality verification, so a eight bit CRC is used. This number is calculated over the full IR respectively IR-DYN packet, excluding payload but with the CID field. The CRC field is set to zero for the calculation.

To calculate this digit, the following formula is used.

$$C(x) = 1 + x + x^2 + x^8 \quad (4.7)$$

4.6.2 CRC's in compressed headers

For compressed packets the CRC is calculated over the RTP/UDP/IP header. This means, the compressor must first rebuild the uncompressed RTP/UDP/IP packet before calculating the checksum.

Note, first the decompressor has to calculate the IP checksum. This is not the same as the described CRC checksum.

The initial content of the CRC register is preset to all one's.

To calculate the seven bit CRC, following formula is used:

$$C(x) = 1 + x + x^2 + x^3 + x^6 + x^7 \quad (4.8)$$

To calculate the three bit CRC, following formula is used:

$$C(x) = 1 + x + x^3 \quad (4.9)$$

4.7 Packettypes

The ROHC-standard defines several different packets. They differ in size and of course in its contents. All possible packets are listed below, in descending

order. At the end there were feedback packets for the two bidirectional modes.

- IR packet
- IR-DYN packet
- UOR-2 packet
- UOR-2-ID packet
- UOR-2-TS packet
- UO-1 packet
- UO-1-ID packet
- UO-1-TS packet
- R-1 packet
- R-1-ID packet
- R-1-TS packet
- UO-0 packet
- R-0 packet
- R-0-CRC packet
- FEEDBACK-1 packet
- FEEDBACK-2 packet

In this implementation only packets which contains a crc checksum are used, and of course which are enable in the unidirectional mode. For simple feedback it is the FEEDBACK-1 packet used.

4.7.1 Context identifier byte

At the beginning of each packet, there is a context identifier, abbreviated CID, needed. This could be a short one, like in this implemtation, or it can be used a long one. With the short CID it is possible to handle up to 15 different streams over one link. If more are desired, long CID's are needed. In common applications these is never used, only in very complex environment will use this. So this CID-byte looks as followed. The CID-field could be a number between one and 15. If it is zero long CID's are used, and more CID-bytes are expected

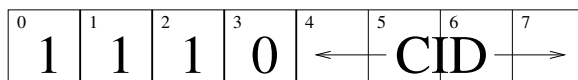


Figure 4.9: CID-byte

4.7.2 IR packet

With the IR packet the full static and dynamic context will be transmitted. This packettype is used at least in start-up phase of a transition. Afterwards this packet is only used, when several packets fail, or when there should be any changes in the static context. The IR packet header is shown in figure 4.10. The IR packet contains first the identifier byte, which is used

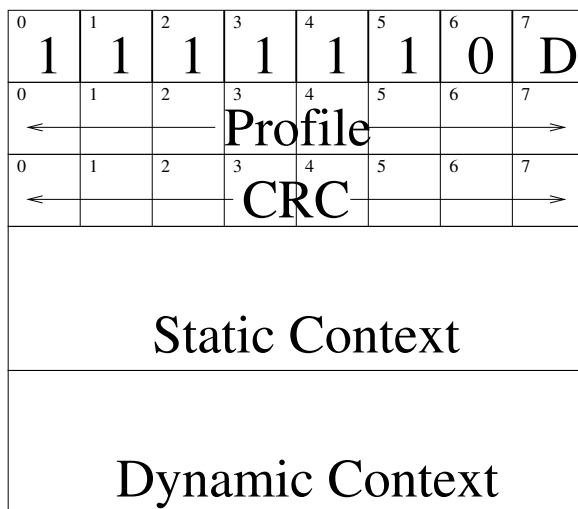


Figure 4.10: IR-packet

to detect the packettype. With clearing the D bit, it's possible to send an IR packet without dynamic context. This will be used only rare, when static context changed but dynamic not. Second byte defines the profile. The profile contains information about the ROHC compression that is used. This could be:

- 0x0000 for uncompressed ROHC
- 0x0001 for RTP/UDP/IP ROHC compressed packet
- 0x0002 for UDP/IP ROHC compressed packet
- 0x0003 for ESP/IP ROHC compressed packet
- ... future work

It's very important that the decompressor is initialised correctly, so it's obvious, that there is a CRC checksum. To get a high error correction probability, a 8-bit CRC is used. This sum is calculated over the entire IR packet, without the payload, but within the CID-byte. For calculation, these unknown CRC bits are set to zero.

4.7.3 IR-DYN packet

This packet is nearly the same as the IR packet. But only the dynamic context is contained. It is used to refresh the dynamic part if there occur any changes. See figure 4.11 to learn more about the construction of a IR-DYN packet. It contains the same fields as the IR packet. The CRC

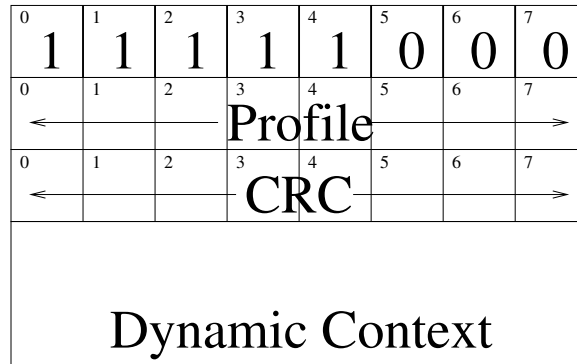


Figure 4.11: IR-DYN packet

checksum is calculated in the same manner as for the IR packet. For more informations about this, please see the section of the IR packet 4.7.2.

4.7.4 UOR-2-TS packet

The UOR-2-TS contains the compressed *TS_SCALED* and the compressed sequence number. The header of a UOR-2-TS packet is shown in figure 4.12 The first three bits identify the UOR-2 packet. The rest of this byte contains

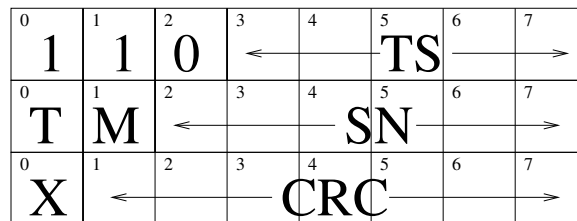


Figure 4.12: UOR-2-TS packet

the compressed timestride. When the T bit is true, a UOR-2-TS packet is expected, should it be zero an UOR-2-ID. M is the RTP marker bit. The

sn-field contains the compressed sequence number. The X bit marks that an extension is preset. Extensions does not make sense in unidirectional mode, that's the reason that they aren't implemented. The CRC checksum is composed of seven bits. For this type, the packet is completely decompressed, and then the calculation is done over the full decompressed RTP/UDP/IP-header. The new value of the ip identifier will be calculated from the sequence number.

4.7.5 UO-1-TS packet

A packet of this type contains the same information fields as the UOR-2-TS packet. But the fields of the sequence number and of the crc are smaller. Please look at figure 4.13 for detailed informations. The first two bits identify

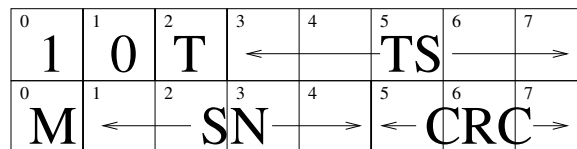


Figure 4.13: UO-1-TS packet

the UO-0 packet. With the T bit, the UO-1-TS and the UO-1-ID packets are distinguished, one for the TS and zero for the ID. The *TS_SCALED* field is here five bits too. In the second octet the first bit is for the marker as well. The next four bits are for the compressed sequence number. With four bits, only small changes could be updated of course. The CRC checksum can use only three bits. Apparently this checksum isn't very secure, only eight different crc's are possible.

4.7.6 UO-1-ID packet

This packettype is very similar to the UO-1-ID. But instead of the *TS_SCALED*, the compressed *ID_OFFSET* is transmitted. For the content see figure 4.14. For the description of the content look in the section of the UO-1-TS.

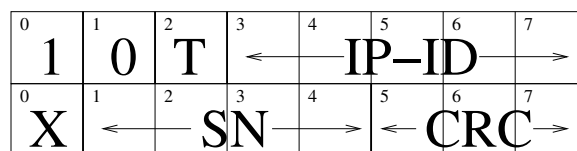


Figure 4.14: UO-1-ID packet

It is exactly the same, excepted the *TS_SCALED*. This field is used for the *ID_OFFSET* and the T bit is zero of course.

4.7.7 UO-0 packet

With this packettype the highest compression level is achieved. Below a UO-0 packet is shown. A UO-0 packet contains a 3-bit crc and 4 bits for



Figure 4.15: UO-0 packet

the compressed sequence number. With this information it can update the sequence number. With the decompressed sequence number it's possible to calculate the ip identifier and the timestamp value as well.

4.7.8 FEEDBACK-1 packet

This packet is an ACK feedback, that signalise successful decompression and context update at decompressors site. It returns the CID of the used context for decompressing an the sequence number of the decompressed packet [4.16](#). The code field indicates the size of the feedback data ioctets.

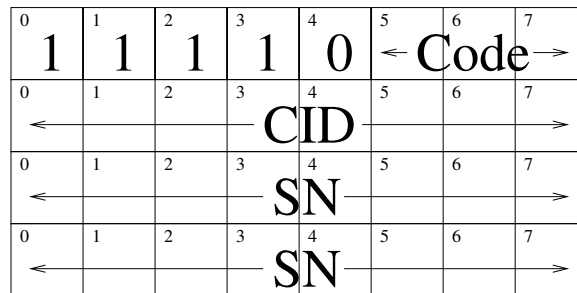


Figure 4.16: FEEDBACK-1 packet

Chapter 5

Existing ROHC-Project

At the beginning of this diploma thesis the goal was to extend the existing open source project of a ROHC (RObust Header Compression) implementation with the compression profile IP/UDP/RTP. The implementation could be found at ¹. The existing project currently implements a compressor / decompressor part and the compression profiles UDP, UDP-Lite, IP-Only and uncompressed. The compressor and decompressor are working in the unidirectional (U-Mode) and the bidirectional optimistic (O-Mode) modes. The Project from sourceforge site is currently running under a Linux kernel, 2.4.20 and a PPP connection.

To install and run the existing open source project the deployed computers were running with a Fedora Core 4 distribution under a Linux kernel 2.6.11. To set up the PPP daemon and use the PPPoE (Point-to-Point Protocol over Ethernet) it was required to download and install the considering `rp-pppoe` and `pppd` packages for the Fedora Core 4. Before compiling these packages, it was needed to replace some source files in the packages with files that come along with the ROHC project. But this was the first problem, because there were some differences in the files that must be replaced. There were declarations used in the ROHC files that were removed in the files from the `rp-pppoe` and `pppd` packages. This led to compiling errors.

Concerning these problems the decision was to use the suggested `pppd` version 2.4.2(b1) package. These actions solved the problems above. But when trying to compile the ROHC project, there occurred new error messages, which refer to errors in the C-files from the ROHC project. Some errors came due to the fact that the including path of included kernel header files differs from the path denoted in the ROHC files. These were not so hard, after adjusting the include path the errors disappeared. But other errors referred to ROHC specific definitions in the source files `pppoe.c` and `ppp_generic.c`. These errors could not be easily removed with adjustments in the code, because this destroyed the functional efficiency of the ROHC project. Now, the

¹<http://sourceforge.net/projects/rohc>

next step was to use the Linux kernel 2.4.20, because the project must be tested with this kernel according the informations at the homepage. The hope was that with this method the errors disappear automatically. But this was not the case even though the used kernel and the required packages matched with the versions suggested at the homepage of the ROHC project. The last hope was to receive help from the mailing list on the sourceforge page and from the authors of the existing project. Neither the project administrators nor the persons who entered similar problems at the mailing list could give some advise to solve the prolems that appeared.

The causes listed above brought about the decision to implement an new interpretation of the ROHC described in the RFC 3095. But in consideration to the short space of time up to the end of the diploma thesis, it is not possible to implement the whole ROHC described in the standard track. Therefore the implementation should basically include a framework with compressor and decompressor operating in unidirectional mode. Further the ROHC profile IP/UDP/RTP should be included to the framework. Also an acknowledge feedback that signalise successful decompression of a compressed ROHC packet should be used to improve the performance. More details about the implementation can be found at the corresponding chapter [6](#).

Chapter 6

Implementation

6.1 Compressor

6.1.1 Compressor framework

At the beginning of the ROHC program a compressor framework would be set up. The setup routine initialise a compressor with the maximum number of context identifier (CID) and the used profile. In this simplified implementation only the short CID up to 15 and the profile IP/UDP/RTP are possible.

When the compressor receives a packet from the pcap network sniffer, he first determine which compression profile should be used according to the initialising values of the framework. In this case it is the profile mentioned above. But in future it could be implemented more profiles like described in the RFC 3095 [4].

In the profile specific compression part the compressor first extract all headers and the payload that comes along with the captured packet. The ethernet header is used to determine if the received packet represents a feedback packet or not. A feedback packet is signalled by the packettype `0x5555` in the packettype field if the compressor use the ethernet wrapped function. If it is ip wrapped a feedback is present if the value in the protocol field is `0xCF`. See figure 6.1.

In this implementation only a ACK-feedback in a feedback-1 packet is supported. The feedback routine first look up the value of the CID in the packet. With this number it is possible to find the corresponding context, with the required feedback parameters. The context contains a parameter that saves the sequence number of the latest received feedback. Because the sequence number should be increased by one for each sent packet, the returned number in the feedback should also increase by the same amount. If packet or context loss occurs, the received sequence number in the feedback has a deviant behavior to this structure. In that case the compressor would step back to the next lower state and try to repair the context at the decom-

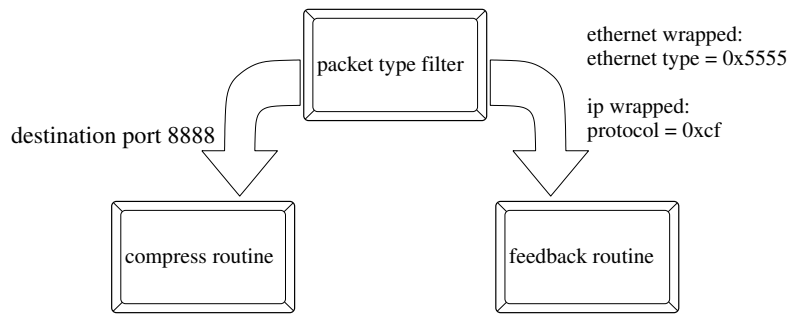


Figure 6.1: packet filter

pressor by sending a context refreshing packet like IR or IR-DYN packets. Another function of the feedback is to force an up transition to a higher state, if the context is completely established between compressor and decompressor before the optimistic approach limit is reached. This happens if the compressor gets a valid sequence number back from the decompressor. For an abstract overview see figure 6.2

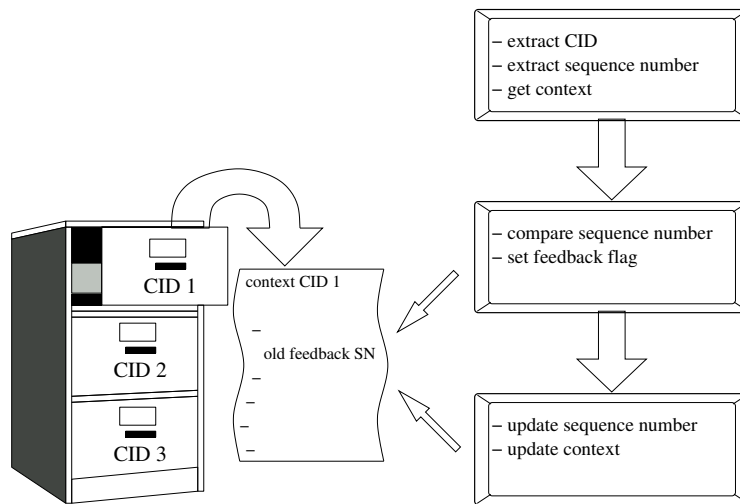


Figure 6.2: feedback routine

Receiving a non feedback packet prompts the compressor to compress the packet according to an existing context or to create a new context. But first there would be performed a header check to determine the validity of the packet. This means, the compressor check if the IP header has version 4 and the IP protocol field contains the UDP type. Further would be controlled that the IP header length is 20 bytes. This could be done with the IP header length field.

Afterwards the compressor search the appropriate context for this packet.

Therefore the compressor takes the first used context in the context table and compares the parameters which defines its characteristic. These parameters are the IP source and destination address, the UDP source and destination port and the RTP synchronization source SSRC. If the parameters does not match to the context the compressor takes the next one until there are no more existing contexts. In this case there would be allocated free memory space and created a new context. If a context is found or a new context is successful created the compressor now knows the context for compressing the current packet.

As a next step the byte order of the header entries would be checked and it would be detected if the IP identifier is a random number or if it increase in a derivable way from the sequence number. Is the IP identifier random, an efficiently compression of the header fields could not be done, because the IP identifier must be send uncompressed in every packet. Are the header field in network byte order the NBO flag is set true. In the other case the flag is false. With this information the decompressor knows in which order the header bytes must be placed in the rebuilt packet header.

After the previous steps, now the compressor is ready for encoding the packet headers. Is it the first packet of a context, the encoding routine enters in a special subroutine. In this cycle the payloadtype in the RTP header would be extracted. With the payloadtype value the compressor knows the codec which is established between the endpoints of the communication channel. Now, it is feasible to calculate the default TS_STRIDE of two consecutive uncompressed packets.

$$TS_STRIDE = SAMPLE_RATE * PACKET_SIZE \quad (6.1)$$

For example, in the case of audio with the GSM codec, the default sample rate is 8kHz and one voice frame may cover 20ms and it is carried in one RTP packet. In this case the RTP timestamp increment is always $8000 * 0.02 = 160$ between two packets. Please note that these are only default values, so the TS_STRIDE can change his value during a communication session.

After calculation the TS_STRIDE the compressor evaluates the TS_OFFSET value and the scaled timestamp TS_SCALED .

$$TS_OFFSET = TS \quad \text{mod} \quad TS_STRIDE \quad (6.2)$$

$$TS_SCALED = \frac{TS - TS_OFFSET}{TS_STRIDE} \quad (6.3)$$

The next step is to send an IR packet including all header informations, and additional the TS_STRIDE .

For all other packets the compressor works through the following scheme. First the compressor calculates the ID_OFFSET and the new TS_STRIDE . The ID_OFFSET is the distance between the value of the IP identifier and the sequence number. Now the TS_STRIDE is calculated as the change between the current timestamp and the immediate previous one.

$$ID_OFFSET = ID_i - SN_i \quad (6.4)$$

$$TS_STRIDE = TS_i - TS_{i-1} \quad (6.5)$$

The new TS_STRIDE is not added directly to the context. First it is checked if it is equal to zero. In this case the value of TS_STRIDE stay as is, because the data frame is divided into several packets. The TS_OFFSET and the TS_SCALED are computed in the same way as described above.

After recalculating the encoding parameters the compressor decide which state should be used to send the compressed packet. Occur differences among static context part, i.e the RND flag, DF flag, X flag and payloadtype. The send static flag will be set to true because it is needed to refresh the context with a IR packet. Are there some varieties between values in the dynamic part of the context the send dynamic flag is set to true and it is required to send a update packet like the IR-DYN packet. An other point which force a potential state transition is in the case of an UDP checksum change. This value could only be transmitted with a packet that refresh the static and/or the dynamic part, i.e a IR or IR-DYN packet. The figure 6.3 shows the compressors state machine with the transition events.

After updating the current state the compressor evaluates the number of bits needed to send the W-LSB encoded sequence number, the W-LSB encoded IP identifier offset and the scaled timestamp. How to encode these values exactly can be found in the theory part of this documentation 4.5.2. According to the required bits above, the compressor decides which packet must be sent to transmit all important informations. The packets which can be sent are different between each state. In IR state the compressor send a IR packet to initialise or refresh the context at decompressor side. In this implementation the FO state is able to sent two forms of packet. The first is a IR-DYN packet to refresh the dynamic part of the context and the second is an UOR-2-TS packet. The UOR-2-TS enables only a timestamp update in addition to the sequence number. The other fields in the dynamic and static context part stay as is. An UOR-2-TS packet is also able to send the RTP marker bit, so it is not needed to send the whole dynamic context, if this bit changes. In SO state are four ROHC packet types available. One of these is the UOR-2-TS well known from the FO state. So the compressor have not to step back even the marker bit changes. The other packets are the UO-1-ID and the UO-1-TS. These packets allows updating the IP identification or the

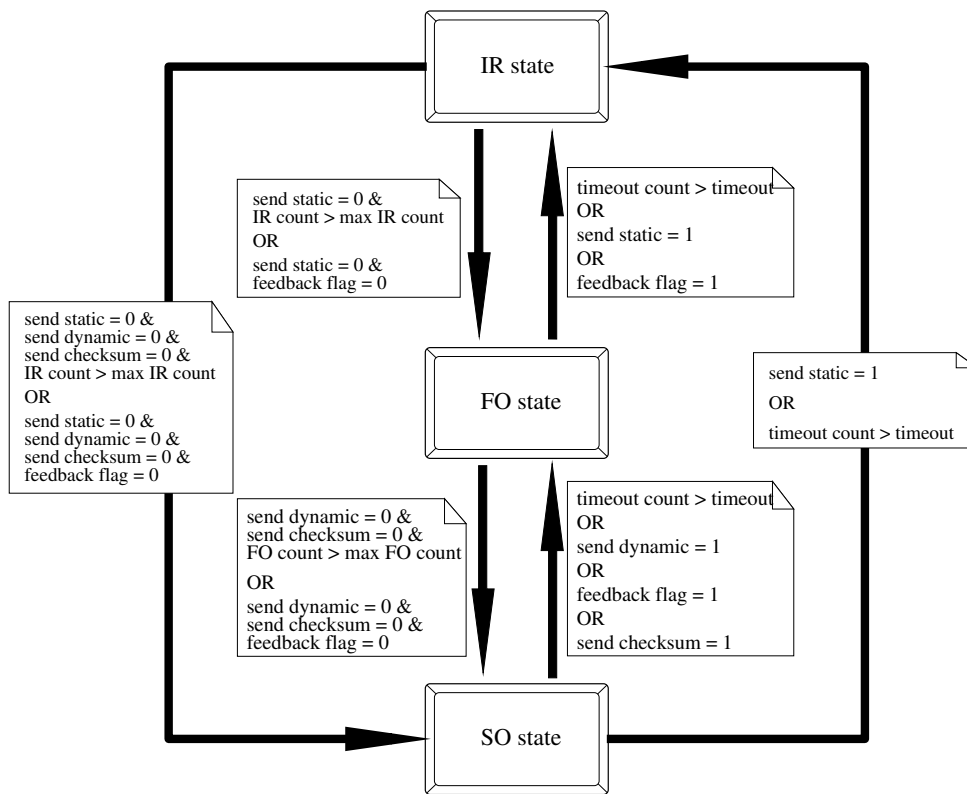


Figure 6.3: compressor state machine

timestamp according to the index at the end of the name. The last packet is the UO-0 packet. This packet offers the best compression, because it sends only sequence number informations.

The detailed structure of these packets is also explained in the theory section of this documentation 4.7.

After deciding the packet type that allows to send the whole updating information without waste a lot of unused bits, the compressor routine delivers to the packet generator routine a pointer to the context with all encoded parameters.

This routine compose the desired ROHC packet type according to the sheme listed in the corresponding chapter of these documentaion 4.7.

If the packet is built, the CRC checksum would be calculated over the oblige packet fields. Afterwards the ROHC packet is ready to send and would be delivered to the libnet routine that releases the compressed packet on the network.

At last the compressor routine will update the corresponding context with the current values which would be needed to compress the next incoming packet of the same packet stream. The parameters included to this step are

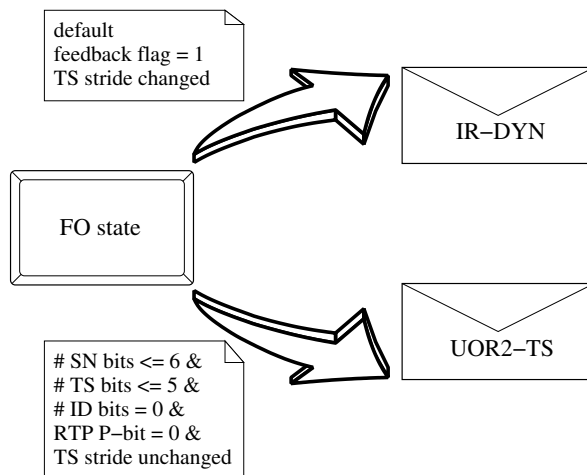


Figure 6.4: decide packet for FO state

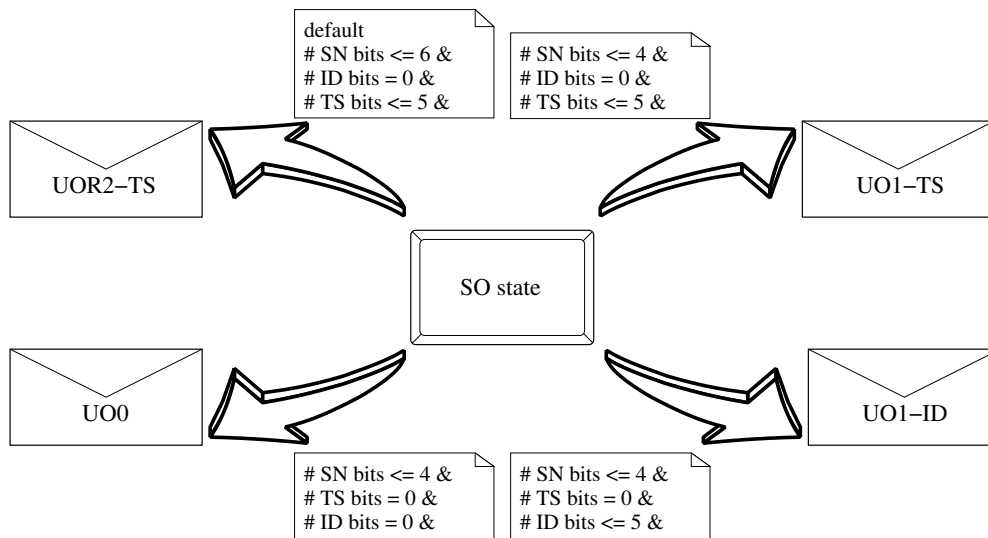


Figure 6.5: decide packet for SO state

the current IP, UDP and RTP headers. Further the RND, NBO flags and the TS_SCALED and TS_STRIDE would also be stored in the context.

6.2 Decompressor

6.2.1 Decompressor framework

In the initialisation part, a sniffer is set up. This is done with the pcap-library. With this library it's possible to filtrate the incoming packets with a specific ethernet type. In this project, the number 0x8888 is used to do this. This number has been chosen, because there are usually no other packets with the same type. Of course it's possible to sniff packets with defined ip-addresses or ports etc. But a ROHC-packet contains only the destination mac-address as well as the source address, and the ethernet type number. Further in this part, the contexts are set up. This means, that the memory is allocated for these variables. And the values that can be set now will be set. This are the initial state to no context and initial mode to unidirectional. Note that in this implementation only the unidirectional mode is used. So this parameter will never be changed.

Once the packet sniffer is initialised, the program stays in a loop, and evoke a callback function when a packet is captured, which handles the captured packets.

In the callback function, first the context identifier number is resolved. With this the packet could allocate to its context. In this implementation only CID values from one to 15 are possible. With this information the decompressor knows in which state it is for this packet.

The next step is to extract the packettype. The appropriate handle function can be called now. But it's possible, that the decompressor stays for the actual CID in a state where not every packet is allowed (see figure 6.1). If the packet isn't allowed, it simple discard this packet, and stays in the same state as before.

6.2.2 Handling of IR packets

First the received CRC number is copied in a local variable. The same number in the packet is deleted. Over this packet, the CRC will be calculated. Is this value equal to the one which was received, the packet has been transmitted without any errors. The following operation are done:

- The local context is updated with the received one.
- The reference values in the lsb-structures are updated with new received one.
- From the context and the actual payload, an original RTP/UDP/IP packet is regenerated to forward it.
- A feedback packet is sent to the compressor. This packet contains in unidirectional mode only the CID and SN.

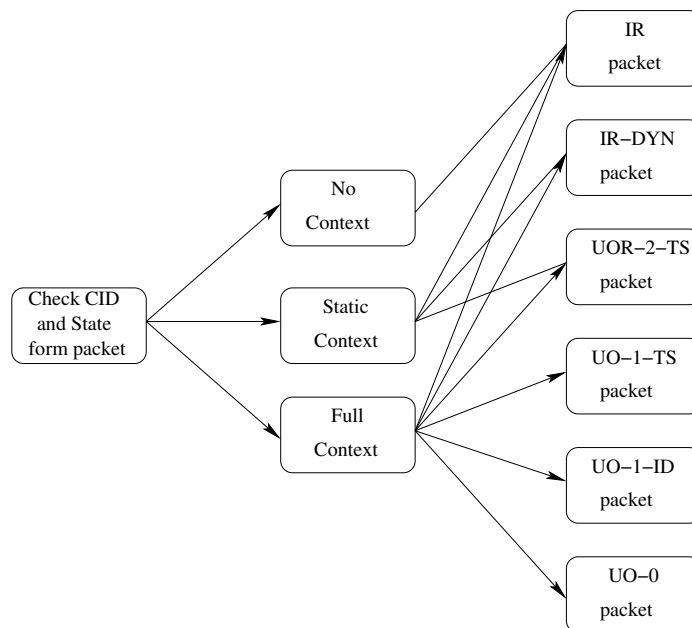


Figure 6.1: Decompressor actions

- The state for this CID is changed into full context.

If the CRC is wrong, the packet will be discarded. No error corrections are possible here, because no compressions are used in this packettype.

6.2.3 Handling of IR-DYN packets

The handling of this packettype is nearly the same. The only different is, that not the full context is updated after a correct CRC verification, but only the dynamic part.

6.2.4 Handling of UOR-2-TS packets

Compressed packets are more difficult to handle, because they have to be decompressed.

First of all, the old context is copied to backup variables. The CRC value from the packet is saved in a local variable. Then the received compressed values for *TS_SCALED*, sequence number and if identifier offset will be decompressed with the lsb decoding method. The timestamp and the ip identifier value will be calculated as it is explained in the theory part of this document.

These values are written into the context. With this context a original RTP/UDP/IP packet is reconstructed. Over this headers, which covers 40 bytes, a seven bit CRC is calculated. If this is equal to the received one, the

reconstructed packet is sent, and the reference values in the lsb-structures are updated.

When the CRC check fails, the sequence number is replaced by the next older one. With this modification the decompressor tries to calculate the right CRC value again. If this will succeed, the procedure above will be done. If this fails again, it is looked for a wraparound of the sequence number. If this was occurred, it will be corrected, and the CRC check will be done one more time. If this check results with the right value, the happening that will occur are listened above. Otherwise an error must be occur on the transmission link. The data is discarded and the packet isn't sent. The old context which was saved at the beginning is copied back. The reference values aren't updated of course.

If the packet was correct transmitted, the decompressor will change its state to full context, if it isn't there anyway. If it fails, a counter variable is increased by one. When this counter pass over a predefined level the decompressor will go a state backwards. This means from full context to static context or from there back to no context.

This counter is reseted with every successful decompressed packet. So the wrong packets must appear coherent. This is why one wrong packet is acceptable, but if there are more erroneous datas, the opposite side may take the wrong reference values to compress the values. This makes it necessary to refresh the context, probably only the dynamic one.

6.2.5 Handling of UO-1-TS and -ID packets

Handle a UO-1 packet is similar to this of a UOR-2-TS. But instead of three compressed values only two are transmitted. The *TS_SCALED* and the sequence number in the TS packet, the ip identifier and the sequence number in the ID packet. These packets are used, when only the according value increase changed.

The two transmitted values are decompressed like in the UOR-2-TS packet. The third value can be calculated from the sequence number. The timestamp will be increased by $TS_STRIDE*n$ where n is the step size of the sequence number. The ip identifier is increase by the same delta as the sequence number.

Regeneration, CRC check, error correction and the following actions are done in the same manner as they are described for the UOR-2-TS packet, but the CRC check is now only three bits.

6.2.6 Handling of UO-0 packets

With this packettypes, only the sequence number is transmitted. So this value is decompressed as usual. With this number it's possible to calculate the timestamp and the ip identifier, because they will be increase by the

same delta as before.

Any other action is equal to this of a UO-1 packet.

6.2.7 Additional informations and problems

To send the packets, the libnet library is used. The feedback packets can be detect again on the basic of the ethernet type number. Sending the decompressed and reconstructed packets to another host is possible, but in this application it wasn't necessary, because it was interesting to know how good the performance is.

One not clearly solved problem is that when the GSM voice codec is used, the payload size is only 33 bytes. Together with the ROHC and ethernet header it results packets with only 49 to 51 bytes. But these packets are to short to fit the minimum packet length. So then zero padding will occur. This problem is only with this mentioned codec, and of course only on ethernet. The fact that this standard is used mostly for wireless links mitigates this problem significant.

In the implementation on ethernet there is a fix value implemented. In the version over ip, this problem is no longer available, because there are additional ip headers.

6.3 Graphical User Interface

At the beginning of the work, there was no need for a graphical user interface. But with the increasing amount of code a simple way to change the parameters and visualization of the packet stream was needed. To configure the decompressor and the compressor a small GUI was designed. Due to the already existing code the GUI is not proper object oriented designed. It had to fit between the existing code, because there was no time to rewrite it. As library GTK+1.2 was chosen, because its native interface to C and its widespread distribution in the Linux world. To layout the design Glade 0.6.4 was used.

6.3.1 Building GUI

The first step to create a graphical user interface is to layout the window area using Glade. With Glade buttons, textareas and labels easily can be arranged. Afterwards the GUI source code can be exported to your project. Glade provides a set of helper files to compile all dependencies of the GTK library. This files generate a Makefile for your system. Also output of every build is a bunch of *.c files: main.c, interface.c and the callbacks.c.

main.c: In this file the GUI initializes its windows and the event loop.

```

GtkWidget *window1;
gnome_init ("gui-decomp", VERSION, argc, argv);
window1 = create_window1 ();
gtk_widget_show (window1);
gtk_main ();

```

After this calls the GUI window pops up and waits for user input. If the user edits or clicks on a button, an event signal is generated and the according callback function is called. In this callback function the user input can be processed and other functions can be called. In this project the user data is collected and the compressor or decompressor function is called with the provided parameters. To allow the GUI to response to user input during the compressor or decompressor is running the

```

while (gtk_events_pending());
gtk_main_iteration();

```

is called between packet processing.

callbacks.c: In this file the functions are found, which are called if the user makes an input. According to the input a signal calls the right function. To illustrate this a short example:

```

on_button2_clicked(GtkButton *button, gpointer user_data)
{
    gdk_exit(0);
    gtk_main_quit();
}

```

The `on_button2_clicked` function is invoke if the user pushes the `button2` in this case the exit button. The following calls try to close the GUI on a gentle way, but the `libcap` capture loop prevents the GUI from exiting graceful. At the moment it is not possible to break the loop and close the GUI at the same time.

interfaces.c: This is the file, where Glade stores the code which builds the the GUI window. Small changes to the GUI can be done directly in this file. For major changes the project should be edited with Glade. Therefore the glade project file `*.glade` is load into Glade.

6.3.2 The two setup windows

These two figures show screenshots of the developed GUI. It is obvious that the compressor GUI contains more field to change than the decompressor GUI. Lets first have a look at the compressor.

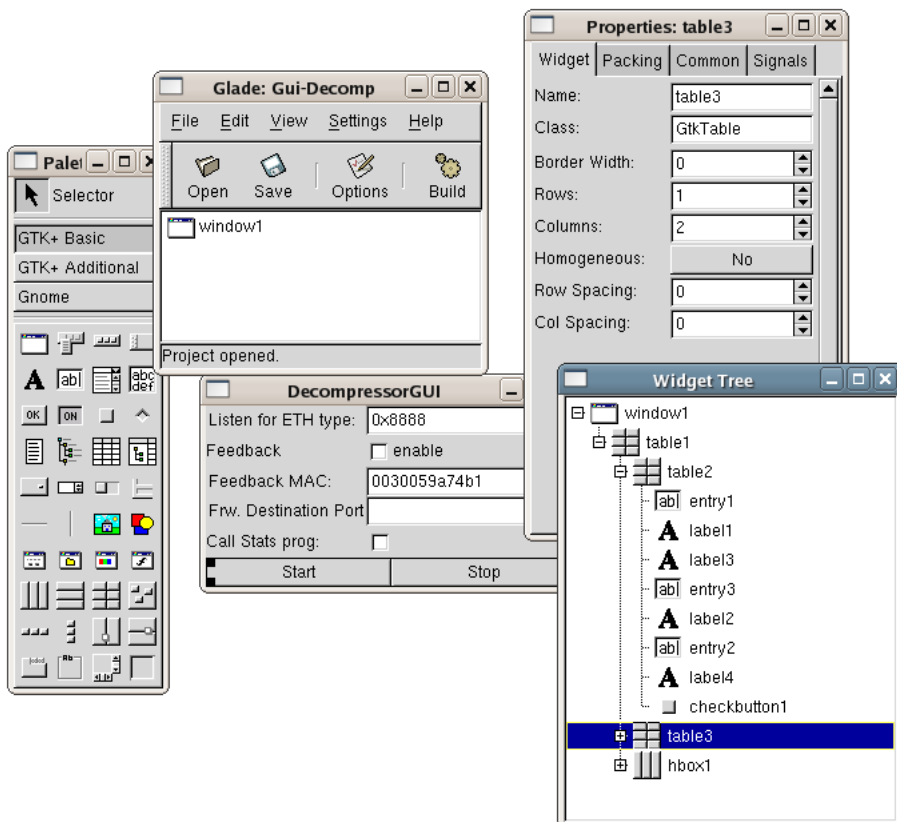


Figure 6.2: Screenshot of the Glade GUI designer

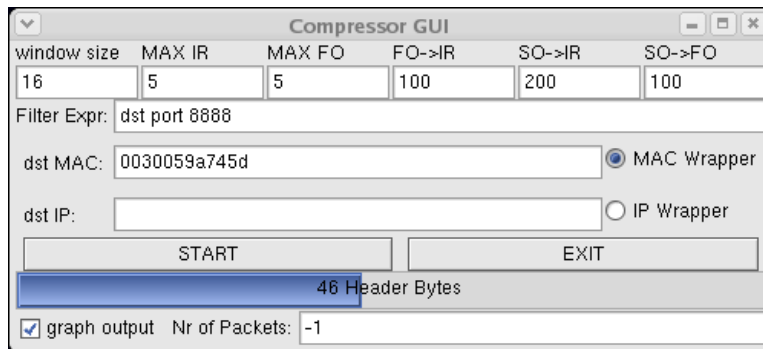


Figure 6.3: The compressor GUI

- Window Size: Determines the size of the w-lsb algorithm. This affects directly the efficiency and the robustness.
- MAX IR: This Value sets the maximal count of successive IR packets.
- MAX FO: This sets the maximal count of FO successive state packets.
- FO->IR: After how many packets the compressor transients from FO to IR state.
- SO->IR: The same for SO to IR.
- SO->FO: And the same again for the SO to FO state transition. For a detailed explanation of these states and transitions please refer to chapter 4.4.1.
- Filter Expression: This sets the port on which the compressor listens to outgoing RTP traffic. This filter expression is fed to the pcap library. Refer to chapter 6.4.1.
- Destination MAC: If the ROHC stream should be wrapped in Ethernet frames, this field has to be filled with a valid MAC address. And the radiobutton on its right hand has to be activated.
- Destination IP: Really the same like the destination MAC, but the ROHC packet is wrapped into a whole IP4 packet. Also the according button has to be checked.
- START: This button starts the capturing and compressing.
- EXIT: This button only produces trouble, its better to hit ctrl-C.
- Progress Bar: This bar indicates the average compression rate.
- Graph output: This checkbox activates a statistic file and a real time plot of the outgoing ROHC stream (see figure 6.4).

- Nr of Packets: The value entered here determines the amount of captured and compressed packets, a (-1) will never end.



Figure 6.4: Graphical output from the compressor.

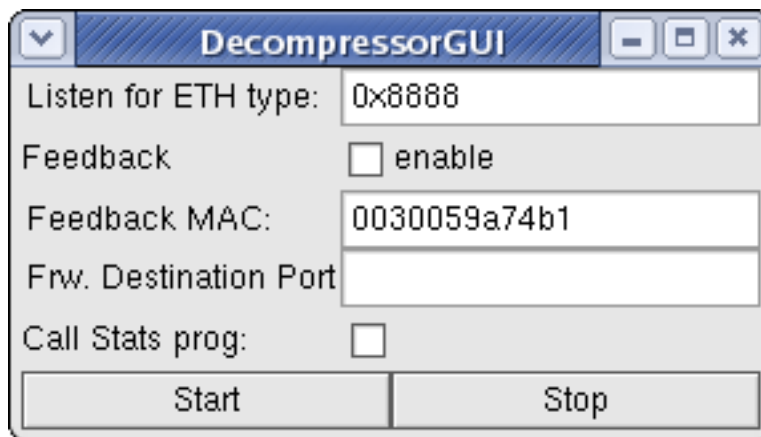


Figure 6.5: The decompressor GUI

As you can see the decompressor offers less options than the compressor. You only specify which ethernet type contains the rohc packets. If the compressor should send an acknowledge for every successful decompressed packet. The

Mac address where the feedback is send. And finally if the statistic program on the decompressor side should start up.

6.3.3 Statistic tool

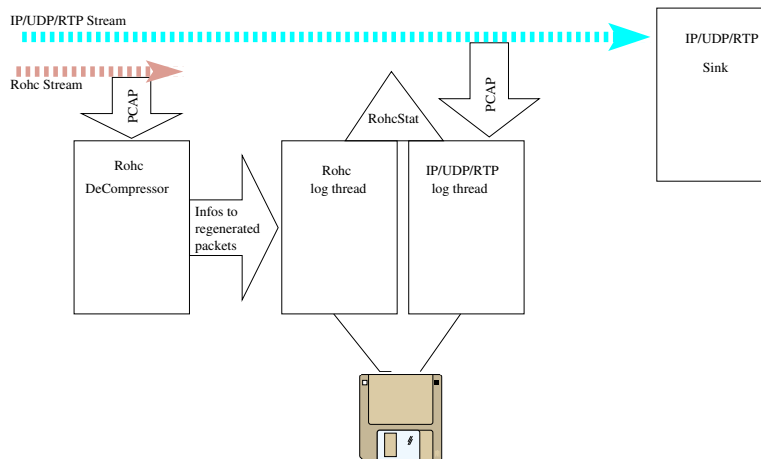
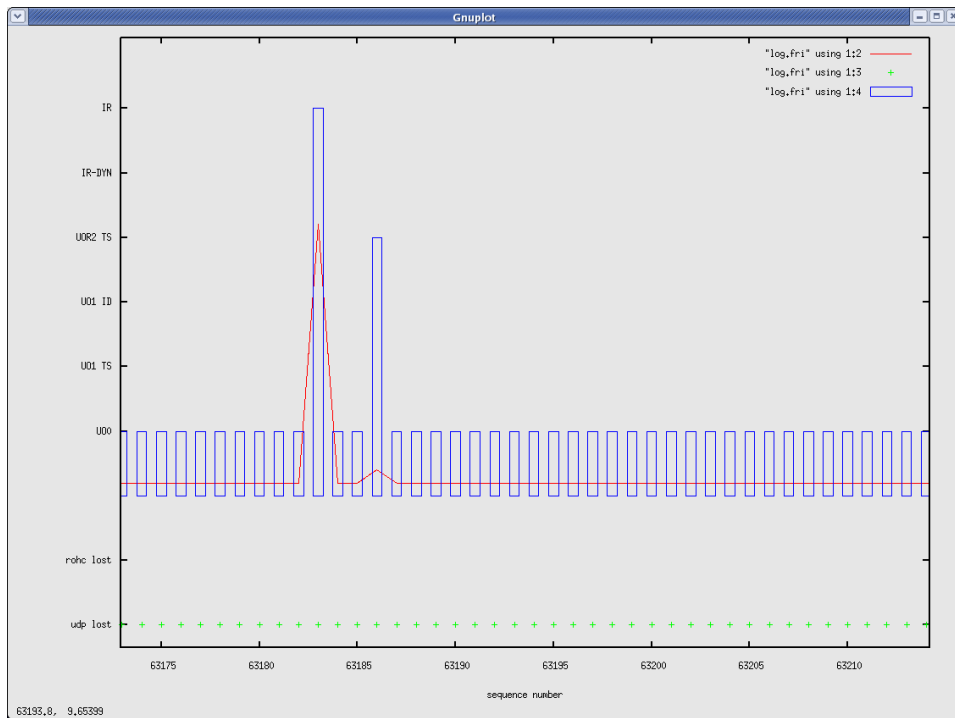


Figure 6.6: Outline how the rohstat program works

On the decompressor side its important to record the incoming packet stream. This makes it possible to compare the ROHC compressed stream with the uncompressed IP/UDP/RTP stream. With this data it is possible to analyze the performance gain or reduction. The statistic tool is a additional program, which is started from the decompressor. It is opened with the `popen{"/.rohstat", "w"}` command, this command opens the specified program an generates a pipe to the stdin of it. Through this pipe it is possible to send arbitrary data to the receiver. If a packet was decompressed successfully it sends the regenerated header through the pipe inclusive what ROHC packet type it was. When the rohstat program is open it starts two threads, one is listening to the stdin and the other starts a pcap environment to capture incoming IP traffic. A logfile is generated, where both threads write their data. Figure 6.6 illustrates how this works. To check if a packet is lost the file is filled with 300 sequencenumber ahead of the first one captured. Afterwards every incoming sequencenumber is checked against this list, this allows to log if a packet loss occurred. To avoid both threads writing at the same time a global mutex is set. This mutex is locked from the thread that is actually writting to the file. To visualize the datastreams the logfile is send very fifty packets to gnuplot, which displays a nice plot. Gnuplot is also opened with `popen("gnuplot", "w")`. This allows to send the cooman console input commands to gnuplot like:

```
fprintf(plot, "plot \"log.fri\" using 1:2 with lines, \"log.fri\"
using 1:3, \"log.fri\" using 1:4 with boxes\n");
```



This plots the received Packets against the sequencenumber. The format how the data is stored in the log file is :

```
SequenceNumber   RochHeaderSize   UdpHeaderSize   packetType
```

Where the packetType is a number between 0 and 60. And it is drawn with blue boxes, so you can easily determine which packet type was received. Successfully received UDP packets are shown as green crosses. And the thin red line shows how many bytes the ROHC header carried.

6.4 Packet Stream

To be able to compress a IP/UDP/RTP packet stream you have to access the data. This has to happen after the data passed the TCP/IP Stack and before they reach the data link layer (see figure: 6.7). In a common Linux system this is the interface between the TCP/IP stack and the device driver. The TCP/IP Stack usually is compiled into the kernel and the network device driver is loaded as a module. This circumstance prevents from catching the packet stream and only send the ROHC packets. To catch and interrupt the IP/UDP/RTP stream you have to edit the kernel or the device driver source code. This is no a suitable solution, because this leads to a solution that can not easy be run on other computers. To avoid modifying the kernel or device driver, another solution was chosen. Using the two libraries: pcap

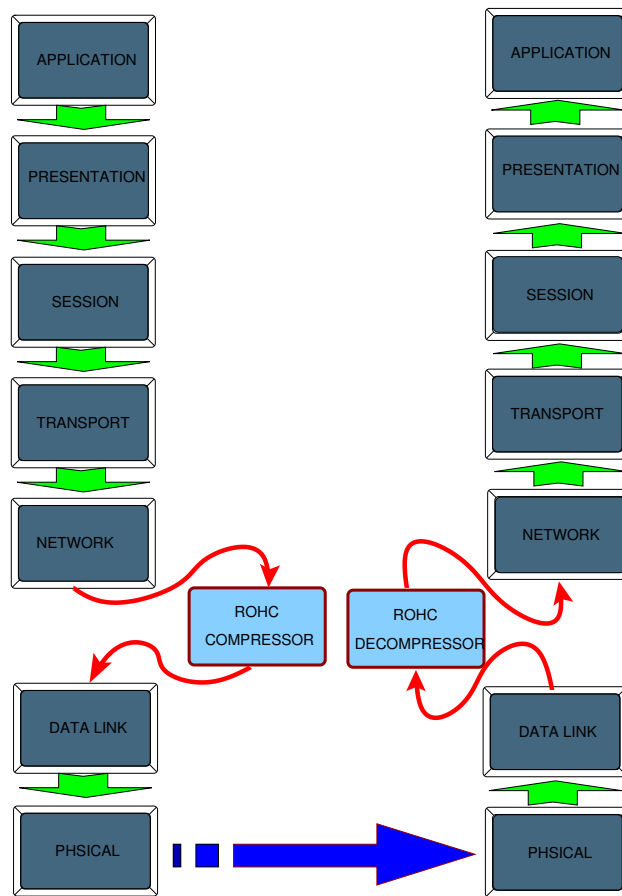


Figure 6.7: The OSI layer and ROHC

and libnet, it is possible to sniff the out-and incoming packet stream and reinject the ROHC packets. This generates a second packet stream, parallel to the IP/UDP/RTP stream. Both streams are send over the same network device, so a performance comparsion of both can be made at the target host.

6.4.1 PCAP Library

Pcap provides a high level interface to very low level network capturing. It is used in a lot of applications to monitor network traffic. In this work the library is used to access the out- and incoming IP/UDP/RTP packet stream.

The PCAP was installed using yum. The installed version is 0.8.3 from the Fedora repository. Using PCAP is quite simple and needs five basically steps.

1. Determining the interface which is used. On Linux systems this may be something like: eth0 or eth1.

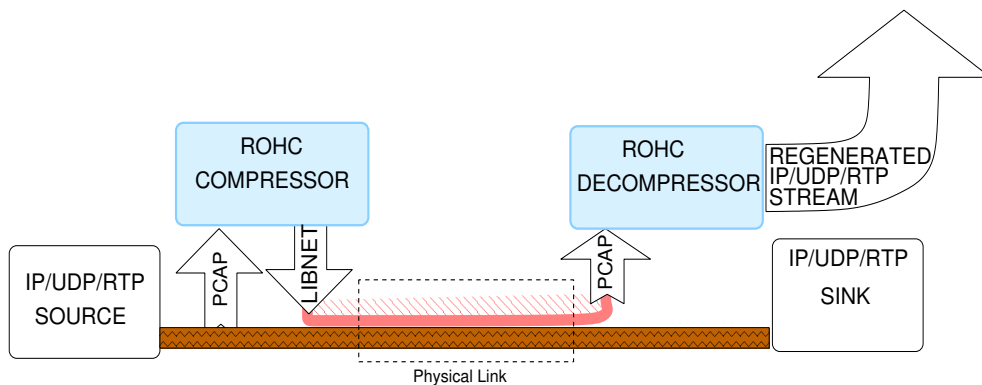


Figure 6.8: The packet stream is captured between the network and the data link layer.

```
char *dev, errbuf[PCAP_ERRBUF_SIZE];

dev = pcap_lookupdev(errbuf);
if (dev == NULL) {
    fprintf(stderr, "Couldn't find default device:
%s\n", errbuf);
    return(2);
}
```

2. Initialize PCAP: `char *device` sets the device, `int snaplen` sets how many bytes should be captured each time, `int promisc` set to promiscuous mode, `int to_ms` determines the read-out timeout in ms and finally `char *ebuf` is the pointer to a string, where the error message is stored.

```
pcap_t *pcap_open_live(char *device,
int snaplen, int promisc, int to_ms,
char *ebuf)
```

3. Set the filter. PCAP has the ability to filter the traffic, in a very effective way, according to a set of rules. Applying those rules to PCAP needs the following steps. `Pcap_t *p` is the pointer to a initialised PCAP environment, `struct bpf_program *fp` the pointer where the compiled filter is stored, `char *str` a pointer to the human readable filter string, `int optimize` indicates optimization (0=false and 1=true) and the `bpf_u_int32 netmask` specifies the netmask to use.

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char
```



Figure 6.9: Example of a ROHC packet wrapped by an IP packet



Figure 6.10: Example of a ROHC packet wrapped by an Ethernet frame

```
*str, int optimize,
bpf_u_int32 netmask)
```

4. The sniffing happens in the `pcap_loop()` this loop calls for every received packet a call back function. This loop iterates till `int cnt` is reached or the `pcap_breakloop(pcap_t *p)` is called. `pcap_handler callback` is a pointer to the callback function which processes the received packet and `u_char *user` is usually set to NULL.

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback,
u_char *user)
```

5. If the sniffing is over you have to clean up the pcap stuff by calling the `void freecode(bpf_program*)` and `void pcap_close(pcap_t *p)`.

6.4.2 LibNet

The Libnet library provides a similar interface like Pcap but it is used to generate and inject packets to the network. Depending in which mode Libnet is used it is even possible to modify ethernet frames. In this implementation it is used to generate ethernet frames with ROHC payload, or common IPv4 packet with ROHC payload. Due to the flexibility of the Libnet library it would also be possible to inject a couple of other packet formats. The whole Libnet is not documented very well. But there are a few examples on the internet and the `.h` files contain a lot of comments. The library is extensively used in the ROHC compressor to send the compressed packets, also the decompressor makes use of it, to send the feedback information. On both, compressor and decompressor, the ROHC packets can be wrapped by ethernet frame or IPv4 frames, as you can see in figure 6.9 and 6.10. Using libnet is very similar to use pcap. First of all you have to initialize a libnet context, and a libnet `p\tag`, which would be use to assembly different protocol header.

```
libnet_t *l
libnet_ptag_t eth
```

The next function is called to setup the libnet socket to send the data packets. The `int injection_type` specifies if you want a socket on the TCP/IP stack or if you want to bypass the stack and write your own data to the data link layer (`LIBNET_RAW4` or `LIBNET_LINK`). To inject ethernet frames you need to use the `LIBNET_LINK` option and to build IP4 packets `LIBNET_RAW4`. The `*device` char points to a string containing the device name, usually `eth0`.

```
libnet_t* libnet_init (int injection_type,  
char *device, char *err_buf)
```

To generate the packets a whole bunch of functions is provided. For this project only two of them were used. The `libnet_build_ethernet()` and the `libnet_build_ipv4()`. Both of them have a extensive list of arguments. The detailed discussion is available at source ¹. Just to mention the most important arguments: `u_int8_t prot` that contains a specific number to identify the packet on the decompressor side. With the pointer to the `u_int8_t payload` we submit our ROHC packets.

```
libnet_build_ipv4(u_int16_t len, u_int8_t tos, u_int16_t id,  
                u_int16_t frag, u_int8_t ttl, u_int8_t prot,  
                u_int16_t sum, u_int32_t src, u_int32_t dst,  
                u_int8_t *payload, u_int32_t payload_s, libnet_t *l,  
libnet_ptag_t ptag);
```

Quite the same applies to the `libnet_build_ethernet()` function. Here the `type` field is used to identify the ROHC packets.

```
libnet_build_ethernet(u_int8_t *dst, u_int8_t *src,  
                    u_int16_t type, u_int8_t *payload, u_int32_t payload_s,  
libnet_t *l, libnet_ptag_t ptag);
```

To clean up and close the socket opened with libnet you simply have to call the void `libnet_destroy(libnet_t *l)`.

¹<http://libnet.sourceforge.net/>

Chapter 7

Test scenario

7.1 Scenario design

A test scenario has been realised which looks like in 7.1. On the client the compressor is running. The decompressor runs on the server. The router is configured with NIST-net ¹. The computers are connected over crossover

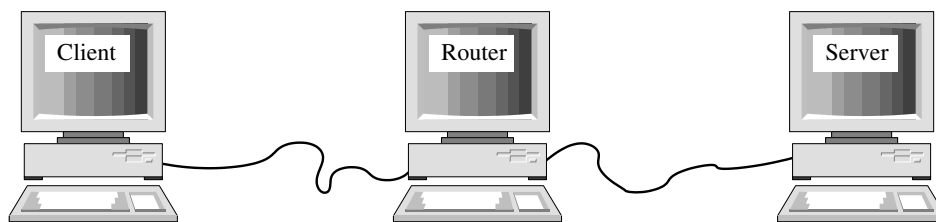


Figure 7.1: Scenario situation

ethernet cables. That it is possible to route the packets with NIST-net, it's necessary to send IP packets instead of ethernet packets. That means all ROHC packets, normal as well as feedback, are wrapped in IP packets.

To generate an RTP stream, the free available Linphone ² is used on the client and the server.

On the NIST-net router different parameters could be emulated. So it's possible to insert a latency, packet loss, and many more. For this test scenario it was interesting to limit the bandwidth. If only a small bandwidth is useable, the ROHC compressed stream must be much better than the uncompressed one.

Also an interesting case is when some bits are changed. The real advantage of ROHC will be visible when a specific bit error rate could be used. Then the probability to receive a unuseable packet is much smaller, because the packets are smaller itself. But with NIST-net it's only possible to gener-

¹<http://snad.ncsl.nist.gov/itg/nistnet/>

²<http://www.linphone.org/>

ate packet loss. Against packet loss, ROHC has no advantage. Because no packet is coming, it couldn't be repaired.

The ROHC performance was tested also without any bad influences. This is nearly the same situation as when a direct connection (7.2) is available. Using this scenario, it's recommended to send the packets direct over ether-

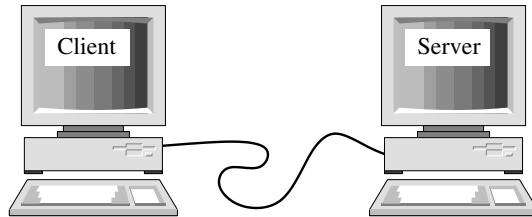


Figure 7.2: Direct connection

net, without any IP headers.

7.2 Functionality

The test scenario with the NISTnet router didn't work until the very end of this diploma thesis. Without this router, the connection was errorless. Everything works great.

7.3 Results

The performance of the ROHC link is very good. A lot of bandwidth can be save to transmit exactly the same data. The feedback was received successful too.

Unfortunately the robustness couldn't be tested, because the connection over the router didn't work.

7.4 Configuration

On the test computers the Linux distribution Fedora Core 4 was installed. The kernel version was 2.6.11 and 2.6.14. The version of the libpcap library was 0.8.3-13. The one of the libnet library 1.1.2.1-6. The version of NISTnet is 3.0a. Linphone has version 1.0.1.

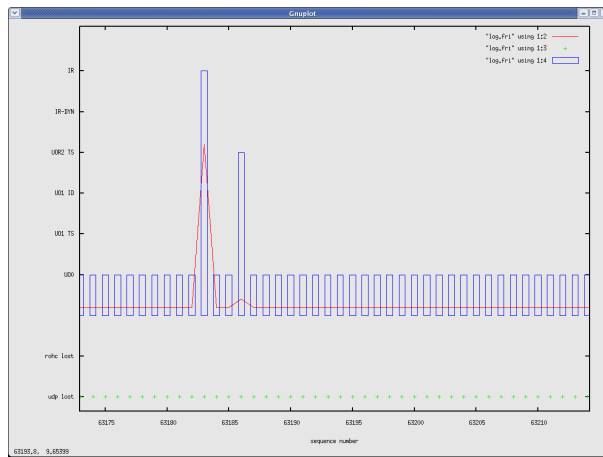


Figure 7.3: Decompressor Graph

Chapter 8

Conclusion

8.1 Results

- A implementation of the compressor and decompressor in unidirectional mode according to the standard RFC 3095 has been realised.
- For a comfortable handling of this parts, a GUI has been developed.
- To show the activities of a ROHC transmission, a self-written plotting application is used.

8.2 Future Challenges

- The integration of the bidirectional optimistic and reliable mode could be the next step
- With this, a extended feedback handling is desired.
- Also a very interesting task will be to test the standart over a link where bit errors could be emulated, because of the smaller size of the packets the ROHC standard will be much better.
- The functionality of the error correction has never really been tested. So doing this could bring more cognitions about the robustness.

8.3 Difficulties

Because of working with a unknown operating system and completly unknown software tools and working environment, a lot of troubles occur. Most of them could be solved but with a high expenditure of time.

- Installation of existing sourceforge project needs a lot of time and ended unsuccessful.

- To get the basic knowledge for working with Linux was time-consuming.
- Just as much to getting well with the developing environment
- Trying to bring up a wireless LAN connection wasted a lot of time.
- Writing the english documentation.
- Using a unknown text manipulation program which is called \LaTeX

8.4 Closing words

This work was starting with a lot of theory about the ROHC and its related standards. The first try to realise this exercise failed because the needed software could not be installed. After this flop, a own implementation was the only way to realise this to achieve this exercise. But the fact that a distinguish part of the available time has been wasted, makes it clear that only a part of the standard could be implemented. And so the decision was to implement the unidirectional mode of the standard.

During this work a lot of time was also spend to Linux specific problems. Because of the fact that the authors was not familiar with this operating system. Several software projects which was tried to install failed because of wrong or too new kernel versions respectively dependent software modules and probably too new hardware.

But with this work we get a lot of new knowledge about Linux and programming with it. We got to know not only the advantage of Linux but unfortunately also the realization that sometimes it can be hard to install new devices. Especially if you do not find the convenient drivers for the hardware and the kernel version.

Further we could refresh and extend our knowledge about network standards. To study the RFC was not easy, because we found this paper is written in a very abstract style. But it was good experience to look to the discussions at the mailing list. So you got a lot of different possibility in which way the problem could be solved. At the end we can say it was a interesting and satisfying work despite to the various problems that occurs. We think we solved the exercise with a satisfactory result even we could not settle the whole implementation of the ROHC standard.

Dino Mani	Markus Gaugler	Christoph Frehner
dmani@hsr.ch	mgaugler@hsr.ch	cfrehner@hsr.ch

Chapter 9

Glossary

CID	context identifier
CRC	cyclic redundancy check
FC	full context state of decompressor
FO	first order state of compressor
GUI	graphical user interface
IP	internet protocol
IR	initialisation and refresh state of compressor
LAN	local area network
LSB	least-significant bits
NBO	flag indicating whether the IP-ID is in network byte order
NC	no context state of decompressor
O-mode	bidirectional optimistic mode
RFC	request for comments
R-mode	bidirectional reliable mode
RND	flag indicating whether the IP-ID behaves randomly
ROHC	robust header compression
RTP	realtime transport protocol
SC	static context state of decompressor
SO	second order state
SSRC	sending source. Field in RTP header
TS	timestamp
UDP	userdatagram protocol
U-mode	unidirectional mode
WLAN	wireless local area network
W-LSB	window based LSB encoding

Bibliography

- [1] **Author:**J. Postel
Title:*User Datagram Protocol (UDP)*
August 1980
<http://www.ietf.org/rfc/rfc768.txt>

- [2] **Title:***Internet Protocol (IP), Protocol Specification*,
September 1991
<http://www.ietf.org/rfc/rfc0791.txt>

- [3] **Title:***Realtime Transport Protocol (RTP)*,
January 1996,
<http://www.ietf.org/rfc/rfc1889.txt>

- [4] **Authors:**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,
H.Zheng
Title:*RObust Header Compression (ROHC)*,
July 2001
<http://www.ietf.org/rfc/rfc3095.txt>

- [5] **Authors:** R. Gern, D. Weber
Title:*NETWORK EMULATOR FOR CDMA2000 & GPRS/GSM*
December 2001