Optimal syntax for packet header compression specification

Julije Ozegovic
University of Split, FESB Split
Split, Croatia
E-mail: julije.ozegovic@fesb.hr

Abstract: **Packet header compression is an important issue for integration of cellular wireless networks with global Internet. Lot of effort to develop robust compression scheme is recently undertaken. Among the others, EPIC and EPIC-lite schemes take approach of formal compression specification. In this paper, based on practical implementation experience, the EPIC-lite scheme extension with optimal syntax for packet header compression specification is proposed.**

1. INTRODUCTION

Former widespread of wireless cellular networks and Internet brings to the front issues of their integration. Internet stack of protocols, designed for flexibility some three decades ago, do not perform well in cellular wireless environment because of poor payload to packet ratio. In particular, IP/TCP for data and IP/UDP/RTP for voice, with their 40+ octets of header, can waste up to 75% of valuable radio bandwidth.

Since it is not likely to change protocol stack in global Internet, wireless networks should be enhanced to transport Internet packet more efficiently. Idea to develop special protocol stack for wireless area with gateways (proxies) connecting it to Internet is abandoned in favor of Internet packet header compression.

Header compression approach has advantage of exploiting strict format of header fields, which can yield much better compression ratio compared to conventional methods based on redundancy. In large proportion of flows that utilize short packets, header compression is sufficient. In some flows, like HTTP documents, compression based on redundancy can be applied to payload part of the packet.

This paper is organized as follows. In Chapter two, an overview of header compression schemes is given. In Chapter three, the proposed syntaxes for header compression specification are analyzed. In Chapter four, the experience gained through the experimental implementation of EPIC-lite scheme is disclosed. Chapter five is used to propose optimal syntax for header compression specification. Conclusions and future work are given in Chapter six.

2. PREVIOUS WORK

CSLIP [8] was inspired with the need to obtain ergonomic response times in interactive data traffic like Telnet: a 200 ms character echo on serial line connection. It is interesting that actual numbers are very close to voice-data integration constrains. No error recovery mechanism was provided.

Deflate [7] is a proposition for general data compression on Internet. It uses standard LZ77 [17] compression algorithm combined with Huffman encoding of control data (literals, distances, and lengths). Both are later considered for packet header and payload compression.

IP header compression (IPHC) [5, 6] is a major improvement of previous attempts. It introduces concepts of context, context identifier (CID), encoding method and robustness. Robustness against packet losses is achieved by "twice" algorithm and feedback from the decompressor. IP/TCP and IP/UDP headers are thoroughly analyzed, including TCP options and IPv6 extension headers.

Compressing IP/UDP/RTP Headers [2] is generally extension of [8]. However, table encoding is proposed, and idea of multiple contexts is introduced. Heuristics of detecting the RTP header existence is elaborated.

Robust Header Compression (ROHC) [1] is a proposal of the IETF ROHC working group. The main contribution to previous work, especially [2], is in robustness enhancement, and in separation of rules and profiles.

Robustness is achieved by introducing three modes of operation, unidirectional (without decompressor feedback), bi-directional optimistic and bi-directional reliable. Multiple contexts per flow are specified, where r contexts can provide safe decompression after r-1 consecutive packet losses.

ROHC framework defines packet formats, which serve as common platform for transport of packets with compressed headers. Various protocol headers can be compressed using appropriate "profiles", set of rules used to achieve maximal compression rate. Itself, [1] defines ROHC uncompressed, IP/UDP/RTP, IP/UDP and IP/ESP in textual form.

3. HEADER COMPRESSION SPECIFICATION

In header compression schemes mentioned earlier, no special effort on formal specification of compression process was involved. In CSLIP [8], C language reference implementation is given. In Deflate [7], a mixture of text, C sample functions and pseudo code is used. In IPHC [5, 6], textual description is used with one algorithm in pseudo code. In [2] and ROHC [1], text with tables is used.

The first atempt to introduce formal header compression syntax is UHCF (Unified Header Compression Framework) [10]. Pearl like specification language is proposed. The major effort in formal compression specification is presented in EPIC-lite (Efficient Protocol Independent Compression) draft [12, 14]. Following the conclusions of 51st IETF meeting, authors of EPIC-lite are proposing modified input syntax presented in Formal notation draft [13]. Simultaneously, Internet draft about Generic notation [9] is issued. All three use modified Augmented Backus-Naur Form (ABNF) [4].

3.1 UHCF syntax

UHCF [10] is a complete framework, actually providing formal compression specification for CSLIP [8] engine. The specification consists of Field, Rule and Action instructions in Pearl like syntax. Profile header comprises protocol identifiers for compressed and uncompressed packets.

Field instructions are used to implicitly define header fields, by specifying their lengths, encoding methods and methods parameters if appropriate. For INFERRED method, a formula written in C should be linked. For DELTA, negative increment can be forbidden to force full header transfer in case of retransmission.

Rule instructions can be used to filter out packets from other protocol stacks (to be sent uncompressed), and Action instructions to initiate particular action when condition is satisfied. State values can be stored in variables.

UHCF syntax is presented on simple IPv4 profile, and more elaborate TCP/IPv4 without options is appended.

3.2 EPIC-lite syntax

EPIC-lite [12] is a complete header compression system under ROHC framework. The independence is obtained by universal specification language used to formally specify a profile, a set of rules used to compress particular protocol stack headers.

Use of ABNF syntax provides definition of rules by combining previously defined rules, or primitive rules (called toolbox methods). The inner interpreter, which executes the rules, is a stack oriented machine with four stacks: uncompressed_data, control, compressed and unc(ompressed)_fields. Some explicit stack-manipulation methods are provided. Benefit of using stacks is smooth parameter passing. Drawback is low readability of formal specification, and usage of stack manipulation methods for complex processing.

Main EPIC-lite strength is "OR" operator to perform choice of methods or rules. It provides possibility to encode particular field in optimal manner. The chosen set of methods for processed header is called "format". Each format is identified by Huffman code token, calculated using the probability of format. To achieve format probability, each method is assigned probability proportional to its usage frequency. Huffman tokens guaranties optimal encoding.

Unfortunately, complex protocols like IP/TCP end with numerous formats, some of them with extremely low probabilities and therefore long Huffman tokens. These formats are discarded, and replaced with partially compressed header (IR-DYN). To provide acceptable compression of flows that behave differently, formats can be grouped in format sets. In this case, format discard is performed on set basis.

Special case is with ORed rules. Rule can include multiple field compressions, and fail if any of the field compressions fail. In that case, interpreter must go back to the OR point, and try next ORed rule. Going back is complicated because stacks must be restored to original content at the OR point.

Multiple format sets are defined using FORMAT method, a specific OR point where format set choice is signaled to the decompressor. Number of format sets equals product of possible choices at various FORMAT points.

Protocol options (like in TCP) are handled with LIST method, in combination with OPTIONAL and U methods. List method provides possibility to encode optional fields, and indicates option presence and order to the decompressor. Unfortunately, order code requires N*logN bits for N possible options, which is frequently more than 32 bits. Additional complication is that various option combinations can dramatically increase number of possible formats, thus lowering their probabilities. To avoid this, U method redirects compressed fields to unc_fields stack, whose content is appended to the compressed header. This way, compressed options are out of the compressed header, and are not covered by Hufmann code tokens. To process them, signaled order and presence codes are used.

Among exceptions, let us mention IP checksum and NBO (Network byte order). IP checksum can be recalculated at the decompressor. NBO tests a field for swapped bytes, and reverts them if necessary. This is indicated by NBO bit left on the uncompressed_data stack under the tested field.

3.3 EPIC-lite formal notation extension

EPIC-lite original specification was criticized because of explicit usage of stack manipulation methods. To avoid this, Formal notation draft [13] provides new LABEL method.

Treating original header as uncompressed_data stack in EPIC-lite [12], header field processing order is determined by the physical field order of the header. Processing of the particular header field can be postponed by storing it on the control stack, and retrieving it back when necessary (with possible rotation of the control stack).

In Formal notation draft [13], a variable is created using LABEL method, and field content is stored there. NEXT-FIELD method reads content of the specified label, and pushes it on top of the uncompressed_data stack.

Using LABEL method, special MSN (Master Sequence Number) methods in EPIC-lite became obsolete. Predefined (reserved) label MSN is used instead. LIST and UNCOMPRESSED methods are redefined to use labeling mechanism.

3.4 Generic notation

Generic notation [9] presents alternative solution of input syntax and consequently virtual compressor-decompressor machine. Based on EPIC-lite [12], and using ABNF, it introduces several new concepts. The profile declaration consists of two parts: the uncompressed (original) header format and compressed header format.

To solve the problem of stack manipulation, a concept of variable is introduced. In uncompressed header declaration, each header field is given a name. Instead of removing some bits from the stack, field names are used for explicit reference. Reference can be done in almost any order, which makes field compression ordering possible. Field names are hierarchical, and follow structure of the header (i.e. protocol.field). The BITS and VALUE methods are used to declare permanent fields in the header. However, only length (number of field bits) is declared, which means that position of the field is relative to all previously declared fields. This is impractical for optional fields (e.g. TCP options), which are declared inside LIST structure. IP options are not covered in presented example.

In Generic notation the same idea from EPIC-lite, of Huffman tokens usage for format identification, is exploited. Compressed header format is defined implicitly with field compression ordering, and compression methods usage. Toolbox methods, similar to EPIC-lite ones, are redefined to use concept of variable names.

The first significant difference is lack of FORMAT method. Instead, compressed header is organized as hierarchical structure of subheaders. Each subheader includes Huffman tokens of its own, and takes the role of single entry in higher layer header. New HUFFMAN and P (probability) methods are introduced to define Huffman coding using the profile source. This gives profile author freedom to use Huffman tokens at will.

Options are covered with new SUBTREE methods, which exploit LIST structure from the uncompressed header declaration. The published example shows complexity of the problem and proposed solution.

4. EXPERIENCE FROM EXPERIMENTAL IMPLEMENTATION OF EPIC-lite

Recently, a project to realize experimental implementation of EPIC-lite [12] was started on University of Split, FESB Split. It resulted with working application that performs file-to-file compression and decompression of captured packet flows [3, 16, 11]. Full EPIC-lite profiles for IP/TCP and IP/UDP/RTP are tested. Compatibility is achieved by successful interchange of compressed files between FESB and authors of [12].

The overall results from the experiment show, that EPIC-lite is complete specification, and that can be used to realize independent software implementation. To accelerate application testing, several simplified profiles were created successfully. More complex functions were included and tested gradually. For final testing, twelve RTP and eight TCP traces were used, from 20 to 500 packets long.

While creating test profiles, the strengths and weaknesses of EPIC-lite BNF syntax were observed. Strength results generally from freedom that BNF notation gives to the profile writer. However, this freedom can be harmful, because profiles that conform to BNF formally, may not always specify correct header processing. This is especially true considering implicit uncompressed_data stack and explicit control stack manipulation. Writing programs with stack machine requires from programmer to keep track of stack content during program development and testing. This is further emphasized with multiple stack machines.

In EPIC-lite, encoding methods simply remove specified number of bits from input_data stack, and push compression result on compressed stack. This means that order of processing must strictly follow the structure of the header. Compressed header takes the same order. Exceptionally, particular field content can be transferred to control stack, and processed later. This results with profile programming errors, without possibility for profile parser to detect them.

Next, ORed rules require special method implementation, which is not specified in draft [12], see [16]. While it is implementation specific, such a method should be declared in future standard to simplify implementation. Also, FORMAT choose function is implementation specific, resulting in

different compression efficiency from different implementations. Some guidelines should be included in future standard.

The processing of fields of variable length is solved with introduction of unc_fields stack, which is also used by the U method inside the LIST/OPTIONAL structure. Implementation of this part was very complex, though successful.

Despite the complexity of some structures, experimental implementation is based on optimized, extended linear profile data structures [3].

5. OPTIMAL SYNTAX FOR HEADER COMPRESSION SPECIFICATION

Based on previous work and experience from EPIC-lite experimental implementation, we propose new syntax for header compression specification. The goals of new syntax are as follows:

1. declare format of uncompressed header explicitly
2. enable random access to uncompressed header fields, except for optional fields
3. declare method parameters literally to simplify referencing
4. declare profile and context check fields for packet classification, as well as protocol identifier fields
5. optimize inter method and rule communication
6. introduce control structures, e.g. IF
7. keep data structures as simple as possible, no unnecessary hierarchy
8. provide for safer profile programming and possible formal profile verification

Declared goals can be obtained if uncompressed and compressed headers are treated as simple list of fields, what they actually are. Header compression and decompression should not differ much from regular header processing in protocol stack software.

5.1 Compression ordering

One of important issues is the problem of compression ordering. Treating input data as bit stack, implies compression ordering to match field structure of the header. This structure was not designed for optimal compression.

One can require random declaration of header fields, random access to them to provide arbitrary compression ordering, and then to reorder compression results in arbitrary manner. Evidently, all that freedom is redundant. Especially, it is not needed to reorder compressed header fields, because it is natural to decompress the header in reverse order than the compression was done.

In EPIC-lite, Huffman tokens refer to list of compression methods actually used to yield compressed header. Free reordering of compressed field gains unnecessary complication in Huffman tokens. The simplest way is to keep compressed fields order as generated at the compression time. Random access to uncompressed header fields provides enough freedom for optimal compression ordering.

To conclude, uncompressed header fields should be declared as random access variables, and compressed header can be realized as implicit bit stack. This simplifies relation between compressor and decompressor, and is in conformance with BNF notation.

5.2 Field declaration

Considering header is a list of fields, each field can be declared with its offset from the beginning of the header and its length, both in bits:

<field_name> = FIELD(<offset>, <length>)

This rule declares simultaneously variable "field_name", which can be referenced at will. The moment of reading field content to variable is implementation specific. Possible solutions are to read all fields at the beginning of header processing, or at the first field reference. Length parameter is needed, because fields lengths are of arbitrary value.

Some fields have predefined values for protocol stack, and are declared using:

<field_name> = FIELD-N(<offset>, <length>, <value>)

Such fields can be used for profile check.

Some fields are located after the variable part of the lower layer header (e.g. after IP options). Since next header offset is known, this information should be used easily providing expression instead of the <offset> parameter:

IHL = FIELD(4,4)

ackno = FIELD(IHL*32+64, 32)

Some fields are optional, and their position is unknown. Such fields are declared with position of first option position, and can be accessed inside option processing structure:

<field_name> = FIELD-O(<offset>, <length>)

<length> of zero can be declared for varying lengths.

It should be possible to declare variables for general use. Here we propose not to declare length, but length information can be stored at the time of initialization.

<variable_name> = VARIABLE

Variable can be initialized at any time:

<rule_name> = STORE(<variable_name> , <expression>)

5.3 Profile declaration

Beyond parameters specified in EPIC-lite, profile header should contain names of rules for profile and context check, protocol identifiers, and field and variable declarations:

```
.........
CO_packet              = <CO_packet_rule>
IR_DYN_packet          = < IR_DYN _packet_rule>
IR_packet              = < IR _packet_rule>
profile_check          = < profile_check_rule>
context_check          = < context_check_rule>
protocol_unc           = <value>
protocol_comp          = <value>
<field_name>           = FIELD(<offset>, <length>)
.........
<variable_name>        = VARIABLE
.........
<profile_check_rule>   = version protocol
<context_check_rule>   = sadr dadr sport dport
```

5.4 Method parameters

Introduction of field declarations, requires redefinition of toolbox methods. Some changes are simple:

```
<rule_name> = IRREGULAR(<field>, <probability>)
<rule_name> = STATIC(<field>, <probability>)
<rule_name> = VALUE(<field>, <value>, <probability>)
```

Using VALUE method, field is checked for value.

```
<rule_name> = LSB(<field>, <length>, <offset>, <probability>)
```

In LSB, field is transferred by length LSB bits.

Some methods are more elaborate:

```
<rule_name> = INFERRED-IP-CHECKSUM(<field>, <length>)
```

where <length> can be declared as IHL*32. In decompressor, IP checksum is recalculated on declared header and inserted in the <field>.

```
<rule_name> = INFERRED-OFFSET(<field1>, <field2>,
                                    <variable_name>)
```

Difference field1-field2 is calculated and stored in variable, which is than compressed using appropriate method.

```
<rule_name> = INFERRED-SIZE(<field>)
```

All parameters needed are contained in <field> declaration, size is recalculated and inserted in <field>.

```
<rule_name> = NBO(<field>, <variable_name>)
```

<field> is checked for network byte order, and modified if needed. Action is indicated with flag stored in <variable_name>.

```
<rule_name> = SCALE(<field>, <v_quotient>, <v_reminder>,
                                    <v_modulo>)
```

<field> is scaled with appropriate modulo, and quotient, reminder and modulo are stored in appropriate variables for later compression.

5.5 ORed rules

As mentioned earlier, BNF notation permits ORed rules. Each OR branching point involves format tree branching, covered by Huffman token.

Besides, Huffman token should cover OR points inside the selected rule. This is possible if all ORed rules deal with the same set of fields. If different fields are involved, current EPIC-lite method of calculating ID bits could fail.

In stack oriented environment with destructive stack pops, stack restoration to original value at OR point is needed in case of method failure. If rule involves compression of several consecutive fields, failure of one compression causes rule to fail, and multiple bit patterns should be restored on the stack.

With header field variables, reading is non-destructive. ORed rules of arbitrary length are inherently supported.

However, OR operation involves rule (or method) trying in order which is implementation specific. For single method this can be acceptable, but for complex rules processing efficiency can be disturbed. Introduction of specific control structure can be considered.

Here we discuss the possible use of conditional branching (IF structure). Suppose that some field or group of fields carry information if specific bit field is set, otherwise it carries zero. In that case, branching decision can be done by testing the (bit) field for non-zero value:

```
<rule_name> = IF(<field>, <rule_true>, <rule_false>)
```

This concept can be extended to field content testing:

```
<rule_name> = IF(<field> = <value>, <rule_true>, <rule_false>)
```

Multiple consecutive IF statements of this kind can serve as primitive CASE structure. To make them coherent structure, usage of ELSEIF, ELSE and ENDIF can be preferred:

```
<rule_name> =
        IF(<field> = <value>, <rule_true>)
        ELSEIF(<field> = <value>, <rule_true>)
        ELSEIF(<field> = <value>, <rule_true>)
        ..............
        ELSE(<rule_false>)
        ENDIF
```

The concept of Huffman token could be applied to single or multiple IF statements for branching decision signaling. However, if branching condition field is encoded after the IF structure, it is known to the decompressor before the IF structure is executed. In that alternative case, no Huffman token bits need to be used for IF structure.

We propose to keep Huffman token signaling of choice for IF structure, to avoid possible errors in profile and achieve compatibility with ORed rules. The same restrictions covering OR points inside selected rules remain.

5.6 Optional fields

The problem of optional fields coding is problem of signaling selected OR choices in variable number of fields environment, combined with arbitrary option occurrence and ordering. Currently considered optional fields found in [12, 13, 9] are TCP options, TCP sack option blocks, RTP CSRC entries and SCTP chunks.

Generating Huffman tokens for optional fields is tricky, because large number of possible combinations exists. This leads to header formats with very small probabilities, and prohibitively long tokens are needed. Even partial header encoding is not possible because of the same reason.

In EPIC-lite [9, 12], compressed fields are redirected by U method to unc_fields stack, whose content is appended after the compressed header of known length. Huffman token does not cover content appended. Decompressor reads appended data when needed, using length information encoded in compressed header part. Huffman tokens are created as if actual data order conforms with one in profile, while presence and order codes are provided to decompressor, to indicate actual ordering. Tokens cover all optional entries, which means that multiple occurrences are not possible if not explicitly included.

In Generic notation, subheaders are provided for option coding. Authors claim that this approach solves the order and presence signaling problem, but no practical confirmation have been published yet.

Here we try to propose optimal syntax for header compression specification, not the actual algorithm itself. However, syntax change often changes properties of corresponding virtual machine. Considering EPIC-lite is proven in practice, the field specification concept is applied to LIST and OPTIONAL methods.

```
<rule_name> = LIST(<length>, <field>, <v_order>,
                   <v_present>, 1*(OPTIONAL(<rule>,
                   <value>)))
```

Length parameter is actually expression like (data_offset-5)*32. Field parameter contains option identifier code. LIST tries to compare <field> with <value> for each optional entry, and executes the matched one. Order and presence variables are updated. This action is repeated, until <length> bits are exhausted.

Selected rule normally reference FIELD-O data, which means that their position is relative to the special list_offset, initially set to zero (first option). After one option is executed, list_offset is updated by length of the offset. This can be done automatically by tracking FIELD-O maximum offset plus length. LIST then rereads <field>, but now with different list_offset. Actually, first option code field name is reused. This is possible because in known protocols, all options begin with code field of the same length.

Another form of LIST uses "end of options" code:

```
<rule_name> LIST-C(<end>, <field>, <v_order>, <v_presence>,
                   1*(OPTIONAL(<rule>, <value>)))
```

This kind of LIST tries to execute option entries until <end> code is recognized.

Finally, lists of the same entry without code identifiers are found, e.g. sack blocks. LIST-N is based on length parameter:

```
<rule_name> LIST(<length>, <v_order>, <v_presence>,
                   1*(OPTIONAL-N(<rule>)))
```

In previous examples, U method is not used. It seems that concept of unc_fields stack is redundant.

Decompressor reads required amount of bits as indicated by the compression involved, referenced by Huffman token. Presence of these bits is indicated with presence code. It is of no concern whether these bits are included inside the compressed header or after it. In our opinion, U method is redundant, and can be excluded from the specification. The only restriction is that optional fields must be treated under OPTIONAL method.

5.7 Variable length fields

Inside the packet format, sometimes fields or groups of fields of variable length exist. TCP options and SCTP chunks of data are of the kind. In EPIC-lite [9, 12], UNCOMPRESSED method can be used to carry such fields uncompressed, with lower efficiency, instead of complex LIST/OPTION encoding.

Uncompressed fields are appended after the EPIC-lite header, which is possibly redundant like the U method is. Proposed syntax is:

```
<rule_name> = UNCOMPRESSED(<v_length>, <field>)
```

Fields with zero length are normally used, and <v_length> parameter can be variable or field, available for encoding after the UNCOMPRESSED is executed.

5.8 Multiple format sets

In practice, flows of the same protocol stack behave differently. Mention interactive traffic, bulk data transfer or Web browsing, transported using TCP. Covering all this possibilities with normal OR functionality can result with large number of header formats. It is better to recognize flow behavior, and apply optimal set of compression formats.

In EPIC-lite [9, 12], FORMAT method is proposed. Here, the same concept is modified to use variables:

```
<rule_name> = FORMAT(<v_selection>, <rule>, 1*<rule>)
```

Format selection result is stored in selection variable, and available for later compression.

5.9 Compressing payloads

Using ABNF, compression of arbitrary length payloads can be specified. This is possible, because variable length fields are already transferred using UNCOMPRESSED method. Providing some compression scheme is adopted, e.g. LZ77 [17], the ABNF rule might look like:

```
<rule_name> = LZ77(<length>, <v_length>)
```

where length bits should be compressed, and total length obtained is stored in variable for subsequent compression.

This possible solution implies that LZ77 is a toolbox method, and that payload is just another field under the EPIC-lite framework. Instead of predetermined compression algorithm, optimal one can be specified using e.g. UDVM [15] byte code.

6. CONCLUSION

In this paper, an overview of proposed header compression standards and compression specification syntaxes is given. Based on experience from experimental realization of EPIC-lite [12], optimal syntax for header compression specification is proposed. It is based on EPIC-lite concept of Huffman tokens header format indication.

The optimal compression ordering is achieved by random access to header fields, and by using implicit compressed stack for compression results. Uncompressed header format is declared independently in profile header, together with other variables, protocol identifiers, and profile and context check rules. Each header field contains position (offset) parameter for random access, except for optional fields whose position is shifted according to previous options.

Random access to header fields provides easy implementation of multi field ORed rules, because input_data restoration is not necessary. Besides ORed rules, alternative IF/ELSEIF/ELSE/ENDIF structure is introduced for further consideration.

Optional fields encoding keeps the same concept from EPIC-lite, modified to use declared header fields. The possible redundancy of using additional unc_fields stack, appended after the compressed header, is discussed. Compression of payload is proposed obeying the same principles.

Proposed syntax satisfies declared goals. Usage of declared fields yields human readability of the profile, and is basis for its formal verification.

Introduced concepts should be elaborated to full standard proposal. Future work includes experimental realization of the concept and verification for functionality, ease of implementation, profile writing safety, and header compression efficiency versus complexity. Comparison to other proposals [12, 13, 9] will be performed.

REFERENCES

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

[2] Casner S, Jacobson V, "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links", RFC 2508, 1999.

[3] Cizmic, M., Vodopija, T., Ozegovic, J.: "EPIC Lite offline processing", SoftCOM 2002.

[4] Crocker D, et al: “Augmented BNF for Syntax Specifications: ABNF”, RFC 2234, 1997.

[5] Degermark, M., Engan, M., Nordgren B, Pink S, "Low-loss TCP/IP Header Compression for Wireless Networks" ACM/Baltzer Journal on Wireless Networks, vol 3, no 5, 1997.

[6] Degermark M, Nordgren B, Pink S.: "IP Header Compression", RFC 2507, 1999.

[7] Deutsch, P. "Deflate", RFC 1951,1996.

[8] Jacobson, V. "Compressing TCP/IP Headers for Low-Speed Serial Links", RFC 1144, 1990.

[9] Liao, H., Zhang, Q., Zhu, W.: "Generic Header Compression Notation for ROHC", draft-liao-rohc-notation-00.txt, 2002.

[10] Lilley, J., Yang, J., Balakrishnan, H., Seshan, S.: "A unified header compression framework for low-bandwidth links", Proceedings of the sixth annual international conference on Mobile computing and networking, p.131-142, Boston, 2000.

[11] Mornar, M., Pezelj, A., Ozegovic, J.: " Testbed for header compression implementation ", SoftCOM 2002.

[12] Price, R., Hancock, R., McCann, S., Surtees, A., Ollis, P., West, M.: "Framework for EPIC-LITE", draft-ietf-rohc-epic-lite-01.txt, 2002.

[13] Price, R., Surtees, A., West, M.: "A Formal Notation for Header Compression", draft-west-rohc-formal-notation-00.txt, 2002.

[14] Price, R., Surtees, A., McCann, S., West, M., Hancock, R., Findlay, D.: "EPIC Provably Optimal Format Encoding for Compression in the Internet", SoftCOM 2002.

[15] Price R, et al, "Signalling Compression (SigComp)", RFC 3320, 2002.

[16] Stula, M., Vidjak, L., Ozegovic, J.: " Program structures for EPIC-LITE experimental implementation ", SoftCOM 2002.

[17] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.