EPIC Lite offline processing

Mia Cizmic*, Tijana Vodopija**, Julije Ozegovic***
*KRON d.o.o Split
**PSE-Split Siemens d.d.
***University of Split, FESB Split
Split, Croatia
E-mail: mia.cizmic@kron.hr, tijana.vodopija@siemens.hr, julije.ozegovic@fesb.hr

Abstract: **Header compression has always been an attractive way of conserving bandwidth over low-speed links. Initially motivated to improve the performance of dial-up modems at 14.4 or 28.8 kbps, header compression is actual again, due to proliferation of wireless systems. EPIC Lite is compression scheme, which instead of being based on protocol-specific standard, represents unified framework for header compression. In this paper, based on experience from experimental implementation, offline processing results are presented and improvements are proposed.**

1. INTRODUCTION

Current research in the field of packet header compression is focused on robustness and formal specification. The last is pioneered by Roke Manor Research - Efficient Protocol Independent Compression (EPIC Lite) scheme [7]. The packet header and thus compression algorithm are defined as "profile", using ABNF formal notation [8]. In such a system, compression is performed in two phases: offline phase to interpret protocol stack profile, and online phase to actually compress (and decompress) packets from data flows.

EPIC Lite scheme [7] is experimentally implemented on University of Split, FESB Split. In this paper, the experience gained for the offline phase is presented, and improvements are proposed. The rest of the paper is organized as follows.

Chapter Two gives introduction to header compression and brief comparison of EPIC Lite and other header compression schemes. Offline and online processing phases are described in Chapter Three followed by profile processing description and proposition of descending sort. Implementation specific solutions related to offline phase are presented in Chapter Four, together with a proposition of offline improvements. Given improvements are related to decreasing time and memory consumption in offline phase, and have no impact on compression ratio.

2. HEADER COMPRESSION

The protocol overhead can significantly increase the packet size of interactive applications that use small packets. It can sometimes take up to 75% of the total network capacity, which is unacceptable for mobile networks. Header compression takes advantage of the precise header format and known behavior of header fields to achieve significant reduction of header size.

One of the earliest examples of header compression was the Thinwire protocol [1], proposed in 1984. It uses a simple 20-bit field to specify which of the packet header's first 20 bytes have changed. Thinwire was relatively protocol independent, but was not optimized for most common protocol headers.

In early 1990, Van Jacobson [2] proposed TCP/IP-specific compression algorithm with many optimizations taking advantage of this protocol stack field behavior. Sending between 3-5 bytes of the 40-byte header in the common case, Van Jacobson is efficient and the most widely deployed header compression protocol. However, it works only with TCP/IP packets, and does not provide robustness.

Since then, compression specifications for a number of other protocols have been written. Degermark proposed additional compression algorithms for UDP/IP and TCP/IPv6 [4]. Detailed specifications for compressing these protocols, as well as others, such as RTP [9] were described in several subsequent RFCs including 2507 [10], 2508 [3] and 2509 [11]. Each of these descriptions specifies a solution for a given protocol.

Despite the development of various compression schemes in recent years, header compression deployment failed. Former schemes were dominantly purposed for slow serial lines, where expected error rates are low thanks to the data link layer protocol functionality. Currently, usage of header compression is expected in the wireless network area, where error rates and round trip times are high. More robust schemes are needed. Development efforts are concentrated

around IETF Robust Header Compression (ROHC) [6] working group.

Unlike known protocol-specific compression techniques, EPIC Lite [7] provides protocol independent specification system for header compression under ROHC framework. It includes a simple human readable profile language, used to describe header properties in high-level manner. Profile language assigns one or more compression methods to each field in the protocol stack to be compressed. Using this input, EPIC Lite derives one or more sets of compressed header formats that can be used to quickly and efficiently compress and decompress actual headers. Identification code is appended in front of the compressed header to communicate used compression format to the decompressor.

3. OFFLINE PROCESSING

3.1 Header processing phases

The idea of header compression is to avoid transferring redundant information. Compression depends on field behavior during the lifetime of a connection. For randomly changing fields, there is not much benefit from the compression. However, there are fields that remain constant, or change slightly from the previous packet value. Some can even be calculated (inferred) from other fields. These fields can be expressed with fewer bits that transfer only the difference, or need not to be transmitted at all.

EPIC Lite toolbox consists of primitive compression procedures called "encoding methods" [7]. One or more encoding methods, along with the probability of usage, are applied to each field by profile. Profile is, in fact, a description of how the chosen protocol stack behaves.

As mentioned before, EPIC Lite header processing has two main phases. First, considered to be an offline phase, is used to convert the input profile into one or more sets of compressed header formats. It is run once. Results are stored at compressor and decompressor.

Second phase, or online phase, is repeated for each packet in the stream. The first step for compressor is to determinate the context relative to which particular packet will be compressed. After choosing right context, state machine could be started. EPIC Lite state machine defines three states (IR, IR-DYN, CO) and determinates which set of header formats will be used. Three state machine states imply three kinds of packets. Beside CO (compressed) packet, there are IR (Initialization and Refresh) and IR-DYN (partially compressed) packets used to obtain and refresh the context.

Depending of stored context values and the state of the machine, compressor chooses between various encoding methods offered by protocol profile. It tries to robustly transfer header content to the decompressor, using as few bits as possible. Each choice of encoding methods maps to unique header format and each header format is represented with unique identification code (indicator flags). Those flags are put in front of the compressed header, Figure 1.

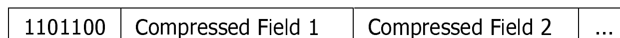

Figure 1 – EPIC Lite compressed header

Indicator flags are version of Huffman encoding [13]. When used in EPIC Lite, the encoded set is simply the set of all possible header formats, where each identifier maps to a compressed header format and vice versa. The property of Huffman coding is that more frequent formats are given shorter identifiers than rarely used ones, which increases compression ratio.

Decompressor reads the indicator flags to determine which compressed header format has been used. This allows the compressed value of each field to be extracted. Using the compressed value of each field and the context, decompressor can apply the encoding methods to reconstruct the uncompressed packet. Fields are decompressed in reverse order compared to compression procedure. This ensures that fields, which are inferred from other field values, are reconstructed correctly. Finally, the decompressor verifies that correct decompression has occurred by applying the header checksum. If the packet is successfully verified, it can be forwarded.

3.2 EPIC Lite input profile

To enable the interoperability between compressor and decompressor, it is important that the EPIC Lite input profiles are specified and processed in an unambiguous manner.

Each EPIC (Lite) profile has eight variables that fully describe profile; three of them (profile_identifier, npatterns, bit_aligment) are used to negotiate towards ROHC framework [6], others are as follows:

max_formats
max_sets
CO_packet
IR-DYN_packet
IR_packet

The max_formats parameter controls the number of compressed header formats to be stored at the compressor and decompressor.

The max_sets parameter controls the total number of sets of compressed header formats to be stored. Profile can have several sets of compressed header formats, but only one set may be in use at a given time.

CO_packet, IR-DYN_packet, IR_packet are encoding methods (in the rest of the paper called profiles) used to

generate CO, IR-DYN and IR packets respectively. Profiles are described using modified BNF (Backus Naur Form) called Augmented BNF [8]. BNF is a "metasyntax" commonly used to describe the syntax of protocols and languages. The language itself is simple consisting of rules and actions, which allow relatively simple parser implementation.

Writing profile consists of describing individual header fields by assigning one or more encoding methods to it. Multiple methods are given using choice ("|") operator, along with the probability that indicates how often encoding method will be used. Encoding method can be one from the toolbox encoding methods, or can be created from toolbox ones by writing a new BNF rule. Therefore, profile has tree-based structure as shown in the Figure 2.

```
Header    = Field_A
            new_rule
            Field_C
Field_A   = STATIC(90%) | IRREGULAR(2,10%)
new_rule  = Field_B1
            Field_B2
Field_B1  = STATIC(100%)
Field_B2  = VALUE(4,1,80%) | VALUE(4,0,20%)
Field_C   = STATIC(70%) | LSB(8,-1,15%) | LSB(16,-1,15%)
```

Figure 2 – Example of profile writing

3.3 Profile processing

Profile provides multiple formats for a protocol stack, depending on the choice of toolbox encoding methods for all fields. For protocol profile introduced in Figure 2, there is a set of 2*1*2*3 = 12 possible header formats. Each header format has a different probability. Since fields are assumed to behave independently, format probability is calculated as product of method probabilities. After all calculations are done, offline processing assigns appropriate indicator flags to each header format. Codes are given using ordinary Huffman algorithm, which assigns the shortest code to most probable header format.

To reduce the amount of memory needed for storing format information, only max_formats header formats are stored at compressor and decompressor per each format set. The formats with the greatest probability to be used are kept. If compressor tries to encode the header using the header format that was discarded because of low probability, state machine would switch to IR-DYN or IR state. Transition to IR state increases compressed header size, but is the trade off between compression efficiency and memory consumption. It is highly recommended for profile writers not to have more than max_formats formats for IR set because it would mean that there are headers which are impossible to compress.

While calculating max_formats header formats, it is not necessary to calculate all possible header formats and than keep the most probable ones. Algorithm offered by Roke Manor is, in fact, tree-recursive algorithm. Formats are generated following the tree structure of BNF specification. At each step, which means inclusion of next tree branch, algorithm combines list of items produced by earlier steps. If new list exceeds the max_formats number, all items with the probability lower then last item probability are discarded because that will not affect the final list order.

During format generation, items are stored in probability based ascending sorted lists. At each step, combined list must be resorted, even if new list of items does not increase number of formats before discarding.

3.4 Descending sort proposition

To apply Roke Manor discarding policy, it is necessary to have probability-based descending sorted lists of items (to be able to compute first max_formats ones). Since lists are kept in ascending order by [7], list must be reversed to descending order before discarding.

In current profile writing practice, multiple methods for a header field are listed in descending probability order. This order is followed when calculating the formats.

Since items enter the list in the way they were written in the profile, and that is dominantly in probability-based descending order, it is somehow natural to keep the list in descending order. This way, number of swapping operations is reduced. Additionally, list is already prepared for discarding procedure, which needs the list in descending order. To optimize offline processing, actually to decrease the number of operations, we propose to keep sorted lists in descending probability order.

Change in sorting order has impact on final list of header formats. As an example, the indicator flags for a set of compressed header formats for earlier profile are given in Table 1.

Header formats are represented as zero-based integer strings. Each integer denotes the enumeration in which toolbox encoding method used to compress appropriate field is written in the profile. Thus, header format 1010 marked with flags 1101100 uses IRREGULAR(2,10%), STATIC(100%), VALUE(4,0,20%), STATIC(70%) encoding methods to compress header fields.

It is obvious that for header formats, which have equal probabilities, computed Huffman codes are the same regardless which kind of sorting was applied. However, they do not indicate the same header format.

FESB University's implementation kept the ascending order for compatibility reasons. We propose the descending sorting to enter the standard.

prob(%)	formats derived by sorting		Huffman codes
	ascending	descending	
0.3	1011	1012	11011111
0.3	1012	1011	11011110
1.2	1001	1002	1101110
1.2	1002	1001	1101101
1.4	1010	1010	1101100
2.7	0011	0012	110101
2.7	0012	0011	110100
5.6	1000	1000	1100
10.8	0001	0002	1011
10.8	0002	0001	1010
12.6	0010	0010	100
50.4	0000	0000	0

Table 1 – Indicator flags for profile from Figure 2

4. IMPLEMENTATION SPECIFIC SOLUTIONS

4.1 Linear profile

Direct interpretation of BNF input language results with profile, which itself has a tree structure. Each node of the profile corresponds to some non-library encoding method, and leaf nodes are toolbox encoding methods.

Figure 3 depicts tree for example profile from Figure 2.

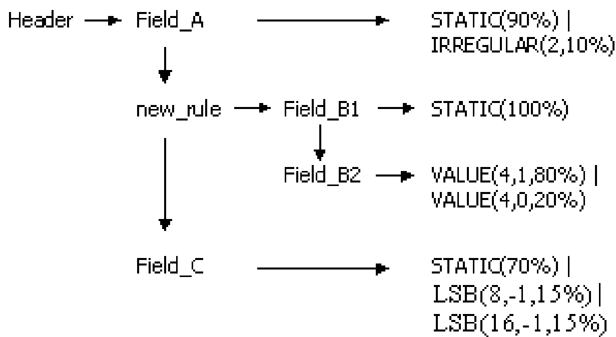

Figure 3 – Tree representation of example profile

Using tree-structured profile, compression is, in fact, tree traversal. Field is encoded only when toolbox method is reached. In this activity, inner tree nodes are redundant, consuming memory and processing time.

To optimize compressor and decompressor operations, FESB implementation is provided with some kind of linear profile. It is structured as double-linked list, which consists from toolbox method's nodes only. Nodes are linked in the order they apply to uncompressed header. Linear representation of example profile is given on Figure 4. More complex encoding methods can also be represented in this way [12].

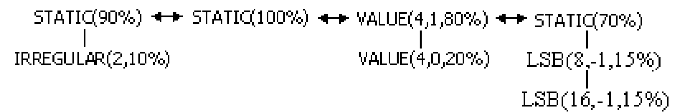

Figure 4 – Linear representation of a profile

Linear profile is created in the offline phase, in parallel with computing header formats set(s). That allows definition of mapping between an integer string and header format as explained in previous Chapter. One-to-one mapping would be much more complicated if the tree structure was dealt with. We propose linear profile to be considered in future standardization process.

4.2 Pre-processing profile proposition

Parsing the profile written in BNF form and converting it to one or more sets of compressed header formats, takes time and processing power due to numerous sorting. Result is a list of max_formats most probable header formats for each set.

Compressor stores information about retained header formats and indicator bits mapped to them. To correctly apply some encoding methods, decompressor needs one extra information, i.e. the compressed header size.

It turns out that each header format is fully described with four parameters: header format identification, compressed header size (in bits), length and value of assigned Huffman code.

All those parts of information can be pre-processed on any platform and appended to the profile. There are two benefits from this approach. First, remaining offline task in compressor - decompressor will consist of parsing the profile specification for data structure generation only. This can be done quickly. Second, the chance to misinterpret indicator bits and header formats will decrease to minimum. Namely, indicator bits will not be calculated, but rather red from the information appended to the profile file.

Proposed format of pre-processed information for the example profile is shown in Figure 5.

```
current_set := 0

1011      | 10    | 8    | DF
1012      | 18    | 8    | DE
...       | ...   | ...  | ...
0000      | 0     | 1    | 0

current_set := 1

...       | ...   | ...  | ...
```

Figure 5 – Proposed pre-processed information format

The keyword current_set denotes the beginning of list of rows, which actually represent header formats retained for given set. Each row has four columns: first column is header format string representation (the same one described before), second is the size of compressed header in bits expressed decimal, third column represents length of indicator bits (decimal), and the last column is the value of indicator bits (hexadecimal). Arbitrary columns separator can be specified, the sign "|" is chosen in the example for human readability.

5. CONCLUSION

Using EPIC Lite to compress new protocol stack, the only needed intervention is to write a suitable protocol profile. Even it is not specially designed for a particular protocol stack, compression ratio obtained using EPIC Lite is comparable to other compression techniques.

Experimental implementation of EPIC Lite, realized at University of Split, FESB Split, proved that efficient compression is obtainable with an application derived from [7].

Experience based on actual implementation motivated some improvement proposals for offline processing phase. Proposed improvements leave the main idea of EPIC Lite unchanged, but could speed up some time-critical parts.

First, descending sort is proposed in the process of calculating max_formats header formats to spare some item swapping. Offline phase can be less time consuming.

Second, linear structure of profile is proposed to optimize online operations (compression and decompression). It allows integer string format representation.

Third, profile pre-processing is proposed, which should append format definition strings and related Huffman indicator bits to the original BNF specification of the profile. Instead of time consuming calculation of max_formats header formats and their indicator flags, offline processing in terminal equipment should only parse BNF file and appended format information to build internal data structures. Additional benefit is the lower probability for terminal equipment to miscalculate formats.

Future work should prove proposed concepts. Current draft proposal [7] needs to be extended, and experimental implementation adjusted accordingly.

REFERENCES

[1] D. Farber, G. Delp, T. Conte, "*A Thinwire Protocol for connecting personal office computers to the Internet*", RFC 914, September 1984

[2] V. Jacobson, "*Compressing TCP/IP Headers*", RFC 1144, January 1990

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

[4] M. Degermark, H. Hannu, L-E. Jonsson, K. Svanbro, "*Evaluation of CRTP Performance over Cellular Radio Networks*", IEEE Personal Communications Magazine, Volume 7, number 4, pp. 20-25, August 2000

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

[6] C. Bormann, C. Burmeister, M. Degermark, H. Fukushima, H. Hannu, L. Jonsson, R. Hakenberg, T. Koren, K. Le, Z. Liu, A. Martensson, A. Miyazaki, K. Svanbro, T. Wiebke, T. Yoshimura, H. Zheng, "*Robust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed*", RFC 3095, July 2001

[7] R. Price, R. Hancock, P. Ollis, A. Surtees, M. West, "*Framework for EPIC-lite*", draft-ietf-rohc-epic-lite-01.txt (work in progress), February 2002

[8] D. Crocker, P. Overel, "*Augmented BNF for Syntax Specifications: ABNF*", RFC 2234, November 1997

[9] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "*RTP: A Transport Protocol for Real-Time Applications*", RFC 1889, January 1996

[10] M. Degermark, B. Nordgren, S. Pink, "*IP Header Compression*", RFC 2507, February 1999

[11] M. Engan, S. Casner, C. Bormann, "*IP Header Compression over PPP*", RFC 2509, February 1999

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

[13] K. Sayood, "*Introduction to Data Compression, Second Edition*", Morgan Kaufmann Publishers, March 2000