Program structures for EPIC-LITE experimental implementation

Linda Vidjak, Maja Stula, Julije Ozegovic
University of Split, FESB Split
Split, Croatia
E-mail: {linda.vidak, maja.stula, julije.ozegovic}@fesb.hr

Abstract: **New header compression techniques are developed, and among them is EPIC-LITE compression scheme, to increase efficiency when transferring Internet packets over the limited cellular radio links. Experimental implementation of EPIC-LITE on University of Split, FESB Split, yielded experience of how to solve particular problems efficiently. In addition, optimal data structures used can be exploited to download preprocessed profiles to terminal equipment.**

1. INTRODUCTION

Internet Engineering Task Force (IETF) Robust Header Compression (ROHC) group is currently working on new compression scheme standardization [2]. The ROHC scheme is designed to compress different packet headers, according to the specified ROHC profiles, over error prone channels.

Efficient Protocol Independent Compression (EPIC-LITE) scheme [1], extends the basic ROHC framework by introducing ABNF (Augmented Backus-Naur Form) [3] based input language for creation of new ROHC profiles. It is currently subjected as a standardization proposal. Under EPIC-LITE, processing can be divide in offline phase, when profile is used to generate data structures, and online phase, when actual compression and decompression takes place.

In this paper, process of developing program and data structures for packet header compression and decompression (online processing) according to the EPIC-LITE framework, and ideas emerged from this activity, are presented.

The paper is organized as follows. In Chapter Two, general EPIC-LITE overview is given. Chapter Three describes some of the basic concepts given in Chapter Two from implementation point of view. Chapter Four is dedicated to the profile. Emphasis is put on implementation and usage of a linear profile structure. Conclusions and future work are given in Chapter Five.

2. EPIC-LITE OVERVIEW

Original packet header is compressed according to the profile, and payload is added to the compressed header. Compressed header doesn't contain just compressed fields. It is extended with Huffman indicator flags, padding bits and uncompressed fields, Figure 1.

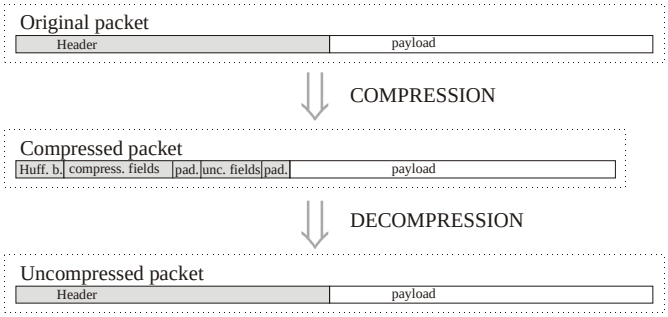

Figure 1 – Compressed and decompresed packet structure

Uncompressed fields are whatever information, compressed or uncompressed, whose format is not covered with indicator flags, but rather with information contained in compressed header itself. EPIC-LITE scheme assures that the original packet can be regenerated from the compressed packet.

The basic parts of compression and decompression are Huffman flags generation, profile, context, encoding methods (EPIC toolbox), and state machine.

Indicator flags or Huffman bits are version of Huffman encoding [4] for each compressed header format. Multiple format exists, because profile writer can specify more than one encoding method for particular field (using "or" operator). The flags are placed in front of the packet to indicate to the decompressor exactly which encoding method has been applied to corresponding header field.

Profile is specification that determines how to compress headers of a certain kind of packet stream over a certain kind of link [2]. Part of a profile for TCP/IP stream is shown on Figure 2.

The context is information that consists of one or more uncompressed header field values from previous packets [2]. The compressor and decompressor both maintain a copy of context, so fields can be compressed relative to their stored values for better compression efficiency. For robustness, compressor can maintain more than one context per flow, and compress headers according to all of them. This way, it is ensured that decompressor can successfully decompress received header, under the sole condition that its context is one of contexts stored at the compressor.

```
TCP-IP-CO      =  INFERRED-IP-CHECKSUM(IPv4-co-header)
                  TCP-co-header
                  CRC(8,100%)

IPv4-co-header =  version
                  header-len
                  tos-co
                  ecn
                  length
                  ip-id-co
                  rf-flag-co
                  df-flag-co
                  mf-flag
                  offset
                  ttl-co
                  protocol
                  ip-chksum
                  src-address-co
                  dst-address-co
version        =  VALUE(4,4,100%)
header-len     =  VALUE(4,5,100%)
tos-co         =  STATIC(99%) | IRREGULAR(6,1%)
ecn            =  IRREGULAR(2,100%)
length         =  INFERRED-SIZE(16,-16)
ip-id-co       =  NBO(16) ; check byte-order
                  FORMAT(ip-id-seq-co, ip-id-rnd)
                  STATIC(100%) ; format selector
                  STATIC(100%) ; nbo flag
```

Figure 2 – Part of TCP/IP stream profile

Encoding methods are functions for compression and decompression of header fields. EPIC-LITE [1] treats ROHC encoding methods as library functions that are called by the modified BNF input language. In [1], the syntax of each encoding method is given using ABNF.

Profile writer determines what method will be used for which field. It is supposed by EPIC-LITE draft that the writer has knowledge about protocol header for which he is writing profile. That knowledge should primarily be about header structure (e.g. at which position field begins, how long the field is in bits) and behavior of a field (e.g. field doesn't change for a flow). Behavior of a field is important to choose the best method for compression of that field.

Header compression [2] can be characterized as an interaction between two state machines, one compressing machine and one decompressing machine. Compressor and decompressor have three states each. Both machines start in the lowest compression state and transit gradually to higher states.

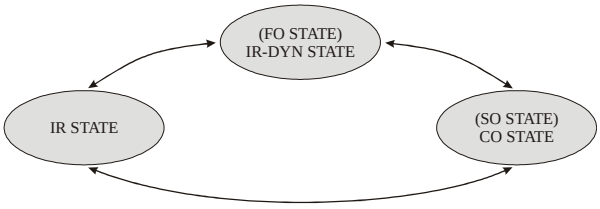

Figure 3 – State machine

According to [1] and [2] there are three states shown on the Figure 3. IR (Initialization and Refresh) state is used to refresh or to initialize static parts of the context at the decompressor. The compressor sends, practically, full header in this state using IR profile. IR-DYN (Dynamic Initialization and Refresh) or FO (First Order in [2]) state is used to transmit the fields that differ between two consecutive packets in a stream. It is used when compressor suspects that decompressor has lost full context. CO (COmpressed) or SO (Second Order in [2]) state is state where compression is optimal. Compressor enters this state when it is sufficiently confident that decompressor has enough information (full context) to decompress a packet.

3. BASIC IMPLEMENTATION ISSUES

EPIC-LITE scheme implies two main phases of processing. Offline processing includes parsing textual profile to generate indicator flags and data structures suitable for online processing [4]. Online processing is used to actually compress and decompress headers.

The experimental implementation described here concerns file to file compression and decompression. Input data for compressor are packets (TCP/IP, RTP) captured to the file, while decompressor uses compressed packet headers from file generated by the compressor.

3.1 Input data

Through EPIC-LITE draft [1] several bit stacks (uncompressed_data stack, compression_data stack, unc_fields stack) and one item stack (control_data stack) are introduced. EPIC-LITE actually specifies multiple stack oriented machine.

Experimental application treats input data as an array of bytes, loaded to the memory buffer in the same order as they were received from the communication channel, currently from the captured flow input file. This approach was chosen to optimize bit manipulations and prepare the code for processing actual packets stored in communications subsystem structures (like Linux *sk_buff* [6]).

3.2 Bit manipulation

Packet header fields have different lengths varying form 1 bit (flags) up to more than 32 (IpV6 source address, some TCP options). Bit manipulation, like extracting 1 bit from a byte, reading bits from 7-th to 14-th bit in 16 bit word etc. is not something that is natural neither to programming languages nor to processors. Processors are made to operate on 8, 16, 32 or 64 bit long parts of memory. Practically, there is no way to read just 1 or 3 bits from a bit stream. Processor always has to access at least 1 byte and manipulate it to extract wanted bits.

In experimental implementation, each field value is extracted and stored in the 32 bits word, aligned right. This is possibly done only once. The length information is found in profile structure or context. To extract field value, shifting and "and" masks are used. The opposite process is used when assembling compressed header. Actual read and write algorithms are byte oriented to flexibly manipulate arbitrary long fields. Since virtual machine is stack oriented, extract operation is assumed to be destructive.

Bit manipulation procedures are further complicated with two requirements: to preserve original input data array, and to append compressor generated data to the input data. Preservation is needed after format failure because of low probability, and after multi field rule failure. The problem is resolved by using special addition to the uncompressed data stack, called "help stack". Appended data is pushed to the help stack, and read first before access to input data is possible, Figure 4.

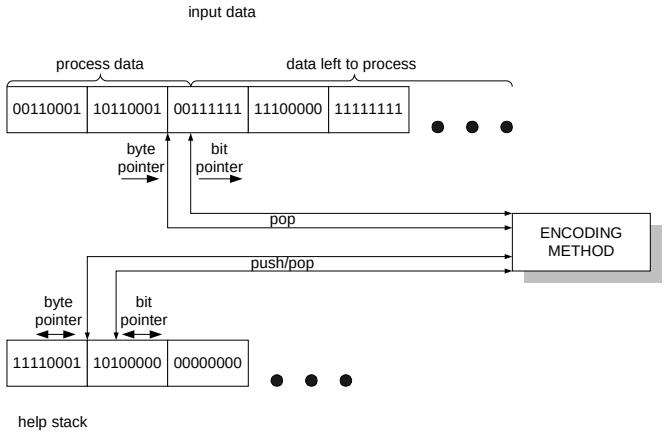

Figure 4 –Bit stack manipulation

This way, stack read is actually nondestructive, because original data is not overwritten. It can be restored by simply moving stack pointer back to appropriate value.

3.3 Encoding methods

According to practical implementation experience, encoding methods are classified in two main groups:

- *Elementary methods* – are encoding methods that process single header field. This is done without modifying field order and uncompressed packet data. Methods in this group are STATIC, VALUE, IRREGULAR, IRREGULAR-PADDED, INFERRED -IP-CHECKSUM, and LSB.
- *Complex methods* – are methods that process more than one field. Those are methods like: FORMAT, LIST, CRC, "MACRO". In this group we also put methods that change field order like STACK encoding methods, than U and UNCOMPRESSED, as well as methods that put some additional information to compressed header that doesn't exist in original header like NBO flag, FORMAT flag and MSN.

Method classification is later used for development of linear profile structures.

3.4. Multi field rules and "or" operator

Multi field rules are BNF rules that process several consecutive header fields. Such rules can be connected with choice "or" operator. This special case is called ORed rules [5]. It is executed in a way that first ORed rule is used. If any of methods inside first rule fails, the whole rule fails, and next ORed rule is tried for compression.

Complexity emerges from a fact that if one ORed rule fails, all context, input and output data values have to be restored to a state before ORed rule was applied.

In experimental implementation it was necessary to code ORed rules as a separate method. New "MACRO" method is introduced, despite not being specified in the EPIC-LITE draft. However, this method is transparent, and implicitly defined when ORed rules are specified in the profile.

Implemented MACRO method is by function very similar to FORMAT method, but choice is indicated through indicator flags, rather than through explicit indication.

3.5 State machine

Compressor and decompressor are implemented like independent program functions, Figure 5. Compressor initially starts in IR state and in this state compresses r consequent packets, where r is number of contexts per flow. Experiments were conducted with r=3. Then compressor moves to CO state. IR-DYN state is reached in two cases, indicator flags were discarded, or all encoding methods stated in the profile for a field have failed.

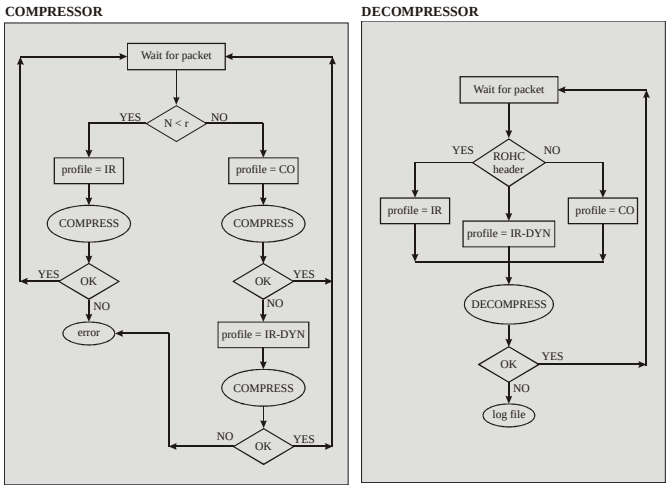

Figure 5 – State machine flow chart

State machines call inner interpreter functions to perform actual compression and decompression. All parameters needed for state machine and inner interpreter operations are contained or linked to a master structure, unique per flow.

4. PROFILE STRUCTURES

One of the main strength of EPIC-LITE specification is protocol independence, gained from the ABNF input language for compression specification. It can be used to simply generate particular protocol stack application, or to implement universal compressor - decompressor machine. In later case, profile is used to generate required functionality on demand: offline and online phases mentioned before emerge, and inner interpreter concept is introduced to process whatever methods profile did specify.

Experimental implementation uses profile parsed to a linked list of data structures to be interpreted by inner interpreter compressor and decompressor machines. Each structure actually represents one encoding elementary or complex method or rule, and contains all necessary parameters, Figure 6.

bToolbox parameter indicates if method concerned appears in integer string representation of used header format or not [4]. If not, default value of 0 is used in integer string. *value* is pointer to the list of encoding method parameters, while *length* is pointer to header field length. *offset* is offset parameter, as well as *prob* is probability parameter, as specified for particular method. *method_id* parameter is used for method identification. *list_order* is useful as branch indicator for LIST method.

```
typedef struct encoding_method
{
        bool bToolbox;
        VALUE_LIST* value;
        LENGTH* length;
        int offset;
        u16 prob;
        u8 method_id;
        u8 list_order;

        encoding_method* Child;
        encoding_method* Or;
        encoding_method* Next;
        encoding_method* Prev;
};
```

Figure 6 – Profile parameters

Several pointers to structures of the same kind are provided, to accommodate proper profile representation. *Child* is a pointer to the child encoding method of complex structures like FORMAT, *Or* to the next optional method or rule, *Next* to the next field encoding method or rule, and *Prev* pointer to the previous field encoding method or rule. *Next* pointer is used by compressor to traverse forward the profile, while decompressor uses *Prev* to traverse backwards.

4.1. Linear profile

Experimental implementation was developed in several phases. The first profiles used contained only elementary methods, so it was possible to parse profile text file to a linked list shown on Figure 7. Usage of *Next*, *Prev*, and *Or* pointers is illustrated based on beginning of profile from Figure 2. Lack of *Child* pointer can be noticed.

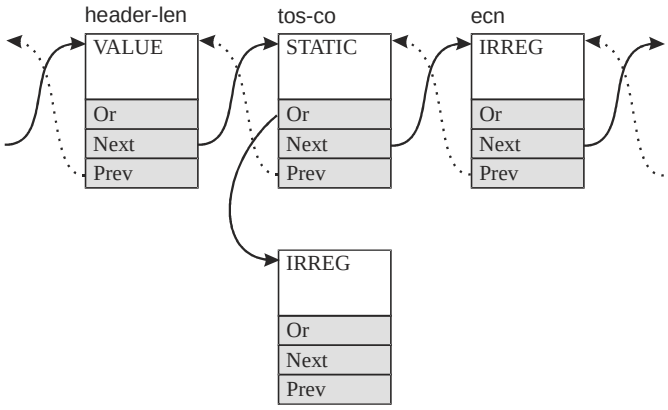

Figure 7 – Linear profile

Although EPIC-LITE profile has a tree structure based on possibility to define new rules using previous ones, simple profile was easily parsed to almost linear list structure.

Compressor inner interpreter passes from the beginning of the linked list towards its end. Each structure in the list contains information about header field that compressor is just to compress, so by passing through profile list compressor actually passes along the uncompressed header.

If field length is not among method parameters (e.g. STATIC encoding method), it can be obtained form the context, where it is initiated using IR headers. Compressor than reads amount of bits defined by field length from uncompressed_data stack to 32-bit long variable, incrementing stack pointer for the value of field length.

Compressed fields are used to construct compressed header by concatenating them in reverse order.

After reading indicator flags from beginning of the compressed header, decompressor calculates list of used methods in reverse order, in accordance with compressed fields order. For each field, decompressor reads amount of bits defined by profile or encoding method. Passing backward through linear list of structures, decompressor inner interpreter passes through the compressed header.

Using almost linear list structure described, several benefits were encountered. Calculation of method string from indicator flags is simple, inner interpreter operations are optimized, and list itself resembles header format closely.

4.2 Extended Linear Profile

More elaborated profiles introduced complex methods like FORMAT, MACRO and LIST. These methods actually perform additional branching and are close to tree structure of ABNF. Due to benefits of almost linear format, its properties were to be preserved.

The solution was found in additional branching by introducing new *Child* pointer. The solution used is called extended linear format, still much simpler than full tree structure of complex profile. In Figure 8, data structure for profile from Figure 2 is disclosed.

Extended linear profile generation is described in [4].

5. CONCLUSION

This paper provides basic implementation issues of EPIC-LITE scheme. Experimental implementation has shown that functional compressor and decompressor application can be made based on EPIC-LITE specification.

Implementation developed includes solution and optimization of bit manipulation, encoding method classification, and state machine functionality. In particular, a need for additional method to support ORed rules was recognized, and MACRO method was introduced.

Experience gained in data structures for simple profiles, which were almost linear, implied the same strategy to be used for complex profiles. Extended linear profile structure is successfully applied and used by compressor and decompressor inner interpreters. Generation of linear profile structure was supported by the offline part of the experimental implementation.

The significance of extended linear profile is, that in optimized manner complex profiles are supported, while enabling the format indicator flags to be downloaded to online processes in simple syntax. Doing so, terminal equipment need not perform complex Huffman generation procedures.

Future work will provide additional testing of inner parser and extended linear profile. Transition to processing actual packets, stored in communications subsystem structures, should also be included.

REFERENCES

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

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

[3] Crocker D., et all, "Augmented BNF for syntax specifications: ABNF", RFC 2234, 1997

[4] Cizmic, M., Vodopija, T. ,' Ozegovic, J.: "EPIC lite offline processing", Softcom 2002.

[5] Ozegovic, J., "Optimal syntax for packet header compression specification", Softcom 2002.

[6] Mornar, M., Pezelj, A.,' Ozegovic, J.: " Testbed for header compression implementation ", SOFTCom 2002.

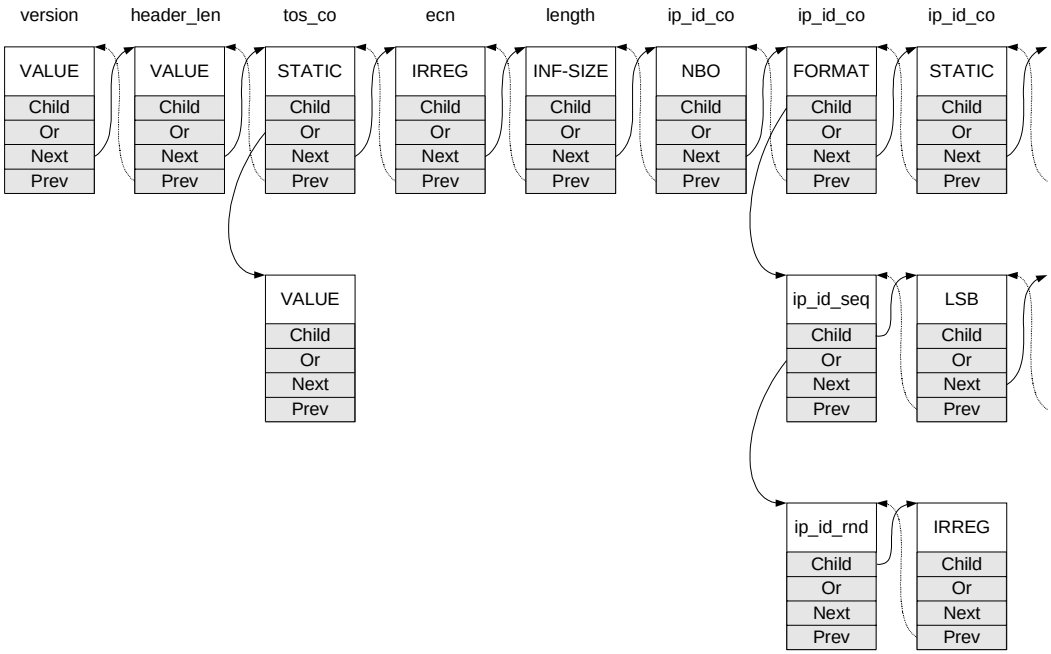

Figure 8 – Expanded linear profile