

# EPIC: provably optimal format encoding for compression in the Internet

Richard Price, Abigail Surtees, Stephen McCann, Mark West, Robert Hancock, Dave Findlay

Siemens/Roke Manor Research Limited

Romsey, UK

E-mail: {richard.price, abigail.surtees, stephen.mccann, mark.a.west, robert.hancock, dave.findlay}@roke.co.uk

**Abstract:** The topic of header compression has recently benefited from an increasing level of research due to the popularity of packet-based wireless networks. The ultimate goal of this research is a set of techniques that achieve optimal compression efficiency without sacrificing ease of implementation. In this paper, a technique to efficiently generate and support multiple header formats is proposed.

## 1. INTRODUCTION

This paper explores the different facets of header compression, first considering the various dimensions of the efficiency problem such as the choice of field encodings, the exploitation of redundant header information, and the selection and encoding of the compressed header. It then explores in more detail the mathematics of the EPIC (Efficient Protocol Independent Compression) algorithm and its basis in Huffman encoding. When parameterised by an accurate description of the protocol stack, EPIC can generate a set of compressed header formats that is provably optimal in a well-defined sense.

## 2. PROTOCOL HEADER BEHAVIOUR

Protocols such as IP are designed to run on fixed networks where bandwidth is plentiful, and introduce a large overhead over bandwidth-constrained links such as those found in a wireless network. When used to carry speech the overhead due to IP can be up to 75% of the total network capacity, which is unacceptable for the wireless environment. Consequently there is merit in investigating techniques to reduce this header overhead.

A typical protocol header is divided into a number of distinct bit patterns known as fields. Each field carries a different piece of information such as the protocol version, the length of the header etc. For example, the 40-byte IPv6 header [1] is divided as shown in Figure 1.

Byte 0	1	2	3
Version	Class	Flow Label	
Payload Length		Next Header	Hop Limit
Source Address (16 bytes)			
Destination Address (16 bytes)			

Figure 1 - Fields contained in an IPv6 header

Header fields typically contain highly structured data; the value of a field can often be inferred from the contents of other fields in the header, or can be derived from an instance of the same field in a previous header. This redundancy offers considerable scope for compression.

A number of schemes exist for the compression of protocol headers (e.g. Van Jacobson [2], Degermark [6], ROHC [3]), which variously make trade-offs between efficiency, flexibility and simplicity.

A typical compression scheme exploits the redundancy of a protocol header by classifying fields depending on their behaviour. A field that can be inferred from the values of other fields in the header may be classified as INFERRED; a field that remains constant between successive headers may be classified as STATIC etc.

The following table lists some examples of field behaviours used by Van Jacobson compression of TCP/IP headers:

Classification	Behaviour
VALUE	Fixed, well-known value
STATIC	Constant relative to previous header
INFERRED	Inferred from other header fields
DELTA	Small change from previous header
IRREGULAR	No discernable pattern

Table 1 - Van Jacobson field classifications

The compressed version of each header must be lengthwise self-describing (i.e. the compressed header must carry its own length value to indicate the point at which the header ends and the payload begins) and it must be aligned to a multiple of  $A$  bits where  $A$  is typically 1 or 8 depending on the characteristics of the link.

A header compression scheme generally offers one or more handcrafted *compressed header formats* designed to carry the compressed versions of each field. Fields classified as VALUE, STATIC or INFERRED can be derived at the decompressor without requiring any bits to be transmitted. Fields classified as DELTA require sufficient bits in the compressed header to communicate the delta change between the field and its preceding value. Fields classified as IRREGULAR have no discernible pattern and must be transmitted in full in the compressed header.

Additionally, if a particular field has more than one possible behaviour then the compressed header format must contain an indicator code to specify how the field has been compressed. For example, Van Jacobson compression specifies that five fields in the TCP/IP header can behave either as STATIC (no change from previous header) or as DELTA (small change from previous header). The first byte in the Van Jacobson compressed header contains a set of indicator flags to specify how each of these five fields has been encoded in the current header.

The efficiency of a conventional header compression scheme depends on the number and complexity of the handcrafted compressed header formats. More complex formats can more accurately capture the behaviour of the protocol headers, leading to a higher overall compression ratio. However the trade-off is that the compression scheme becomes more difficult to design and to implement.

### 3. THE EPIC APPROACH

This chapter gives an overview of the EPIC algorithm for compression of protocol headers [7]. The idea behind EPIC is to introduce a generic protocol notation that can be used to capture the behaviour of a protocol or protocol stack in a well-defined manner. This protocol notation can then be used by EPIC to automatically generate a set of compressed header formats that are optimally efficient for the described protocol.

#### 3.1. GENERIC PROTOCOL NOTATION

The underlying principal behind Van Jacobson and other header compression schemes is to classify each header field in terms of its expected behaviour, and then to construct a compressed version of the field that encodes the same information using fewer bits.

This section describes a generic protocol notation that can be used to succinctly capture the behaviour of each field in a header. The generic notation is a version of the well-known BNF (Backus-Naur Form) metasyntax [4] commonly used to describe new protocols and languages.

The generic notation is based around the concept of a “rule”, which can be used to describe the behaviour of a portion of the header to be compressed: a single field, a protocol, or even the entire protocol stack. New rules are defined recursively in terms of existing rules. This allows more complex behaviours to be constructed from simpler behaviours: for example the BNF rule for a protocol can be constructed from the BNF rules describing each individual field; the BNF rule for a complete protocol stack can then be constructed from the BNF rules for each protocol in the stack.

The format of a new rule in the generic protocol notation is as follows:

```
new_rule = ruleA1 | ruleA2 | ... | ruleAI
         ruleB1 | ruleB2 | ... | ruleBJ
         ruleC1 | ruleC2 | ... | ruleCK
         :           :           :
```

The symbol “|” denotes that a choice of BNF rules is available. So in the above example, the new BNF rule states that exactly one of ruleA<sub>1</sub>, ..., ruleA<sub>I</sub> occurs, followed by exactly one of ruleB<sub>1</sub>, ..., ruleB<sub>J</sub>, then followed by exactly one of ruleC<sub>1</sub>, ..., ruleC<sub>K</sub> etc.

Clearly it is impossible to define new BNF rules in terms of existing rules ad infinitum, so a number of fundamental BNF rules must be provided to define a set of simple behaviours for a field in the protocol header. Some examples of fundamental BNF rules are given in Table 2:

Fundamental BNF rule	Meaning
VALUE ( $N, K, P$ )	$N$ -bit field of value $K$
STATIC ( $P$ )	Constant relative to previous header
INFERRED-SIZE ( $N, K, P$ )	$N$ -bit field taking value (PACKET SIZE - $K$ )
DELTA ( $K, P$ )	Offset of between 0 and $2^K - 1$ from last header
IRREGULAR ( $N, P$ )	$N$ -bit field with no discernable pattern

Table 2 - Fundamental BNF rules

Note that each of the fundamental BNF rules contains a probability value  $P$  that expresses the percentage of headers for which the BNF rule is expected to apply. This allows the more common rules to be compressed using fewer bits than the rarely used rules.

```
Simple_Header = Field_A
              Field_B
Field_A       = STATIC(90%) | IRREGULAR(2,10%)
Field_B       = VALUE(6,0,80%) | VALUE(6,1,20%)
```

Figure 2 - Example of generic protocol notation

As an example, Figure 2 uses the generic protocol notation to describe a simple header containing just two fields. Field A is a 2-bit field that for 90% of the time remains constant between successive headers. However, for 10% of the time Field A behaves unpredictably and its value must be transmitted in full. Field B is a 6-bit field that randomly alternates between two values; it is 0b000000 for 80% of the time and 0b000001 for 20% of the time:

A more complex example is given in Figure 3. In this case the generic protocol notation is used to describe a complete IPv6 header:

```

IPv6_Header = Version
             Traffic_Class
             ECT_Flag
             CE_Flag
             Flow_Label
             Payload_Length
             Next_Header
             Hop_Limit
             Source_Address
             Dest_Address

Version      = VALUE(4,6,100%)
Traffic_Class = STATIC(99%) | IRREGULAR(6,1%)
ECT_Flag     = STATIC(99%) | IRREGULAR(1,1%)
CE_Flag     = VALUE(1,0,99%) | VALUE(1,1,1%)
Flow_Label  = STATIC(100%)
Payload_Length = INFERRED-SIZE(16,288,100%)
Next_Header  = VALUE(8,4,100%)
Hop_Limit    = STATIC(99%) | IRREGULAR(8,1%)
Source_Address = STATIC(100%)
Dest_Address  = STATIC(100%)

```

Figure 3 - Describing IPv6 in the generic protocol notation

The next step is to construct a mechanism for automatically converting the BNF description of a protocol into a set of compressed header formats. A simple compression scheme similar to Van Jacobson might generate the compressed header formats by concatenating the compressed versions of each field. Where a choice of more than one BNF rule is available for a field, one or more indicator flags could be added to specify which rule has been selected.

However the above approach is inefficient when there is a single commonly used compressed header format, as this common format will contain several indicator flags whereas one would in fact suffice. To improve the compression efficiency it is possible to first “multiply out” the BNF rules for each field, giving a table in which every row assigns exactly one fundamental BNF rule to each field in the header.

	Field A	Field B	$P$	$K$
1	IRREGULAR(2)	VALUE(6,1)	2%	2
2	IRREGULAR(2)	VALUE(6,0)	8%	2
3	STATIC	VALUE(6,1)	18%	0
4	STATIC	VALUE(6,0)	72%	0

Table 3 - Possible field behaviours for the example protocol

Table 3 illustrates this step for the simple example of Figure 2. The table also lists the overall probability  $P$  that a header can be compressed using each assignment of BNF rules, and the number of bits  $K$  required to transmit the compressed versions of each field.

Note that the overall probabilities are derived by multiplying the probabilities for each fundamental BNF rule (i.e. the fields are assumed to behave independently). In practice this is not a restriction however, because two fields that behave in a dependent manner can always be treated as a single field (albeit with somewhat more complex behaviour).

Determining the possible behaviours of every field in advance allows the compressed headers to be more efficiently encoded. Each compressed header format now only requires a single indicator code that specifies which BNF rule has been assigned to every field in the header, followed by any additional bits required for the compressed versions of each field.

The indicator code should take into account the probability that the compressed header format will be used, with more common formats allocated a shorter indicator code than rare formats. One method for determining a set of indicator codes is to apply Huffman encoding [5] to the probabilities that each compressed header format will be required. The advantage of Huffman encoding is that the indicator codes generated by the algorithm are *optimally efficient*: in other words it is not possible to construct a more efficient set of indicator codes for the compressed headers (assuming that the probabilities that each compressed header format will be used have been accurately determined).

Note however that the compressed headers themselves are not necessarily optimal, because the indicator codes are constructed without taking into account the number of bits required to transmit the compressed versions of each field. So if a particular compressed header format requires  $K$  bits to transmit the compressed fields, the restriction is that all  $2^K$  corresponding headers must have the same compressed length. Removing this restriction would require a separate Huffman code to be generated for every compressed header, rather than reusing the same Huffman code for all headers corresponding to one compressed header format.

Whilst it is theoretically possible to generate a distinct Huffman code for every compressed header, extensive use of the IRREGULAR rule makes this approach computationally infeasible. For example, the BNF description of an IPv6 header includes a compressed header format that contains two IRREGULAR (8, 1%) rules and an IRREGULAR (1, 1%) rule. This compressed header format alone would require  $2^{17} = 131072$  leaf nodes in an ordinary Huffman tree.

If an optimally efficient set of compressed headers is desired then the basic Huffman algorithm must be enhanced.

### 3.2. HIERARCHICAL HUFFMAN

The Hierarchical Huffman (HH) algorithm is a modified version of Huffman encoding designed to efficiently handle the case where many leaf nodes contain equal probabilities. HH generates a set of codes identical to those generated by the ordinary Huffman algorithm, and so in particular the optimal efficiency of the codes is retained.

The HH algorithm makes two modifications to ordinary Huffman. Firstly, as well as the probability value  $P$  the leaf nodes of the HH algorithm each contain an index value  $N$ . An

HH leaf node with probability  $P$  and index  $N$  is equivalent to  $N$  ordinary Huffman leaf nodes each with probability  $P$ .

Secondly, whereas the ordinary Huffman algorithm always generates a set of bit-aligned indicator codes, the HH algorithm can ensure that the resulting codes are an integer multiple of  $A$  bits for arbitrary values of  $A$ . This is especially useful when the underlying link only accepts compressed headers with a certain alignment (e.g. certain links require all headers to be byte-aligned).

When constructing the HH tree, the first step is to define a value  $M := (2^A - \text{TOTAL\_N}) \bmod (2^A - 1)$ , where  $\text{TOTAL\_N}$  is the sum of all of the index values for all of the leaf nodes. If  $M > 0$  then a new node is created with probability 0% and index  $M$ . This “padding” node is added to ensure that the resulting Huffman codes are all an exact multiple of  $A$  bits.

Next, the active node with the smallest probability value is discovered. Suppose that this node has probability value  $P_j$  and index value  $N_j$ . One of the following three steps is then taken depending on the value of  $N_j$ :

1.) If  $N_j$  is a multiple of  $2^A$  then a new node is created with probability  $P_j * 2^A$  and index  $N_j / 2^A$ . The node with probability  $P_j$  and index  $N_j$  is joined to this new node and the branch is labelled with an “X”.

2.) If  $N_j$  is not a multiple of  $2^A$  and  $N_j > 2^A$  then two new nodes are created. The first node has probability  $P_j$  and index  $N_j \bmod 2^A$ , while the second node has probability  $P_j$  and index  $R := 2^A * \lfloor N_j / 2^A \rfloor$  where  $\lfloor K \rfloor$  denotes the largest integer not larger than  $K$ . The node with probability  $P_j$  and index  $N_j$  is joined to both of the new nodes. The branch to the node with index  $R$  is labelled “D0”, and the branch to the node with index  $N_j \bmod 2^A$  is labelled “DR”.

3.) If  $N_j < 2^A$  then the next  $K - 1$  active nodes in order of increasing probability are discovered such that if the nodes have probabilities  $P_2, \dots, P_K$  and indices  $N_2, \dots, N_K$ , then  $K$  is the smallest value with  $N_j + \dots + N_K \geq 2^A$ .

a.) If  $N_j + \dots + N_K > 2^A$  then two new nodes are created. The first node has probability  $P_K$  and index  $(2^A - N_j - \dots - N_{K-1})$ , while the second node has probability  $P_K$  and index  $R := (N_j + \dots + N_K - 2^A)$ . The node with probability  $P_K$  and index  $N_K$  is joined to both of the new nodes. The branch to the node with index  $R$  is labelled “D0” and the branch to the node with index  $(2^A - N_j - \dots - N_{K-1})$  is labelled “DR”.

b.) A new node is created with probability  $(P_j * N_j + \dots + P_{K-1} * N_{K-1} + P_K * (2^A - N_j - \dots - N_{K-1}))$  and index 1. For all  $J$  between 1 and  $K - 1$  inclusive, the node with probability  $P_J$  and index  $N_J$  is joined to this new node and the branch is labelled with the integer  $(N_j + \dots + N_{J-1})$ . The node with probability  $P_K$  and index  $(2^A - N_j - \dots - N_{K-1})$  is also joined to the new node and the branch is labelled with the integer  $(N_j + \dots + N_{K-1})$ .

Note that when a new node is created, the existing nodes to which it is joined become “inactive” and no longer take part in building the HH tree.

Figure 4 illustrates the bit-aligned HH tree generated for the simple protocol of Figure 2. Each leaf node in the HH tree represents one of the compressed header forms shown in Table 3:

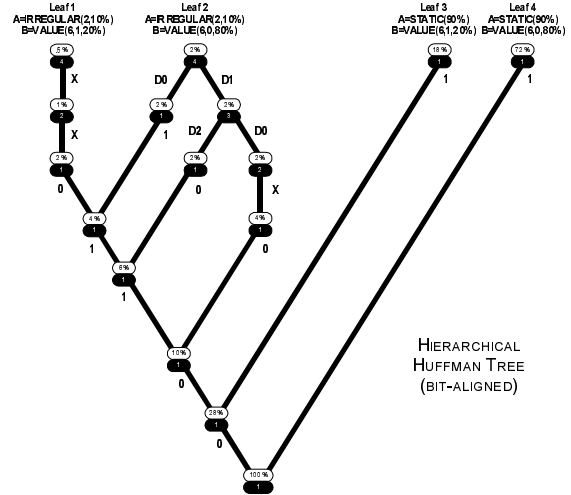


Figure 4 - Bit-aligned HH tree

Figure 5 illustrates the HH tree generated for the same protocol but aligning the headers to multiples of 2 bits (i.e. the case  $A = 2$ ):

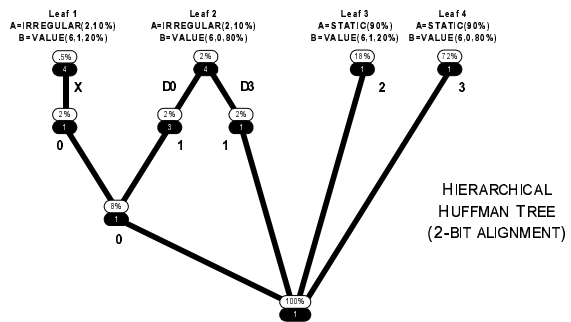


Figure 5 - HH tree with 2-bit alignment

Compressing headers using EPIC is straightforward. Firstly, a compressed header format is selected that is capable of encoding all of the fields in the uncompressed header. This format corresponds to a certain leaf node of probability  $P$  and index  $N$  on the HH tree. Also, the compressed version of each field in the header is concatenated to form a single bit pattern, which is interpreted as an integer  $M$  from 0 to  $N - 1$ . The compressed header is then constructed by parsing the HH tree from leaf node to root as follows:

1.) If the next branch is labelled “X” then the  $A$  bits with value  $(M \bmod 2^A)$  are appended to the front of the compressed header, and  $M$  is replaced by  $\lfloor M / 2^A \rfloor$ .

2.) If the node splits to follow branches labelled “D0” and “DR” then if  $M < R$ , the branch labelled “D0” is followed. Otherwise  $M$  is replaced by  $M - R$  and the branch labelled “DR” is followed.

3.) If the next branch is labelled with an integer  $J$  then the  $A$  bits with value  $(J + M)$  are appended to the front of the compressed header and  $M$  is set to 0.

When decompressing a header, the HH tree is parsed from root to leaf node using the bits contained in the compressed header to select the route through the tree. The integer  $M$  is initialised to 0 and then reconstructed as follows:

1.) If the next branch is labelled “X” then the next  $A$  bits are read from the compressed header and interpreted as an integer  $J$ . The integer  $M$  is then replaced by  $M * 2^A + J$ .

2.) If the next branch is labelled “DR” then  $M$  is replaced by  $M + R$ .

3.) If the next branches are labelled with integers then  $A$  bits are read from the compressed header and interpreted as an integer  $M$ . The branch is followed with the largest label  $J$  such that  $M \geq J$ , and  $M$  is replaced by  $M - J$ .

When a leaf node is reached the choice of compressed header format is known. The integer  $M$  is interpreted as a bit pattern which contains the compressed version of each field in the header. These two pieces of information are sufficient to reconstruct the original uncompressed header.

Table 4 gives examples of headers generated by the simple protocol of Figure 2, and their compressed equivalents in both the bit-aligned and 2-bit aligned cases:

Uncompressed header		Compressed header	
Current	Previous	Bit-aligned	2-bit aligned
11 000001	00 000000	0 0 1 1 0 1 1	00 00 11
11 000001	11 000001	0 1	10
11 000000	11 000001	1	11
01 000000	11 000000	0 0 0 0	00 10

Table 4 - Compressed headers for the example protocol

### 3.3. RESULTS

Table 5 compares the performance of EPIC with the ROHC scheme [3] for compressing RTP/UDP/IPv4 headers. The table shows the average compressed header size for a number of different RTP flows, together with the percentage improvement offered by EPIC:

Flow	Size (bytes)		Improvement
	ROHC	EPIC	
Regular RTP stream	1.42	1.42	0%
RTP stream with talk-spurts	1.71	1.51	13%
Small jumps in IP-ID	2.44	2.09	17%
Periodic changes in IP TOS	2.16	1.73	25%

Table 5 - Results for compression of RTP/UDP/IPv4

Note that the ROHC headers include a CRC and a sequence number to provide robustness over a lossy link: to ensure a fair comparison the same CRC and sequence number are included in the EPIC compressed headers.

## 4. CONCLUSIONS

This paper has presented the EPIC scheme for the compression of protocol stacks such as TCP/IP or RTP/UDP/IP.

A generic protocol notation, based on BNF syntax, is introduced that allows the behaviour of a protocol or protocol stack to be captured in a well-defined manner. Moreover, it provides possibility to specify multiple encoding methods for a field to better track its behaviour. This way, multiple formats can be specified.

To identify header format chosen for actual compression, a Huffman indicator code is proposed. To generate it, field encodings are characterised with expected probability of usage. With assumption that fields' behaviour is independent, format probability is gained as simple product of chosen method probabilities.

The Hierarchical Huffman algorithm is then defined so that the description of a protocol can be converted into a set of compressed header formats that is provably optimal in a well-defined sense.

Proposed concepts are presented for standardisation [7,8].

## REFERENCES

- [1] Deering S and Hinden R, “Internet Protocol, Version 6 (IPv6)”, RFC 1883, 1995.
- [2] Jacobson V, “Compressing TCP/IP Headers for Low-Speed Serial Links”, RFC 1144, 1990.
- [3] Bormann C, et al, “Robust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed”, RFC 3095, 2001.
- [4] Crocker D, et al, “Augmented BNF for Syntax Specifications: ABNF”, RFC 2234, 1997.
- [5] Nelson M and Gailly J-L, “The Data Compression Book”, M&T Books, 1995.
- [6] Degermark, M. et al, "Low-loss TCP/IP Header Compression for Wireless Networks" ACM/Baltzer Journal on Wireless Networks, vol 3, no 5, 1997.
- [7] Price, R. et al, "Framework for EPIC-LITE", draft-ietf-rohc-epic-lite-01.txt, 2002.
- [8] Price, R. et al, "Enhanced TCP/IP Compression for ROHC", draft-price-rohc-epic-03.txt, 2002.