Robust Header Compression for CCNx 1.0: *Bit Aligned*

Draft 0

Marc Mosko

Aug 25, 2015

Overview

- Previously defined byte-aligned compression
  - TL pairs and Vs start on byte boundary
  - Uses dictionary substitution
  - Loses efficiency due to byte-aligned nature
- Current algorithm
  - Uses bit-aligned code words
  - Uses Huffman tree to generate codes
  - Still allows plain 'T' and 'L' encoding for uncommon symbols not in codebook.

More Details

- Assign probability weights to previous dictionary entries
  - Throw out the ‘4-bit’ ones, use new ‘variable length’ encoding, so fewer symbols.
- Build Huffman Tree
- Encode
  - Symbols in Huffman tree use that code word.
  - Otherwise use a compacted TL encoding.
  - Still allows for learned symbols.
  - Fixed header encoding the same.

TL Compression (bit aligned)

(t = type bit, l = length bit, z = compressor key)

Uncompressed Format: (“000” fixed header)

```
t{16} l{16}                         (16-bit T & 16-bit L)
```

Compressed Formats: (“1xx” fixed header)

```
0(Huffman Z and L)
10z{4} l{4}                    (learned 4-bit Z & 4-bit L)
110t{8} l{8}                   ( 8-bit T &  8-bit L)
1110t{16} l{10}                (16-bit T & 10-bit L)
11110t{16} l{16}               (16-bit T & 16-bit L)
111110z{11}                    (learned – 2K entries)
1111110z{24}                   (learned – 16M entries)
```

Formats with a ‘t’ encode dictionary misses.
Formats with a ‘z’ encode dictionary hits.

Huffman encoded keys

(t = type bit, l = length bit, z = compressor key)

Compressed Formats: ("1xx" fixed header)

0(Huffman Z and L)

The encoded value Z is a Huffman code (i.e. the path in a Huffman tree).

The token string represented by a leaf node in the tree is
- A fixed T and fixed L (i.e. a 4 byte string)
- A series of TL pairs with no intermediate Values.
- A fixed T and L bit length (e.g. 0x0002 with 4-bit L)
- A fixed T and an encoded L (see next page)

Variable length L

Covers common short lengths with few bits. Long lengths take more bits. Designed so maximum representable value is 65535 (the maximum for an L).

Prefix	**Length bits**	**Range start**	**Range end**
0b0	l{3}	0	7
0b10	l{5}	8	39
0b110	l{8}	40	295
0b1110	l{10}	296	1319
0b11110	l{15}	1320	34087
0b111110	l{14}	34088	50471
0b1111110	l{13}	50472	58663
0b11111110	l{12}	58664	62759
0b111111110	l{11}	62760	64807
0b1111111110	l{9}	64808	65319
0b11111111110	l{7}	65320	65447
0b111111111110	l{6}	65448	65511
0b1111111111110	l{4}	65512	65527
0b1111111111110	l{3}	65528	65535

Example

Token String	Length	Code
0x0001	Variable	1
0x0000	variable	01
0x0013	variable	000
0x0002 0x0020	32 bytes	001001
0x0003 0x0004 0x0002 0x0000 0x0004 0x0004	4 bytes	001000

```
0x0001, 0x005D,   // Interest             01,11000110101,
0x0000, 0x0025,   // Name                 001,1011101,
0x0001, 0x0008,   // NameSeg              01,1000000,
‘parc.com’,                               ‘parc.com’,
0x0001, 0x0010,   // NameSeg              01,1001000,
‘compression.pptx’,                       ‘compression.pptx’,
0x0013, 0x0001,   // Chunk                0000,0001,
{0x00},                                   {0x00},
0x0002, 0x0020,   // KeyId restriction    0001001,
{32-byte string},                         {32-byte string}
0x0003,0x0004,    // validation alg       0001000,
0x0002,0x0000,    // CRC32C
0x0004,0x0004,    // Validation payload
{4-byte string}                           {4-byte string}
```

Example

```
0x0001, 0x005D,   // Interest           01,11000110101,
0x0000, 0x0025,   // Name               001,1011101,
0x0001, 0x0008,   // NameSeg            01,1000000,
‘parc.com’,                             ‘parc.com’,
0x0001, 0x0010,   // NameSeg            01,1001000,
‘compression.pptx’,                     ‘compression.pptx’,
0x0013, 0x0001,   // Chunk              0000,0001,
{0x00},                                 {0x00},
0x0002, 0x0020,   // KeyId restriction  0001001,
{32-byte string},                       {32-byte string}
0x0003,0x0004,    // validation alg     0001000,
0x0002,0x0000,    // CRC32C
0x0004,0x0004,    // Validation payload
{4-byte string}                         {4-byte string}

- 776 bits total           34% less     - 511 bits total
- 448 bits data                         - 448 bits data
- 328 bits overhead        81% less     -  63 bits overhead
```

Using the byte-aligned form

```
0x0001, 0x005D,   // Interest           0x32, 0xc2
0x0000, 0x0025,   // Name               0x5d, 0xc0,
0x0001, 0x0008,   // NameSeg            0x25, 0x18,
‘parc.com’,                             ‘parc.com’,
0x0001, 0x0010,   // NameSeg            0xc2, 0x10,
‘compression.pptx’,                     ‘compression.pptx’,
0x0013, 0x0001,   // Chunk              0x41,
{0x00},                                 {0x00},
0x0002, 0x0020,   // KeyId restriction  0x82,
{32-byte string},                       {32-byte string}
0x0003,0x0004,    // validation alg     0x84,
0x0002,0x0000,    // CRC32C
0x0004,0x0004,    // Validation payload
{4-byte string}                         {4-byte string}

- 776 bits total           31% less     - 536 bits total
- 448 bits data                         - 448 bits data
- 328 bits overhead        73% less     -  88 bits overhead
                                           25 bits more
```

Information Theory Analysis

Uncoded	TL Information	Compressed
0x0001, 0x005D,	8 bits	01,11000110101,
0x0000, 0x0025,	8 bits	001,1011101,
0x0001, 0x0008,	7 bits	01,1000000,
‘parc.com’,		‘parc.com’,
0x0001, 0x0010,	8 bits	01,1001000,
‘compression.pptx’,		‘compression.pptx’,
0x0013, 0x0001,	4 bit	0000,0001,
{0x00},		{0x00},
0x0002, 0x0020,	3 bit	0001001,
{32-byte string},		{32-byte string}
0x0003,0x0004,	0 bit	0001000,
0x0002,0x0000,	4 bit	
0x0004,0x0004,	0 bit	
{4-byte string}		{4-byte string}
328 bits	42 bits	63 bits
781% over		50% over

Example Huffman Tree

Code word	token_string	bytes
000	0x0013	Variable
001000	0x0003 0x0004 0x0002 0x0000 0x0004 0x0004	4 bytes
001001	0x0002 0x0020	32 bytes
0010100	0xF000	Variable
0010101	0x0004	Variable
001011	0x0003 0x0020	32 bytes
0011000	0x0004 0x0004	4 bytes
0011001	0x0003 0x0012	18 bytes
0011010	0x0003	Variable
0011011	0x0002	Variable
00111000	0x0003 0x0034 0x0006 0x0030 0x0009 0x0020	32 bytes
00111001	0x0006	Variable
00111010	0x0005	Variable
001110110	0x0009 0x0020	32 bytes
001110111000	0x0009 0x0010	16 bytes
001110111001001100	TBD	Variable
001110111001001110	TBD	Variable
001110111001001111	TBD	Variable
00111011100101	0x0019 0x0004	4 bytes
00111011100110	0x0003 0x000C 0x0004 0x0008 0x0009 0x0004	4 bytes
00111011100111	0x0002 0x0004	4 bytes
0011101110100	0x0019 0x0002	2 bytes
0011101110101	0x0002 0x0000	0 bytes
001110111011	0x0019 0x0001	1 byte
00111011110	0x0004 0x0010	16 bytes
00111011111	0x0004 0x000E	14 bytes
001111000000	0x0004 0x0014	20 bytes
001111000001	0x0003 0x000C	12 bytes
001111000010	0x000B 0x0226	550 bytes
001111000011	0x000B 0x00A2	162 bytes
0011110001	Counter Def	Variable
00111100100	0x000B 0x0126	294 bytes
00111100101	0x0009 0x0004	4 bytes
00111100110	0x0003 0x00CE 0x0006 0x00CA 0x0009 0x0020	32 bytes
00111100111	0x000F 0x0008	8 bytes
0011110100	0x0008 0x0011	17 bytes
0011110101	0x0003 0x0014 0x0004 0x0010 0x0009 0x0004	4 bytes
0011110110	Token Def	Variable
0011110111	Dict ACK	Variable
0011111000	0x0003 0x0004	4 bytes
0011111001	0x0005 0x0001	1 byte
001111101	0x0006 0x0008	8 bytes
00111111	0xF001	Variable
01	0x0000	Variable
1	0x0001	Variable

Conclusion

- A TLV-aware bit-aligned compressor
  - – Exploits knowledge of the TLV structure.
  - – Encodes common TL strings with Huffman code.
  - – Misses encoded with compact TL forms.
  - – Still allows for learning additional code words.
  - – About 30% – 40% more efficient than byte-aligned compressor