Exploring HTTP/2 Header Compression

Kazuhiko Yamamoto
Internet Initiative Japan Inc.
kazu@iij.ad.jp

Tatsuhiro Tsujikawa
Z Lab Corporation
ttsujika@zlab.co.jp

Kazuho Oku
Fastly
kazuhooku@gmail.com

ABSTRACT

For every HTTP/1.1 request and response, almost the same set of headers is transferred. This wastes bandwidth, the most expensive resource of the browser-server communication. To solve this issue, header compression for HTTP/2 was standardized. During the standardization, we found that one element of the compression technology, the so called reference set, contributes little to the compression ratio while its mechanism is complicated. With our proposal, the reference set was removed and the specification and implementations were drastically simplified. For high performance implementation of header compression, we devised token-based reverse indices, length guessing for Huffman encoding, and pre-calculated state transitions for Huffman decoding.

KEYWORDS

HTTP/2, Header Compression, Compression Ratio, High Performance

ACM Reference format:
Kazuhiko Yamamoto, Tatsuhiro Tsujikawa, and Kazuho Oku. 2017. Exploring HTTP/2 Header Compression. In *Proceedings of CFI'17, Fukuoka, Japan, June 14-16, 2017,* 5 pages.
https://doi.org/10.1145/3095786.3095787

1 INTRODUCTION

One of the issues of HTTP/1.1 [18] is header redundancy. Almost the same set of headers is transferred between a browser and a server for every request and response. Because HTTP/1.1 is a stateless protocol, large cookie values are used in request headers to implement state. It is said that the average size of request headers is 800 bytes. This wastes bandwidth – the most expensive resource in browser-server communications.

Google Inc. designed and implemented SPDY [14] to solve the issues of HTTP/1.1, including low concurrency and head-of-line blocking. Their solution for the redundant header issue was header compression based on DEFLATE [17]. Subsequently, the CRIME attack [2] showed that this method is insecure. SPDY compresses the entire set of headers. If an attacker can inject a header, they are able to observe the varying size of compressed headers, even in an encrypted channel.

Though the Internet Engineering Task Force (IETF) [15] standardized HTTP/2 [16] based on SPDY, it took different approach for header compression from SPDY's. HTTP/2 header compression, so called HPACK [23], compresses header by header, to resist the CRIME attack.

We participated in the standardization of HTTP/2 and HPACK and continuously checked the inter-operability of HPACK with our three independent implementations:

- HPACK library included in nghttp2 [11] — the de facto reference implementation of HTTP/2 written in C.
- HTTP/2 library included in H2O [4] — a high performance HTTP/2 server written in C.
- HTTP/2 library [10] [24] written in Haskell [20] — it implements all compression methods described later.

This paper summarizes our experiences on HPACK. Contributions of this paper are as follows:

Specification: we show that one element compression technology, the so called reference set, contributes little to the compression ratio, and we propose to remove it from the HPACK drafts. This discovery resulted in a much simpler specification and hence better inter-operability.

Implementation: we found implementation techniques to improve performance, including token-based reverse indices, length guessing for Huffman encoding, and pre-calculated state transitions for Huffman decoding.

It is highly probable that our implementation techniques can be used in future protocols such as QUIC, a UDP-based multiplexed and secure transport [12].

This paper is organized as follows: Section 2 explains how we contributed to simplify the specification of HPACK. We show implementation techniques to improve performance in Section 3, and evaluate this in Section 4. Section 5 describes conclusions and future work.

2 SIMPLIFYING THE SPECIFICATION

This section discusses the HPACK specification. The critical elements of HPACK (RFC7541) are as follows:

Static table: a predefined table whose entry is either a header name or a header name-value pair. Each entry can be accessed through an index. For example, 31 is the key for "content-type" and 8 is for the pair of "accept-encoding" and "gzip, deflate". Since the typical length of indices is seven bits, transferring indices instead of strings saves bandwidth.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CFI'17, June 14-16, 2017, Fukuoka, Japan
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5332-8/17/06...$15.00
https://doi.org/10.1145/3095786.3095787

Dynamic table: a table similar to the static table, but entries are registered dynamically. The dynamic table has an upper limit to its size. If the registration of a new entry causes table overflow, old entries are removed. Each endpoint uses two dynamic tables, one each for sending and receiving per connection.

Huffman encoding [19]: encoding more frequently used letters in header names and header values with fewer bits. The mapping is statically defined.

Also, HPACK draft 08 [5] or earlier defined the reference set:

Reference set: a set of indices to represent headers. A client saves the reference set of the headers of a request. When sending the next request, it prepares the reference set of the next request and calculates difference between the previous one and the current one. It requests addition and deletion for indices which have newly appeared or disappeared, respectively. If the number of common indices is large enough, the total length gets shorter.

2.1 Removing Reference Set

The algorithm of HPACK decoding (decompression) is unique and clearly described in the specification. On the other hand, that of HPACK encoding (compression) is not defined concisely. We explored the design space of HPACK encoding and created eight methods. Table 1 describes their names and use/non-use of the element. Note that the reference set depends on both the static table and the dynamic table.

Table 1: Eight compression methods of HPACK draft 08

	static table	dynamic table	reference set	Huffman encoding
Naive	not used	not used	not used	not used
NaiveH	not used	not used	not used	used
Static	used	not used	not used	not used
StaticH	used	not used	not used	used
Linear	used	used	not used	not used
LinearH	used	used	not used	used
Diff	used	used	used	not used
DiffH	used	used	used	used

The HTTP/2 community in Japan [9], made test cases for HPACK inter-operability [7]. We verified the correctness of our implementations with these test cases. Also, the community calculated the compression ratio [1] for several HPACK implementations with header sample data [8] provided by the IETF. The sample data consisted of 31 sets, where each set included multiple headers (the minimum was 2 and the maximum 646). The compression ratio was the length of compressed headers divided by that of original headers.

Using this sample data and this calculation, we computed the average compression ratio for the eight methods(Figure 1). Since the format of HPACK has overhead, the compression ratio of “Naive”, which uses no element technologies, is over 1.0.

Figure 1 shows:

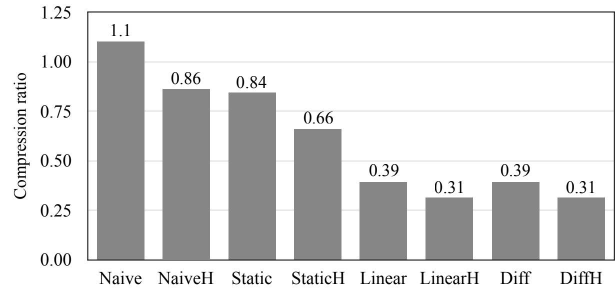

Figure 1: The average compression ratio of eight compression methods. The smaller, the better.

- The static table is effective (e.g. Naive vs Static)
- The dynamic table is effective (e.g. Static vs Linear)
- Huffman encoding is effective (e.g. Naive vs NaiveH)
- The reference set is ineffective (e.g. Linear vs Diff)

In fact, Huffman encoding saves 31.36 bytes per frame on average, compared to only 1.57 bytes for the reference set. This is because the number of common indices tends to be small as the typical length of indices is seven bits.

Based on this study, we advised the IETF to remove the reference set. The advantages of this proposal are as follows:

- The reference set is a complex mechanism. For instance, it has a special corner case [13]. If the reference set is removed, the specification gets much simpler, resulting in improved inter-operability.
- The compression ratio is almost the same even if the reference set is removed.
- Without the reference set, the order of headers is always preserved on the decoding side.

Nobody could show effective cases for the reference set. As a result, the reference set was removed in HPACK draft 09 [6]. This also made implementations much simpler. 24.5% (704/932) and 8% (2249/2450) lines of code were removed from the main part of HTTP/2 library in Haskell and that of nghttp2, respectively.

3 HIGH PERFORMANCE IMPLEMENTATION

This section summarizes techniques to improve the performance of HPACK encoding and decoding. Our profilings shows the following have room for improvement:

- Converting headers to indices in HPACK encoding
- Huffman encoding in HPACK encoding
- Huffman decoding in HPACK decoding

We will discuss these items one by one in this Section. Complexity of operations is in worst-case unless explicitly mentioned.

3.1 Headers to Indices

In HPACK decoding, we convert indices to header names or header name-value pairs. This is accomplished by accessing

the static and dynamic tables in $\mathcal{O}(1)$. In HPACK encoding, however, naive implementations must search the tables to convert header names or header name-value pairs into indices. This operation is $\mathcal{O}(n)$ where n is the *length* (the number of entries) of the tables. The length of the static table is 61. The length of the dynamic table can be 128 at most for the initial case where the table *size* (the total size of entries) is 4,096 bytes and the minimum size of entries are 32 bytes.

For a high performance implementation, it is wise to use reverse indices instead of searching the tables directly. It is necessary to find indices from both a header names and header name-value pairs. Logically speaking, one can make use of a finite map of finite maps where the outer keys are header names and the inner keys are header values. Two lookups in $\mathcal{O}(\log n)$ are necessary for this scheme. Hash tables can be used instead of infinite maps. If the chained hashing whose lookup operation is $\mathcal{O}(1)$ average-case complexity is used for simplicity, implementations should beware of hash-collision where its lookup operation results in $\mathcal{O}(n)$ worst-case complexity.

We can reduce two lookups to one with *tokens* for header names. The tokens are enumerations whose members are organized by the header names defined in the static table. A token holds an index to reverse index arrays described later, and other useful information statically defined. Note that tokens are also used for header name comparison in $\mathcal{O}(1)$ in the HTTP/2 servers using our libraries. As shown below, only one string comparison is necessary to convert a header name to a token:

```
const token_t *
to_token(const char *name, size_t len) {
    switch (len) {
    ...
    case 3:
        switch (name[2]) {
        case 'a':
            if (memcmp(name, "vi", 2) == 0)
                return TOKEN_VIA;
            break;
        case 'e':
            if (memcmp(name, "ag", 2) == 0)
                return TOKEN_AGE;
            break;
        }
        break;
    case 4:
        switch (name[3]) {
        case 'e':
            if (memcmp(name, "dat", 3) == 0)
                return TOKEN_DATE;
            break;
        case 'g':
            if (memcmp(name, "eta", 3) == 0)
                return TOKEN_ETAG;
            break;
        ...
        }
    ...
    }
    return NULL;
}
```

With these tokens, one can implement reverse indices with one lookup. Here is an example of the data structures:

(1) An array of finite maps for the static table. The array is accessed with the tokens in $\mathcal{O}(1)$ and the finite maps are accessed with header values in $\mathcal{O}(\log n)$. Indices for header name-value pairs and header names can be resolved. Note that most finite maps in this reverse index are empty because header values are not defined for most header names in the static table. This is an immutable data structure.
(2) An array of mutable finite maps for the dynamic table. The array is accessed with the tokens in $\mathcal{O}(1)$ and the finite maps are accessed with header values in $\mathcal{O}(\log n)$. Indices for header name-value pairs whose header names are defined in the static table can be resolved.
(3) A mutable finite map for the dynamic table. The finite map is accessed with header name-value pairs in $\mathcal{O}(\log n)$. Indices for header name-value pairs whose header names are not defined in the static table can be resolved.

The look-up algorithm with these three reverse indices is as follows:

- For a header name defined in the static table, reverse index 2) is accessed with the corresponding token and the corresponding infinite map is looked up with a header value:
  - If an index for the name-value pair is found, it is returned.
  - Otherwise, reverse index 1) is accessed with the token.
    - If the corresponding infinite map is not empty and an index for the name-value pair is found using the value, return it.
    - Otherwise, an index for the header name is returned.
- For a header name not defined in the static table, reverse index 3) is used:
  - If an index for the name-value pair is found, return it.
  - Otherwise, return nothing.

Since most infinite maps in the index 1) are empty, this algorithm uses only one lookup in most cases.

3.2 Huffman Encoding

The format of Huffman encoded strings is length-value. The byte count of an integer varies, and the length of Huffman encoded strings is not known in advance. So, one naive implementation encodes a string in a temporary buffer, obtains the resulting length, encodes the length in the target buffer, and copies the result from the temporary buffer. Another naive implementation calculates the result length by traversing a string in advance, then encodes the length and the string.

The former is slow due to a string copy and the latter is slow because of two traversals.

We found that the byte count to represent the result length can be guessed. The following shows how many byte counts are necessary for integer ranges:

byte count	integer range
1	0 – 126
2	127 – 254
3	255 – 16510

In our experiments, Huffman encoding can compress by 20 percent on average. Thus, we can roughly guess the resulting length using a factor of 0.8 and estimate the byte count from the table above.

We also noticed that the length of a Huffman encoded string is larger than that of the original string in some cases, typically for cookie values which use ASCII symbols. In these cases, we should use the original string instead of the Huffman encoded string.

The following is our one traversal algorithm of Huffman encoding, which chooses a shorter string, without using a temporary buffer:

- Get the length of an input string (l_o) and its byte count (b_o).
- Prepare a buffer whose length is $l_o + b_o$.
- Guess the result length with the factor of 0.8 (l_e) and its byte count (b_e).
- Huffman encode the string starting from the position of b_e.
  - If we reach the end of the buffer during this encoding, encode l_o in the beginning of the buffer and copy the input string as is.
  - Otherwise, obtain the real length of the result (l_r) and its byte count (b_r).
    - In the rare case where b_e is not equal to b_r, move the result appropriately within the buffer and encode l_r in the beginning of the buffer.
    - Otherwise, encode l_r in the beginning of the buffer.

3.3 Huffman Decoding

Naive implementations of Huffman decoding transit the Huffman binary tree bit by bit. This is slow. To improve performance, we adopted a method to calculate transition destinations by n bits basis in advance [22]. Logically, this converts the Huffman binary tree to an 2^n-way tree.

As the encoded headers always have byte boundary with padding, 2, 4 and 8 are reasonable candidates for n. If n gets larger, the performance gets better, but more memory is necessary. The number of tree nodes can be calculated by 256×2^n. Since 2^n-way trees are static data, they can be shared for all sessions between a client or a server. Currently, nghttp2 and H2O use 4 for n while the HTTP/2 library in Haskell uses 8.

4 EVALUATION

This section evaluates the techniques introduced in Section 3. For measurement, we used a Xeon E5-2650Lv2 (1.70 GHz / 10 core / 25MB) without hyper threading x 2 with 64G memory for hardware and CentOS 7.2. To stabilize the benchmark results, the CPUs are set to the performance mode. The HPACK library used is the one in Haskell with GHC(Glasgow Haskell Compiler) [21] 7.10.3. The benchmark framework is criterion [3]. Each encoder/decoder encodes/decodes the set of 646 headers described above sequentially. Criterion repeats this operation in several times and calculates a mean time according to a statistical technique called bootstrapping.

Figure 2 shows the performance progression of HPACK encoding for LinearH. The labels of the x-axis are as follows:

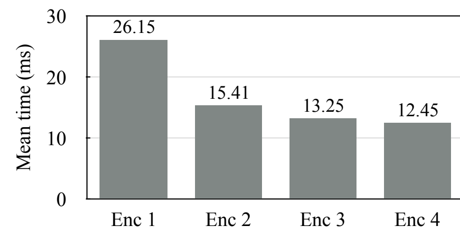

Figure 2: The performance progression of HPACK encoding for LinearH. The smaller, the better.

- Enc 1 — Linear search to convert headers to indices with the calculation of the result length in Huffman encoding.
- Enc 2 — Introducing reverse indices based on the finite map of finite maps to Enc 1.
- Enc 3 — Introducing the token based reverse indices to Enc 2
- Enc 4 — Introducing the length guessing for Huffman encoding to Enc 3.

With the all techniques described in Section 3.1 and 3.2, the final compression method is 2.10x faster than the original.

Figure 3 shows the performance of the six currently valid encoding methods. Roughly speaking, the differences between Naive and Static and between Static and Linear are the overhead of reverse index 1) and reverse index 2)/3), respectively. The performance trend of differences between Naive and NaiveH, Static and StaticH, and Linear and LinearH is the opposite, improving as more complex header tables are used. This is because fewer strings need to be encoded explicitly with Huffman encoding when the static and dynamic tables are used.

Figure 4 shows the performance of HPACK decoding with a 2^n-way Huffman tree against data encoded with LinearH. 2^4 and 2^8-way Huffman trees are respectively 2.69x and 3.92x faster than the original binary tree.

Since Haskell is a high-level programming language with a strong type system, it enables one to implement various methods described here quickly without suffering from bugs.

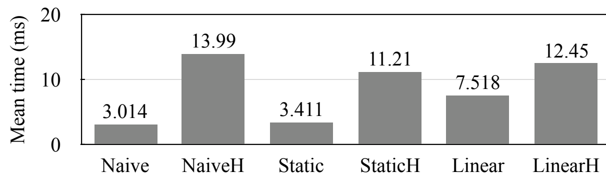

Figure 3: The performance of six compression methods for HPACK encoding. The smaller, the better.

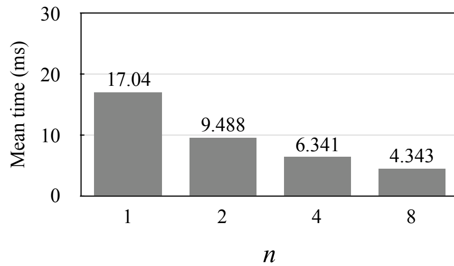

Figure 4: The performance progression of HPACK decoding for LinearH with 2^n way Huffman tree. The smaller, the better.

However, the generated binaries have overhead when compared with implementations in C with the same algorithms. We took the performance of HPACK decoding of nghttp2 (n is 4) in the same benchmark environment and the same data set with clang version 3.4.2 as C compiler. The result is 1.735 ms, which is 3.65x faster than that of Haskell.

5 CONCLUSION AND FUTURE WORK

The header compression, HPACK, is a key technology for HTTP/2 to solve the header redundancy issue of HTTP/1.1. We contributed by simplifying the HPACK specification, removing reference set which is complicated but contributes little to compression ratio. Also, we showed three implementation techniques to improve the performance of HPACK encoding and decoding. HPACK encoding with our techniques is 2.10x faster than the naive implementation. Also, our 4-bits based HPACK decoding is 2.69x faster than the original bit-by-bit implementation. We pursue further work on performance improvement of HPACK encoding and decoding.

6 ACKNOWLEDGEMENT

We would like to express gratitude to Randy Bush for reviewing this paper. We deeply thank other members of the HTTP/2 community in Japan. The authors would like to show our gratitude to three anonymous referees for their valuable feedback.

REFERENCES

[1] Compression Ratio. https://github.com/http2jp/hpack-test-case/wiki/Compression-Ratio.
[2] The CRIME attack. http://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf.
[3] A criterion tutorial. http://www.serpentine.com/criterion/tutorial.html.
[4] H2O. https://github.com/h2o/h2o.
[5] HPACK - Header Compression for HTTP/2. https://tools.ietf.org/html/draft-ietf-httpbis-header-compression-08.
[6] HPACK - Header Compression for HTTP/2. https://tools.ietf.org/html/draft-ietf-httpbis-header-compression-09.
[7] HPACK test case. https://github.com/http2jp/hpack-test-case.
[8] HTTP Header Samples. https://github.com/http2/http_samples.
[9] HTTP/2 Japan Local Activity. https://http2.info/.
[10] The http2 package. http://hackage.haskell.org/package/http2.
[11] Nghttp2: Http/2 c library. https://nghttp2.org/.
[12] QUIC: A UDP-Based Multiplexed and Secure Transport. https://tools.ietf.org/html/draft-ietf-quic-http.
[13] Re: Understanding how HPAC draft-02 works. http://lists.w3.org/Archives/Public/ietf-http-wg/2013JulSep/1135.html.
[14] SPDY Protocol. http://dev.chromium.org/spdy/spdy-protocol/.
[15] The Internet Engineering Task Force (IETF). https://www.ietf.org/.
[16] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (http/2), 2015. RFC7540.
[17] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3, 1996. RFC1951.
[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1, 1997. RFC2068, RFC2616 and RFC7230–7235.
[19] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the I.R.E.*, 1952.
[20] S. Marlow et al. *Haskell 2010 Language Report*, 2010.
[21] S. Marlow and S. P. Jones. The Glasgow Haskell Compiler. In *the Architecture of Open Source Applications*, volume 2. 2012. http://www.aosabook.org/en/ghc.html.
[22] R. Pajarola. Fast Prefix Code Processing. In *Proceedings of IEEE ITCC Conference*, 2003.
[23] R. Peon and H. Ruellan. HPACK: Header Compression for HTTP/2, 2015. RFC7541.
[24] K. Yamamoto. Experience Report: Developing High Performance HTTP/2 Server in Haskell. In *Proceedings of Haskell Symposium*, 2016.