Conversation Starter:

HTTP/2 Delta Compression & Binary Optimized Header Encoding (BOHE*)

James M Snell

* Pronounced as "bow"

A Quick Review...

Mark Nottingham and others have collected a growing list of HTTP/1.1 traffic traces... stored on Github as har files...

https://github.com/http2/http_samples

These contain lots of request and response header examples... I went through and generated some stats about those headers...

https://github.com/jasnell/compression-test/tree/master/counts

A Quick Review...

Let's look at an example, shall we?

https://github.com/jasnell/compression-test/blob/master/counts/google.com.har.txt

In a collection of request and response messages to Google.com, we have **98** separate HTTP Messages. In these, there are a total of **1,528** header fields. The values of these headers account for **26,278** bytes sent over the wire.

Let's break that down by specific examples...

A Quick Review...

In that sample of **98** messages, the Set-Cookie header field appears only **3** times but accounts for **866** bytes, **3.30%** of the overall total. By it's nature, the Set-Cookie header is highly variable*

The Via header field appears **91** times and accounts for **6,006** bytes, **22.86%** of the overall total. It's value rarely changes over multiple requests.

** it's value frequently changes across multiple requests

A Quick Review...

The Date header field appears **91** times and accounts for **2,639** bytes, **10.04%** of the overall total. The average length of each instance is **29** bytes and the value is moderately variable.

The Cache-Control header appears **87** times and accounts for **2,070** bytes, **7.88%** of the overall total. The average length of each instance is **23.79** bytes but ranges anywhere between **7** and **56** bytes. The value is generally static (typically, only max-age value changes)

A Quick Review...

The User-Agent field appears **97** times and accounts for **10,282** bytes, **24.33%** of the overall total... even tho there is only a single value for the entire set.

All date values combined account for around **7,256** bytes across **253** instances.

A Quick Review...

The point of all this is to say this:

- Some header fields are highly redundant -- they frequently repeat the same values over and over within a single session, wasting bytes-on-the-wire (example: User-Agent)
- Some header fields have extremely low density -- they waste bytes with inefficient text encodings (example: all date header fields)

Doing Better with Optimized Encodings

Here's the basic idea:

For HTTP/2, let's allow headers to optionally have binary values. Each header would have a binary or text flag

If the value is text, the HTTP/1 format is used. If the value is binary, the optimized HTTP/2 format is used.

The encodings will be designed to support 1.1 <=> 2.0 transformations with little or no data loss.

Some examples...

Dates in HTTP/1 are inefficiently encoded as text... Example:

```
Mon, 25 Jun 2012 14:34:28 GMT
```

Instead, let's set a new epoch (e.g. midnight on Jan 1, 1990) and use seconds since that epoch instead, then encode that using an unsigned variable length integer (uvarint) ... we go from **29** bytes, to **4** bytes.

Some examples...

Applying this to our sample set, we get...

Date Header :

HTTP/1 format -> **2,639** bytes

Binary format -> **364** bytes (**86.21%** reduction)

Last Modified Header:

HTTP/1 format -> **2,117** bytes

Binary format -> **292** bytes (**86.21%** reduction)

Other optimizations we can make...

Let's go ahead and use uvarint for all numeric header fields (status, content-length, age, etc)

status drops from 3-bytes to 2-bytes

content-length drops from a range of 1-6 bytes, to a range of 1-3 bytes

age drops from a range of 1-9 bytes, to a range of 1-5 bytes.

Other optimizations we can make...

Let's also try optimizing Set-Cookie and Cache-Control ...

Binary Set-Cookie format:

```
+----------------------------------------+
|H|S|B|P|M|X|X|X|len(key)|key|len(val)|val|
+----------------------------------------+
|len(path)|path|len(domain)|domain|expires|
+----------------------------------------+
|num_params|... repeating param key block |
+----------------------------------------+
H = HttpOnly bit, S = Secure bit
B = Binary value bit, P = Optional params bit
M = Expires is max-age delta not date
len(key) = 1 byte, len(val) = 4 bytes, len(path) = 2 bytes, len(domain) = 2 bytes
expires = uvarint, num_params = 2 bytes only if P is set
```

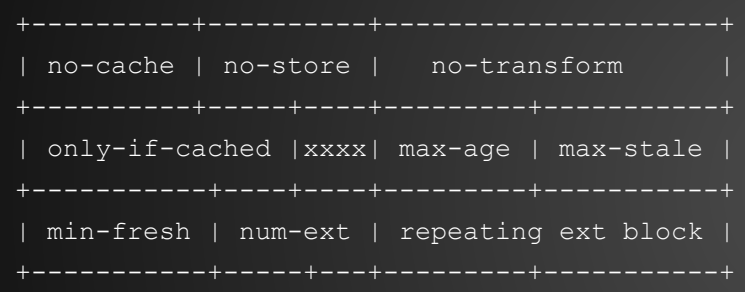
Using this format, we go from **866** bytes in the sample set, to only **132** bytes, an **84.76%** reduction in size without any loss of data.

Plus, the encoding can be easily translated back to its HTTP/1 form.

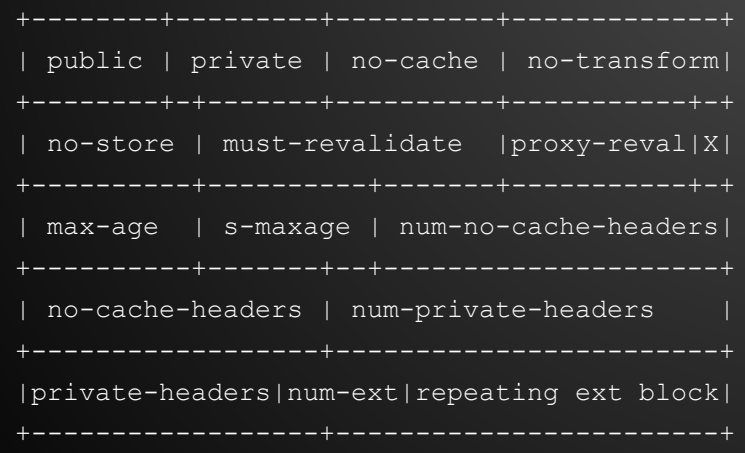
Other optimizations we can make...

How about Cache-Control?

Requests:



Responses:



The no-cache, no-store, no-transform, only-if-cached, public, private, must-revalidate and proxy-reval fields are all single bit flags.

max-age, max-stale, min-fresh, num-ext, s-maxage, num-no-cache-headers, num-private-headers and num-ext are all uvarint's

By applying this encoding to the sample set, we go from **2,070** bytes down to **736**, a **64.44%** reduction in size without any loss of data... and we can easily translate back to HTTP/1

Big picture view...

In the google.com sample set, by applying just the date, number, set-cookie and cache-control optimizations, we save **8,601** bytes total.

Other examples:

Flickr (http://goo.gl/gi2Wd) - Total Unencoded Headers = **108,746** bytes / **219** msgs
With Binary Encoding we save **20,997** bytes

LinkedIn (http://goo.gl/V1K3K) - Total Unencoded Hdrs = **28,882** bytes / **127** msgs
With Binary Encoding we save **10,978** bytes

YouTube (http://goo.gl/61ijT) - Total Unencoded Headers = **79,452** bytes / **381** msgs
With Binary Encoding we save **23,959** bytes

Let's look at Delta Encoding...

Binary-Optimized Header Encoding helps increase data-density but does nothing to address the data-redundancy issue...

Let's find a way to avoid sending redundant data over the wire...

CRIME makes stream-compressors like gzip unusable.

(Developed by Roberto Peon...)

So what is Delta Compression exactly?

In a nutshell, Only Send The Bits That Change.

Say you open a connection and send a request that includes 10 headers.

Then you send another request with 10 headers but only one has a different value than the first request.

With Delta, you tell the server to reuse the nine unchanged headers and only send the new one.

Delta.. a closer look

```
(you know it's serious when we switch to a fixed-width font!)

Let's send a request!

:method  = GET
:path    = /foo/bar/baz
:scheme  = HTTP
:host    = example.net
:version = 2.0
```

Delta.. a closer look

Let's send a request!

```
:method  = GET
:path    = /foo/bar/baz
:scheme  = HTTP
:host    = example.net
:version = 2.0
```

We know that certain headers and values are extremely common, let's take a second to define a "default dictionary" of headers and assign each a number...

```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
```

Delta.. a closer look

```
Let's send a request!

:method  = GET
:path    = /foo/bar/baz
:scheme  = HTTP
:host    = example.net
:version = 2.0
```

Now that we have our table, let's substitute the headers in our request for the ID's in our default dictionary...

```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
```

```
Delta Encoding:

[
  {op:trang,start:1,end:3},
]
```

Delta.. a closer look

Let's send a request!

```
:method  = GET
:path    = /foo/bar/baz
:scheme  = HTTP
:host    = example.net
:version = 2.0
```

For the headers that aren't in the default dictionary, let's add them (kvsto) and assign them an ID

```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
4 => :path = /foo/bar/baz
5 => :host = example.net
```

Delta Encoding:

```
[
  {op:trang,start:1,end:3},
  {op:kvsto,key:':path',
val:'/foo/bar/baz'},
  {op:kvsto,key:':host',val:'example.net'}
]
```

Delta.. a closer look

```
Let's send a request!

:method  = GET
:path    = /foo/bar/baz
:scheme  = HTTP
:host    = example.net
:version = 2.0
```

```
Delta Encoding:

[
  {op:trang,start:1,end:3},
  {op:kvsto,key:':path',
val:'/foo/bar/baz'},
  {op:kvsto,key:':host',val:'example.net'}
]
```

```
Then we send an optimized
encoding of the Delta to
the server. Since it uses
the same default
dictionary, it is able to
reconstruct the message.
```

```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
4 => :path = /foo/bar/baz
5 => :host = example.net
```

Delta.. a closer look

Now let's send a second request, but change only the :path and add a new header:

```
:method       = GET
:path         = /foo/bar/xyz
:scheme       = HTTP
:host         = example.net
:version      = 2.0
Cache-Control = no-cache
```

If we look at our table, we see that :method, :path, :scheme, :host and :version are already there. We need to change the value of :path by turning off the old value and creating a new one...

```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
4 => :path = /foo/bar/baz
5 => :host = example.net
```

Delta Encoding:

```
[
  {op:toggle,id:4},
  {op:clone,id:4,val:'/foo/bar/xyz'},
]
```

Delta.. a closer look

Now let's send a second request, but change only the :path and add a new header:

```
:method       = GET
:path         = /foo/bar/xyz
:scheme       = HTTP
:host         = example.net
:version      = 2.0
Cache-Control = no-cache
```

Then we need to add the Cache-Control header...

```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
4 => :path = /foo/bar/baz
5 => :host = example.net
6 => :path = /foo/bar/xyz
7 => Cache-Control = no-cache
```

Delta Encoding:

```
[
  {op:toggle,id:4},
  {op:clone,id:4,val:'/foo/bar/xyz'},
  {op:kvsto,key:'Cache-Control',
   val='no-cache'}
]
```

Delta ops: toggle (switch on/off), trang (toggle range), kvsto (key-value store), clone (clone existing), eref (ephemeral reference)

Delta.. a closer look

And now a third request! Removing the Cache-Control and back to the original path...

```
:method       = GET
:path         = /foo/bar/baz
:scheme       = HTTP
:host         = example.net
:version      = 2.0
```

Everything we need is already in the dictionary!

Delta Encoding:

```
[
  {op:toggle,id:6},
  {op:toggle,id:4},
  {op:toggle,id:7}
]
```

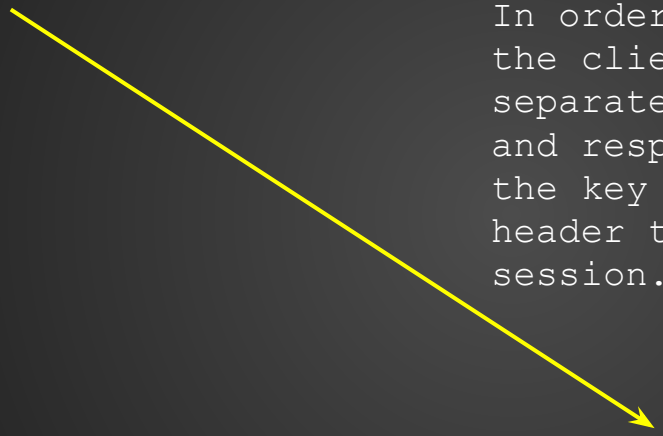
```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
4 => :path = /foo/bar/baz
5 => :host = example.net
6 => :path = /foo/bar/xyz
7 => Cache-Control = no-cache
```

Notice how we don't have to tell it to use :method, :scheme, :host and :version... those don't change so those aren't sent!

Delta.. a closer look

But wait a second...

What about this thing?



In order for Delta Encoding to work, the client AND server have to maintain separate state tables for all requests and responses. Those have to include the key name and values for every header that's been used in a given session.*

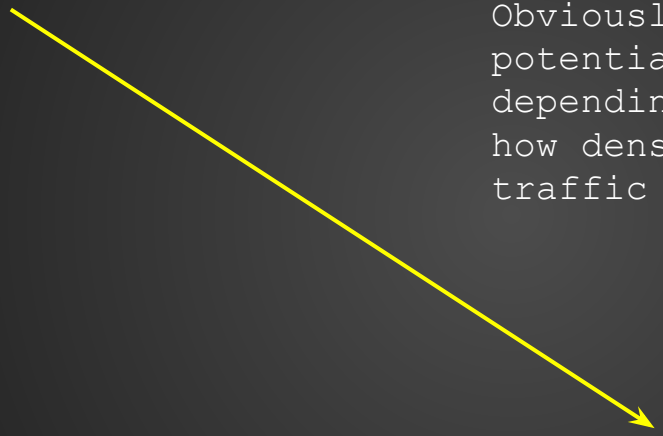
```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
4 => :path = /foo/bar/baz
5 => :host = example.net
6 => :path = /foo/bar/xyz
7 => Cache-Control = no-cache
```

* Up to a configured limit set by the receiver

Delta.. a closer look

But wait a second...

What about this thing?



Obviously, these tables have the potential of becoming quite large depending on how variable the data is, how densely it is encoded and how much traffic is on the server.

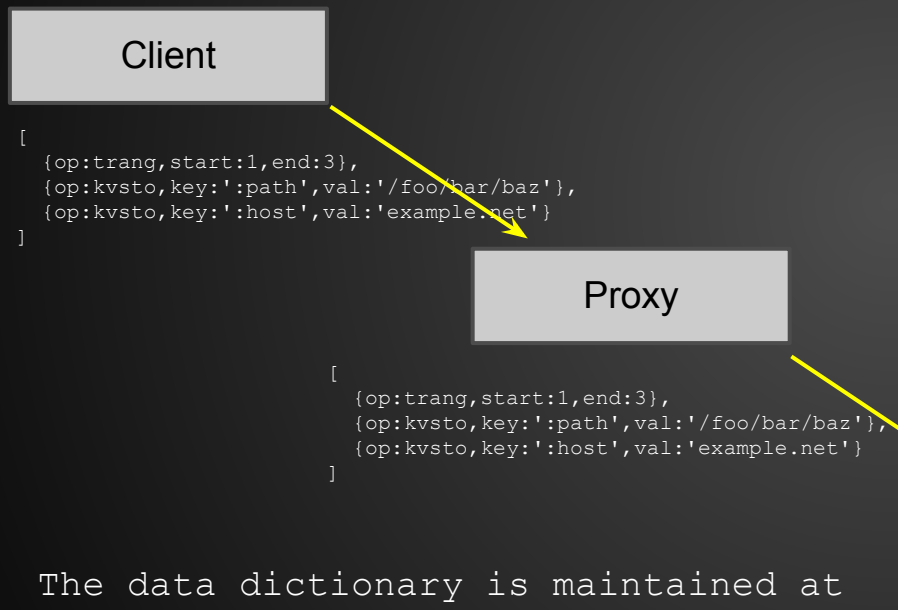
```
1 => :method  = GET
2 => :scheme  = HTTP
3 => :version = 2.0
4 => :path = /foo/bar/baz
5 => :host = example.net
6 => :path = /foo/bar/xyz
7 => Cache-Control = no-cache
```

Optimized binary encoding can help, but we do not yet have a really good idea of the impact this additional state will have on middleboxes and servers.

This is a critical issue!!

Speaking of Middleboxes...

How does Delta encoding work with proxies and such?

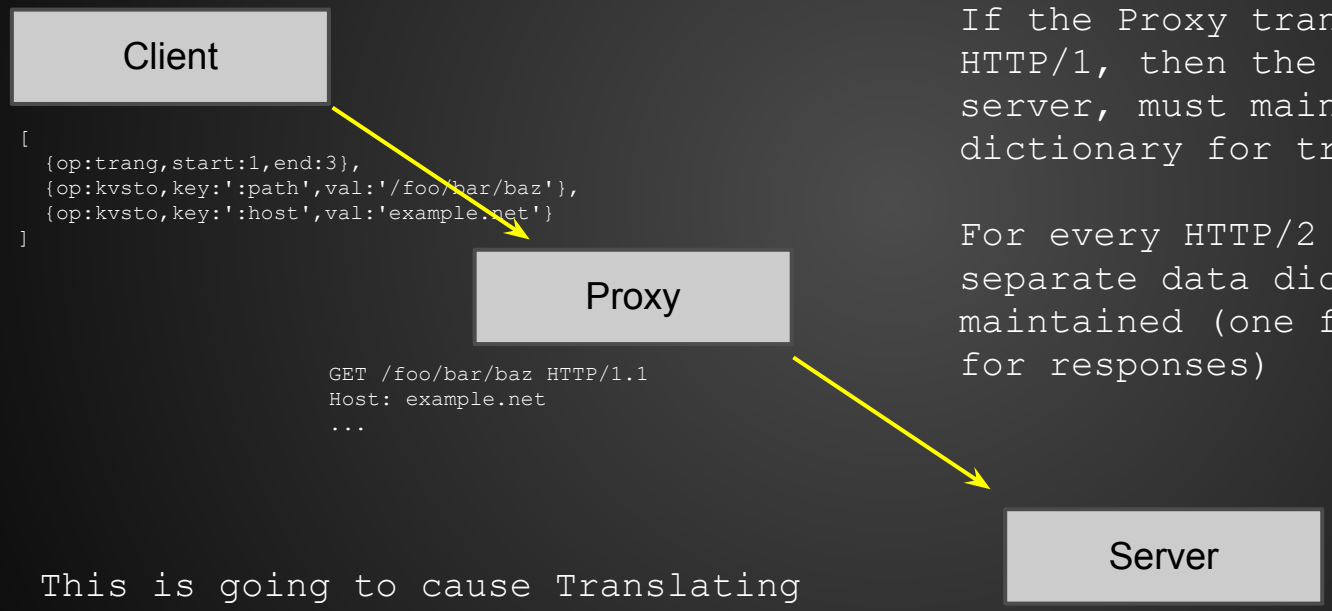


Let's say we have a Client sending an HTTP/2 request through a proxy, which also uses HTTP/2 to talk to the server... the client sends it's delta encoded message to the Proxy, which opens a connection to the server and sends along it's own delta encoded message. Basically this is just a passthrough...

The data dictionary is maintained at the Client and the Server. The Proxy has to maintain a minimal map of id's to translate back and forth. **We currently do not have any good measures of how much additional state the proxy needs to keep track of!**

Speaking of Middleboxes...

But what happens if the proxy and server use HTTP 1.1 to communicate?



If the Proxy translates HTTP/2 to HTTP/1, then the proxy, not the server, must maintain the data dictionary for translation.

For every HTTP/2 <=> HTTP/1, separate data dictionaries must be maintained (one for requests, one for responses)

This is going to cause Translating Proxies to store a significant amount of state depending on the current load on the proxy. **Again, We do not yet have any measurements of this yet!**

Some additional points on Delta...

- The receiver gets to put an upper limit on the size of the state table. The lower the limit, the worse the compression will be.
- Delta uses huffman coding on header text values, with different huffman-tables for the request vs response. the huffman values are EOF terminated, with the start of each string byte-boundary aligned (for easy/cheap) addressing.
- Delta uses 'header groups' (by default, unless the server says otherwise, you get only one of these). In that case, the set of 'toggl' instructions may change, as a 'header group' is simply a collection of things that we say are in a set of headers.
- It's not yet clear if there needs to be a way for the receiver to tell the sender to reset the compression state. This could be dangerous.
- Delta provides the most benefit with highly redundant, low-variability header fields. Still need mechanisms to increase data density.
- So far, it ***appears*** that Delta is resistant to CRIME style attacks.

BOHE + Delta...

By combining Binary Optimized Header Encoding with Delta, we reduce redundancy and improve data density. This lowers both bits on the wire and the size of the stored compression context.

BOHE also makes parsing header data much more efficient.

Need to collect a lot more data!

What next?

0. Set specific priorities and goals for improvement

1. Identify appropriate binary-optimized encodings for selected headers

2. Test Delta Encoding thoroughly using as much sample data as possible. We need input from developers!