Network Working Group
Internet-Draft
Expires: September 30, 2012

W. Tarreau
Exceliance
A. Jeffries
Treehouse Networks Ltd.
A. de Croy
Qbik New Zealand Ltd.
P-H. Kamp
Varnish Cache Project
March 29, 2012

Proposal for a Network-Friendly HTTP Upgrade
draft-tarreau-httpbis-network-friendly-00

Abstract

This document proposes an upgrade to HTTP messaging which aims at being faster, more robust and more friendly to mobile networks than the current version, while retaining the same semantics and offering a high enough compatibility level to make it possible to implement highly efficient gateways between existing implementations and this presently described version, thus offering a smooth upgrade path for legacy applications.

Table of Contents

1. Introduction

HTTP/1.1 relies on a base designed 15 years ago for use in a context which has significantly evolved over the years. Applications have become mostly stateful with sessions spanning over multiple connections. Network intermediaries have been installed everywhere between clients and servers for various purposes ranging from caching and filtering to load-balancing and off-loading. Enterprise networks rely on HTTP for almost all inter-server communications. Mobile networks are becoming prevalent in HTTP traffic, and at the same time they suffer from important constraints imposed by the medium, such as a higher latency and a higher loss rate than wired networks. HTTP itself is a very verbose protocol which magnifies issues specific to these environments. Web usage has changed, with social networks connecting millions of people and resulting in some sites having to deal with hundreds of thousands of concurrent connections, and front end components having to forward incoming requests to the proper server as quickly as possible.

Economics have changed too, making it attractive for some groups to attack business-critical sites. DDoS authors rely on the ratio between the cost of processing traffic for their victim versus the cost of building the attack. HTTP has inherited 15 years of improvements and total backwards compatibility with the original design, making it hard to parse and process, with a number of ambiguous situations left to the implementation's choice. Current model's corner cases with its moderately high parsing cost contributes to the success of these attacks by making it quite expensive for server-side components to ignore undesired requests.

1.1. Background

Many internet users rely on asymmetric links to connect to the net (POTS, ADSL, HSPA, ...). Downstream to upstream ratios of 4:1 are quite common, sometimes reaching high figures like 20:1 or even more in ADSL2+ or HSDPA.

HTTP relies on header-based messages in both directions, with bodies more often in the response messages than in request messages, resulting in the upstream traffic being mostly composed of headers. Most header field names and values are repeated unchanged over multiple requests or responses from the same sender.

For historical reasons, request headers are much larger than response headers. The User-Agent and Referer header fields usually take a significant size, and cookies can be so large that some sites prefer to register a separate domain for statics to save the browser from sending them when fetching static objects.

The one-request-at-a-time model is not suited at all to high BDP links such as the ones used in mobile environments. The only way to fill at least one direction of the link bandwidth on high latency links such as HSDPA is to fetch many objects in parallel. Pipelining enables this but is not supported by all servers, so user agents are often configured to use a large number of concurrent connections instead in order to parallelize objects retrieval, wasting bandwidth with payload-less TCP packets, wasting server resources, and taking more time to converge to the optimal CWND.

Many sites involve a large number of small objects to compose a page, typically smaller than 2 kB ([WebMetrics]), which make it hard to fill the downstream link before filling a smaller upstream even when pipelining is used.

Still, the shortcomings above are probably transient. With HSPA+ reaching 168 Mbps downstream and 20 Mbps upstream in 3GPP-Rel10 ([4gamericas]), and with Google's advice of running TCP stacks with INITCWD=10, it seems reasonable to expect that request header size on the wire will not remain the limiting factor forever, which implies that reducing the number of round trips and header processing costs will become more important than optimizing the network usage reduction alone.

1.2. Improvements

This proposal focuses on four improvements over HTTP/1.1 :

- Binary encoding of headers fields : header field names are encoded and their sizes advertised to speed up lookup
- Grouping of common headers fields : a section defines all header fields common to several subsequent messages, avoiding repetition
- Request and response multiplexing : requests and responses may be delivered in parallel and out of order
- Layering model : more friendly to intermediaries, saves header field lookups and memory copies

Backwards compatibility is an absolute requirement so that gateways can be built to present HTTP/1.1 servers to the world with the new protocol version. This should become even more obvious at mobile operators where it is likely that gateways will present the whole HTTP/1.1 internet to mobile users in HTTP/2.0. Therefore, semantics must not be affected.

2. Principles of operation

This draft proposes a mechanism to exchange messages in parallel over an established bidirectional connection with support for out of order

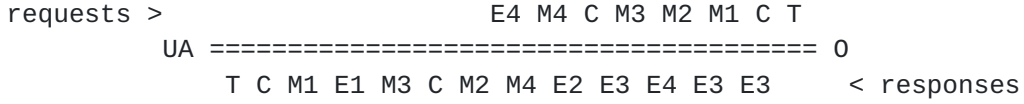
processing and delivery.

In order for messages to flow in both directions out of order, some delimiters are needed. Thus, the protocol is a stream of frames which can be of the following types :

- Transport Frame : this frame is only allowed once in each direction and advertises a set of header fields that the sender knows are invariant for this connection and that must be considered present for all messages passing over that connection
- Common Frame : this frame may appear as often as needed and advertises sets of header fields that the sender thins will be common to several upcoming messages and are worth advertising only once
- Message Frame : this frame holds a request or response message with message-specific header fields but without any message body
- Entity frame : this frame carries all or part of a message body
- Control frame : various control frames such as Ping/Pong/Pause/ Abort/Close are planned but not described here yet (TBD)

Frames which are part of the same message will generally include the reference to the request which initiated the frame, which simply corresponds to the request arrival order over the connection. This is particularly important since responses may appear in any order.

If we note 'T', 'C', 'Mx' and 'Ex' the Transport Frame, Common Frame, Message Frame number 'x' and Entity Frame number 'x', the stream between a user agent (UA) and an origin server (O) could be represented like this :



In the diagram above, the client has sent 4 requests and the server has responded to all of them in a slightly different order and with some payload interleaved. In general, over a connection, there will be in each direction zero or one Transport Frame, zero or a few Common Frames, one or more Message Frames, and zero or more Entity Frames.

2.1. Frame encoding

NOTE: the proposed encoding is a work in progress and subject to change

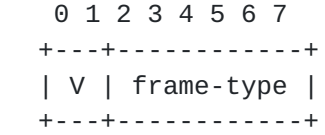
Frames use reasonably low overhead. Some frames will need to indicate a request number, while others won't. All frames start with a frame type octet indicating the frame type and the HTTP version.

Frame types between 0 and 31 are standard frames and have their own format. Frames types 32 to 63 are extension frames which all follow the same unambiguous format. Such frames are not described here and are left for future work or may even be dropped if considered unneeded.

In order to associate frames to a given request, response frames and Entity frames will include a 16-bit request number. The request number correspond to the arrival order of the request over the connection and automatically wraps past 2^16, meaning that no more than 65536 outstanding requests are supported over a single connection. In practice this should be more than enough considering that :

1. current HTTP implementations only support one outstanding request;
2. TCP congestion and losses affect all requests at the same time, so it is unlikely that browsers will push more than a few hundreds requests in parallel.

The two higher bits of the frame type octet indicate the HTTP version, and the lower 6 bits indicate the frame type :



V stands for the HTTP version. Possible values for these 2 bits are:

- 00: HTTP/1.0
- 01: HTTP/1.1
- 10: HTTP/2.0
- 11: other version

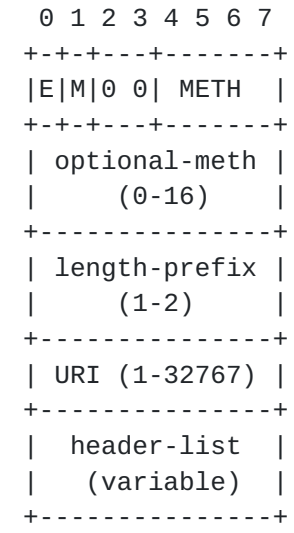
The frame type is defined below :

```
frame     = frame-type frame-body
          = %x00 tra-frame          ; Transport Frame
          / %x01 com-frame          ; Common Frame
          / %x02 req-frame          ; Request Frame
          / %x03 sts-frame          ; Status Frame
          / %x04 sef-frame          ; Small Entity Frame
          / %x05 mef-frame          ; Medium Entity Frame
          / %x06 lef-frame          ; Large Entity Frame
          / %x07 hef-frame          ; Huge Entity Frame
          / %x08 trl-frame          ; Trailers Frame
          / %x09 abt-frame          ; Abort Frame
          / %x0A-1F                 ; reserved frame  (control etc...)
          / %x20-3F ext-frame       ; extension frame

tra-frame  = header-list            ; Transport Frame
com-frame  = header-list            ; Common Frame
trl-frame  = header-list            ; Trailers Frame
ext-frame  = frame-len opaque       ; extension frame
frame-len  = 4*OCTETS               ; 32-bit frame length encoding
```

2.1.1. Request Frame (frame type = 2)

The Request Frame is a Message Frame composed of a bit indicating if an Entity Frame is expected for this request, a method, a URI and an optional header list.



- E : Entity is present. One or more Entity Frames are expected if this bit is 1, while 0 indicates no entity is attached to this request.
- M :

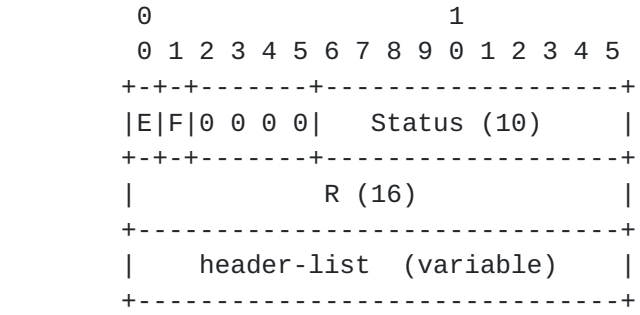
0: METH contains the method length minus 1, between 1 and 16 bytes, and the method follows in the optional-meth field
1: METH contains a method number among the following values and no optional-meth field is provided :
0: OPTIONS
1: GET
2: HEAD
3: POST
4: PUT
5: DELETE
6: TRACE
7: CONNECT
other: TBD

- optional-meth: this is the method written in plain text then M=0.
- length-prefix: this is the number of octets representing the request URI encoded as a 15-bit quantity between 0 and 32767 on either 1 or 2 octets, using the variable length encoding described in the header field encoding section.
- URI: this is the request URI, it is of exactly length-prefix octets
- header-list: this is the encoded list of headers specific to this request, see below.

In many cases, this frame alone will be enough to send a complete request, which will then be as small as just a frame-type octet followed by 1 byte for the method, one byte for the URI length, the URI itself and the null byte to end the header list. This sums up to the URI length plus 4 bytes.

2.1.2. Status Frame (frame type = 3)

The Status Frame is composed of a bit indicating if an Entity Frame is expected for this response, a bit indicating if this response is a final response or an interim response, a status and a request number.



- E : Entity present. One or more Entity Frames are expected if this bit is 1. 0 indicates no entity is attached to this response.
- F : Final response. All responses except those with status 1xx are final and have this bit set. Responses 1xx are not final and have this bit cleared.
- Status : This is the HTTP status encoded over 10 bits.
- R : this is the associated request number encoded on 16 bits.
- header-list: this is the encoded list of header fields specific to this response, see below

2.1.3. Entity Frame (frame types = 4..7)

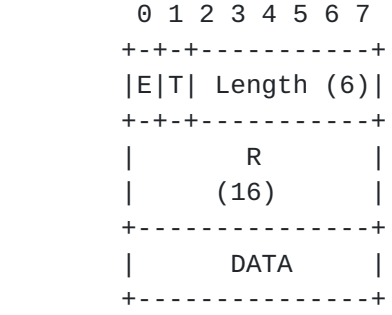
The Entity Frame is composed only of payload which in principle is very comparable chunked encoding. The payload length is encoded on a variable size so for this we have 4 types of Entity Frames which are totally similar except for the data length encoding :

- Small frames : length is encoded on 6 bits (64 bytes max). These frames are useful for uploading small contents such as credentials, as well as to send an empty final frame.
- Medium frames : the length is encoded on 22 bits (4 MB max). These will probably be the most common ones.
- Large frames : the length is encoded on 32 bits (4 GB max). These ones might also be very common.
- Huge frames : the length is encoded on 64 bits (18 EB max). These ones will probably only be used in CDN environments where use of sendfile() is desirable for very large files, when multiplexing is not involved.

An entity length contains a bit indicating if more Entity Frames are expected, a bit indicating if a Trailers Frame is expected, a length, a request number, and data.

2.1.3.1. Small Entity Frame (frame type = 4)

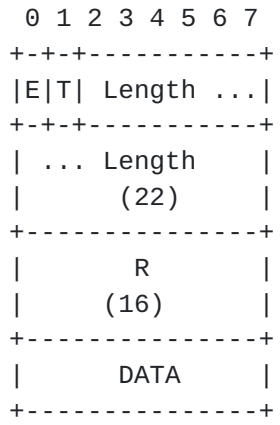
This is the smallest Entity Frame, which can be used to transfer between 0 and 63 bytes of payload and can be as small as one single byte (0).



- E : More Entity Frames present. One or more Entity Frames are expected if this bit is 1, while 0 indicates this is the last Entity Frame for this request number.
- T : 1 if a Trailers Frame is expected, otherwise zero.
- Length : this is the length of the entity data in octets, encoded on 6 bits.
- R : this is the associated request number encoded on 16 bits.
- DATA (0..Length bytes) : entity payload.

2.1.3.2. Medium Entity Frame (frame type = 5)

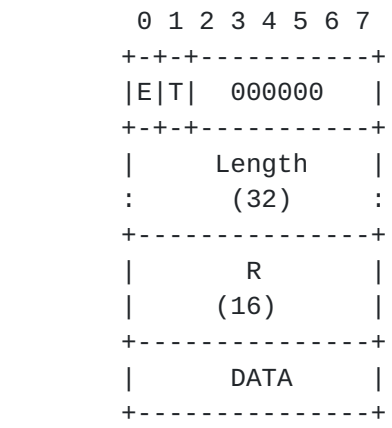
This frame type combines the small length field with 16 more bits to encode up to 22 bits of length.



- E : More Entity Frames present. One or more Entity Frames are expected if this bit is 1, while 0 indicates this is the last Entity Frame for this request number.
- T : 1 if a Trailers Frame is expected, otherwise zero.
- Length : this is the length of the entity data in octets, encoded on 22 bits, with the 6 higher offset bits in the first octet.
- R : this is the associated request number encoded on 16 bits.
- DATA (0..Length bytes) : entity payload.

2.1.3.3. Large Entity Frame (frame type = 6)

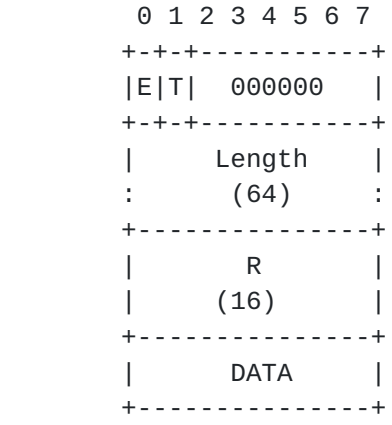
This frame type only uses a 32-bit length field.



- E : More Entity Frames present. One or more Entity Frames are expected if this bit is 1, while 0 indicates this is the last Entity Frame for this request number.
- T : 1 if a Trailers Frame is expected, otherwise zero.
- Length : this is the length of the entity data in octets, data encoded on 32 bits.
- R : this is the associated request number encoded on 16 bits.
- DATA (0..Length bytes) : entity payload.

2.1.3.4. Huge Entity Frame (frame type = 7)

This is the largest Entity Frame, used to code up to 64-bit lengths.



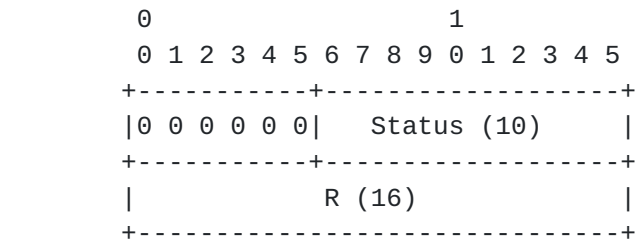
- E : More Entity Frames present. One or more Entity Frames are expected if this bit is 1, while 0 indicates this is the last Entity Frame for this request number.
- T : 1 if a Trailers Frame is expected, otherwise zero.
- Length : this is the length of the entity data in octets, encoded on 64 bits.
- R : this is the associated request number encoded on 16 bits.

- DATA (0..Length bytes) : entity payload.

2.1.4. Abort Frame (frame type = 9)

The Abort Frame is composed of a status and a request number. It is returned by a server if an error caused the request to be aborted in the middle of a transfer. It may also be emitted by a client which wishes to abort a transfer (either download or upload) without breaking the connection. The receiver of such a frame must immediately stop any communication with this request number and not expect any further data for this request number in the same direction. The connection is not affected and other requests continue their normal work.

TBD: it seems to make sense to have an ACK frame (or maybe respond with an ABRT frame) for this frame in case of a client abort so that the client knows the server has really stopped sending anything for this request.



- Status : This is the HTTP status encoded over 10 bits in case of a server-initiated abort. TBD: would this be useful to let a client tell the server what it wants to abort ? Maybe intermediaries could tell servers the client is gone.
- R : this is the associated request number encoded on 16 bits.

2.1.5. Header fields encoding

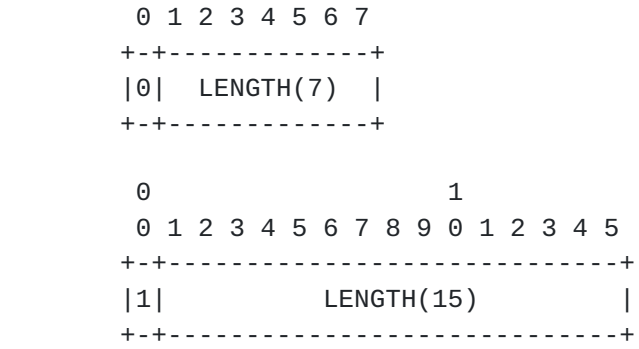
Header fields have two parts, one which is the field-name and one which is the field-value. A header-list is defined as a sequence of header fields terminated by and end-of-headers tag (%x00).

Based on the observations from Appendix A, the current proposal suggests to encode header field names either as a registered well-known field-name identifier, or as a 7-bit name length followed by the header's name. This operation will permit to reduce up to 127 header names to one single byte each. For optimal efficiency, the assignment of header names to entries has to be done based on wider analysis. It is suggested that no more than half of the possible entries are assigned, in order to leave room for newer headers, or for dynamically assigned header fields.

In order to support larger field values, the field-value is encoded as a variable sized length-prefix followed by a value.

```
header-list    = *( header-field ) end-of-hdr
header-field   = field-name field-value
field-name     = common-hdr / rare-hdr / rsvd-hdr
field-value    = length-prefix *( octet )
rare-hdr       = hdr-len token  ; token is [hdr-len] octets
hdr-len        = %x01-7F        ; header names may be up to 127 bytes long
common-hdr     = %x80-FE        ; 127 possible header names
rsvd-hdr       = %xFF           ; for future extensions if needed.
end-of-hdr     = %x00           ; this was the last header.
```

The length-prefix is used to efficiently encode a length which most of the time is small but sometimes needs to be large. The principle is that small lengths between 0 and 127 are encoded on a single octet, and lengths between 128 and 32727 are encoded on two octets. (TBD: decide if we should encode 128 to 32895 instead). This is appropriate for field-values and for the request-URI :



2.2. Grouping headers

Observations from Appendix B suggest that it is worth grouping headers for multiple consecutive messages over a single connection. Some of these headers will be connection-specific and should be common to all messages transported over the connection, while other ones will be common to a group of messages.

This proposal thus introduces the notion of sections to communicate header fields. These sections have a different lifetime. They are only valid for a hop-by-hop connection, and have no end-to-end meaning. The header fields will be split into three sections :

- Transport Header Fields
- Common Header Fields
- Message Header Fields

The Transport Header Fields section holds all headers fields that are

specific to the connection and invariant over all the connection. These headers are transmitted in a Transport Frame only once at the beginning of the connection and never after that. The recipient of any message always considers the Transport Header Fields when parsing a message coming over that connection. While a user agent may use this section to present a number of invariant header fields such as the User-Agent, Accept or Host, intermediaries which are able to multiplex requests over a single connection will probably not use it much, maybe only for rare constant header fields such as Via, or even Host if the connection was opened for a specific Host field value. It is important to note that since this header field section only applies to a hop-by-hop connection, only context-specific header fields will be there so all header fields present there must be considered after those of all other sections in order to maintain ordering (eg: chaining multiple Via fields).

The Common Header Fields section holds a number of headers which are common for a number of subsequent requests, and may be updated at any time. These header fields are transmitted in a Common Frame. All headers fields contained in the Common Header Fields section are implicitly present in any subsequent message until the next Common Header Fields section is encountered, which voids and replaces any previous Common Header Fields section. Header fields eligible to this section are all those which are expected to appear multiple times over a connection, without necessarily being invariant. A user agent will likely use this section to send Cookie, a Referer or even Authentication credentials. A multiplexing intermediary may use this section when forwarding multiple requests at once from the same user agent, or to store almost invariant headers fields such as Host. All header fields present in this section must be considered after the Message Header Fields section and before the Transport Header Fields section.

The Message Header Fields section represents all header fields that are attached to a given message (request, response, trailers...). The recipient of a message will reconstruct the original message headers by concatenating the Message Header Fields section, the last Common Header Fields section and the Transport Header Fields section. Respecting this order is important so that some hop-by-hop header fields are correctly appended last (for instance, Via or X-Forwarded-For).

All these sections are proper to a connection only. Each hop is free to rearrange them as it likes for the other side connection if it estimates it is appropriate, provided that the resulting set of header fields remains the same once reassembled.

Doing this is not only interesting for the sender which saves

upstream bandwidth, but also for the recipient which has to process much less header fields for each message. If an intermediary has to rewrite, insert or delete a header field which is in either the Transport or Common section, it only does so once, and not for every request or response. Common rewriting practices include rewriting the Host header field in requests and removing the Server header field from responses. Another example of CPU savings if gained by not having to perform more layer7 inspection than necessary. For instance, a front load balancer which selects the target server based on the Host header field alone might simply splice the client and the server connection together when it receives a Host header field from the client in the Transport Frame.

Example of request path with a client, a load balancer and two servers. All connections are fresh new, both from the client to the LB, and the LB to the servers. Hence, all request numbers start at 0 on each connection. TF, CF and RF designate the Transport Frame, the Common Frame and the Request Message Frame respectively.

Here, the LB maps the request numbers between the connections :

- C: Req #0 <=> dynamic: req #0
- C: Req #1 <=> static: req #0

- C: Req #2 <=> static: req #1

2.3. Sending Requests

A client wishing to send requests does not need to verify that the recipient accepts enough requests. It simply writes a new request message to the stream, which implicitly gets a new request number. If the recipient is not reading, the request will just wait somewhere along the path as it does with usual HTTP pipelining.

If a client wishes to send a request with a body, it must not send multiple interleaved bodies from different requests unless it has verified that the recipient is willing to process them. Otherwise, it would be possible to enter a deadlock with interleaved partial bodies sent to a server which supports only one outstanding request at a time. The proper way to proceed is to send the first request without prior check, but if other request bodies have to be interleaved before the first request is complete, then the client must first make use of the Expect: 100-Continue header field and wait for the server to send the non-final 100 response corresponding to the same request, thus proving it is able to read multiple requests at once. In practice this is not an issue since clients sending multiple POSTs at once are not common.

Note that this restriction does not apply to response bodies from the servers, as the servers will always respond to requests that have been received, so for each response, it is certain that there is a client listening.

3. Connection Setup

The protocol is designed to operate over various stream-based bidirectional connections, and to be upgradable from HTTP/1.1, offering a smooth upgrade path to existing applications.

A client wishing to use this protocol to communicate with an origin server for which the protocol support is unknown will send the first request in HTTP/1.1 format, with an additional Upgrade: HTTP/2.0 header :

```
GET / HTTP/1.1
Host: www.example.com
Connection: Upgrade
Upgrade: HTTP/2.0
...
```

If the server does not support the new protocol, it will simply respond to the client using HTTP/1.1 :

```
  HTTP/1.1 200 OK
  Content-length: 243
  Content-type: text/html
  ...
```

However, if the server supports the new protocol, it will first emit an interim response then will immediately respond with the final response in HTTP/2.0, just as if it had received the first request in HTTP/2.0 :

```
  HTTP/1.1 101 Switching Protocols
  Connection: Upgrade
  Upgrade: HTTP/2.0

  [ tra-frame ] [ com-frame ] [ sts-frame ] ...
```

4. Improving the handshake to save bandwidth

In order to save network exchanges, two new hop-by-hop header fields could be registered :

- Http2-Th : list of the headers fields to keep in the Transport Header Fields section after the Upgrade
- Http2-Ch : list of the headers fields to keep in the Common Header Fields section after the Upgrade

This way, a client could make the server keep various information such as the Host and User-Agent in the Transport Header Fields section and the Referer as a Common Header Fields section for next requests, so that only the request-uri has to be sent after the upgrade :

```
  GET / HTTP/1.1
  Host: www.example.com
  User-Agent: Mozilla/5.0 (X11; U; Linux i686 (x86_64); en-US; rv:1.9.1.16)
Gecko/20101210 SeaMonkey/2.0.11
  Accept: text/css,*/*;q=0.1
  Accept-Language: en-us,en;q=0.5
  Accept-Encoding: gzip,deflate
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
  Referer: http://www.ietf.org/meeting/83/index.html
  Cookie: styleSheet=1
  Connection: Upgrade, Http2-Th, Http2-Ch
  Upgrade: HTTP/2.0
  Http2-Th: Host, User-Agent, Accept, Accept-Language, Accept-Encoding, Accept-
Charset
  Http2-Ch: Referer, Cookie
  ...
```

5. Improving the handshake to save time

Some minimum testing suggests that many consecutive requests will only vary by the request-uri. This is the case for instance, for requests for static objects fetched from a same host. In this situation, the sender would like to benefit from HTTP pipelining/ multiplexing without knowing whether the whole chain supports the protocol upgrade. The solution consists in enumerating the expected upcoming requests in a specific header field, that the recipient will decide to consider as individual requests sharing the same Common Header Fields section and Transport Headers Fields section.

These additional requests will take number 1 and onwards. The recipient will just have to indicate in a header field of the handshake response the highest number of the pending requests its is willing to process. If the client does not receive this header field in the response handshake, then it knows that the next hop to the server does not support this optimization and it is free to send these requests individually once the handshake completes.

For this we would register two more hop-by-hop headers fields, one for the request and one for the response :

- Http2-Reqs : comma-delimited list of request-uri represented as quoted-strings.
- Http2-Accepted-Reqs : integer number representing the number of the last accepted request for which a response message will be delivered

Example :

```
  GET / HTTP/1.1
  Host: www.example.com
  User-Agent: Mozilla/5.0 (X11; U; Linux i686 (x86_64); en-US; rv:1.9.1.16)
Gecko/20101210 SeaMonkey/2.0.11
  Accept: text/css,*/*;q=0.1
  Accept-Language: en-us,en;q=0.5
  Accept-Encoding: gzip,deflate
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
  Referer: http://www.ietf.org/meeting/83/index.html
  Cookie: styleSheet=1
  Connection: Upgrade, Http2-Th, Http2-Ch, Http2-Reqs
  Upgrade: HTTP/2.0
  Http2-Th: Host, User-Agent, Accept, Accept-Language, Accept-Encoding, Accept-
Charset
  Http2-Ch: Referer, Cookie
  Http2-Reqs: "/css/ietf.js", "/css/ietf.css", "/css/ietf4.css", "/css/
ietf3.css"
  ...

  HTTP/1.1 101 Switching Protocols
  Connection: Upgrade
  Upgrade: HTTP/2.0
  Http2-Accepted-Reqs: 4

  [ tra-frame ] [ com-frame ] [ sts-frame ] ...
```

6. Directions for future work (TBD)

This draft in its state currently lacks a number of things :

- the frame encoding could be much better with some specific fields always at the same position (for instance, the request number).
- date formats have not been discussed but are expensive to parse at the moment and cause issues with header folding due to the comma. A binary encoding of a single scalar (eg: epoch in milliseconds) would be much more suited.
- multiple header occurrences might be better handled by having a repetition of the header value than by keeping the comma inside the header field value. Several options will have to be explored.
- watch out other working groups (eg: hybi) to see how extensions may be efficiently added at a low cost (eg: per-frame compression, ...)
- determine if some sets of features are more suited to the current most common usage (loading a web page in a graphical browser) than to some other usages such as interactive use of XHR, displaying widgets on a TV, forwarding a request between a load balancer and an origin server, or making one's backups online ;

some of the SPDY experience will probably be useful here.

- identify what is needed to operate over datagram-based transport protocols such as UDP and if it is worth having a single protocol for all transports.
- protocol handshake if another port is to be used.
- use delta encoding for header updates ? Would this void the need for Transport Header Fields ?
- replace "Host" with "Base" which would include a scheme ?

7. IANA Considerations

The Upgrade field header value "HTTP/2.0" might require a IANA assignment.

8. Acknowledgements

This document was produced using the xml2rfc tool [RFC2629].

9. Change log [RFC Editor: Please remove]

draft-tarreau-httpbis-network-friendly-00: original version, 2012-03-29.

10. References

10.1. Normative References

10.2. Informative References

[4gamericas] "4G Mobile Broadband Evolution - 3GPP Release 10 and Beyond", 2011, <http://www.4gamericas.org/documents/4G%20Americas_3GPP_Rel-10_Beyond_2.1.11%20.pdf>.

[RFC2629] Rose, M., "Writing I-Ds and RFCs using XML", RFC 2629, June 1999.

[RFC2991] Thaler, D. and C. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection", RFC 2991, November 2000.

[RFC4864] Van de Velde, G., Hain, T., Droms, R., Carpenter, B., and E. Klein, "Local Network Protection for IPv6", RFC 4864, May 2007.

[RFC6296] Wasserman, M. and F. Baker, "IPv6-to-IPv6 Network Prefix

Translation", RFC 6296, June 2011.

[RFC6438] Carpenter, B. and S. Amante, "Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels", RFC 6438, November 2011.

[WebMetrics] Ramachandran, S., "Let's make the web faster - Web metrics: Size and number of resources", 2010, <http://code.google.com/speed/articles/web-metrics.html>.

Appendix A. Analysis of header field occurrences

An analysis of 30797 requests received by a server located behind a load balancer indicates that a small set of headers is very common :

- 101 different header names were found in requests
- 9.6 headers on average were present in each request
- headers total 648 bytes per request on average
- 4 header names were present in 100% of the requests (Host, User-Agent, Accept, X-Forwarded-For)
- 4 header names were present in 94% of the requests (Accept-Language, Connection, Accept-Encoding, Referer)
- 1 header name was present in 75% of the requests (Cookie)
- 4 header names were present in more than 10% of the requests (Accept-Charset, UA-CPU, Keep-Alive, Cache-Control)
- 3 header names were present in more than 5% of the requests (Via, If-Modified-Since, If-None-Match)

The analysis of the responses was even comparable, with only 22 different header names (one single site) :

- 8.6 headers on average were present in each request
- headers total 257 bytes per request on average
- 3 header names were present in 100% of the requests (Server,Date,Connection)
- 2 header names were present in 97% of the requests (Content-Type,Content-Length)
- 1 header name was present in 67% of the requests (Last-Modified)
- 9 header names were present in more than 10% of the requests (ETag, Accept-Ranges, Expires, Cache-Control, Pragma, P3P, Vary, Content-Encoding, X-Pad)
- 2 header names were present in more than 5% of the requests (Cache-Control, Set-Cookie)

It is also worth noting that 40 different header names represent 562532 of the 564043 header occurrences (99.73%). These header names alone are responsible for 175 bytes per request on average.

Appendix B. Analysis of header field length

The analysis above shows that many request headers are almost always identical. Among the 648 bytes per request, we can see that :

- The User-Agent header is sent with every request yet does not change. This header alone was responsible for 145 bytes on average per request.
- The Referer header is sent with every request, while it remains unchanged for 9.75 requests on average, sometimes with up to 38 requests using the same. This header accounts for 91 bytes per request on average.
- The Cookie header is sent with 75% of the requests and only changes on average once every 9.6 such requests. It accounts for 184 bytes per request.
- The Accept-Language, Accept-Encoding, Accept-Charset and Accept headers are constant across all requests and account for 121 bytes per request.
- The transport-specific headers such as Connection, Host, X-Forwarded-For and Keep-alive did not change for a given client. Together they account for 84 bytes per request on average.
- In the end, only If-Modified-Since and If-None-Match were changed at almost very request. These ones are found in 11% of the requests where they account for 47 bytes on average.

The analysis of the responses showed that header values were even more constant, with only the following ones changing with almost every request :

- Content-Length (found in 94% of the responses)
- Last-Modified (found in 67% of the responses)
- ETag (found in 61% of the responses)

Authors' Addresses

Willy Tarreau
Exceliance
R&D Produits reseau
3 rue du petit Robinson
78350 Jouy-en-Josas
France

Email: w@1wt.eu
URI: http://www.exceliance.fr/

Amos Jeffries
c/- 130 Fox St
Hamilton East
Hamilton, 3216
New Zealand

Phone: +64 21 293 4049
Email: amos@treenet.co.nz
URI: http://treenet.co.nz/

Adrien de Croy
Qbik New Zealand Ltd.
28 York St
Parnell
Auckland 1052
New Zealand

Email: adrien@qbik.com
URI: http://www.wingate.com/

Poul-Henning Kamp
Herluf Trollesvej 3
Slagelse, DK-4200
Denmark

Phone: +45 21 72 05 25
Email: phk@varnish.org
URI: http://varnish.org/