Network Working Group
Internet-Draft
Expires: September 2, 2012

R. Trace
A. Foresti
S. Singhal
O. Mazahir
H. Nielsen
B. Raymor
R. Rao
G. Montenegro
Microsoft

Mar 2012

HTTP Speed+Mobility
draft-montenegro-httpbis-speed-mobility-01

Abstract

The design of HTTP--how every application and service on the web communicates today--can positively impact user experience, operational and environmental costs, and even the battery life of the devices you carry around.

Improving HTTP starts with speed. Apps--not just browsers--should get faster too. More and more, apps are how people access web services, in addition to their browser. Improving HTTP should also make mobile better, particularly to ensure great battery life and low network cost on constrained devices. People and their apps should stay in control of network access. Finally, to achieve rapid adoption, HTTP 2.0 needs to retain as much compatibility as possible with the existing Web infrastructure. Done right, HTTP 2.0 can help people connect their devices and applications to the Internet fast, reliably, and securely over a number of diverse networks, with great battery life and low cost.

This document describes "HTTP Speed+Mobility," a proposal for HTTP 2.0 that emphasizes performance improvements and security while at the same time accounting for the important needs of mobile devices and applications. The proposal starts from both the Google SPDY protocol and the work the IETF has done around WebSockets. The proposal is not a final product but rather is intended to form a baseline for working group discussion.

Status of this Memo

This Internet-Draft will expire on September 2, 2012.

Copyright Notice

Table of Contents

1. Introduction

Over the course of its almost two decades of existence, the HTTP protocol has enabled the web to experience phenomenal growth and change the world in more ways than its creators might have imagined. HTTP's designers got many design principles right, including simplicity and robustness. These charateristics allow billions of devices to support and use HTTP in a multitude of communication scenarios.

Improving HTTP starts with speed. Web sites have become complex. A single site could comprise hundreds of different elements (from images to videos to ads to news feeds and so on) that need to get retrieved by the client before the page can be fully displayed. Users expect all of this to happen securely and instantly across all their devices and applications. In many scenarios, HTTP fails to meet these expectations. Speed improvements need to apply not only for browsers but also for apps. More and more, apps are how people access web services, in addition to their browser. A key attribute of mobile applications is that they may access only a subset of the web site's data, relying on local application logic to process the data and create a presentation and interaction layer.

At the core of the speed problem is that HTTP does not allow for out-of-order or interleaved responses. This requires the establishment of multiple TCP connections for concurrency (pipelining is formally supported by the protocol but is seldom implemented in practice). The overhead in terms of additional roundtrips and dealing with TCP slow start causes a significant performance penalty. This leads to a variety of issues, such as additional round trips for connection setup, slow-start delays, and potentially connection rationing: the client may not be able to dedicate too many connections to any single server, and the server needs to protect itself from denial-of-service attacks. As a result, users are often disappointed in the perceived performance of websites.

Improving HTTP should also make mobile apps and devices better. When HTTP was first developed mobile communication was virtually non-existent, but today the mobile Web is an integral and fast-growing part of the Web. The different conditions on mobile communications require rethinking of how protocols work. For example, people want their mobile devices to have better battery life. HTTP 2.0 can help decrease the power consumption of network access. Mobile devices also give people a choice of networks with different costs and bandwidth limits. Embedded sensors and clients face similar issues. Mobile considerations require that HTTP be network efficient while simultaneously being sensitive to the limited power, computation, and connectivity capabilities of the client device. To support mobile

devices, HTTP needs to be able to "scale down" to allow clients to control the level of data received, the format of that data, and even the timing of that data.

1.1. Overview

This draft describes our proposal for "HTTP Speed+Mobility". The approach proposed focuses on all the web's end users---emphasizing performance improvements while at the same time accounting for the important needs of mobile devices and applications.

The proposal's intended outcome is a protocol that can be quickly and widely adopted in the industry, and start delivering real value to end users without imposing undue burden on hardware and software vendors, as well as administrators of legacy equipment. Implementors should also find it easy to understand due to the familiarity of some of its key concepts, which are aligned with innovations that were adopted in recent IETF specifications like WebSockets. Most important, the proposal seeks to establish a baseline for working group discussion on the potential improvements that would define HTTP 2.0.

This HTTP Speed+Mobility proposal adheres to the following principles:

- Maintain existing HTTP semantics. The request-response nature of the HTTP protocol and semantics of its messages as they traverse diverse networks must be preserved. Any deviation from this principle would represent an extension to HTTP and should be treated as such.

- Maintain the integrity of the layered architecture.

- Use existing standards when available to make it easy for the protocol to work with the current web infrastructure including switches, routers, proxies, load balancers, security systems, DNS servers, and NATs. For example, the proposal reuses the WebSockets handshake and framing mechanism to establish a bidirectional link that is compatible with existing proxies and connection models.

- Be as broadly applicable and flexible as the current protocol, and keep the client in control of content. For example, the proposal does not mandate the use of TLS or compression, leaving those features up to the client to negotiate based on its specific security, computation, and communication needs.

- Account for the needs of modern mobile clients, including power efficiency and connectivity through costed networks.

These principles are described in more detail below.

1.1.1. Maintain existing HTTP semantics

HTTP at its core is a simple request-response protocol. The working group has clearly stated that it is a goal to preserve the semantics of HTTP. Thus, we believe that the request-response nature of the HTTP protocol must be preserved. The core HTTP 2.0 protocol should focus on optimizing these HTTP semantics, while improving the transport via a new session layer. Additional capabilities that introduce new communication models like unrequested responses must be treated as an extension to the core protocol, and explored separately from the core protocol.

1.1.2. Layered Architecture

HTTP relies on an in-order, reliable transport to ensure delivery of application data. TCP has almost exclusively provided the reliable, ordered delivery of HTTP messages from one computer to another since its inception. TCP accounts for adverse network conditions such as congestion, or other unpredictable network behavior. Any HTTP 2.0 proposal should leverage the reliable transport and not attempt to replicate functions generally accepted as addressed by other layers.

Conversely, any proposals for enhancing functionality typically provided by other layers of the networking stack (e.g., congestion control provided by the transport layer) should be brought to the attention of, and discussed in, proper IETF forums (e.g., TCPM WG).

During the HTTPbis charter proposal discussion, the security and applications area directors suggested an additional paragraph on security work and authentication. If new work is undertaken in this regard, it should be done by existing IETF security groups in this area.

1.1.3. Use of Existing standards

HTTP 2.0 should prefer models that are compatible with the existing Internet and, where possible, reuse existing protocol mechanisms. One primary example is in protocol negotiation where the WG should avoid a proliferation of methods, and instead consider using the HTTP 1.1 Upgrade header as it is used in the WebSocket protocol. This will help HTTP 2.0 to be readily deployed on the existing internet, and maintain compatibility with existing web sites and client environments (such as some educational networks).

1.1.4. Client is in control of content

HTTP is used in a vast array of scenarios and a variety of network architectures. There is no "one size fits all" deployment of HTTP. For example, at times it may not be optimal to use compression in certain environments. For constrained sensors from the "Internet of things" scenario, CPU resources may be at a premium. Having a high performance but flexible HTTP 2.0 solution will enable interoperability for a wider variety of scenarios. There also may be aspects of security that are not appropriate for all implementations. Encryption must be optional to allow HTTP 2.0 to meet certain scenarios and regulations. HTTP 2.0 is a universal replacement for HTTP 1.X, and there are some instances in which imposing TLS is not required (or allowed). For example, a "random thought of the day" web service has very little need for it, nor does a sensor spewing out a temperature reading every few seconds.

Because of the variety of clients on the Internet and the number of connection scenarios, clients are in the best position to define what content is downloaded. The browser or app has firsthand information on what the user is currently doing and what data is already locally available. For example, most of the browsers in use today have powerful caches that should be leveraged to store web elements that change infrequently.

Increasingly, apps, rather than browsers, originate HTTP requests. The content retrieved by apps is usually different from that downloaded by browsers; in fact, multiple apps may access the same content for different purposes. Each app may access different subsets of the server content, with different priorities, and in different sequences according to their own rendering requirements and user interaction models. The server cannot always know the needs or intents of a particular application.

HTTP 2.0 proposals should not force the browser or app to download content that has not been requested and may already be cached. Furthermore, the client must have the option to decline unwanted or unneeded content. Clients need the ability to inform the server about cached elements that do not need to be downloaded. Ideally this feedback from the client to the server would allow for incremental approval of content to enable an efficient "push" extension to deliver the right content, with the right security and with the right formatting.

1.1.5. Network Cost and Power

Any new protocol for transporting HTTP data on the Internet must also take into account the types of systems and devices that use HTTP and

how they are connected to the Internet. The growth of the Internet of the next decade (and longer) will be fueled by mobile apps and mobile devices, as well as by the cheap, limited-capability devices envisioned by the "Internet of Things." For all these devices, speed is only one design tenet: considerations about battery life, bandwidth limitations, processor and memory constraints, and various policy mandates will also challenge designers and users. For instance, the user of a device connected over mobile broadband may need to minimize the amount of data sent in order to conserve bandwidth, minimize power usage and monetary cost of communication. Furthermore, transmitting the same amount of data may have radically different power implications depending on how the transfer is structured: for example, when operating over a mobile broadband interface it is more efficient to use a single larger transfer than to space out the transmission in multiple smaller transfers. Multiple transfers may cause multiple radio transitions between low and high powered states, causing additional battery drain.

In short, the choice among speed, cost, and power is not a simple one. At times, speed may be the most important consideration. Other times, bandwidth cost or battery life may be the deciding factor. HTTP 2.0 must allow developers to optimize for the specific constraints of their problem space (which might change over time) rather than imposing a monolithic solution to a generic problem. For example, server push is a good optimization for many scenarios where content updates to web pages revisited over time are infrequent, the client has plenty of bandwidth as well as the needed processing power to either handle the updates instantly, or cache them for later processing. On the other hand, it is not likely to be appropriate in situations where content is being transmitted over a costed link. Neither it will be when the client is running several applications that use network bandwidth concurrently, and bursty, server-initiated content transmissions would interfere with their smooth operation. Rather than forcing developers to choose between using all the features of HTTP 2.0 or sticking with HTTP 1.1, it would be better to provide mechanisms for developers to fine tune the capabilities of HTTP 2.0 to a specific set of requirements.

In summary, the goals of higher speed, lower cost, lower power may often be aligned. For instance, having less data sent on the wire will allow pages to load faster, allow the radio to power down sooner and consume less bandwidth. But given the variety of the scenarios where HTTP 2.0 will be used, this will not always be the case. For example, a device whose battery is about to run out, or whose cache is near capacity can provide a better user experience by disabling server push updates while retaining the other optimizations available in HTTP 2.0. Accordingly, the working group should consider power and cost as well as speed.

1.2. Definitions

Client: The endpoint initiating the WebSocket session.

Connection: A transport-level connection between two endpoints.

Endpoint: Either the client or server of a connection.

Frame: A header-prefixed sequence of bytes sent over a HTTP Speed+ Mobility WebSocket.

Server: The endpoint which did not initiate the WebSocket session.

Session: A synonym for a WebSocket.

Session error: An error on the WebSocket.

Stream: A bi-directional flow of bytes across a virtual channel within a HTTP Speed+Mobility session.

Stream error: An error on an individual stream.

1.3. Protocol Overview

This protocol comprises four parts:

1. Setting up a session (Handshake): Uses WebSocket upgrade

2. Session maintenance and framing: Uses WebSocket framing, including control frames such as keepalive (PING-PONG) and WebSocket Close

3. Multiplexing within the session: Uses SPDY [I-D.mbelshe-httpbis-spdy] stream semantics implemented via a WebSocket extension

4. HTTP layering: Same as SPDY

WebSocket provides a standards-based (RFC 6455) model for establishing a bi-directional session (or a socket) between a client and a server across the web. The RFC describes the following:

- A mechanism to create a session between a client and a server (Upgrade) and optionally secure the session using TLS

- A light-weight framing model to send data asynchronously and bi-directionally within the session

- A set of control messages to keep the session alive (PING-PONG), and to close the session (CLOSE)

- An extension model to optionally layer semantics such as multiplexing and compression

In keeping with our principle to leverage existing standards where possible, this HTTP Speed+Mobility proposal uses WebSocket as the session layer between the client and the server. Using WebSocket as a session layer has several advantages. First, we do not have to invent a new set of control messages, since we can use the ones defined by the WebSocket standard. Second, network intermediaries (the middleboxes) do not have to be modified to cope with a new protocol for establishing and maintaining bidirectional sessions across the web. Finally, clients and servers have the flexibility to decide whether they want to secure the session or not.

Using WebSocket also makes it easy to enable multiplexing within the session. In fact, this proposal takes the concept of streams and the stream related control messages as defined in SPDY, and models them as a WebSocket extension. Barring some important issues as noted in the issues section, the HTTP layering on streams is identical to what was presented in the SPDY proposal.

Furthermore, this proposal removes all congestion management control frames proposed in SPDY, in accordance with the principle of preserving a layered architecture. Instead, any TCP issues raised in the SPDY proposal should be submitted to the relevant working group for consideration.

Finally, this proposal regards server push as being outside of the scope of HTTP 2.0 because it is not in line with existing HTTP semantics. Having said that, given the relevance of server push with mobility and in anticipation of such an extension, this proposal does offer some thoughts on server push in section 5.

2. Setting up the session

The opening handshake is the standard WebSocket handshake based on HTTP Upgrade. To advertise support for the HTTP 2.0 extension, the client request MUST include the "http2" extension token in the |Sec-WebSocket-Extensions| header in its opening handshake:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin: http://example.com
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: http2
```

To accept the HTTP 2.0 extension requested by the client, the server MUST include the "http2" extension token in the |Sec-WebSocket-Extensions| header in its opening handshake. Otherwise, the client MUST fail the WebSocket connection:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Extensions: http2
```

For more details, please refer to [RFC6455].

3. Session layer and Framing

At the end of an HTTP upgrade as described above, the bi-directional WebSocket between the client and the server becomes the new session layer. In keeping with the principle around re-using existing standards, the session layer for HTTP Speed+Mobility uses the WebSocket base framing protocol for both data frames and control frames.

3.1. WebSocket framing protocol

Once connected, the client and server can exchange framed messages using the framing protocol shown below. For more details, please refer to RFC6455.

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-------+-+-------------+-------------------------------+
 |F|R|R|R| opcode|M| Payload len |    Extended payload length    |
 |I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
 |N|V|V|V|       |S|             |   (if payload len==126/127)   |
 | |1|2|3|       |K|             |                               |
 +-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
 |     Extended payload length continued, if payload len == 127  |
 + - - - - - - - - - - - - - - - +-------------------------------+
 |                               |Masking-key, if MASK set to 1  |
 +-------------------------------+-------------------------------+
 | Masking-key (continued)       |          Payload Data         |
 +-------------------------------- - - - - - - - - - - - - - - - +
 :                     Payload Data continued ...                :
 + - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
 |                     Payload Data continued ...                |
 +---------------------------------------------------------------+
```

3.2. WebSocket Keepalive messages

Keepalive messages in WebSocket are modeled using the ping and pong control frames.

The Ping frame contains an opcode of 0x9. Upon receipt of a Ping frame, an endpoint MUST send a Pong frame in response, unless it already received a Close frame. A Ping frame may serve either as a keepalive or as a means to verify that the remote endpoint is still responsive.

The Pong frame contains an opcode of 0xA. A Pong frame is sent in response to a Ping frame. A Pong frame MAY be sent unsolicited. This serves as a unidirectional heartbeat. A response to an

unsolicited Pong frame is not expected.

For more details, please see [RFC6455].

3.3. WebSocket Close

Closing a session uses the standard WebSocket close handshake as defined in [RFC6455]. The GOAWAY control frame in SPDY is replaced by the WebSocket Close control frame. GOAWAY specific data is mapped as follows:

- Status Code is replaced with the status code in the WebSocket Close frame. For example:

  * OK (0) is replaced by 1000 (normal closure)

  * PROTOCOL_ERROR (1) is replaced by 1002 (protocol error)

- Last-good-stream-id is carried as extension data in the WebSocket Close frame.

3.4. WebSocket errors (Session errors)

The SPDY proposal details session errors and determines that a GOAWAY frame MUST be sent when that happens. For the HTTP Speed+Mobility protocol, closing a session MUST use the WebSocket Close frame as described in section 3.3 of this document.

For best performance, it is expected that clients will not close open TCP connections until the user closes the HTTP app or navigates away from all web pages referencing a connection, or until the server closes the connection. Servers are encouraged to leave connections open for as long as possible, but can terminate idle connections if necessary.

4. Streams Layer

Once the session is established, HTTP Speed+Mobility allows creating streams to send and receive HTTP data. The stream operations and semantics are borrowed directly from SPDY. As noted earlier, WebSocket is the protocol used for framing data that is sent and received within the session (and consequently each stream). Stream operations (such as SYN_STREAM) are modeled using a WebSocket extension.

4.1. Modeling SYN_STREAM in a WebSocket frame

The SYN_STREAM frame is carried as extension data (as seen in section 3.1) in a binary data frame. The opcode is set to 0x2. A possible future refinement is for SYN_STREAM to use a control opcode reserved for WebSocket extensions as defined in section 5.8 Extensibility in RFC6455.

The payload length is the length of the "Extension data" (which is the length of the SYN_STREAM frame) + length of "Application data (zero for SYN_STREAM). In other words, the payload length is as shown below:

o Control (1 bit)

o Version (15 bits)

o Type (16 bits)

o Flags (8 bits)

o Length (24 bits)

o Length of SYN_STREAM specific data

The "Payload data" is defined as "Extension data" concatenated with "Application data." The payload data for a SYN_STREAM frame consists of the SYN_STREAM frame (shown below) tunneled "as-is" in the Extension data of the WebSocket frame.

```
+----------------------------------+
|1|    version    |        1       |
+----------------------------------+
|  Flags (8)  |  Length (24 bits)  |
+----------------------------------+
|X|           Stream-ID (31bits)   |
+----------------------------------+
|X| Associated-To-Stream-ID (31bits) |
+----------------------------------+
| Pri|Unused | Slot |              |
+-------------------+              |
| Number of Name/Value pairs (int32) |   <+
+----------------------------------+    |
|     Length of name (int32)       |    | This section is the "Name/Value
+----------------------------------+    | Header Block", and is compressed
|           Name (string)          |    | unless opted out.
+----------------------------------+    |
|     Length of value  (int32)     |    |
+----------------------------------+    |
|          Value   (string)        |    |
+----------------------------------+    |
|           (repeats)              |   <+
```

A Multiplexing Extension for WebSockets [draft-tamplin-hybi-google-mux] is being designed in the HyBi working group that also defines a multiplexing model. There is an opportunity to converge these designs for use by both WebSockets and HTTP Speed+Mobility.

4.2. Compression

Throughout this document, header compression is enabled by default. However, either the client or the server may opt out of using compression when transmitting headers. This opt out model is described with added flags in the SYN_STREAM, HEADERS and SYN_REPLY frames.

4.3. Control Frames

The following set of stream-related control frames are taken directly from SPDY.

- o SYN_STREAM

- o SYN_REPLY

- o RST_STREAM

o HEADERS

All of the frames are identical to SPDY with the few additions described below:

4.3.1. SYN_STREAM

In addition, this protocol adds two new flags: one to make Compression opt out, and one to make Server Push opt in.

o 0x04= FLAG_NO_HEADER_COMPRESSION: indicates the Name/Value header block is not compressed.

o 0x08 = FLAG_PUSH_ALLOWED: a stream created with this flag allows the server to push related responses in separate unidirectional streams. This flag MUST only be sent by the client.

4.3.2. SYN_REPLY

In addition, this protocol adds one new flag, to allow opting out of compression.

o 0x04= FLAG_NO_HEADER_COMPRESSION: indicates the Name/Value header block is not compressed

4.3.3. HEADERS

In addition, this protocol adds one new flag, to allow opting out of compression.

o 0x02= FLAG_NO_HEADER_COMPRESSION: indicates the the Name/Value header block is not compressed.

4.4. SPDY frames removed in this proposal

This proposal simplifies the session control messages to remove items that are redundant to WebSockets control frames, break compatibility with existing HTTP semantics, or implement concepts best addressed at the transport layer. The reasons for the deletions are outlined as follows:

SETTINGS: The information in the settings control message are concepts best reserved for the transport layer.

PING: WebSockets already has a keepalive mechanism in Ping / Pong. Other functions, such as RTT estimation, are associated with flow control, which is a function of the transport layer.

GOAWAY: Replaced with the WebSockets Close Frame which is documented in sections 5.5.1 and 7 of WebSockets [RFC 6455]

WINDOW_UPDATE: Flow control is a function of the transport layer.

CREDENTIAL: This is removed from HTTP Speed+Mobility because we believe it is not compatible with options such as TLS SNI. For this proposal, a session MUST only target one origin as described in [RFC6454].

5. HTTP Layering

This proposal adopts the HTTP integration model used by SPDY. The request-response semantics would be the same as well as stateless authentication.

The places where HTTP Speed+Mobility differs from SPDY are a result of its relationship with WebSockets and the removal of the CREDENTIAL frame.

While not addressed in this proposal, stateful authentication is something that can be added to the proposal and is captured in the Open Issues for Discussion section.

Lastly Section 5.3 articulates some thoughts on server push and discusses mechanisms to allow control from the client.

5.1. Connection Management

By default, and because it reuses the WebSocket handshake, HTTP Speed+Mobility uses port 80 for unsecured connections and port 443 for connections tunneled over Transport Layer Security (TLS) [RFC2818].

Clients SHOULD attempt to use a single HTTP Speed+Mobility connection to a given origin [RFC6454]. The server MUST be able to handle multiple connections from the same client and MUST be able to handle concurrent establishments and disconnects. As noted above, a client MUST only send requests for a single origin over a HTTP Speed+Mobility connection.

For a secure connection, if the client provides a Server Name Indication (SNI) extension during TLS handshake then all subsequent SYN_STREAM messages on that connection MUST specify a Host specification that exactly matches the server name provided in the SNI [RFC4366]. If the server receives a SYN_STREAM with a non-matching Host specification then it MUST respond with a 400 Bad Request. If the client receives a SYN_STREAM with a non-matching Host specification then it MUST issue a stream error.

5.2. Use of GOAWAY

HTTP Speed+Mobility replaces the GOAWAY message with a WebSockets CLOSE message per section 2.2 of this document. The last-stream-ID is included in the CLOSE message as extension data to provide the opportunity for graceful server shutdown.

5.3. Server Push Transactions

The HTTP Speed+Mobility protocol does not enable server push by default, requiring instead that the client explicitly request it using the FLAG_PUSH_ALLOWED flag in the SYN_STREAM frame sent from the client to the server. Server push is a new concept introduced in SPDY wherein a server pushes content to a client even if the client may not have requested it. We have disabled server push by default because it violates two of our design principles, namely preserving HTTP semantics and keeping the client in control of content.

Server push, as described in the SPDY proposal, has the limitation that the client cannot communicate its push requirements to the server. The result is that the server may push content to the client which is already cached locally, or to a client with a full cache, thereby causing unnecessary cache evictions. Furthermore, the server does not have sufficient context to know whether the access is coming from a browser or an app. We see a trend toward "tailored" apps, where each app may access different subsets of the server content, with different priorities, and in different sequences according to their own rendering requirements and user interaction models. For small devices as defined in the "internet of things", it is likely that there will be devices that run highly specialized apps to consume exactly what they need given limited cache space. In all of these scenarios, server push would result in needless traffic to the client resulting in bandwidth consumption and reduction in battery life.

We believe that for server push to be truly effective, HTTP 2.0 requires a feedback model enabling an app to give context to the server about its push needs. This proposal does not formally define such a mechanism. One way to enable this capability is for a client to include identifiers for content it already has in its cache, and send this as a hint in a SYN_STREAM message. The server could now use this information to only push deltas to that known cached content.

This is an area that requires significant working group discussion. Given the principle around maintaining existing HTTP semantics, we need to determine in the working group if server push should remain a part of HTTP 2.0.

6. Open Issues for Discussion in the Workgroup

During the drafting of this proposal a number of question came up that warrant deeper investigation. This is by no means a complete list of discussions around HTTP 2.0 but simply the current list of issues that the authors of this document wanted to explore further in the Working Group

Streams Issues:

- SPDY defines max and min control frame size but does not define what size to start with or how to discover what the endpoint can support. SPDY defines that an endpoint receiving a SYN that is too large must sent a RST with an error FRAME_TOO_LARGE. In order to maintain compression context, does the large SYN need to be decompressed?

- Interleaving Headers and data needs more definition. How does a server process a header before it is fully delivered?

HTTP Issues:

- Are mixed origin requests allowed? How does this work with TLS and SNI?

- What are the changes to the stream layer to naturally enable stateful authentication?

- How can we support chunked encoding.

- Do we want to support HTTP trailers?

Optimizations:

- There is potential optimization between the WebSockets Op codes and the Stream frames. This needs more investigation.

- There is potential optimization with adding settings to the WebSockets upgrade handshake (compression, push, header size, etc.)

- Investigate options for implementing a message for a client to inform server push of cache contents.

Server Push issues:

- Should server push stay in HTTP 2.0 or be defined in a different specification?

- How can a client describe its cached content or indicate its content needs, to facilitate efficient server push behavior

- Should server push negotiation be done as part of the WebSocket handshake rather than inside SYN_STREAM?

Compression issues:

- Should header compression be negotiated at a session level as part of the WebSocket handshake rather than transmitted on a per packet basis?

Security Issues:

- Is there a DDOS possibility with the way stateful authentication is specified?

- How does interleaving requests for cross origin content over TLS work? Are there vulnerabilities there?

7. Acknowledgements

Thanks to the following individuals who have also contributed with discussions and text: Dave Thaler, Ivan Pashov, Jitu Padhye, Jean Paoli, Michael Champion, NK Srinivas, Sharad Agarwal and Rob Mauceri.

This document incorporates materials from http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00.

8. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[I-D.mbelshe-httpbis-spdy] Belshe, M. and R. Peon, "SPDY Protocol", draft-mbelshe-httpbis-spdy-00 (work in progress), February 2012.

Authors' Addresses

Rob Trace
Microsoft

Email: Rob.Trace@microsoft.com

Adalberto Foresti
Microsoft

Email: aforesti@microsoft.com

Sandeep Singhal
Microsoft

Email: Sandeep.Singhal@microsoft.com

Osama Mazahir
Microsoft

Email: OsamaM@microsoft.com

Henrik Frystyk Nielsen
Microsoft

Email: HenrikN@microsoft.com

Brian Raymor
Microsoft

Email: Brian.Raymor@microsoft.com

Ravi Rao
Microsoft

Email: RaviRao@microsoft.com

Gabriel Montenegro
Microsoft

Email: Gabriel.Montenegro@microsoft.com