Network Working Group
Internet-Draft
Expires: December 17, 2012

R. Trace
A. Foresti
S. Singhal
O. Mazahir
H. Nielsen
B. Raymor
R. Rao
G. Montenegro
Microsoft
June 15, 2012

# HTTP Speed+Mobility
# draft-montenegro-httpbis-speed-mobility-02

Abstract

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 is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 17, 2012.

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 characteristics allow billions of devices to support and use HTTP in a multitude of communication scenarios. However, it is time to improve upon HTTP 1.1.

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.

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 should also 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.

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 round trips 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 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 targets broad HTTP applicability while emphasizing performance improvements and 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. Implementers 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 a major extension to HTTP and should be treated as such (see section 2.1 in [I-D.iab-extension-recs]).

- 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 multiplexing layer. Additional capabilities that introduce new communication models like unrequested responses should be treated in a different specification and explored separately from this proposal.

### 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).

### 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 use the HTTP 1.1 Upgrade header similar to how 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, 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 sizable portion of HTTP requests and responses actually happen in "backend" scenarios, in which the messages are transported over physically trusted infrastructure between endpoints owned by the same organization. Furthermore, a "random thought of the day" web service or a sensor spewing out a temperature reading every few seconds may choose not to use TLS. In such situations, it may not be worth the additional expense of deploying TLS, nor might it be desirable to hinder caching of the content by encrypting it end-to-end.

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 app 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.

In addition to browsers, apps increasingly 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 that is already 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 will it 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 and 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, whose communication monetary costs are prohibitive, or whose cache is near capacity can provide a better user experience by disabling a capability that consumes bandwidth with potentially unwanted content, while continuing to use other optimizations available in HTTP 2.0. Accordingly, the working group should consider power and cost as well as speed.

## 1.2. Definitions

client: A program that establishes HTTP Speed+Mobility connections for the purpose of sending requests.

connection: A TCP layer virtual circuit established between two programs for the purpose of communication.

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

message: The basic unit of HTTP communication, consisting of a structured sequence of octets matching the syntax defined in [RFC2616] and transmitted via a connection.

request: An HTTP request message, as defined in [RFC2616].

response: An HTTP response message, as defined in [RFC2616].

server: An application program that accepts connections in order to service requests by sending back responses. Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

origin server: As defined in [RFC2616] section 1.3, a server on which a given resource resides or is to be created.

origin: As defined in [RFC6454] section 3.2, a representation of a security principal. Roughly speaking, two URIs are part of the same origin if they have the same scheme, host, and port.

user agent: The client that initiates a request. These are often browsers, editors, spiders (web-traversing robots), or other end user tools.

proxy: An intermediary program that acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them on, with possible translation, to other servers. A proxy MUST implement both the client and server requirements of this specification. A "transparent proxy" is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. A "non-transparent proxy" is a proxy that modifies the request or response in order to provide some added service to the user agent, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering. Except where either transparent or non-transparent behavior is explicitly stated, the HTTP proxy requirements apply to both types of proxies.

endpoint: Either the client or server of a connection.

receiver: Endpoint receiving network data in a HTTP Speed+Mobility session. This can be either the client or the server.

sender: Endpoint sending network data in a HTTP Speed+Mobility session. This can be either the client or the server.

session: A single channel between a client and server over which there will be multiplexed HTTP requests and responses.

session error: An error on the HTTP Speed+Mobility session.

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

HTTP Speed+Mobility is a proposal for an HTTP 2.0 transport protocol that includes multiplexing HTTP content for improving transmission of HTTP content and efficient use of TCP connections.

This protocol comprises four parts:

1. Negotiation: Setting up a session (Handshake) is the WebSocket Upgrade with additional headers.

2. Session Layer: This defines maintenance and framing of a HTTP Speed+Mobility session and is defined as a WebSocket extension [RFC6455].

3. Multiplexing Layer: This defines the framing and maintenance for multiplexing HTTP requests over a single HTTP Speed+Mobility session. This proposal borrows from the SPDY [I-D.mbelshe-httpbis-spdy] stream semantics and is defined as a WebSocket extension.

4. HTTP layering: This proposal borrows from the SPDY [I-D.mbelshe-httpbis-spdy] proposal.

The WebSocket protocol [RFC6455] provides a standards-based model for establishing a bi-directional session 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 the principle to leverage existing standards where possible, this HTTP Speed+Mobility proposal uses WebSockets as the session layer between the client and the server. Using WebSockets as a session layer has some 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, clients and servers have the flexibility to decide whether they want to use TLS or not.

Using WebSockets 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, and models them as a WebSocket extension.

Furthermore, this proposal specifies a simple receive buffer management scheme based on a credit control mechanism.

Finally, this proposal regards server push as being outside of the scope of HTTP 2.0 itself, because it is not in line with existing HTTP semantics. Having said that, given the benefits of populating the client cache proactively, we believe that the Working Group should create a specification separate from HTTP 2.0 to define such a solution.

### 1.3.1. 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 server. The server MUST be able to handle multiple connections from the same client and MUST be able to handle concurrent establishments and disconnects.

## 1.4. Proxies

Based on the existing Internet, proxies are an important consideration for any HTTP 2.0 proposal. There are many cases where the presence of a proxy (both explicit and transparent) will impede negotiation of any new protocol. In existing environments, the only reliable method of traversing proxies with non-HTTP 1.x communications is by tunneling over TLS / SSL.

However, given the importance of HTTP 2.0 and the desire to continue to use proxies, we believe that proxies will eventually adopt HTTP 2.0 and will support communication without TLS, although such adoption may take a long time.

WebSockets provides the best of both environments. WebSockets may be negotiated over a secure tunnel to traverse an incompatible proxy or may be used in the clear, when appropriate, with a proxy that understands HTTP 2.0.

## 2. Negotiation

HTTP Speed+Mobility negotiates a session using the WebSockets handshake based on HTTP Upgrade. To advertise support for the HTTP 2.0 extension, the client request MUST include the "x-httpsm" extension token in the |Sec-WebSocket-Extensions| header in its opening handshake:

```
GET /default.htm HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade, X-InitialCreditBalance
Origin: http://example.com
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-httpsm
X-InitialCreditBalance: 131072
```

To accept the HTTP 2.0 extension requested by the client, the server MUST include the "x-httpsm" 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, X-InitialCreditBalance
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Extensions: x-httpsm
X-InitialCreditBalance: 65536
```

The Sec-WebSocket-Extensions defines the version of the protocol. For incompatible future revisions to the protocol, the extension name will need to be revised.

This draft defines a new header to declare the initial credit balance for endpoints that need to use flow control. This header is defined in Section 5.3 below.

HTTP Speed+Mobility may be extended to allow for new negotiated options by adding new headers to the upgrade exchange.

When the negotiation of HTTP Speed+Mobility is successful, the server MUST respond to the GET request with a SYN_REPLY message with an Stream ID of 1, containing the response to the original GET request. Any required data frames for this response MUST be identified with the stream ID of 1. For more information on SYN_REPLY see Section 4.2.2 below.

For more details on WebSockets, refer to [RFC6455].

## 3. Session layer and Framing

At the end of the WebSockets upgrade as described above, the bi-directional WebSocket between the client and the server becomes the new session layer. The session layer for HTTP Speed+Mobility uses the WebSocket base framing protocol for both data frames and control frames.

### 3.1. Opening and Closing Sessions

One of the motivations for a multiplexing solution is to have a more efficient use of the TCP transport. Implementations should minimize the number of connections to reduce the impact of TCP slow start and to avoid latency from creating new connections. Ideally there will be a single session between a client and a server. An implementation SHOULD use this session to multiplex the maximum amount of data between the two endpoints. Implementations MAY create multiple simultaneous sessions between two endpoints.

For best performance, it is expected that a client will not close open TCP connections until it is certain that it no longer has use for it (e.g., the user closes the HTTP app or navigates away from all web pages referencing a connection), or until the server closes the connection. Servers SHOULD leave connections open for as long as possible, but MAY terminate idle connections if necessary.

### 3.2. Origin of Multiplexed Content

A single session MAY contain HTTP content from multiple origins. A client implementation SHOULD only multiplex requests destined to multiple origins into a single connection under the following conditions:

- Anonymous / Clear: For sessions that do not require authentication or SSL/TLS, implementations MAY multiplex content to multiple origins in the same session. This is the primary use case for sending requests to a Proxy.

- Basic / Digest Authentication: For sessions to an origin server that requires per-request authentication, implementations MAY multiplex content to multiple origins.

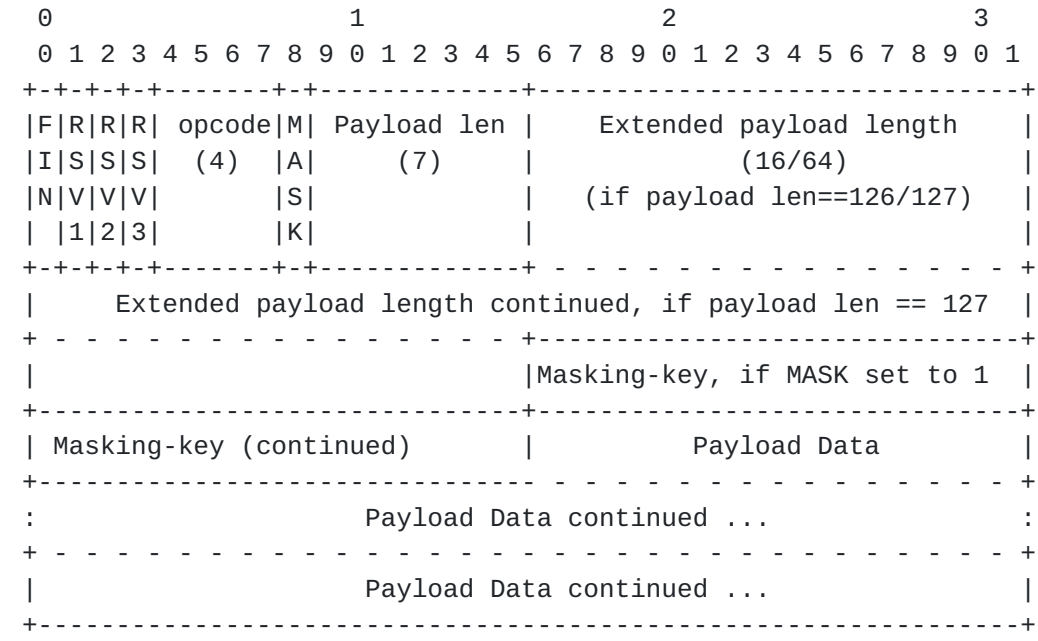
- Multi-Part Authentication (e.g., Kerberos): To be done.

- For a secure connection, if the client provides a Server Name Indication (SNI) extension during the TLS handshake then all subsequent SYN_STREAM messages (see Section 4.2.1) on that connection MUST specify a Host specification that exactly matches

the server name provided in the Server Name Indication (SNI) (Section 3.1 of [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.

## 3.3. WebSocket Framing Protocol

This specification defines the x-httpsm WebSocket extension to enable multiplexing of HTTP content within a single WebSocket session. Once the upgrade is accepted, the client and server can exchange framed messages using the WebSockets framing protocol. The standard WebSocket frame from [RFC6455] is included for reference.




The payload data for this extension is multiplexed streams as defined in Section 4 below.

The x-httpsm extension defines 4 extension opcodes to establish and maintain streams:

(opcode TBD) - SYN_STREAM: See Section 4.2.1.

(opcode TBD) - SYN_REPLY: See Section 4.2.2.

(opcode TBD) - RST_STREAM: See Section 4.2.3.

(opcode TBD) - CREDIT_UPDATE: See Section 4.2.4.

### 3.4. Closing HTTP Speed+Mobility Sessions

Closing a session uses the standard WebSocket close handshake as defined in [RFC6455].

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 from SPDY. As noted earlier, WebSockets 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 implemented as a WebSocket extension.

### 4.1. Stream Management

#### 4.1.1. Stream Creation

A stream is created by sending a SYN_STREAM (Section 4.2.1). The first stream is created by the GET request that initiates the upgrade to HTTP Speed+Mobility and will have a stream ID of 1. Each subsequent SYN_STREAM sent by the client will increment the stream ID by 1. Stream IDs do not wrap; when a client or server cannot create a new stream id without exceeding a 32 bit value, it MUST NOT create a new stream.

If a server receives a SYN_STREAM with a stream id which is less than any previously received SYN_STREAM, it MUST issue a session error (Section 4.1.5.1) with the status PROTOCOL_ERROR.

It is a protocol error to send two SYN_STREAMs with the same stream-ID. If a recipient receives a second SYN_STREAM for the same stream, it MUST issue a stream error (Section 4.1.5.2) with the status code PROTOCOL_ERROR.

Upon receipt of a SYN_STREAM, the recipient can reject the stream by sending a stream error (Section 4.1.5.2) with the error code REFUSED_STREAM. Note, however, that the creating endpoint may have already sent additional frames for that stream which cannot be immediately stopped.

Once the stream is created, the creator may immediately send data frames for that stream, without needing to wait for the recipient to acknowledge.

Both endpoints can send data on the stream.

#### 4.1.2. Stream Data Exchange

Once a stream is created, it can be used to send arbitrary amounts of data. Generally this means that a series of data frames will be sent on the stream until a frame containing the FLAG_FIN flag is set. The FLAG_FIN can be set on a SYN_STREAM (Section 4.2.1), SYN_REPLY

(Section 4.2.2), or a data (Section 4.3) frame. Once the FLAG_FIN has been sent, the stream is considered to be half-closed.

### 4.1.3. Stream Half-Close

When one side of the stream sends a frame with the FLAG_FIN flag set, the stream is half-closed from that endpoint. The sender of the FLAG_FIN MUST NOT send further frames on that stream. When both sides have half-closed, the stream is closed.

If an endpoint receives a data frame after the stream is half-closed from the sender (e.g. the endpoint has already received a prior frame for the stream with the FIN flag set), it MUST send a RST_STREAM to the sender with the status STREAM_ALREADY_CLOSED.

### 4.1.4. Stream Close

There are 3 ways that streams can be terminated:

Normal termination: Normal stream termination occurs when both sender and recipient have half-closed the stream by sending a FLAG_FIN.

Abrupt termination: Either the client or server can send a RST_STREAM at any time. A RST_STREAM contains an error code to indicate the reason for failure. When a RST_STREAM is sent from the stream originator, it indicates a failure to complete the stream and that no further data will be sent on the stream. When a RST_STREAM is sent from the stream recipient, the sender, upon receipt, should stop sending any data on the stream. The stream recipient should be aware that there is a race between data already in transit from the sender and the time the RST_STREAM is received. See Stream Error Handling (Section 4.1.5.2).

TCP connection teardown: If the TCP connection is torn down while un-closed streams exist, then the endpoint must assume that the stream was abnormally interrupted and may be incomplete.

If an endpoint receives a data frame after the stream is closed, it must send a RST_STREAM to the sender with the status PROTOCOL_ERROR.

### 4.1.5. Error Handling

The framing layer has only two types of errors, and they are always handled consistently. Any reference in this specification to "issue a session error" refers to Section 4.1.5.1. Any reference to "issue a stream error" refers to Section 4.1.5.2.

#### 4.1.5.1. Session Error Handling

A session error is any error which prevents further processing of the session layer or which corrupts the session compression state. When a session error occurs, the endpoint encountering the error MUST send a WebSockets CLOSE [RFC6455].

#### 4.1.5.2. Stream Error Handling

A stream error is an error related to a specific stream-id which does not affect processing of other streams at the session layer. Upon a stream error, the endpoint MUST send a RST_STREAM (Section 4.2.3) frame which contains the stream id of the stream where the error occurred and the error status which caused the error. After sending the RST_STREAM, the stream is closed to the sending endpoint. After sending the RST_STREAM, if the sender receives any frames other than a RST_STREAM for that stream id, it will result in sending additional RST_STREAM frames. An endpoint MUST NOT send a RST_STREAM in response to an RST_STREAM, as doing so would lead to RST_STREAM loops. Sending a RST_STREAM does not cause the HTTP Speed+Mobility session to be closed.

If an endpoint has multiple RST_STREAM frames to send in succession for the same stream-id and the same error code, it MAY coalesce them into a single RST_STREAM frame.

### 4.2. Stream Control Frames

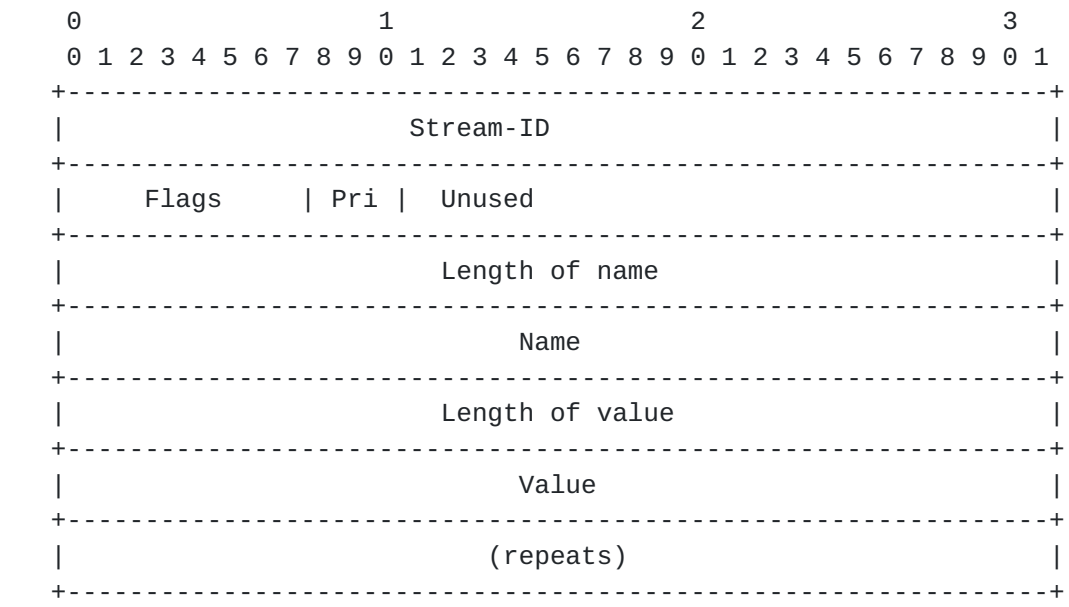
In Speed+Mobility four new opcodes are introduced:

- SYN_STREAM
- SYN_REPLY
- RST_STREAM
- CREDIT_UPDATE

In addition, all frames in HTTP Speed+Mobility include a 32-bit stream identifier in the Extension data.

#### 4.2.1. SYN_STREAM

The SYN_STREAM control frame is used to initiate a new stream and send the headers for a request. SYN_STREAM is specified as the extension opcode in the WebSocket frame. The SYN_STREAM Extension data is carried in the WebSocket payload:




Flags: Flags related to this frame. Valid flags are:

0x01 = FLAG_FIN: marks this frame as the last frame to be transmitted on this stream and puts the sender in the half-closed (Section 4.1.3) state.

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

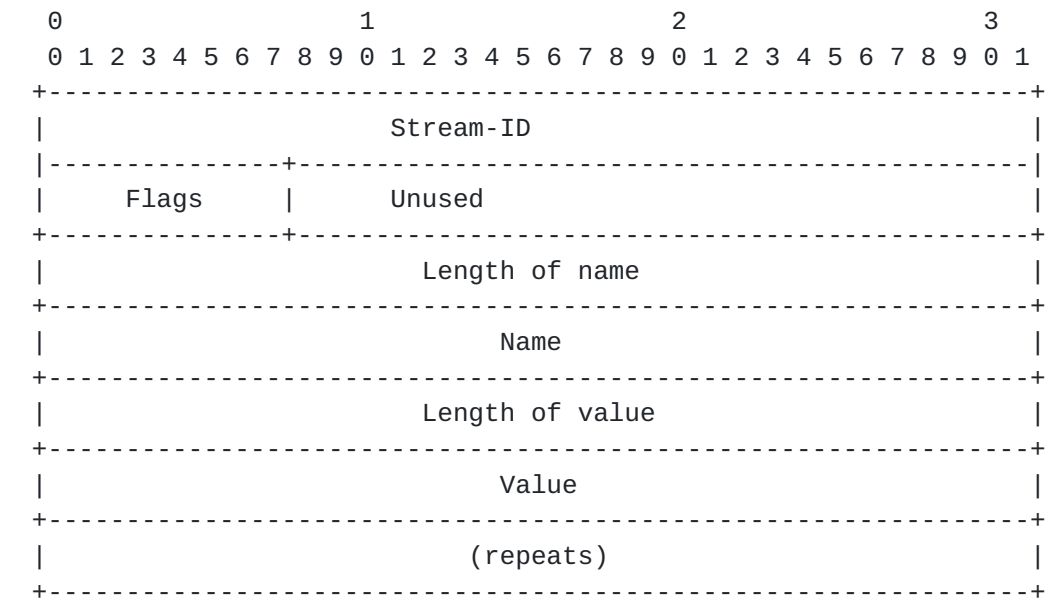
Priority: A 3-bit priority (Section 5.1) field.

Unused: 21 bits of unused space, reserved for future use.

Name/Value Header Block: A set of name/value pairs carried as part of the SYN_STREAM. See Section 4.4.

### 4.2.2. SYN_REPLY

The SYN_REPLY control frame indicates the acceptance of a stream creation by the recipient of a SYN_STREAM control frame. SYN_REPLY is specified as the extension opcode in the WebSocket frame. The SYN_REPLY Extension data is carried in the WebSocket payload:




Flags: Flags related to this frame. Valid flags are:

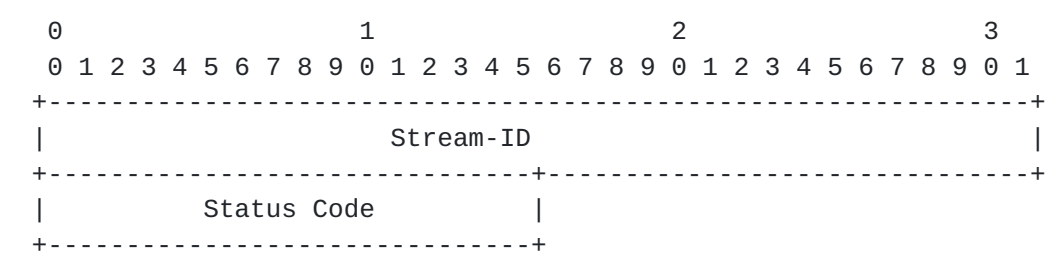
0x01 = FLAG_FIN: marks this frame as the last frame to be transmitted on this stream and puts the sender in the half-closed (Section 4.1.3) state.

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

Name/Value Header Block: A set of name/value pairs carried as part of the SYN_STREAM. See Section 4.4.

### 4.2.3. RST_STREAM

The RST_STREAM control frame allows for abnormal termination of a stream. When sent by the creator of a stream, it indicates the creator wishes to cancel the stream. When sent by the recipient of a stream, it indicates an error or that the recipient did not want to accept the stream, so the stream should be closed. RST_STREAM is specified as the extension opcode in the WebSocket frame. The RST_STREAM Extension data is carried in the WebSocket payload:



Status code (16 bits): An indicator for why the stream is being terminated. The following status codes are defined:

1 - PROTOCOL_ERROR: This is a generic error, and should only be used if a more specific error is not available.

2 - INVALID_STREAM: This is returned when a frame is received for a stream which is not active.

3 - REFUSED_STREAM: Indicates that the stream was refused before any processing has been done on the stream.

5 - CANCEL: Used by the creator of a stream to indicate that the stream is no longer needed.

6 - INTERNAL_ERROR: This is a generic error which can be used when the implementation has internally failed, not due to anything in the protocol.

7 - FLOW_CONTROL_ERROR: The endpoint detected that its peer violated the flow control protocol.

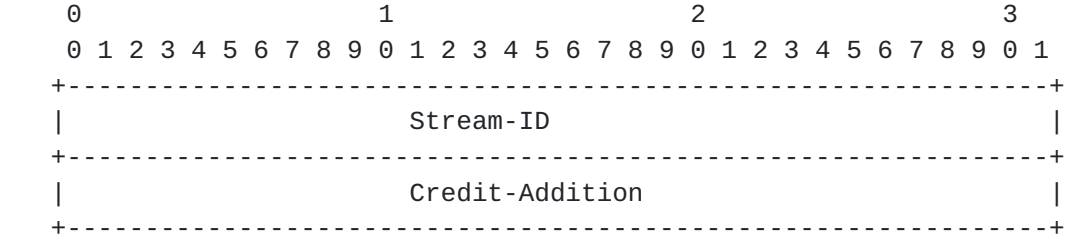
8 - STREAM_IN_USE: The endpoint received a SYN_REPLY for a stream already open.

9 - STREAM_ALREADY_CLOSED: The endpoint received a data or SYN_REPLY frame for a stream which is half closed.

Note: 0 is not a valid status code for a RST_STREAM.

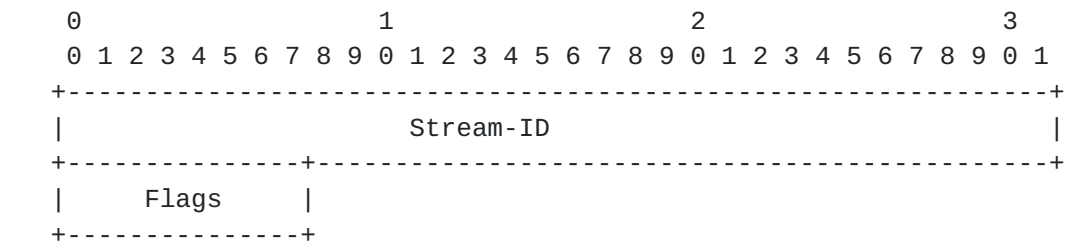
After receiving a RST_STREAM on a stream, the recipient must not send additional frames for that stream, and the stream moves into the closed state.

### 4.2.4. CREDIT_UPDATE




Credit-Addition: The value, in bytes, that the recipient must add to the stream's credit balance. The value ranges from 0 to 4294967295 (0xffffffff) inclusive. 4294967295 (0xffffffff) is a special value that designates "infinite" (see Section 5.5).

## 4.3. Data Frames




Stream data frames are modeled as WebSocket binary data frames with extension data:

Flags: Flags related to this frame. Valid flags are:

0x01 = FLAG_FIN: signifies that this frame represents the last frame to be transmitted on this stream. See Stream Close (Section 4.1.4) below.

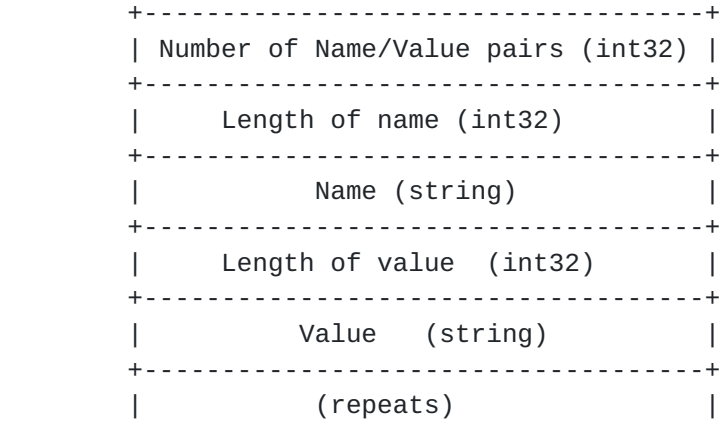
Data frame processing requirements:

If an endpoint receives a data frame for a stream-id which is not open, it MUST send issue a stream error (Section 4.1.5.2) with the error code INVALID_STREAM for the stream-id.

If the endpoint which created the stream receives a data frame before receiving a SYN_REPLY on that stream, it is a protocol error, and the recipient MUST issue a stream error (Section 4.1.5.2) with the status code PROTOCOL_ERROR for the stream-id.

## 4.4. Name/Value Header Block

The Name/Value Header Block is found in the SYN_STREAM and SYN_REPLY control frames, and shares a common format:



Number of Name/Value pairs: The number of repeating name/value pairs following this field.

List of Name/Value pairs:

Length of Name: a 32-bit value containing the number of octets in the name field.

Name: 0 or more octets, 8-bit sequences of data, excluding 0.

Length of Value: a 32-bit value containing the number of octets in the value field.

Value: 0 or more octets, 8-bit sequences of data, excluding 0.

Each header name must have at least one value. Header names are encoded using the US-ASCII character set and must be all lower case. The length of each name must be greater than zero. A recipient of a zero-length name MUST issue a stream error (Section 4.1.5.2) with the status code PROTOCOL_ERROR for the stream-id.

Duplicate header names are not allowed. To send two identically named headers, send a header with two values, where the values are separated by a single NUL (0) byte. A header value can either be empty (e.g. the length is zero) or it can contain multiple, NUL-separated values, each with length greater than zero. The value never starts nor ends with a NUL character. Recipients of illegal value fields MUST issue a stream error (Section 4.1.5.2) with the status code PROTOCOL_ERROR for the stream-id.

## 4.5. Compression

The Name/Value Header Block is a section of the SYN_STREAM and SYN_REPLY frames used to carry header meta-data. This block MAY be compressed using zlib compression. Within this specification, any reference to 'zlib' is referring to the ZLIB Compressed Data Format Specification Version 3.3 as part of [RFC1950].

For each HEADERS compression instance, the initial state is initialized using the dictionary specified in [I-D.mbelshe-httpbis-spdy] section 2.6.10.1.

Implementations MUST support header compression as specified in [I-D.mbelshe-httpbis-spdy] except for the following.

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.

# 5. Flow Control

## 5.1. Stream Priority

Each stream has a 3-bit priority field where 7 represents the highest priority and 0 represents the lowest priority. The stream priority is specified in the SYN_STREAM and cannot be re-specified for the lifetime of the stream.

When selecting data to send, the sender SHOULD select the data from the highest priority stream that has data ready for transmission. If multiple streams of the same priority have data ready for transmission then the sender SHOULD be fair in sending data between those streams. See Section 5.7.

## 5.2. Credit Control

Credit control is used by memory-sensitive endpoints to advertise their limited buffering capability. This is to prevent the sender from sending too much data, in a given time interval, thus causing the recipient's buffers to overflow.

An endpoint MAY demand that its peer honor credit control. An endpoint MUST honor the credit control if the peer demands it. Section 5.3 explains how an endpoint demands credit control.

Credit control is directional and is demanded by an endpoint to control how much its peer can send.

An endpoint that is honoring its peer's credit control will maintain a credit balance, for each stream, that controls how much data the endpoint can send to its peer. The credit balance is always in units of bytes. The demanding endpoint will send CREDIT_UPDATE messages, for a given stream, to update how much data the honoring peer is allowed to send. The credit balance applies to the data payload of data frames. Credit control is applied on an HTTP S+M per-hop basis.

## 5.3. Credit Control Declaration

During the HTTP S+M handshake, an endpoint MAY demand that the peer honor credit control when sending data, for all streams, on that connection. If the endpoint does not demand credit control, then it MUST NOT send CREDIT_UPDATE messages.

Credit control is demanded by specifying an HTTP header in the GET that upgrades the HTTP/1.1 connection to HTTP S+M. The header name is "X-InitialCreditBalance". The header value indicates the initial credit balance that the peer has for sending data on streams. The

header value is a base-10 number ranging from 0 to 4294967294 (0xfffffffe), inclusive. If the header is not present then that indicates the endpoint does not advertise credits and will never send CREDIT_UPDATE messages on that connection.

In the following example, the client does not advertise flow control because it wants uninhibited responses throughput. Thus the server will send data frames to the client without credit tracking. However, the server indicates an initial credit balance of 64KB, which means the client will keep track of the CREDIT_UPDATE messages from the server to know when it can send data frames for a given stream.

Upgrade Request:

```
GET / default.htm 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: x-httpsm
```

Upgrade Response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade, X-InitialCreditBalance
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Extensions: x-httpsm
X-InitialCreditBalance: 65536
```

## 5.4. Credit Balance Updates

If an endpoint is honoring credit control then the endpoint MUST maintain a credit balance for each of the streams on that connection. The client MUST NOT send more data than there is credit available. Upon sending a data frame, the endpoint MUST decrement the credit balance by the number of bytes in the payload of the data frame. Upon receipt of a CREDIT_UPDATE message, the endpoint MUST increment the credit balance by the amount indicated in the CREDIT_UPDATE message. If the resultant sum exceeds 4294967294 (0xfffffffe) then that is a stream error. The demanding endpoint knows the initial credit balance and the amount of data received thus far so it MUST NOT emit a CREDIT_UPDATE message that would cause the credit balance to exceed 4294967294 (0xfffffffe).

### 5.5. Turning Credit Control Off for a Stream

If an endpoint demanded credit control then all streams start with the specified initial credit balance. Any time, before having sent a frame with FLAG_FIN set on the stream, the demanding endpoint MAY emit an "infinite" CREDIT_UPDATE message to terminate any further credit control on that stream. Upon sending an "infinite" CREDIT_UPDATE, the sender MUST NOT send any more CREDIT_UPDATE messages for that stream. Upon receipt of an "infinite" CREDIT_UPDATE message, the sender has an unlimited number of credits.

### 5.6. Increasing and Decreasing Stream Credit

An endpoint MAY increase the credit available to the peer by specifying a value in the CREDIT_UPDATE message that is larger than how much data was sent by the peer or consumed. For example, having demanded an initial credit balance of 64KB, the endpoint may send a CREDIT_UPDATE of 512KB for a newly created stream shortly after creation, thus increasing the available credit for that stream to 576KB.

An endpoint MAY replenish less credit by specifying a value in the CREDIT_UPDATE message that is smaller than how much data was actually consumed. For example, after demanding an initial credit balance of 64KB and upon receiving 40KB of data, the endpoint may not send back a CREDIT_UPDATE message thus forcing the available credit down to 24KB. Note that it is not possible for an endpoint to revoke credit that it already advertised to the peer.

### 5.7. Implementation Guidance and Considerations

This document does not mandate a specific algorithm for selecting data to send from amongst multiple streams. The exact logic used will be implementation-specific. Within a priority level, the implemented algorithm should try to be fair to avoid one or more streams from monopolizing the send opportunities and hence starving the other streams. One such solution would be to implement a Deficit Round Robin scheme within a priority class and have a higher priority always preempt a lower priority.

This document does not mandate a specific algorithm for deciding when to send CREDIT_UPDATE messages. For example, a simple implementation may always emit a CREDIT_UPDATE immediately upon consuming the received data. Another implementation may coalesce multiple CREDIT_UPDATE messages into one. Yet another implementation may delay emitting a CREDIT_UPDATE message until a specific time or the next set of received data, whichever comes first, to reduce packet chatter.

This document does not mandate a specific algorithm for adjusting the credit balance. For example, implementations may monitor their memory state to determine when they can afford to increase or reduce the credit balance. Other implementations may also interface with the lower stack layers (e.g., TCP) to compute bandwidth-delay-products to tune the credit balance. Some implementations (e.g., devices) may be very constrained and may not have any logic to tune the credit balance.

# 6. General Notes

## 6.1. 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.

This proposal does not support some HTTP concepts as documented in [RFC2616] including Chunked Encoding and HTTP trailers.

While not addressed in this proposal, stateful authentication is something that will be addressed at a later date

## 6.2. Relationship to SPDY

This proposal borrows on many of the concepts of the SPDY proposal. There are some key areas where we differ from SPDY as outlined below.

Much of where HTTP Speed+Mobility differs from SPDY are a result of its relationship with WebSockets where we use the existing standard for the following:

Negotiation: Uses WebSockets Upgrade. This also negotiates streams settings and version allowing the simplification of the streams frames

Session Framing: Defined as a WebSockets Extension. Allows reuse of the length and opcode data to simplify the streams frames.

Lastly, this document simplifies the number of messages in the streams layer.

## 6.3. Server Push

Server push is a new concept introduced in [I-D.mbelshe-httpbis-spdy] wherein a server pushes content to a client even if the client may not have requested it. This is an area that requires significant working group discussion. Given the principle around maintaining existing HTTP semantics, we are not documenting it here and would like to see the working group document this separately from HTTP 2.0.

## 6.4. Open Issues

There are a number of open issues that are still under 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.

### 6.4.1. Flow Control

Describe how intermediaries may add/ adjust credit control parameters.

Deeper investigation into frame buffering requirements.

What to do if a control frame is too big. What to do in the case of a buffer overrun.

Do we want to add the ability to change priority on a stream?

### 6.4.2. Streams Issues

Do we need to negotiate maximum streams in the Upgrade header?

## 7. Acknowledgements

Thanks to the following individuals who provided helpful feedback and contributed to discussions on this document: Dave Thaler, Ivan Pashov, Jitu Padhye, Jean Paoli, Michael Champion, NK Srinivas, Sharad Agarwal and Rob Mauceri.

This document incorporates materials from [I-D.mbelshe-httpbis-spdy].

# 8. References

## 8.1. Normative References

[RFC1950] Deutsch, L. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.

[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.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, December 2011.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, December 2011.

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

## 8.2. Informative References

[I-D.iab-extension-recs] Carpenter, B., Aboba, B., and S. Cheshire, "Design Considerations for Protocol Extensions", draft-iab-extension-recs-14 (work in progress), June 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