Interim Report

Details:

Title:	“Web Protocol: A modern replacement for HTTP”
Author:	Christopher Jamie Hall (1031815)
Supervisor:	Prof. Omer F. Rana
Moderator:	Prof. Ralph Martin
Module:	CM0343
Credits:	40

Abstract

This project aims to create WP; a modern replacement for HTTP, making use of connection multiplexing to reduce the number of TCP sessions used to one (§2.1), binary headers to save overhead, and a fixed framing structure to accelerate parsing (§2.3). This also allows ongoing data streaming and connection testing (pinging). Drawing inspiration from Google's SPDY protocol, WP is designed to be simple and consistent. To test, measure, and improve WP a library has been written in the Go programming language, since one of the language's design goals was to write excellent web servers (§3.2).

Using an initial version of the WP library, and Go's built-in HTTP library, comparable web apps were developed using each protocol. The results from these tests showed that, with the base version of the library, the data efficiency gains made were swiftly outweighed by the content itself; a saving of 11.18 kilobytes only reducing the total data traffic by 2.39% (§3.4). However, the complete protocol should increase these savings, and the specialist functions like pinging and ongoing streaming will give substantial savings. During the ping testing, the data saving was 80.11%, meaning that in the app's expected use it would save 299.93 megabytes.

The future work for the project is to complete the Go library implementation, perform further tests to evaluate the additional features, and to improve and adjust the protocol accordingly (§4).

Table of Contents

1: Introduction

1.1: Initial plan

In the response to the initial plan, two specific matters were raised, which I shall address immediately:

"Is WP your invention?"

Although it draws strong influence from Google's SPDY protocol[1], the Web Protocol (WP) is purely my invention, and was originally designed during the summer holiday in preparation for the project.

"No time seems to have been allocated to design of the protocol, and thinking about what it should include, etc: there is too much emphasis on implementation."

Although some subtle improvements have been made during the project to resolve issues identified during testing, the main protocol design and specification is complete. Rather than rely on theoretical or simulated analysis to compare WP and Hypertext Transfer Protocol[2] (HTTP), I have decided to create similar applications in both protocols, so that realistic results can be obtained. The difference between HTTP's text-based approach and WP's binary structure means that text parsing plays a role in performance differences; one which is difficult to simulate.

The aim of the project is to supplement HTTP's features set with data streaming and connectivity checking (pinging), and to improve data efficiency by reducing the number of TCP sessions used and the size of the headers. Another key goal is to produce an easily-used implementation of the protocol, so that it sees actual use. Much theoretical research becomes shelved or lost, so providing an open-source library enabling the protocol's use is important in giving it some permanence.

2: Background

2.1: HTTP

There are several factors in HTTP's limited performance which WP attempts to address. The most important is session multiplexing, since it gives several benefits and improvements. In addition to this, binary framing, single-send headers, suggest and push messages, and data streams provide further benefits to data efficiency and help minimise latency. WP also provides features to supplement HTTP's capability, such as ping and data streams. These are discussed in more details in the approach section.

The research performed by Google on HTTP performance, shortly before work began on SPDY, showed that the main factor in determining user-side web performance was not bandwidth, but latency[3]. With the global latency average at roughly 100ms, the minimum round-trip time (RTT) is approximately 200ms; almost all of the 250ms usually allowed to give good perceived responsiveness[4]. Since this latency is used in both directions, even a small reduction gives a substantial improvement on performance. The main goal for SPDY is reducing latency in HTTP traffic, since it is fully compatible with HTTP at the application level; simply changing how the data is transmitted over the network. By contrast, WP aims to replace HTTP altogether, and thus does not suffer from the same legacy issues, such as text headers.

2.2: TCP

The main performance problems suffered by HTTP are caused in the underlying Transport Control Protocol (TCP), for a number of reasons. The only way to have multiple simultaneous requests in HTTP is to use multiple parallel TCP sessions. Although this gives network parallelism, it also introduces several issues. Each new TCP session begins with a three-way handshake, resulting in three times the latency before any data can be sent. In situations where the latency is already very high, such as in mobile internet connections (which can average at around 400ms[4]), this means that there is a very noticeable delay before the request is even sent. In this example, there would be a delay of over a second before each request can be sent; giving very poor perceived performance.

The situation becomes worse than this would suggest, thanks to TCP slow start. In order to handle connections with asymmetric bandwidth fairly and stably, TCP starts all sessions with a small window size (normally fewer than 3 segments[5]), slowly increasing the data throughput as more data is sent, or until packet loss occurs. While this does not affect performance unduly in long-term connections, it can have a significant impact when each connection only transfers a small amount of data, since it will not reach the connection's actual bandwidth. Even high-speed fibre networks will experience poor data throughput when transferring small volumes of data in multiple connections. Conversely, putting high traffic through a single connection accelerates the window expansion process; swiftly overcoming slow start.

Although HTTP 1.1 supports pipelining (allowing multiple requests to be sent sequentially), the protocol does not have any multiplexing system so the responses must be sent in the same order as the requests they serve. This means that any large or costly responses early on in the sequence will cause a bottleneck; holding back any subsequent responses on the same connection which may be ready to send.

This leaves a catch-22 situation where the client can either use a single connection and risk bottlenecks occurring, or use multiple connections and suffer diminished data efficiency. SPDY's solution to this problem is to multiplex data requests ('streams') onto a single connection; giving concurrent data transmission to avoid bottlenecks. This minimises the impact of TCP's slow start, provides parallel requests, and doesn't suffer HTTP pipelining's flaw of sequence dependence.

2.3: Streams

To perform this stream multiplexing, each stream is assigned a connection-unique identity number, which is provided with each packet. Further structure is provided in the stream ID, by using odd numbers for client-initiated streams and even IDs for server-initiated streams. More details are provided in the SPDY protocol specification[1].

Adopting a very similar stream system, WP gains the same benefits in efficiency and concurrency. Furthermore, WP uses the streams idea to implement very efficient data streaming for ongoing requests, such as Twitter feeds and other update systems. With a single response header for the stream and one 7-byte frame for each transmission, the streaming has minimal overhead; especially when compared to the HTTP equivalent, where polling would need to be used; possibly including a new TCP connection with each poll.

2.4: Headers

Another source of data inefficiency in HTTP is that each request's headers are normally the same, even when multiple requests are made to the same server. The client's user-agent, the server's hostname, and cookies tend to remain the consistent between requests; meaning that unnecessary is sent. To reduce this inefficiency, WP has a specific packet for sending repeated headers, allowing these details to be sent once and stored in the session memory for subsequent requests, meaning that only volatile headers like cache information are sent with each request. With increased use of cookies for advertising purposes, as well as session identification, this stands to give a significant saving in data usage.

To improve performance, SPDY compresses its headers; something not possible in HTTP. WP takes this further by setting aside the most common headers and forming them within the packet frames, made as specific as possible. For example, the HTTP Charset header is modified; using UTF-8 by default and allowing a single bit flag to upgrade to UTF-16. Similarly, the choice of language is transmitted in a byte consisting of 255 flags corresponding to a list of acceptable languages. These methods reduce the frame size of each request substantially, especially when combined with single-use headers to avoid repetition.

2.5: Alternatives

As mentioned, the SPDY protocol already attempts to solve many of the same problems as WP. The main difference is that SPDY maintains full compatibility with HTTP; simply changing how the data is transmitted. By contrast, WP aims to replace HTTP altogether; changing some of the key semantics of the protocol. For example, there are no methods in WP (GET, POST, etc); simply different packet types. This also makes integration of new features more consistent. Whereas SPDY is not normally handled directly by the developer, this indirection is not possible with its new features, such as push messages.

Using a new protocol altogether also allows a broader scope for its use, such as ongoing data streaming. While the SPDY protocol's structure would allow streaming data, it does not make use of this to preserve compatibility with HTTP's semantics.

There were several other protocols which attempted to fix the same problem, but which were not as directly relevant as SPDY. Details of these are given in the Other References section.

3: Approach

3.1: C

The first stage was to implement a basic WP server in C, since this gave the most consistent runtime performance and seemed most appropriate for what could eventually become kernel-level code. Having implemented the core of the protocol, both in a server and a simple client, initial testing could begin. This showed that gains were being made, but that further refinement was required. Having reduced the number of different packets and dropped SPDY's concept of opening streams before use, more tests took place.

3.2: Protocol development

The initial specification for WP was very similar to SPDY[1]. The main differences in structure were that since SPDY encompasses HTTP data, it contains a variable number of HTTP headers within many of its packet headers. As a standalone protocol, WP does not support HTTP and thus does not contain these variable-sized headers. Furthermore, to allow streaming data, WP has always had separate Data Response and Data Content packets.

Initial testing of the C implementation showed that the TCP overhead of sending Stream Requests before each Data Request, and a Stream Response before every Data Response resulted in WP actually using more data than HTTP. To address this, the protocol was changed to drop the requirement that streams be opened explicitly by dedicated Stream Request/ Response packets, and instead be opened implicitly by either a Data Request or a Data Push. This made more sense, since the additional stream safety provided by dedicated requests did not provide any known benefit, given that one stream ID is not more important than any other, and invalid streams can still be refused. This change also gave considerable efficiency gains and was far more simple to implement.

Since Stream Requests held the original mechanism for transmitting one-use headers, a new method was required. Where the protocol version was previously sent with each Stream Request, each connection should use a single protocol version. As a result, the Hello packet was created with the task of handling single-use data, such as certain headers and the protocol version.

To aid parsing and provide rigid structure, certain common headers were reformed into a dedicated binary structure. For example, where HTTP handles request language with a comma-separated string, it made more sense in WP to have a specific language header in the Hello packet, using a single byte to denote the number of language choices (with 0 representing indifference to language), and that number of bytes following; each byte giving the selection of a language from a numbered list of accepted languages. This means that requesting a response from one of five languages requires only six bytes; one for the number of choices and one for each choice. This avoids parsing potentially vague responses and overlaps (such as en_uk and en_gb). A similar process was followed for the expression of MIME types.

Using a completely new protocol also gave the opportunity to fix several issues which had arisen during the early days of the web, such as time and caching. HTTP uses ETags as the main mechanism for determining whether a user agent has already cached the current version of a resource. The method for determining an ETag has never been specified, but it normally involves taking a cryptographic hash of the resource. Since this has to be calculated every time each resource is requested, it is incredibly inefficient. Meanwhile, the date relating to a resource is normally sent in human-readable form; one which requires considerable parsing to be usable by a computer.

WP solves both these problems in one; providing the resource's date as a 64-bit value in UNIX epoch time. This is easily parsed, and can then be transformed into whatever human-readable format may be desired. Meanwhile, this same value is used to determine caching. The user agent sends the date of the resource it has cached (or 0 if the resource has not been cached). If the server holds a resource with a larger date (or a dynamic response), it sends the resource, with that version's date (or 0, for dynamic responses which should not be cached). This requires no additional processing, since the file system holds the resource's last modified date.

Another historic issue in HTTP is the user-agent string. During the early browser wars, user-agents swiftly grew, adding various details of their lineage and similarity, with irrelevant specifics being added for compatibility. By now, a user-agent is almost an essay, with far more detail than is necessary. For example, the Google Chrome user-agent includes "(KHTML, like Gecko)", even though WebKit is now separate from KHTML. With the opportunity of the new protocol, WP standardises and compacts user-agents to a strict formula. WP user-agent strings should contain the engine and version, browser and version, OS and version, architecture (using Plan 9 designation)[2]. This reduces:

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.95 Safari/537.11 to:

webkit/537.1,chrome/21.0.1180.89,macosx/10.7.4,6

which corresponds to:

WebKit version 537.1
Google Chrome version 21.0.1180.89
Mac OS X version 10.7.4
on a 64-bit architecture.

This change saves 71 bytes and gives a more useful, concise result.

3.3: Go

It swiftly became clear that large-scale, realistic testing could not take place without substantial further development, since the effectiveness of WP's push/suggest feature could not be compared against multiple HTTP requests without strong HTML, CSS, and JavaScript parsing to identify additional resources to fetch. Furthermore, a higher-quality HTTP parser would give more realistic performance. Accordingly, the project has transitioned language to Go[6].

While still compiling to native code and thus giving consistent performance, Go was designed with server construction a primary goal; intended for building the back-end for much of Google's extensive server infrastructure. Its standard library contains a thorough HTTP library and a very effective API, making complex web applications simple to build. By porting the WP library from C to Go, and adjusting it to provide the same interface and structure as the HTTP library, a single application can be build for both protocols with minimal change from the programmer's perspective. This also gives a far better implementation of the HTTP library than I could have achieved on my own. In creating the WP library in Go, the protocol is also more likely to see real-world use after the project's conclusion.

3.4: UML sequence diagram

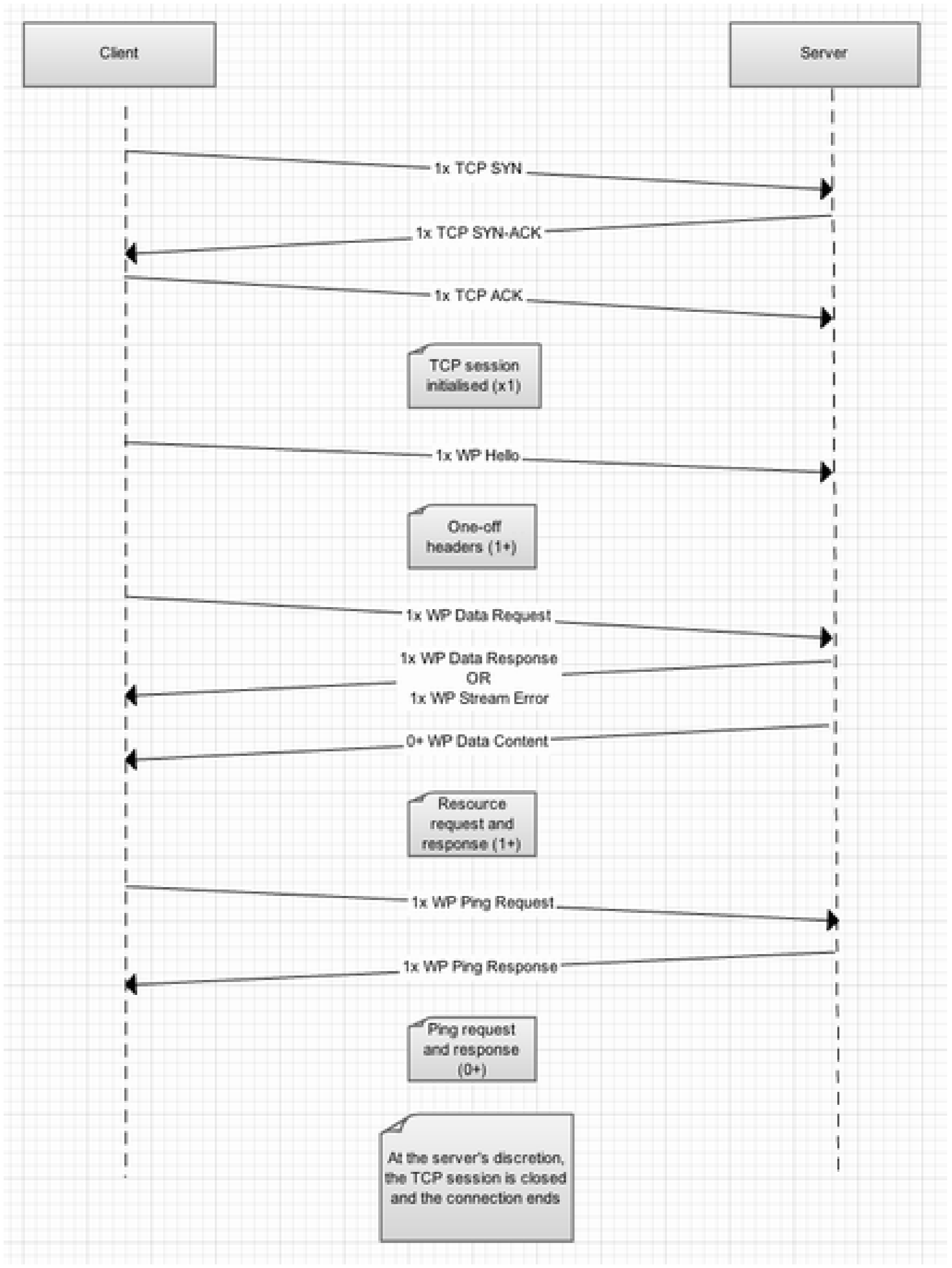
Client
Server
1x TCP SYN
1x TCP SYN-ACK
1x TCP ACK
TCP session initialised (x1)
1x WP Hello
One-off headers (1+)
1x WP Data Request
1x WP Data Response OR 1x WP Stream Error
0+ WP Data Content
Resource request and response (1+)
1x WP Ping Request
1x WP Ping Response
Ping request and response (0+)
At the server's discretion, the TCP session is closed and the connection ends

Figure 1: UML sequence diagram describing a WP session

3.5: Initial tests

Although the current implementation of the WP protocol in Go is incomplete, it is sufficient for initial tests. The first comparison between the two protocols with the new implementation used an existing HTTP web app I had developed in Go. The application contained some additional processing which required concurrent handling of requests; a feature not yet stable in the WP library. As a result, that component of the app was excluded from the test for both protocols. The app consists of a web server for the path /web/ and a pinging mechanism on path /ping to determine the connectivity. This was used by the client-side app to make sure that the received data was up-to-date, since it is a time-sensitive application.

During the test, each protocol made a single request to the /web/ path, receiving an HTML document. This was parsed with Go's experimental HTML library; returning an array of links to content that would be required by a web browser to render the page. The code to perform this task was identical in both protocols. These resources were then requested automatically and the responses parsed for further required content. Ten seconds after the test began, each protocol made a single ping request and the experiment concluded. In actual use, the ping would be resent every ten seconds, but a single ping was sent for this test, in order to analyse its size and speed.

3.6: Results

The test was measured with Wireshark[7], so that the data could be inspected and analysed thoroughly afterwards. The packet captures are attached in appendices 1 and 2. The results have been compiled in appendix 5. For both protocols, the results were fairly consistent, with HTTP varying in data sent by 6.20 kilobytes (1.33% of the mean) about the center and WP by 112 bytes (0.024% of the mean). On average, WP provided savings of 11.18 kilobytes (2.39% of the mean) across the file requests; using 110.47 fewer packets (54.43% of the mean).

The results from the pings (Appendices 3 and 4) are somewhat more interesting. With no variance in either protocol's data usage across 24 iterations each, the data is very consistent. Each time, WP uses almost a kilobyte less data (80.11% less) and 9 packets fewer (69.23% fewer). Since these savings are made on every single ping, a long-running application making frequent pings would see substantial performance gains. The application, from which this test was originally derived, pings once every ten seconds and is expected to run for a full weekend, with approximately 10 instances running. In this situation, a switch to WP would save 299.93 megabytes of data (80.11%) and would actually save load on the server, since in the experiment the HTTP requests' TCP session tended to stay open for quite some time after the response had been sent; whereas WP used a single TCP session for all pings.

4: Design

4.1: WP

In addition to providing the same core functionality as HTTP, the key design principles behind WP are simplicity and consistent structure. These help ensure minimal parsing, and high data efficiency, since the size of the data being received is always known, so precisely that much memory is allocated. The full protocol design is as follows.

Upon a connection being made, the first WP message must be a Hello packet, containing any single-use headers, such as the user-agent and target hostname. Subsequent Hello packets may be sent if the headers change. Streams form part of the core structure of the protocol and only Hello and Ping packets do not belong to a stream, since they affect the overall protocol.

4.2: Streams

Streams are initiated by either a Data Request or a Data Push, depending on the stream direction. A stream's direction is determined by the origin of the first message. Streams started by the client are denoted client streams have odd stream ID numbers; initiated consecutively and starting at 1. Client streams must be initiated by a Data Request, to which the response must be either one Stream Error or by one Data Response, optionally followed by one or more Data Content packets. Server streams have even stream ID numbers; initiated consecutively from 2. Note that 0 is never a valid stream ID. Server streams must be initiated by a Data Push, to which there is an optional Stream Error response. The server is free to assume that there will be no error and send one or more Data Content Packets with or after the Data Push.

Assuming an error does not occur, a stream ends in one of two ways. Either a packet is sent with the FIN flag set, in which case that is deemed the last valid packet on that stream and the stream is closed, or a Stream Error is sent with the FINISH_STREAM status code. Since there are no resources used in keeping a stream open (assuming the connection is still in use elsewhere), it is common to leave streams open, rather than send an extra packet to close them.

Stream IDs are stored as unsigned 16-bit integers; giving 65534 valid stream IDs (since 0 is not valid). If ever a stream ID larger than this is required, the ID does *not* wrap around; a new connection must be started. Doing so does not implicitly end the existing connection.

4.3: Errors

If a Data Request or Data Push is sent on a stream which is already in use, the recipient must reply with a Stream Error with the STREAM_IN_USE status code. If a new stream is initiated with an invalid stream ID, the recipient must reply with a Stream Error with the REFUSED_STREAM status code. If a non-initiating packet (such as a Data Content packet) is received on an invalid stream, the recipient must reply with a Stream Error with the INVALID_STREAM status code. In all three of these cases, the Stream Error's status data should be set to the expected stream ID for a new stream initiated by the sender. For example, if the first message in a connection is a Data Request from the client on stream 2, the server must reply with a Stream Error with status code REFUSED_STREAM and status data 1, since the request should have arrived on stream 1.

If, when parsing a packet, an invalid component is found, or a packet otherwise fails to parse, the recipient must respond with a Stream Error with status code PROTOCOL_ERROR. This signifies that data may have become scrambled or out of sync, and recovery from this point is unlikely. As a result, the connection should be severed and a new connection begun.

The connection must be begun by a Hello, since this contains the protocol version being used by the initiator (normally the client). WP implementations are expected to be fully backwards-compatible, unless security considerations decide otherwise. If an unsupported version is stated in the opening Hello packet, be it insecure or more recent than the recipient, the receiver must respond with a Stream Error with the status code UNSUPPORTED_VERSION. This should be accompanied with the highest supported version if the received version was too recent, or the smallest supported version if the received version was too old. The server may choose to end the connection at this point, or allow the client to try with a different protocol version. The server is unlikely to allow a second attempt if the first attempt was an insecure version.

If a stream which has been closed receives further packets which would otherwise be valid (normally Data Content packets), the recipient must ignore the data and respond with a Stream Error with status code STREAM_CLOSED. Since these data packets may have been sent before the stream was closed, this is not an error of protocol. However, if these packets continue, the recipient may choose to send a PROTOCOL_ERROR Stream Error.

WP defaults to using UTF-8 in all protocol headers and header data. Either side may request the use of UTF-16 instead by sending a Hello with the appropriate flag. If, on receiving this request, an implementation does not support UTF-16, it must respond with a Stream Error with status code UNSUPPORTED_ENCODING. This does not end the connection, and any subsequent packets sent with UTF-16 code points which occur outside the range supported by UTF-8, those packets will trigger further UNSUPPORTED_ENCODING Stream Errors, where the Stream Error references the stream ID on which the unsupported packet was sent.

In cases where either side suffers an error not relating to WP, it should attempt to send a Stream Error with status INTERNAL_ERROR. This represents an unrecoverable situation and the connection is not expected to continue.

If a Stream Error is received with a status code that corresponds to a particular stream (such as STREAM_REFUSED), but which has no stream ID, the recipient must respond with a Stream Error with status STREAM_ID_MISSING.

4.4: Connections

Since WP uses an ongoing TCP session, there has to be some session management to prevent popular servers from being overloaded. Ideally, the session should be left open until it is no longer needed. However, a server can choose to end connections prematurely if it nears its maximum load. The ideal way to do so is to refuse any new streams, with the expected stream ID set to 0, and to finish any open streams. Once all streams have been closed, the connection can be closed safely.

4.5: Packets

All WP packets are identified by the first three bits. These give the number of the packet type. Since each packet type's headers are of fixed size and all packets are at least 8 bits, the first byte of a packet can be read to determine how much extra data to read. The size of any further data is included in the header, so the data sizes are always known.

- `Packet 0: Hello (8+ Bytes)`

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  0  |  Flags  |    Version    | Language size | Hostname size |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|User-Agent size| Referrer size |      Header size (16 bits)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Headers    |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Flags:           See hello flags:
     UTF-16:    bit 0 (7)
Version:         WP version number. (8 bits)
Language size:   See languages (8 bits).
     - 0 means accept all.
Hostname size:   Length of hostname (8 bits).
User-Agent size: Number of chars in the User Agent.
Referrer size:   Number of chars in the referrer URI.
Header size:     Size of any remaining headers.
```

- `Packet 1: Stream Error (6 Bytes)`

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  1  |  Flags  |       Stream ID (16 bits)     |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Status    |      Status Data (16 bits)    |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
Flags:     See Error flags:
      Finish:    bit 0 (7)
Stream ID: See streams (16 bits).
Status: 8 bit code, 16 bit data:
      List of codes:
      0:  [INVALID].
      1:  Protocol error (ends session).
      2:  Unsupported version (send supported version).
      3:  Unsupported encoding.
      4:  Invalid stream (send expected stream ID).
      5:  Refused stream (send expected stream ID).
      6:  Stream closed.
      7:  Stream in use (send expected stream ID).
      8:  Stream ID missing.
      9:  Internal error (ends session).
      10: Finish stream
```

● Packet 2: Data Request (17+ Bytes)

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  2  |  Flags  |       Stream ID (16 bits)     |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Resource size (16 bits)    |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Header size (32 bits)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Cache Timestamp                        |
|                           (64 bits)                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       =================                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Res (0-65535) | Headers (cookies etc)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Flags:           See request flags.
      Finish:          bit 0 (7)
      Upload:          bit 1 (6)
Stream ID:       See streams.
Resource size:   Number of chars in the resource name.
      - 0 is invalid and triggers PROTOCOL_ERROR.
Header size:     Number of bytes in the request header.
Cache Timestamp: Time of most recent version (64 bits).
      - 0 means not cached.
Res:             URI (1-65,535B).
Headers:         Header content.
```

* Packet 3: Data Response (18+ Bytes)

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  3  |  Flags  |       Stream ID (16 bits)     | Response code |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Header size (32 bits)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Cache Timestamp                        |
|                           (64 bits)                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           MIME type           |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       =================                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Headers    |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Flags:          See data response flags.
      Finish:    bit 0 (7)
      Streaming: bit 1 (6)
Stream ID:      See streams.
Response code: 2-bit response type, 6-bit subtype.
Header size:   Number of bytes in the response header.
Cache:          Timestamp of when file was last changed.
      - 0 means do not cache.
MIME type:     3-bit type, 13-bit subtype.
```

- Packet 4: Data push (13+ Bytes)

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  4  |  Flags  |       Stream ID (16 bits)     |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Source stream ID (16 bits)   |    Resource size (16 bits)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Header size (32 bits)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           MIME type           |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       =================                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Res (0-65535) |    Headers    |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Flags:            See data push flags.
Stream ID:        This push’s stream ID.
Source stream:    The request which triggered this push.
Resource size:    Number of bytes in the URI.
Header size:      Number of bytes in the push header.
MIME type:        3-bit type, 13-bit subtype.
Res:              Resource URI.
```

- Packet 5: Data content (7+ Bytes)

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  5  |  Flags  |       Stream ID (16 bits)     |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Size (32 bits)                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       =================                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Flags:       See data flags (5 bits).
     Finish:    bit 0 (7)
     Streaming: bit 1 (6)
Stream ID:   See streams.
Size:        Number of bytes in the WP datagram.
```

- Packet 6: Ping request (1 Byte)

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  6  | Ping ID |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Ping ID:        Number identifying this request (5 bits).
```

- Packet 7: Ping response (1 Byte)

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  7  | Ping ID |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Ping ID:        Number identifying the request (5 bits).
```

5: Conclusions

5.1: Results

Although the efficiency gains in serving web content have been comparatively small compared to the data transmitted, WP has demonstrated that its extra features (in this case, pinging) can give considerable savings. Once concurrent stream serving has been implemented, long-term ongoing data streaming can be tested, which should give even greater efficiency gains.

Various other work should also be completed. The rest of the protocol needs to be implemented fully to complete the code library. The interface provided also needs to be improved and supplemented to make the protocol's use simpler and more effective. Furthermore, larger and more-diverse experiments need to be undertaken. Specifically, a large variety of file sizes and number of files should be sent, to determine what effect this has on file serving efficiency gains. Alternative applications using data streaming should also be devised and measured.

5.2: Future plans

The initial plan suggested that modified versions of the Chromium web browser and Apache Httpd web server be written to provide real-world use for the protocol. Given the switch to Go, WP servers are incredibly easy to write; requiring only 14 lines of code (see appendix 6). As a result, a modified form of Apache's Httpd would be unnecessary. Although providing a web browser compatible with WP would greatly increase its likelihood of use, initial tests have shown that the efficiency gains from serving web content are slim. To make best use of the protocol's efficiency gains, a client-side API to the ping and streaming functions would need to be provided. This would be a great use of the protocol, but would require modifying not only the browser's network code, but also its JavaScript engine; a monolithic task. As a result, I no longer feel that such work would be feasible.

Glossary

Connection	A single TCP session, which is explicitly opened and closed.
Pinging	Sending a simple message to test connectivity.
Stream	A single request or push of data in WP. This consists of a unique 16-bit stream ID number.
Streaming	Data which may be sent in a single burst, or in multiple packets; possibly separated by a long time. Examples include video data which is sent piece by piece, rather than downloaded as a single file, and asynchronous data such as a Twitter feed, or instant messaging.

Table of Abbreviations

HTTP	HyperText Transfer Protocol
RTT	Round-Trip Time
TCP	Transmission Control Protocol
UCS	Universal Character Set
UTF	UCS Transformation Format
WP	Web Protocol

Appendices

1. wp.pcap (attached)
2. http.pcap (attached)
3. wp_ping.pcap (attached)
4. http_ping.pcap (attached)
5. results.txt (attached)
6. wp_server.go (attached)

References

1 Belshe, M. and Peon, R. *SPDY Protocol - Draft 3* [Online]
Available at: http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3
[Accessed: 12 Dec 2012]

2 Berners-Lee, T. et al. June 1999. *Hypertext Transfer Protocol -- HTTP/1.1* [Online]
Available at: http://www.w3.org/Protocols/rfc2616/rfc2616.html
[Accessed: 12 Dec 2012]

3 Grigorik, I. November 2012. *SPDY and the Road Towards HTTP 2.0* [Online]
Available at: https://www.youtube.com/watch?v=SWQdSEferz8
[Accessed: 12 Dec 2012]

4 Sprint Inc. 2012. *Important Coverage Information* [Online]
Available at: http://shop2.sprint.com/en/coverage/support/important_coverage_info_popup.shtml
[Accessed: 12 Dec 2012]

5 Allman, M. et al. September 2009. *TCP Congestion Control* [Online]
Available at: http://tools.ietf.org/html/rfc5681
[Accessed: 12 Dec 2012]

6 Pike, R. et al. *The Go Programming Language* [Online]
Available at: http://golang.org
[Accessed: 12 Dec 2012]

7 Wireshark Foundation. *Wireshark* [Online]
Available at: http://www.wireshark.org
[Accessed: 12 Dec 2012]

Other references

Shelby, Z. et al. December 2012. *Constrained Application Protocol (CoAP)* [Online]
Available at: http://tools.ietf.org/html/draft-ietf-core-coap-13
[Accessed: 12 Dec 2012]

Fette, I. et al. December 2011. *The WebSocket Protocol* [Online]
Available at: http://tools.ietf.org/html/rfc6455
[Accessed: 12 Dec 2012]

Stewart, R. September 2007. *Stream Control Transmission Protocol* [Online]
Available at: http://tools.ietf.org/html/rfc4960
[Accessed: 12 Dec 2012]

Nottingham, M. December 2012. Hypertext Transfer Protocol Working Group [Online]
Available at: https://datatracker.ietf.org/wg/httpbis/charter/
[Accessed: 12 Dec 2012]
Note: HTTP 2.0 is still in only the proposals stage, but it is expected to use SPDY as a starting point, and thus the discussion on SPDY applies equally to HTTP 2.0.