

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

The purpose of this project was to design and implement Web Protocol (WP); a modern replacement for HTTP, which makes use of request multiplexing to reduce the number of TCP sessions used to one (§2.2), binary framing, and a fixed frame structure to accelerate and simplify parsing (§2.4). This also provides data streaming and server push facilities. Drawing inspiration from Google's SPDY protocol, WP is designed to be simple, consistent and quick. To test, measure, and improve WP, a library has been written in the Go programming language, since one of Go's design goals was to write high-performance web servers.

Using the WP library and Go's HTTP library, simple web servers and clients were developed for each protocol. These were then used to serve and fetch the average website, on the average network connection, according to data from the HTTP Archive and Google (§4.1.1). Where HTTP fetched a typical website in an average of 14.6 seconds, and HTTPS in 20.4 seconds, WP took just 4.1 seconds; averaging 3.7 times faster than HTTP and 5.1 times faster than HTTPS.

The future work for the project is to add the last details to the WP library in Go, to perform a restructuring to enable more appropriate handling of the protocol, and to connect it into the HTTP library, enabling a single server to handle both protocols simply (§5.1).

Acknowledgements

WP was partly inspired by the SPDY protocol, and works on similar principles. As a result, credit goes to Mike Belshe and Roberto Peon for their work designing SPDY and writing the SPDY/3 specification, which is available at <http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3> .

Further thanks go to my supervisor, Prof. Omer Rana, for his encouragement and guidance.

Table of Contents

1: Introduction	
1.1: The Problem	6
1.2: Goals, Requirements, and Deliverables	6
2: Protocol Design	
2.1: Starting Point	9
2.2: Request Parallelism	9
2.3: Streaming Data	10
2.4: Simplification	10
2.5: Security	11
2.6: Frame Structure	11
2.6.1: Hello	11
2.6.2: Stream Error	12
2.6.3: Data Request	14
2.6.4: Data Response	15
2.6.5: Data Push	16
2.6.6: Data Content	18
2.6.7: Ping Request and Ping Response	19
3: Implementation	
3.1: Adaptation of "net/http"	20
3.2: Frame Parsing	20
3.3: Top-Level Server Changes	21
3.3.1: Handler Interface	21
3.4: Top-Level Client Changes	21
3.5: Networking Structure	23
3.5.1: Server Structure	23
3.5.2: Structure Deficiencies	24
3.5.3: Parsing	24
3.5.4: Illustration	24
3.5.4.1: Creation of a Listener	24
3.5.4.2: Serving a Connection	25
3.5.4.3: Parsing Received Data	27
3.5.4.4: Parsing a Frame	28
3.5.4.5: Sample Handler	29
3.5.4.6: Writing Data	30
3.5.4.7: Stream Conclusion	31
4: Results and Evaluation	
4.1: Atomic Resource Testing	32
4.1.1: Preface	32
4.1.2: Experiment	33
4.1.3: Results	41
4.1.4: Analysis	41
4.1.4.1: SPDY	41
4.1.4.2: Number of Packets Sent	42
4.1.4.3: Data Transmitted	42

4.1.4.4: Data per Packet	43
4.1.4.5: Time	44
4.1.5: Extrapolation	46
4.2: Streaming Resource Testing	46
4.2.1: Preface	46
4.2.2: Experiment	47
4.2.3: Results	54
4.2.4: Analysis	56
5: Future Work	
5.1: Implementation Development	57
5.2: Adoption Requirements	57
6: Conclusions	59
7: Reflection	60
Table of Abbreviations	61
References	62

1: Introduction

1.1: The Problem

The Web Protocol (WP) was designed to address the two main problems with HTTP in the modern web. These are its inadequate support for connection parallelism, and the difficulties and inefficiency in parsing a text-based protocol. In addition to this, WP provides further features missing from HTTP, such as server push, connection testing (pinging), and full streaming data support. WP also takes the opportunity of a completely new protocol to use a structure more appropriate to the modern web; standardising user agent identifiers, resource caching, and response codes to a coherent structure, rather than one developed by implementation over several years. Where HTTP provides named methods for controlling documents (like GET, PUT, and DELETE) and debugging the connection (like CONNECT AND TRACE), WP uses a simple request/response approach, where requests may optionally become 'Upload' requests and contain data.

With the web now experiencing an enormous change as a result of the huge popularity of smart phones and the mobile web, large, complex websites are seeing increasing traffic from high-latency mobile internet connections. Most phone users have network latency of 400 ms and over^[1]; meaning that sites with a large number of unique domains induce many simultaneous TCP connections, reducing performance enormously. With the average website using 16 different domains^[2], and most web browsers supporting up to six simultaneous connections per domain^[3], a huge number of TCP handshakes has to be performed, each requiring a full round-trip of around 800 ms for mobile internet (assuming the 400 ms latency). This serious performance issue is addressed directly by WP, which uses just one connection per domain. This would (like SPDY) encourage the abolition of dummy subdomains used to enable more simultaneous connections for HTTP parallelism, thereby reducing the number of different domains per page.

1.2: Goals, Requirements, and Deliverables

The key goals of the project were:

1. **To design the Web Protocol.**
2. **To develop this design in a reference implementation.**
3. **To test and evaluate the implementation with a comparison against HTTP.**
4. **To write a protocol specification to standardise WP.**

Although my initial plan suggested implementing WP in an open source web browser, such as Firefox or Chromium, to provide more effective testing, I soon realised that to gain full advantage of the protocol, the browser's entire network stack would need to be modified; a truly enormous task. With only eight months in which to complete the project, I decided that this was not achievable, so it was removed from my goals.

The most important requirements were:

1. **The protocol must be able to perform any web-related task for which HTTP can be used.** As a protocol intended to replace HTTP for the modern web, it would not fulfil its role very well if it could not do everything that HTTP can.
2. **WP must have higher performance than HTTP in the most important cases.** Although WP's additional features are useful, its main feature is its performance, so this needs to be superior to HTTP for most web browsing.
3. **WP should outperform HTTP on every relevant use.** Although it would be acceptable for WP to be slower than HTTP in some edge cases, provided it was faster in the most common uses, ideally it would always be faster. While initial versions of WP used more control frames per request, and were thus slower than HTTP when serving very small resources, changes to combine frames into the current structure mean that WP is at least as quick as HTTP in every case.
4. **WP must remain solely in the application layer.** The internet's development as a completely distributed network has resulted in all manner of different kinds of proxy server and switch forming the many backbones of the core networks. These already interfere to some extent with application-layer protocols, such as performing various levels of caching, and in many cases, modifying HTTP headers. The developers of both SPDY and WebSockets advise the use of SSL/TLS when using these application-layer protocols, since some proxies silently drop packets they cannot understand^[4]. With even application-layer protocols suffering this kind of unpredictability with internet proxies, WP could not expect reliable performance if it depended on changes to the transport layer or below. Accordingly WP uses TCP at the transport layer and TLS at the session layer.

The main deliverables of the project are:

1. **A library implementing WP in the Go programming language.**
2. **The WP protocol specification.**

The library is called `net/wp`, in Go's package notation, and was written in its entirety in this project. Although a lot of the code was inspired by, and some copied from, Go's standard library package `net/http`, I modified and adapted most of the code for WP. Similarly, although some inspiration was drawn from the SPDY protocol, I designed WP as a standalone protocol to replace HTTP. The library can be used for three main tasks. It provides parsing capabilities for analysing WP frames, from any `io.Reader`, including network connections, files, and file archives. The library also provides application programming interfaces (APIs) with which WP clients and servers can be written. Although a simple interface for making individual requests is available, both the client and server APIs provide a more advanced system for handling multiple connections to multiple domains, or from multiple clients, respectively.

The server API provided by `net/wp` works by giving the `wp.ResponseWriter` interface, which extends the `http.ResponseWriter` interface by adding methods specific to the features new to WP, like data streaming and server push. Functions and data structures which satisfy the `HandlerFunc` and `Handler` interfaces respectively are registered to specific request paths. When a request arrives at the server, the handler registered to the path being requested is called, and provided with a structure containing the request and an implementation of `wp.ResponseWriter`. The response writer's methods are then used to construct the response, which is sent to the client. This allows the server-writer to construct a potentially very advanced server, without requiring any knowledge of the underlying protocol. In `net/http`, this structure and interface forms one of the big selling points of the Go language, by fulfilling its design goal of aiding and simplifying the creation of high-performance web servers.

The so-called “new client API” I designed provides a web browser-like interface for making long-term sessions similar to a single browser tab. This can include connections to multiple different servers simultaneously. The mechanics of the API are covered in more detail in Top-Level Client Changes (§3.4), but by writing a data structure which satisfies the `wp.ResponseWriter` interface, the client can have complete control over a client session, without needing to implement any WP-specific code. This was used extensively in my testing and evaluation of WP, since it simplified the design of the testing clients considerably, and completely decoupled the semantics of the test from the specifics of the protocol implementation. While the structure of the server API is the same as that of Go’s `net/http`, I designed and implemented the new client API myself.

2: Protocol Design

2.1: Starting Point

Network protocols are highly complex systems to consider; many (such as TCP) have been in active development for almost forty years. Accordingly, it would have been unwise to start from nothing with WP. With the next iteration of HTTP using SPDY draft 2 as a starting point, it seemed appropriate to do the same, although WP actually started from SPDY draft 3. SPDY aims to increase performance by multiplexing many request/response streams onto a single TCP session, and by giving a regular binary structure to aid parsing and validation. These were the two main factors driving WP, so SPDY seemed an excellent starting point. The most immediate differences between the two protocols are that SPDY forms a framing layer beneath regular HTTP, which changes how the same data is written to the network, whereas WP is a completely separate protocol intended to replace HTTP, and that WP focuses very specifically on streaming data, rather than the traditional single request/single response mechanism in HTTP, which SPDY continues.

2.2: Request Parallelism

HTTP functions by sending a single request, and receiving a single response. If multiple requests must be sent, the only two options for performing them in parallel, are multiple connections and pipelining. While using multiple TCP connections can give true parallelism, it incurs the overhead discussed in the interim report. Furthermore, it has other limits. The HTTP/1.1 specification states^[5] that a user agent should not have more than two simultaneous connections to a particular host. In spite of this, most modern browsers allow up to six connections to the same host^[3], purely to improve performance. Even this is not enough for many large websites, though. A common practice in high-performance websites is so-called domain sharding, where extra subdomains are created to enable more simultaneous connections to the same server. This gives greater parallelism, but increases the inefficiency produced by so many TCP connections.

Another common technique, introduced in HTTP/1.1, is request pipelining, where multiple requests are sent in sequence on the same connection, without waiting for responses. The server then replies to the requests on a first-in-first-out (FIFO) basis. Although this does not incur the same inefficiencies as multiple connections, it is not commonly used by modern browsers, since a slow response to the first request will hold up subsequent responses. This is particularly common in cases where the first response requires database queries or high volumes of I/O. The unpredictability this causes generally leaves pipelining avoided by clients.

Both SPDY and WP parallelise their requests by multiplexing requests onto a single TCP connection with an abstract stream model. Each request/response stream is assigned a stream ID unique within that connection. This allows data packets to be interleaved as necessary, while still being unambiguous. This avoids the overhead of opening multiple TCP sessions, saving both data and time. Since a TCP handshake requires one RTT to the server, and SSL/TLS requires a further two, the time savings on high-latency networks can be quite substantial, particularly on mobile devices, where average latency can reach 400 ms^[1]. This is covered in more detail in the interim report.

2.3: Streaming Data

Although SPDY's structure enables streaming data, its current use as a session-layer protocol to support HTTP, leaves it with a more atomic behaviour. At least in the public domain, SPDY has only been used for improving performance in systems that would otherwise have used HTTP. By comparison, WP puts a strong focus on streaming data, where data can be used as soon as it is received, and isn't considered a single block. Since the web is used for more than just exchanging documents, as was originally intended, other protocols and techniques have emerged to perform tasks for which HTTP isn't well suited. One of the most popular examples of this is streaming video content, on sites like YouTube and Netflix. Both of these use proprietary media frameworks added to web browsers via plugins (Adobe Flash and Microsoft Silverlight, respectively). To truly earn its title as the Web Protocol, WP must be able to perform these roles as well as the duties fulfilled currently by HTTP/SPDY.

WP's request/response model differs from HTTP in that it completely separates the response header from the response data. Accordingly, a single response header is always sent to inform the client whether the request was successful, the kind of data being sent in response, and so on. Optionally, a data stream then follows, but this stream may arrive a while after the response header, and may continue for some time. This works perfectly well for transferring documents and other atomic transactions, but is also highly appropriate for streaming media. The only problem WP would encounter in this role is that, since it relies on TCP, it cannot provide quite as high a bandwidth as traditional streaming media protocols like RTP, due to the overhead of TCP framing and ACKs, and its performance will be worse on unstable networks, where packets may be lost and have to be retransmitted. This is the typical problem when using TCP over UDP for real-time content.

This streaming data structure is fairly simple in design; data packets all contain a flags region, and one flag is defined as the 'Finish' flag. A data packet received with the 'Finish' flag enabled concludes that data stream. This may be the first (and thus only) packet in the stream. Until a finishing packet is received, the client should assume that more data is yet to come. If the server does not plan to send any data (for example, the client may have requested a resource that does not exist), the 'Finish' flag is enabled in the response header.

This design is sufficient for most use cases, but there may be some situations where it is useful for the response to consist of discrete segments, which might need to be larger than a single data packet. For these cases, the 'ready' flag can be used to indicate that this packet completes the segment, but that more data may follow.

2.4: Simplification

Several features and dedicated frames in SPDY have been combined into other components in WP. Where SPDY has a specific frame for preventing new connections (GOAWAY), in addition to a general error frame, WP uses a single, unified error packet. Similarly, the functionality provided by SPDY's SETTINGS frame is just one component in WP's hello packet, and SPDY's WINDOW_UPDATE and CREDENTIALS frames are left to lower-level protocols (TCP and TLS, respectively) in WP. These changes were made to try to reduce the number of different frames to simplify parsing slightly, and to make the protocol more simple as a whole.

Finally, SPDY uses special frames for HTTP headers and for name/value pairs for its SETTINGS frame. Both of these were assimilated into the relevant packets as part of the packet's structure, rather than as self-describing components. This tied the data to a specific purpose, and ensured that it did not become

separated during transmission.

2.5: Security

Some of the most common uses of the web require user privacy and security. Online banking and social media are two examples where authentication, integrity, and confidentiality are crucial. In HTTP, these are provided by HTTPS; using SSL or TLS to add cryptography and MACs. These have become the foundation of web security, but are not without issue. The thin barrier between HTTP and HTTPS mean that incomplete or inconsistent use of HTTPS can lose its security benefits. Without use of the Secure attribute, cookies are shared in both protocols' requests, meaning that HTTP requests which are intercepted by a MiTM attack can be read, or even modified, at will. Furthermore, HTTPS sessions can silently be downgraded to insecure HTTP connections, in so-called SSL Stripping attacks^[6]. Although these issues, and others, have been solved to some extent through techniques like HSTS, these solutions have not been universally adopted by servers, or even browsers.

To address this issue, the WP specification mandates that all WP connections must use TLS to provide session-layer security. TLS is specified to avoid legacy attacks on the older SSL protocol. Utilising TLS does not magically provide good security, but web browsers are already becoming fairly good at demonstrating poor use of the protocol when it is in use.

As discussed in the interim report, use of TLS also helps circumvent malicious or poorly-designed web proxies, gateways, and switches which often manipulate or discard protocol traffic they do not understand. For these reasons, WebSockets and SPDY also recommend the use of session-layer security protocols.

2.6: Frame Structure

In addition to saving space, a key reason behind using a fixed binary structure in WP's framing is to simplify and accelerate parsing. The regular structure means that most fields can be accessed with constant offsets, without needing to parse other content. To simplify this further, all WP frames share a common first byte structure, so when a packet is received, its type can be identified from just this first byte. This then gives the minimum size of the buffer to be allocated for the next segment of the frame, which contains the size of any variable-length data in the packet. As a result, pulling the raw data into a data structure is both quick and efficient, where many HTTP libraries allocate up to a kilobyte for initial HTTP headers to look for the Content-Length field.

For a more thorough analysis of the protocol's structure, a description of each frame type follows, complete with packet diagrams. Note that in these diagrams, any blank space indicates a gap left solely for aesthetic reasons, such as to avoid splitting a field over multiple lines. These blanks are not included in the actual data streams.

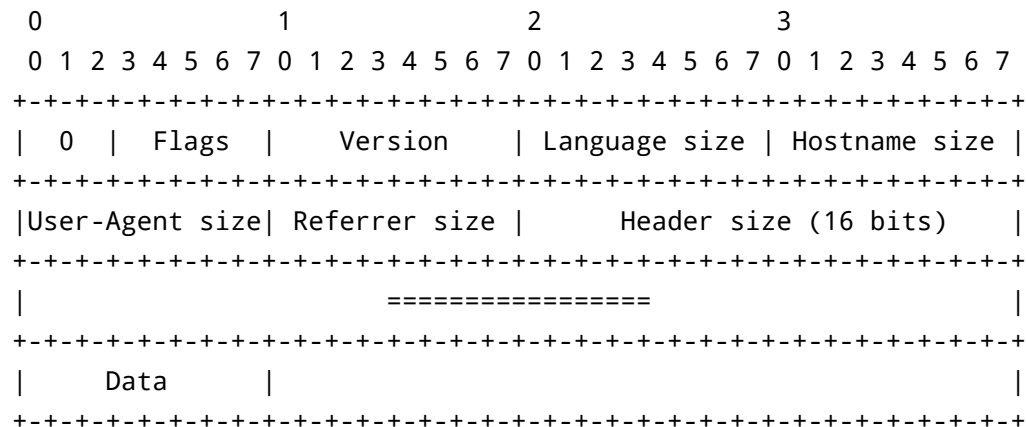
2.6.1: Hello

Hello frames are typically sent once, at the beginning of the connection. Every WP session must begin with the connection initiator (generally the client) sending a Hello frame, which must contain the initiator's protocol version, and should also have the user-agent, target host name, and other details. The client is free to begin sending requests and other data immediately after sending the initial Hello, but this Hello must be the first frame sent. If the recipient (normally the server) supports the proposed protocol version, the connection continues as normal. However, if the server only supports older versions of the protocol, it must reply with an UnsupportedVersion error, containing the latest supported version. The connection is

then terminated and another established by the client to avoid any confusion over stream IDs.

The data sent in the Hello is intended to be data which will not change over the course of the connection. In some cases, subsequent Hello frames may be sent to update these details, but this is unusual. These subsequent frames cannot change the user-agent, protocol version, or hostname, and these fields should be ignored by the recipient. The later frames are most likely to be caused by the client being referred to the host from another page, and thus updating the host to the new referral.

Hello frames consist of at least 8 bytes and have the following structure:



Flags: None defined. Must be 0.

Version: WP version number. (8 bits)
- Must not be 0.

Language size: See languages in the WP specification (8 bits).
- 0 means accept all.

Hostname size: Number of bytes in hostname (8 bits).

User-Agent size: Number of bytes in the user agent.

Referrer size: Number of bytes in the referrer URI.

Header size: Number of bytes in any remaining headers.

Data: Length-specified fields and headers named above.

2.6.2: Stream Error

The Stream Error frame is used to communicate both protocol and application errors, and can also be used to close streams safely. Although it has a stream ID, the Stream Error is the only frame allowed to send a 0 stream ID, since some errors, such as Application_Error apply to the entire session, rather than just one stream. In addition to the stream ID, Stream Errors have status code and status data fields, with the one-byte code illustrating which error has occurred, and the status data providing optional extra information, such as the expected value for a protocol version or stream ID.

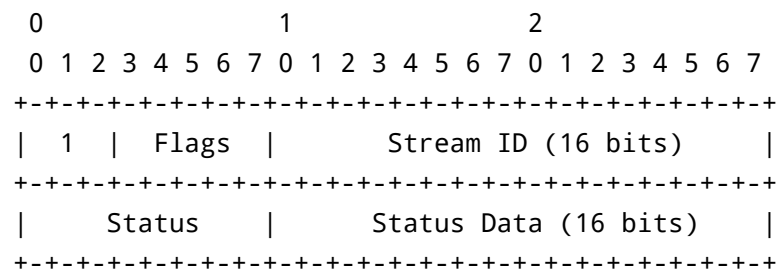
The full details for stream IDs are covered in the WP specification and have been discussed in the interim report. If an endpoint receives a Data Request frame, and that ID is odd when it should be even, or vice versa, then it must reply with a Stream Error with the Invalid_Stream status code. Similarly, if the ID is too large, it should evoke a Refused_Stream code, and if the ID is too small, a Stream_In_Use code. In all three cases, the Stream Error's stream ID should be set to the requested stream ID, and the status data should

contain the valid stream ID for such a request. If any frame is received containing a stream ID set to 0, other than a Stream Error, it must evoke a Stream Error with the `Stream_ID_Missing` status code. Finally, a Data Content frame sent to a stream which has already been closed must evoke a Stream Error with the `Stream_Closed` code. These responses are intended to give the recipient as precise information on the error as possible. In all cases, the offending frame must be ignored. In high-latency connections, it's quite plausible that many frames may be sent with an invalid stream ID before a Stream Error is received. In this case, the recipient of the invalid frames must still issue a separate Stream Error for each one, so that the sender knows that none has been processed.

If the recipient of a frame fails to parse it for whatever reason, such as invalid field values, other than those mentioned above, it must reply with a single Stream Error with the `Protocol_Error` status code, and close the connection. This is because parsing errors are unlikely to be recoverable, since there may be an inconsistency between endpoints on the frame boundaries. As mentioned previously, `Unsupported_Version` Stream Errors also close the current session. If either endpoint suffers an internal error, such as an inconsistency of state, it must attempt to send a Stream Error with the `Internal_Error` code, before closing the connection. Finally, the `Finish_Stream` status code may be used to close a stream in the same manner as using the finish flag in a Data Response or Data Content frame.

As discussed here, the status codes `Invalid_Stream`, `Refused_Stream`, `Stream_Closed`, `Stream_In_Use`, and `Finish_Stream` all require accompanying stream IDs. If any of these Stream Errors is received with a stream ID set to 0, or any other Stream Error is received with a stream ID *not* set to 0, a `Protocol_Error` Stream Error must be sent and the connection closed.

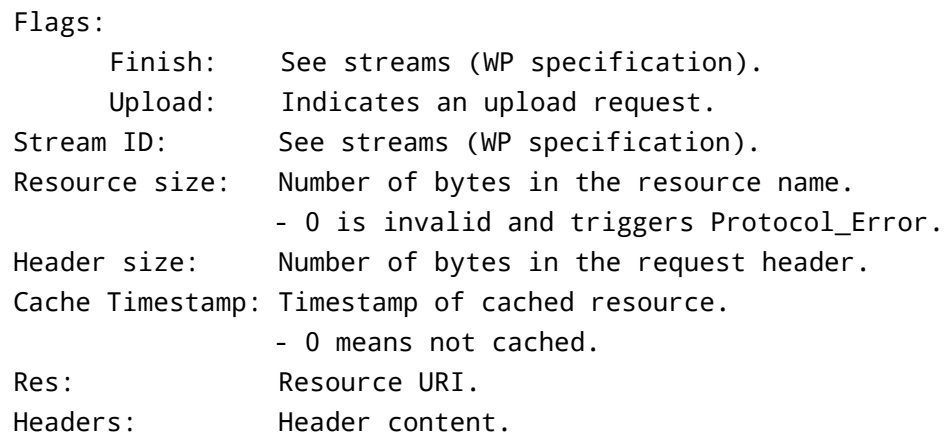
Stream Error frames consist of 6 bytes and have the following structure:



```
Flags:      No flags currently defined. Must be 0.
Stream ID:  See streams (WP specification) (16 bits).
Status:     List of codes:
            1: Protocol error (ends session).
            2: Unsupported version (send version, end session).
            3: Invalid stream (send expected stream ID).
            4: Refused stream (send expected stream ID).
            5: Stream closed.
            6: Stream in use (send expected stream ID).
            7: Stream ID missing.
            8: Internal error (ends session).
            9: Finish stream.
```

2.6.3: Data Request

Data Request frames consist of at least 17 bytes and have the following structure:



2.6.4: Data Response

Data Request frames should normally receive a reply with either a Stream Error or a Data Response. As discussed in Streaming Data (§2.3), there is a very specific separation between Data Response frames and actual data, in order to provide consistent support for streaming data. The most immediate and important field in a Data Response is the Response Code, which indicates the status of the requested resource. The code is split into two parts; a 2-bit response type, and a 6-bit response subtype. These are as follows:

Response types:

- 0 - Success
- 1 - Redirection
- 2 - Client error
- 3 - Server error

Response subtypes:

- 0/0 - Success (cacheable)
- 0/1 - Cached
- 0/2 - Partial (cacheable)

- 1/0 - Moved temporarily (cacheable)
- 1/1 - Moved permanently (cacheable)

- 2/0 - Bad request
- 2/1 - Forbidden (cacheable)
- 2/2 - Not found
- 2/3 - Removed (cacheable)

- 3/0 - Server error
- 3/1 - Service unavailable
- 3/2 - Service timeout

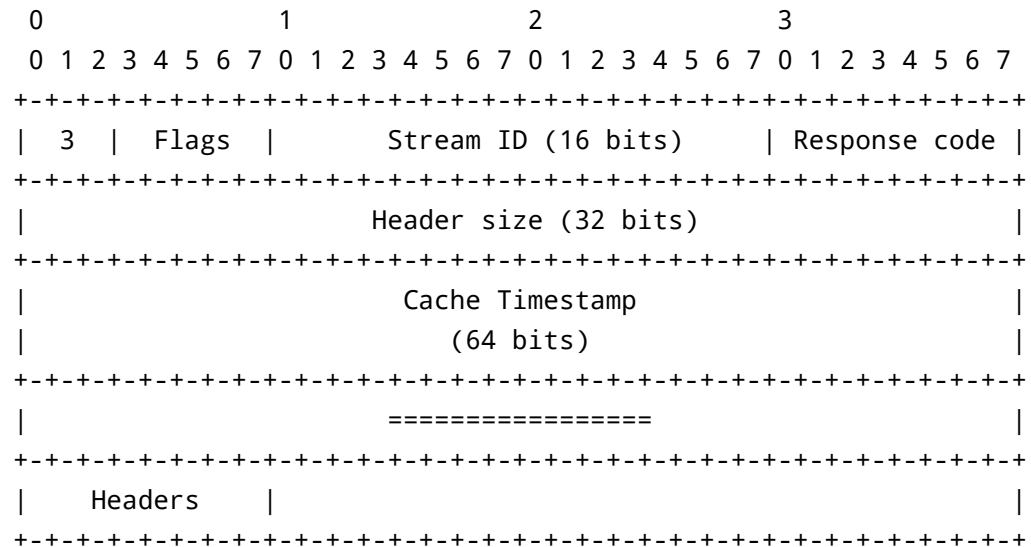
These types and subtypes perform similarly to their HTTP counterparts. The main differences to note are the 0/1, 3/1, and 3/2 responses. A 0/1 response indicates that the Data Request's cache timestamp is equal to the most recent version of the resource, and thus that the cached version should be used. A 3/1 response shows that the response depended on an external service which is unavailable. Similarly, a 3/2 status indicates that the external service request timed out. Ideally, servers should avoid 3/1 and 3/2 responses and instead redirect to an HTML (or other) page explaining the situation to the user, or using an alternative.

If a response is marked cacheable above, the Data Response's cache timestamp field should be used to communicate the time until when the response can be assumed cached. While this time (in Unix nanosecond format) remains in the future, the user agent should consider the response up-to-date. After this point, the timestamp should be presented in the next Data Request to check whether the current version is still correct. If the Data Response's cache timestamp is set to 0, then the response must not be cached.

As in a Data Request, Data Response frames may contain variable-length text fields (such as new cookies, and content encoding declarations), which are stored in the Headers field. Furthermore, as discussed in Streaming Data (§2.3), the Finished flag can be used to indicate that a stream is complete. All 2/* and 3/* responses must set the Finished flag, since these should never be followed by data. Any 1/* responses must leave the Finished flag unset and follow with Data Content frame(s) containing the URL to which the

client should be redirected. Cache responses, such as 0/1 may contain no data and thus set the Finished flag, but most 0/* responses will be followed by data and as a result should not set the flag.

Data Response frames consist of at least 18 bytes and have the following structure:



Flags:

Finish: See streams (WP specification).

Stream ID: See streams (WP specification).

Response code: 2-bit response type, 6-bit subtype.

Header size: Number of bytes in the response header.

Cache: Time of when resource becomes invalid.

- 0 means do not cache.

2.6.5: Data Push

A very common use case in the web is the situation where a user agent has requested a resource and the server responds, knowing that the client will have to request related files immediately after (such as CSS and JavaScript files), but the subsequent requests cannot be issued until the user agent has parsed the initial resource. To solve this problem, SPDY introduced the concept of the server push. In this situation, the server would be able to send the secondary resources to the client directly, rather than waiting for the user agent to parse the initial resource and send a latency-dependant request. The concept works well, and was implemented in SPDY draft 2, which will form the starting point for HTTP 2.0^[7]. Nevertheless, there are situations where the client may already have a cached form of the secondary resource(s). To address this problem, WP introduces the concept of server suggest.

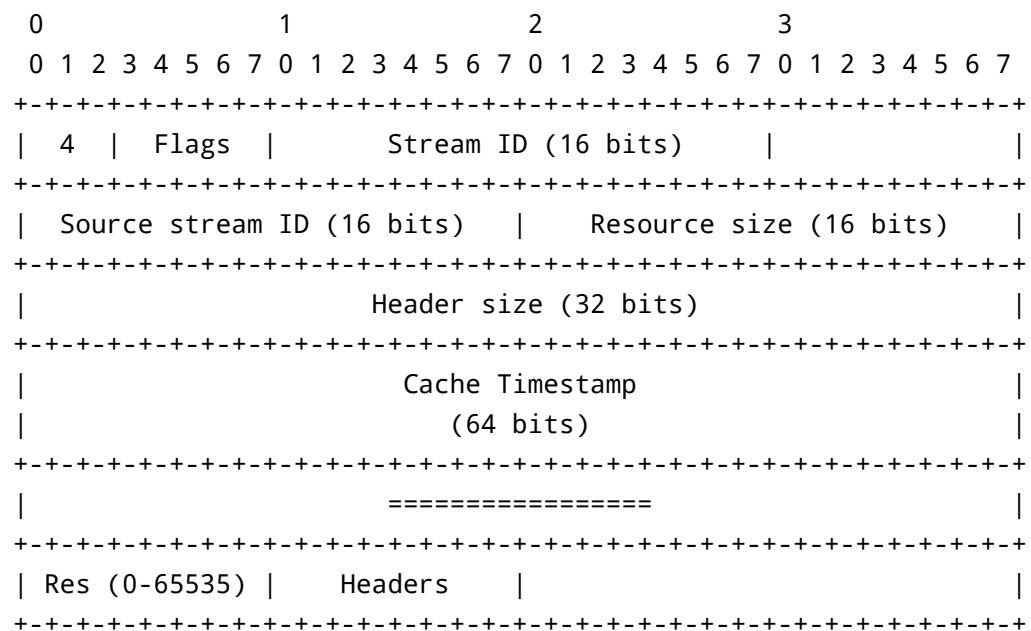
While WP Data Push frames default to similar behaviour to SPDY server pushes, by enabling the Suggest flag, the frame changes from a data stream to a reverse request. The Suggest contains the URI of the resource being suggested, along with its cache timestamp. This prompts the user agent to check their cached version of the named resource. If their timestamp is older than the timestamp attached to the suggestion, then the user agent should immediately submit a request for the resource. Otherwise, they do not need to perform any action, although reading the cached resource into memory may be a useful move. Data Push suggestions will never be followed by data and thus close the stream they use. Whether using a

Data Push for a server push or a suggestion, any variable-length data may be stored in the Headers field, and the Source Stream ID field must be used to indicate which request caused the Data Push, since the user agent may keep different resource sets in separate threads or processes. As mentioned in the interim report and the WP specification, Data Push frames must use an even stream ID, to highlight that it is a server-initiated stream.

If not using the Suggest flag, Data Push frames contain the URI and cache timestamp for the resource being pushed, so that the user agent is aware of which resource it is receiving. The Finished flag must not be sent, since the resource's content must follow. When sending the resource, the data may follow the Data Push immediately, although the client may choose to refuse the push. To do so, the user agent should send a Stream Error with the Finish_Stream status code upon receiving the unwanted Data Push. Note that after this point, the user agent should endeavour not to send Stream Error frames with the Stream_Closed status in response to Data Content frames sending the push data, since the Data Content frames will be being sent for some time before the Stream Error arrives. Similarly, the server should attempt to cease sending data upon receiving the Finish_Stream Stream Error. The server may wish to default to Data Push suggestions, rather than server pushes, if it receives the Finish_Stream code, or to send the Data Push but wait a while before sending any data, in case the user agent refuses all pushes.

The key distinction between Data Push suggestions and pushes is that a push should be used in situations where the chances of the client having a cached version of the resource are very slim. Common examples of this are resources with a far-future cache date, dynamic responses which are never cached, and resources for a situation when caching would be unlikely, such as for new user pages. In cases where the user may well have a cached version of a resource already, a Suggest is preferred, since it does not waste bandwidth with unnecessary data. It is the responsibility of the web application writer to determine which resources should be pushed and which should be suggested.

Data Push frames consist of at least 19 bytes and have the following structure:



Flags:

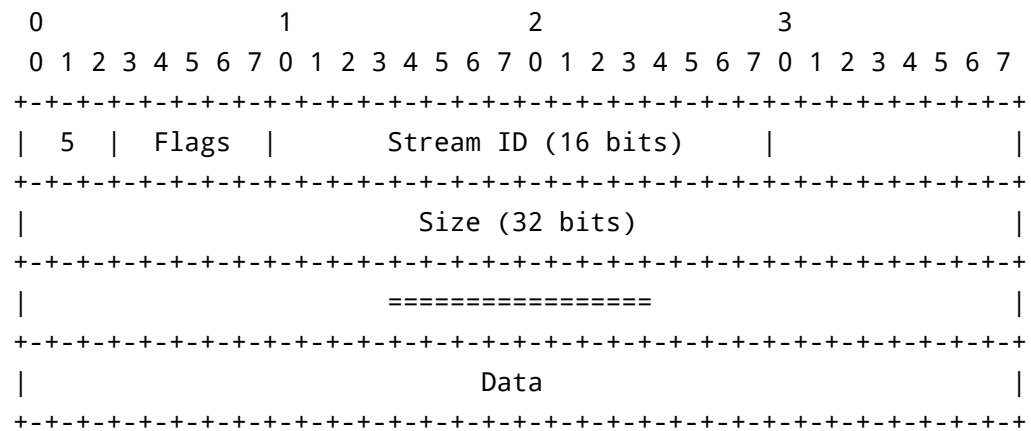
Suggest: This is just a suggestion.

Stream ID:	See streams (WP specification).
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.
Cache timestamp:	Timestamp of when file was last changed. - 0 means do not cache (non-suggest only).
Res:	Resource URI.

2.6.6: Data Content

Several WP frames are followed by data: Data Request frames with the Upload flag set, most Data Response frames, and Data Push frames with the Suggest flag unset. For all of these purposes, the Data Content frame is used. Having a dedicated frame for transferring context-agnostic data is one of the reasons behind the separation between a Data Response and its data, allowing the same structure to be reused for other purposes. Accordingly, the Data Content frame contains only the structure needed to send data: the stream ID on which data is being sent, and the number of bytes being sent. The purpose and use of the Ready and Finish flags is discussed in detail in the WP specification and in Streaming Data (§2.3), but it is an error to send more than one Data Content frame on the same stream with the Finish flag set, since the first closes that stream. As with all WP data, the content is assumed to have UTF-8 encoding, unless it has been declared otherwise in the control frames for the stream, such as in the Header field of a Data Response.

Data Content frames consist of at least 7 bytes and have the following structure:



Flags:

Finish: See streams (WP specification).

Ready: Data is ready to be processed.

Stream ID: See streams (WP specification).

Size: Number of bytes in the whole packet.

2.6.7: Ping Request and Ping Response

In many web applications, it is useful to be able to know whether the server is reachable, such as cloud applications which also have an offline mode. Currently, the only way to perform this is to make an HTTP request to a dedicated URL, which will respond with 204 No Data, since this will inform the user agent that the request succeeded, without expecting any data. Like SPDY, WP provides a dedicated mechanism for this capability, although with a slightly different implementation. While SPDY has a single PING frame, WP uses separate request and response frames, to give clearer semantics and simpler parsing. Both frames consist solely of the standard first byte structure of all WP frames; using the 5-bit flags space for a ping ID. These ping IDs should be used consecutively, starting at 1. When a Ping Request is received, a Ping Response should be sent with maximum priority and the same ping ID. Since the frames are directional, ping IDs do not have separate semantics for odd or even numbers as stream IDs do. If a new Ping Request is to be sent and the most recently-used ping ID is the maximum ping ID (31), the ping ID 1 should be used. 0 is never a valid ping ID, and evokes a Protocol_Error Stream Error.

It's important to note that although SPDY supports a ping mechanism, and that SPDY is implemented in the Google Chrome and FireFox browsers, there is no way yet to make use of this mechanism through the browsers' JavaScript APIs.

Ping Request frames consist of 1 byte and have the following structure:

```
0 1 2 3 4 5 6 7
+---+---+---+---+
| 6 | Ping ID |
+---+---+---+---+
```

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

Ping Response frames consist of 1 byte and have the following structure:

```
0 1 2 3 4 5 6 7
+---+---+---+---+
| 7 | Ping ID |
+---+---+---+---+
```

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

3: Implementation

3.1: Adaptation of “net/http”

The WP library for the Go language^[8] was written both as a proof of concept, and to enable testing of the protocol and comparison with HTTP. The full reasons for picking Go are discussed in the initial report, but one reason was its strong HTTP library, “net/http”. Having a well-written library meant both that I had a starting point for the WP library, and that the experimentation comparing the two protocols would not rely on my ability to implement HTTP; removing a potential element of bias.

Given the quality of the HTTP library and the relevance of its interface, it made an excellent starting point for my own library. The first addition was a file providing an interface to the various WP packets, vastly simplifying data decoding within the actual library. Although these can be useful for the user, as much as possible was abstracted to higher levels. The transmission code was then modified to include the stream handling and state processing, and to add the further error handling required by the different structure, such as errors in stream creation and data sent to closed or invalid streams.

This process was not too challenging, since the existing structure gave clear points where the additions would be made. With similar request/response handling, a lot of the code remained the same, particularly the structure of Handler functions, which are the most common method for the server writer to interact with the protocol.

3.2: Frame Parsing

To simplify the parsing of WP’s binary frames, the first part of the WP library was a file defining data structures for each frame type. Each of these was then given a ‘getter’ and ‘setter’ method for each field to abstract the specifics of each field’s offset and size to a single implementation. Making use of Go’s type declarations (not to be confused with C’s typedef statements), these data structures were defined as being of type byte slice (similar to a byte array), allowing raw data received from network connections to be cast into the relevant WP frame type. Most of these parsing operations were performed with binary shifts, using binary ‘and’ and ‘or’ operators to isolate individual bits where necessary. In addition to this, a further data structure was provided to isolate the first byte of a data stream and to identify which WP frame is being received. This frame is then extracted into the appropriate data structure.

Although the change to Go gave good object orientation and stricter structure, the functional code for extracting individual fields is largely similar to the original C code. To give further abstraction and simplification of the parsing process, the WP library makes good use of Go’s interfaces in providing a data structure which can automate the parsing of WP frames from any data source wrapped by a `bufio.Reader` (a buffered data reader). In turn, a buffered reader can be wrapped around any `io.Reader`; any data structure with a method to read data, which could be a network connection, a file, or even a raw array of bytes. This abstraction allows the WP library to read and parse data from any of these input streams, without needing to know the origin type.

3.3: Top-Level Server Changes

3.3.1: Handler Interface

While the use of Handler functions still remains the main interaction with the average server-writer, the initial implementation could not provide features added by WP, such as streaming responses and server pushes. Implementing these with as little change to the API was a rewarding challenge. The biggest change was to modify the behaviour of `wp.ResponseWriter` so that calling `Write` would send the data immediately, rather than adding to a buffer and sending in a single chunk when the handler returned. This was more in keeping with WP's principle of streaming data, and enabled the sending of very large files which could not fit into a single buffer, as Go's arrays are currently limited to a length of $2^{32}-1$ (a maximum-length byte array could hold up to 4 GB). This also improves performance slightly when sending files, as the data can be sent as it's read from disk, since writing to the connection will often be quicker than reading from disk. Finally, this enables some web-serving optimisations like sending HTML headers before expensive operations like database queries are performed.

The other changes to `wp.ResponseWriter` were the addition of the `Send` method, which is used to indicate the end of a streaming data segment, and methods to push and suggest content. These were slightly tricky to implement, since they crossed the stream boundary, which meant that the caller would not have access to the same connection and state object as they do for their own stream. The proposed solution to this is to have a centralised stream management object, and for each stream to interact through that. Unfortunately, this would require a substantial rewrite of most of the library, for which there was not enough time. This is covered in more detail in Future Work (§5.1). As an interim measure, a method was made available to suggest a particular resource, and another two for pushing either raw data or a named file. These methods each construct the Data Push frames and any additional Data Content frames, and send them in one go, adjusting the connection state accordingly. However, due to the limitations given above, error handling is largely a case of success or failure; if the Data Push is sent successfully, but the Data Content frames fail to send, the library has little alternative to sending a `Protocol_Error` and ending the connection altogether.

As discussed in the streaming data test (§4.2), it became apparent that the initial implementation for sending Data Content frames was inefficient. Originally, a `DataContent` structure was created, encapsulating the data to be sent. Once this was ready, the structure would be sent, meaning that it was considered a single TCP segment. However, this resulted in copying the data into the structure. Although this was fine for other frames, Data Content frames can hold up to 4 GB of data, and copying this much takes a noticeable time. As a result, the library was modified to store just a reference to the data, and to send the frame and its data separately, meaning that the data did not need to be copied. Though difficult to quantify, this was noticeably quicker, and left less work for Go's garbage collector.

3.4: Top-Level Client Changes

Although the existing structure for writing WP servers was fairly satisfactory, the differences between an HTTP client and a WP client are quite substantial, so significant changes had to be made to handle streaming responses. Where the HTTP library's client methods centre around optionally configuring a client and then using an HTTP method like `Get` to fetch a `Response` object, which contains the response body, a WP client may want to interact with the data as it arrives. Accordingly, in addition to atomic functions like `http.Get`, the WP library provides the 'new client API', designed to provide streaming interactivity.

Making use of Go's interfaces, the functions `wp.Fetch` and `wp.FetchJust`¹ take a `ResponseHandler`, which contains the methods used to interact with incoming data. When WP packets are received by `Fetch`, they are parsed and the data presented to the relevant `ResponseHandler` method, where the client can make use of them, including correct error handling and notifications. To simplify its use, the new client API provides two basic `ResponseHandlers`; `DownloadTo` and `DownloadAllTo`², which write the responses to disk. The advantages of these combined functions are that they give a very streamlined way to make full use of the protocol, without requiring understanding of the implementation details. To demonstrate, here is the full code required to download a web page and any required CSS and JavaScript files, assuming the host used WP:

```
package main

import (
    "net/wp"
)

func main() {
    err := wp.Fetch("wp://example.com/", nil, wp.DownloadAllTo("."))
    if err != nil {
        panic(err)
    }
}
```

By comparison, here is the code required to download the same page with HTTP, without automatically fetching the render-required resources:

¹ `wp.FetchJust` will submit the request provided to it. `wp.Fetch` will also present any server suggestions to the `ResponseHandler` and accept server pushes.

² `wp.DownloadTo` will store any requested and pushed resources but will refuse any suggestions. `wp.DownloadAllTo` will accept suggestions and will also parse received HTML and CSS resources and will automatically request any resources a browser would fetch in order to render the page.

```

package main

import (
    "io"
    "net/http"
    "os"
)

func main() {
    response, err := http.Get("http://example.com/")
    if err != nil {
        panic(err)
    }

    file, err := os.Create("./index.html")
    if err != nil {
        panic(err)
    }

    _, err = io.Copy(file, response.Body)
    if err != nil {
        panic(err)
    }
    file.Close()
    response.Body.Close()
}

```

This streamlined API makes it quicker to perform common tasks with the protocol. As the testing framework used in the experimentation shows, implementing a new `ResponseHandler` is fairly simple, and requires minimal knowledge of the underlying protocol.

3.5: Networking Structure

3.5.1: Server Structure

The overall network structure of `net/wp` is essentially identical to that of `net/http`, as I had no reason to change it when I first adapted the package to WP. When a new TCP connection is received, a `conn` structure is initialised with the connection (to which data is written), a pointer to the server which serves the connection, a mutex for state synchronisation, and a simple session state structure. Once the connection is fully initialised, its `serve` method is called, entering a loop which receives a packet, parses it with `connection.readRequest`, performs the appropriate error checking, and if all is well, it constructs `Request` and `response` structures, and passes them to the registered handler for the path being served. Once this handler returns, the response is completed, any additional data is sent, and the loop begins again.

3.5.2: Structure Deficiencies

While this structure is perfectly adequate for HTTP, where all requests on a single connection must be sequential, it is not particularly appropriate for a concurrent protocol like WP. Having discovered this issue, I tried to rectify it by allowing multiple responses to be active at the same time, but synchronisation issues continued to occur until it was clear that I would be wasting time if I continued, so I moved to other tasks. If given time for further development and testing, I would restructure the networking code altogether to create a connection structure to manage reading from and writing to the network, to manage the state of the connection as a whole, and to provide interactions between streams. Separately, I would also make a stream structure to cater for the needs of each individual stream; deferring to the connection where communication between streams was necessary. The connection would then be able to spawn streams independently and to coordinate between them. This would provide a structure far more representative of WP's protocol structure, and far more effective for its implementation. However, this would require at least a month's work, and time constraints on the project did not allow this.

3.5.3: Parsing

Server-side frame parsing is performed by the `ReadRequest` function, which makes good use of the `DataStream` mechanism provided in `wp.go`. This takes a buffered input stream (which is normally a network connection, but could be a file or even an array of bytes, thanks to Go's interface model), scans the first byte to determine its type, and then extracts the frame, reading any variable-size fields' lengths and extracting them too, until the entire frame has been read. This is then returned to `ReadRequest` to be processed. If the received frame is not of an appropriate type, such as when a server receives a server push, then `ReadRequest` returns an error. Otherwise, a `Request` structure is created and populated.

While this is generally adequate, it does place some restrictions on the parsing process. Since `ReadRequest` is called by `connection.readRequest` in the context of a single stream, it can only be called once per stream. This means that upload requests using multiple `Data Content` frames cannot interleave with other frames without breaking the parsing model. To remedy this, the rewrite discussed above would move all parsing into the connection structure, so that interleaved frames can be sent to the appropriate streams. This separation of the connection management layer and the streams is crucial to effective and efficient implementation of WP.

3.5.4: Illustration

To give a clearer indication of how the library functions in practice, the following annotated code snippets follow the creation of a new connection, starting a stream, reaching the server-writer's handler, a response being formed, and that being sent to the client. In most of the code samples some of the implementation details have been removed to give clearer indication of the main code path, without fine details getting in the way. Each snippet from the WP library begins with a comment, giving its source file and line number, so that the unmodified version can be read if desired.

3.5.4.1: Creation of a Listener

The server-writer starts to listen on the network through one of two ways. They can create a `Server` explicitly and call its `ListenAndServe` method, or simply call `wp.ListenAndServe`, which will instantiate a generic `Server` implicitly. This server then creates a TCP network listener and initialises the TLS state if necessary. This then calls `Server.Serve` to enter the connection loop, as shown below:


```

// server.go:1078
func (srv *Server) Serve(l net.Listener) error {
    for {
        rw, err := l.Accept()
        if err != nil {
            // Handle error.
        }

        // Handle timeouts.

        conn, err := srv.newConn(rw)
        if err != nil {
            continue
        }
        go conn.serve()
    }
}

```

Here, an unconditional for loop iterates through accepting a new connection, handling any error if necessary, enforcing timeouts set by the server-writer, creating a new WP connection structure, and starting the new connection in a new goroutine.

3.5.4.2: Serving a Connection

Once the connection has been created, moving from the transport and session layer into the application layer, WP frame parsing begins. Here follows a simplified version of the `conn.serve` method spawned above.

```

// server.go:554
func (c *conn) serve() {
    defer func() {
        // Set up critical error handling.
    }()

    if tlsConn, ok := c.rwc.(*tls.Conn); ok {
        // Complete TLS handshake.
    }

    for {
        w, err := c.readRequest() // returns *response, error
        if err != nil {
            // Handle error.
        }

        // Handle packets other than data requests.
        if w.req.Type == IsHello {
            // Process Hello frame.
            continue
        }

        if w.req.Type == IsPingRequest {
            // Reply to Ping.
            continue
        }

        // Validate Stream ID.

        // Update connection state.

        handler.ServeWP(w, w.req)
        w.finishRequest(false)
    }
}

```

Although most of the details have been abstracted in this snippet, the important factors are clear. `serve` loops indefinitely, using `conn.readRequest` to fetch the next request, performs various stages of error handling and validation on the request, deals with special frames like `PingRequest` frames as appropriate, and checks and updates the connection state (like the `Stream ID`, which must follow strict rules described in section 2.3.2 of the WP specification). Once all this has happened, the server-writer's handler is called; enabling them to process the request as they choose. Once their handler returns, the stream is concluded with `response.finishRequest`, which is shown in [Stream Conclusion \(§3.5.4.7\)](#).

3.5.4.3: Parsing Received Data

The main role performed by `conn.readRequest` is parsing arbitrary data into a WP frame, which can then be handled with protocol logic. To do this, `readRequest` turns to the public `net/wp` function `ReadRequest`, which uses the `DataStream` structure designed for `net/wp`. As this example follows a `DataRequest` arriving on the new connection, an abridged version of `ReadRequest` is shown here, processing the `DataRequest` received:

```
// request.go:138
func ReadRequest(b *bufio.Reader) (req *Request, err error) {
    // Create the Request to be returned.

    data := NewDataStream(b) // Create a DataStream.

    // Get the first byte, which identifies the frame type.
    firstByte, err := data.FirstByte()
    if err != nil {
        return nil, err
    }

    // Find the matching type.
    switch firstByte.PacketType() {

    case IsDataRequest:
        // Parse the DataRequest with DataStream.DataRequest()
        // Populate the Request structure.
        return req, nil

    // ... Handle other frame types with other case statements.

    }
}
```

`ReadRequest` begins by creating the `Response` structure to return, and making a `DataStream` (a specialised buffered input reader which reads data from the network connection), parses it, and provides a parsed version of the received frame. Once the `DataStream` has been created from the network connection, the first byte of the received data is extracted. As discussed in [Frame Structure \(§2.6\)](#), all WP frames share a common first byte, which includes the frame's type, so from this first byte, the `DataStream` is able to determine which frame's parsing method should be used.

`ReadRequest` uses a `switch` control structure on the first byte, using a `case` statement for each frame type. When a `case` statement triggers, as above, the appropriate method is called on the `DataStream`, parsing the frame and returning it. The details of this structure are then used to fill in the fields of the `Request`, which is returned.

3.5.4.4: Parsing a Frame

The actual parsing of a frame is performed in stages. Once the `DataStream` has extracted the first byte, it

uses binary operations to isolate the type number. Then, once `ReadRequest` has used the switch to identify the type, it calls the appropriate parsing method on `DataStream`. This performs the raw data parsing, and returns the populated frame. Continuing the example, the following code snippets show the extraction of the first byte, the isolation of the frame type, and the `DataStream.DataRequest` method.

```
// wp.go:182
func (reader *DataStream) FirstByte() (FirstByte, error) {
    first, err := reader.Peek(1)
    if err != nil {
        return nil, err
    }
    return FirstByte(first), nil
}

// wp.go:237
func (packet FirstByte) PacketType() int {
    return int((packet[0] & 0xe0) >> 5)
}

// wp.go:551
func (reader DataStream) DataRequest() (DataRequest, error) {
    buffer := make([]byte, DataRequestSize)

    // Fetch packet header.
    if _, err := reader.Read(buffer); err != nil {
        return nil, err
    }

    // Resize the buffer to take the headers.

    // Read in the remaining data.
    if _, err := reader.Read(buffer[start:]); err != nil {
        return nil, err
    }

    return DataRequest(buffer), nil
}
```

Here, `DataStream.FirstByte` uses the buffered reader package (`bufio`) to read the first byte on the connection without removing it from the buffer. This means that later calls to `Read` will fetch all the data, without needing to reconstruct the first byte. Next, `FirstByte.PacketType` uses a binary AND to zero the byte, except the bits of the type information. This is then right-shifted until the least significant bit is the rightmost bit. Then, by casting from a byte to an `int`, the type number is returned.

When `DataStream.DataRequest` is called, a simple buffer (a slice of bytes, functionally similar to an array of bytes) is created, with its size set to the minimum length of a `DataRequest` frame. The connection then reads that many bytes into the buffer. The frame is then inspected to parse relevant fields like the size of any variable-length headers, so that the total number of bytes of variable-length data is known. The buffer is then extended to the appropriate size, and the remaining data read into the buffer. Finally, this is cast into a `DataRequest` frame, and is returned to `ReadRequest`, which finishes initialising the `Request` structure.

3.5.4.5: Sample Handler

Once the data has been parsed and the `Request` constructed, it is passed to a `Handler`. Handlers are registered to a particular request path, or a subtree thereof. A small demonstration of this is shown below in an example server application, which has been annotated with comments.

```
package main

import (
    "net/http"
)

// Handler.
func Web(writer http.ResponseWriter, reader *http.Request) {
    // Add HTTP Strict Transport Security.
    name := "Strict-Transport-Security"
    value := "max-age=31536000; includeSubDomains"
    w.Header().Add(name, value)

    wp.ServeFile(writer, reader, "." + reader.RequestURI)
}

func main() {
    // Register the Web handler function to
    // the path subtree "/".
    wp.HandleFunc("/", Web)

    // This actually starts the listening.
    wp.ListenAndServe(":99", "cert.pem", "key.pem", nil)
}
```

This simple application starts a server on port 99 (the default port for WP), where all requests are handled by the `web` function, as this is registered for the root path, and no other handlers have been registered. The handler provided adds an HSTS^[9] header to the response, and then serves the requested resource from the local disk, where the root path (`"/"`) is the current working directory for the server. The `wp.ServeFile` function takes care of the specifics of file serving, like giving 2/2 responses for requested resources which could not be found. Although the current implementation does not have content type detection, it is part of the future work for the library and should not require much extra work, thanks to helper functions in the `net/http` library.

3.5.4.6: Writing Data

Although it has not been called explicitly in this particular handler, data is written to the response by calling `writer.Write`. Since the handler shown here has used `wp.ServeFile`, we cannot see `writer.Write` in action, but it is used by `wp.ServeFile` to send the resource's contents to the client. The actual method implementation is illustrated here:

```
// server.go:180
func (w *response) Write(data []byte) (n int, err error) {
    // Ensure the DataResponse has been prepared.

    // Do nothing if data is empty.

    // Handle content-length.

    // Calculate frame sizes.

    // Create data buffer.
    buffer := make([]byte, responseSize + contentSize)

    if !w.responseSent {
        // Create DataResponse frame.
        w.responseSent = true
    }

    // Create DataContent frame.

    // Send data down the wire.
    w.conn.rwc.Write(buffer)
    w.conn.rwc.Write(data)

    return size, nil
}
```

As can be seen, the process is fairly simple, with the relevant frames being created, initialised, and then written to the network. The `responseSent` variable is simply used to ensure that only one `DataResponse` is sent for each request. Where `net/http` simply stores data given to `response.Write` in a buffer and sends it atomically in `request.finishRequest`, `net/wp` sends data immediately; making use of WP's streaming data semantics to accelerate transmission, since the total amount of data being sent does not need to be known when the `DataResponse` is sent.

By comparison, an HTTP server must either know the full response length for the Content-Length header, or use chunked Transfer-Encoding to provide basic data streaming capabilities, although this introduces the complexity of deciding whether to send immediately or simply buffer. If chunked Transfer-Encoding is used, then any compression must be performed on the chunks individually; not on the data as a whole. Thus, when sending pre-compressed data, chunked Transfer-Encoding may not be possible.

3.5.4.7: Stream Conclusion

Once the handler function for a particular request returns, we can assume that the handling of that stream is finished, since there is no longer any way for the server-writer to send or receive data on that stream. As a result, execution continues to `request.finishRequest`, which is illustrated below:

```
// server.go:423
func (w *response) finishRequest(streaming bool) {
    // Ensure headers are prepared.

    var buffer []byte

    if !w.responseSent {
        buffer = make([]byte, DataResponseSize + len(headers))

        // Construct a DataResponse frame.

        w.responseSent = true
    } else {
        buffer = make([]byte, DataContentSize)

        // Prepare an empty DataContent frame.
    }

    // Send data down the wire.
    w.conn.rwc.Write(buffer)
}
```

To conclude the stream, the 'Finish' flag is used. If, for whatever reason, a `DataResponse` has not yet been sent, then one is constructed, the flag set, and the frame sent. If any data has been sent, however, a `DataResponse` will already have been sent, so an empty `DataContent` frame is initialised instead, and the 'Finish' flag set. Alternatively, a `StreamError` frame could be sent with the `FinishStream` status code.

4: Results and Evaluation

4.1: Atomic Resource Testing

4.1.1: Preface

With WP's focus on the web, the most important evaluation of the protocol would compare it with the existing web protocols in transmitting web content. Although websites vary enormously in size and composition, there are some common trends that can be seen over time. Since November 2010, the HTTP Archive^[2] has performed similar work to a search engine, but rather than indexing the sites' content, the content is analysed for its properties. This data is then made public and available for further analysis. Using this data, it is easy to see that popular websites have grown in many respects. Not only is more data being sent, but each page is using more resources, due to rising integration with social media and other third-party services. This rise in the number of resources, and particularly the number of different hostnames, means that request parallelisation becomes more important for high-performance web networking.

As of mid-March 2013, the average number of resources on a page is 90, totalling 1,311 kB. These resources are spread over an average of 15 different domains. This large number of domains is likely due both to rising use of third-party services and to the use of so-called domain sharding: the process of using multiple subdomains for secondary resource serving, such as images and stylesheets. This has been used enormously to supplement request parallelisation, since most mainstream web browsers only allow up to seven simultaneous connections to a single domain. By sharding domains into multiple subdomains, the number of connections available is increased substantially. However, this large number of parallel connections each requires a full TCP (and possibly SSL/TLS) handshake, costing between one and three RTTs.

The advent and gradual uptake of SPDY, and the comparable design of HTTP 2.0, is hoped to herald a great 'unsharding' process, where resources are moved back into a single domain, so that request multiplexing can have its greatest effect, since although SPDY and WP can multiplex all the requests on a particular domain, a separate connection is required for each subdomain. For the average website, this brings the number of connections from a possible 90 down to 15, but even fewer would be preferable.

Another crucial factor in performing relevant experiments on web performance, is the network latency between the user agent and the server. According to data from Google^[10], the average RTT experienced by Google Search in the United States is 50-60 ms; giving a latency of 25-30 ms. Assuming that this gives a fairly good indication of the average or median latency for the broadband web as a whole, a latency of 30 ms was used in my experimentation.

It should be noted that these figures solely represent the latency for broadband internet connections; cable or faster. The latency experienced on DSL connections, and on mobile internet services, tends to be larger, according to further data from Google^[10]. With US telecoms providers Sprint and Verizon claiming a latency of around 100 ms in their terms of service^[11] for LTE (often marketed as 4G), and latency ranging from 200-450 ms for 3G services^[12], it's clear that any differences established between multiplexing protocols like WP and SPDY, and parallel connection protocols like HTTP, would be magnified when used on mobile internet services.

4.1.2: Experiment

With the great diversity shown across the web; from simple, largely text-based websites like Wikipedia to full-blown web applications like Google Docs (with which this report was written) and media-rich pages like YouTube, the testing required to fully evaluate a new web protocol is worthy of a final year project in itself. The key network characteristics like latency, bandwidth, data frequency, data rate, and data volume often vary substantially on a single page, let alone across a website, or the web as a whole. As a result, my priority was to evaluate the improvements provided by WP in the average case for a modern website, and to extrapolate likely trends for other kinds of web resource, depending on its differences from the average case.

The main website experiment used HTTP Archive data from 1st March 2013 and Google's latency data for the Google Search page in 2012 as sources for the average web site structure and network latency respectively. I generated 89 text files containing "ipsum lorem" text (since the data content is not particularly relevant), and a single HTML document referencing the other files. I was careful to ensure that the total volume of all 90 resources was 1,311 kB; the average total resource size given by HTTP Archive. Similarly, these resources were split across 15 different 'domains', although this was actually achieved with different port numbers, since this still triggers a separate TCP connection for each, but requires less setup. The site was served separately (but simultaneously) with HTTP, HTTPS, SPDY, and WP, using the applications provided in the appendix. Each protocol used a separate port range, with the HTTP test using ports 1200-1214, HTTPS using 1300-1314, SPDY using 1500-1514, and WP using 1400-1414.

Since the Unix loopback interface exhibits far lower latency than the 30 ms average seen in the web, an artificial latency was induced. This was performed^[13] with the IP Firewall tool present in most Unix-like kernels. I was careful to add 30 ms latency on both inward and outward connections for each port used, and to test that the net effect was accurate, by using the Wireshark^[14] application to examine a dummy test.

The tests were performed sequentially by a testing framework written in Go. The framework is optionally provided with the number of times to perform each test; defaulting to one. For each protocol, the framework requests the page, storing the resulting data into a simple buffer. If multiple iterations have been requested, they are performed successively. Once all iterations have been performed, the total volume of data and number of resources received is checked and printed to screen, and the next protocol is started. In each case, a single request is performed first, in case there is a noticeable delay on the first request. The data from this request is dropped on receipt. To illustrate the structure of the test more clearly, a simplified version of the actual framework is presented here. The full version of the framework is attached in appendix 2. The servers are also in the appendices, but were essentially identical, so did not warrant repeating.

```

// test.go
package main

import (
    "bytes"
    "crypto/sha1"
    "crypto/tls"
    "errors"
    "flag"
    "fmt"
    "io"
    "log"
    "net/browser"
    "net/http"
    "net/spdy"
    "net/wp"
    "os"
    "strings"
    "time"
)

func main() {

    // Initialise.

    // Run the tests.
    t1 := HTTP()
    t2 := HTTPS()
    t3 := SPDY()
    t4 := WP()
}

func HTTP() time.Duration {
    // Warm up the connection with a dummy first test.
    // Use Http() to run the actual tests.
    // Display results.
}

```

```

func HTTPS() time.Duration {
    // Warm up the connection with a dummy first test.
    // Use Http() to run the tests, but with encryption enabled.
    // Display results.
}

func SPDY() time.Duration {
    // Warm up the connection with a dummy first test.
    // Use Spdy() to run the actual tests.
    // Display results.
}

func WP() time.Duration {
    // Warm up the connection with a dummy first test.

    results := make([]*TestHandler, *times)

    // Loop through the number of tests requested.
    for i := 0; i < *times; i++ {
        // Prepare the results structure.
        results[i] = &TestHandler{
            files: make([]*bytes.Buffer, 0, 1),
            filenames: make([]string, 0, 1),
            follow: true,
        }

        // Perform the tests.
        start := "wp://localhost:1400/web/index_wp.html"
        wp.Fetch(start, config, results[i])
    }

    // Display results.
}

```

```

// TestHandler is created to use WP's "new client API", as seen
// in the line wp.Fetch(start, config, results[i]) above.
// It stores the file names and their content in the parallel
// slices filenames and files. It also uses the init bool to
// record the start time exactly once, and updates the finish
// every time close is called, so the last call will overwrite
// the previous with the most up-to-date time.
type TestHandler struct {
    filenames []string
    files      []*bytes.Buffer
    follow     bool
    init       bool
    id         int
    start      time.Time
    finish     time.Duration
    size       int64
    number     int
}

// Fetch uses Init to tell the client that a resource has been
// requested, and provides a unique ID for that resource. This
// makes it easy for the client to store the data in an array.
func (t *TestHandler) Init(s string, id int) error {
    // Setup.
    if !t.init {
        t.start = time.Now()
        t.init = true
    }

    t.filenames = append(t.filenames, s)
    t.files = append(t.files, new(bytes.Buffer))
    t.id++

    return nil
}

```

```
// Fetch uses Suggest to ask the client whether they wish to request
// a resource suggested by the server. Its cache timestamp is also
// provided. This TestHandler has the follow boolean set to whether
// it should follow links and suggestions, so we use that to decide.
// It assigns the arguments to _ to ignore them, since they do not
// influence the decision. This means that they do not get added to
// the stack frame.
func (t *TestHandler) Suggest(_ string, _ uint64) (bool, error) {
    return t.follow, nil
}
```

```

// This is the main function of the handler. Fetch calls Handle
// whenever data arrives on the connection; providing the client
// with a Response structure and the unique ID of the resource.
// The TestHandler pulls the content of the Response into its
// data buffer.
// The Close boolean on the Response is set to true if the stream
// that delivered it is now closed. This informs the client that
// if there is any further processing they wish to perform on the
// data, now is a good time to do it.
// We update the finish timer, number of resources fetched, total
// amount of data fetched, and check for any links in the document.
// By returning a slice of strings, with each giving a URL, Fetch
// Automatically handles the creation of new requests, and the
// resulting calls to Init.
func (t *TestHandler) Handle(r *wp.Response, id int) ([]string, error) {
    // Error checking.

    // Write the response data to the buffer.
    io.Copy(t.files[id], r.Body)

    // This was the final packet; the file is now complete.
    if r.Close {

        // Update stats.
        t.finish = time.Since(t.start)
        t.number++
        t.size += int64(t.files[id].Len())

        // If we're not just downloading one file.
        if t.follow {

            // Fetch the links.
            links, err := browser.Links(t.files[id].String())
            return links, nil
        }
    }

    return nil, nil
}

// Fetch calls Close if ever an error occurs, so that
// the client may try to handle the error. It is also

```

```

// called when all requested resources have been
// received and their streams closed. In this case,
// the supplied error will be nil.
func (t *TestHandler) Close(err error) error {

    // We do not handle errors here, so
    // we simply return it. It will then
    // be returned by Fetch.
    if err != nil {
        return err
    }

    // Loop through files.
    for i, file := range t.files {
        // For each file, we open the original file, and calculate
        // its SHA1 cryptographic hash, before getting the hash of
        // the version received and stored in the files buffer.
        // If the hashes do not match, we return an error.
    }

    return nil
}

// The HTTP/HTTPS/SPDY results structure is more simple,
// since there is less direct involvement with the API.
type HttpResults struct {
    number int
    size   int64
    start  time.Time
    finish time.Duration
}

```

```

// Http is called once per test being performed.
func Http(s string, follow bool, results *HttpResults) {
    // Initialise the test.

    // Fetch the starting resource.
    r, err := client.Get(s)

    // Store the data into a buffer.
    n, err := io.Copy(buf, r.Body)

    if follow {
        links, err := browser.Links(buf.String())

        for len(links) != 0 {
            // Initialise the request.

            r, err = client.Get(links[0])
            n, err := io.Copy(buf, r.Body)
            l, err := browser.Links(buf.String())
            links = append(links, l...)
            links = links[1:]
        }
    }

    results.finish = time.Since(results.start)

    // Check the resources were received correctly.
}

func Spdy(s string, follow bool, results *HttpResults) {
    // Spdy functions identically to Http, but with
    // spdy.Transport, rather than http.Transport.
}

```

For the main test, the framework was run with ten iterations, since preparatory tests had indicated that more than this began to cause errors in Wireshark's packet capturing functionality. This is likely due to the large data throughput (over 1 MB per test, with ten tests per protocol and four protocols, in addition to TCP and TLS traffic). I would have preferred to have gathered more data than this, but the reliability of the data was more important than its volume.

4.1.3: Results

The full data set is available in the packet capture, which contains the entire set of data transmitted during the test. From this, I extracted the key data for each iteration. This consisted of the number of packets transmitted, the total number of bytes this comprised, the total time taken to transmit all data, and the average number of bytes sent per packet. These statistics were generally found using Wireshark's filters to isolate a single TCP connection and analyse its characteristics. The one exception is the total time statistic. Since Go's net/http library (and thus my net/wp library) leaves TCP connections open until they are explicitly closed in order to reuse connections, the statistics provided by Wireshark for the duration of the connection bore little relationship to the amount of time sent to transmit the data. To account for this, the amount of time between the final data packet and the first FIN packet of the TCP termination handshake was subtracted from the total time. This adjustment was made to all connections, even in cases where the gap was very small, to ensure consistency.

Since the first connection on each protocol was used to 'warm up' the connection, its results often varied noticeably from the rest of the data, and was thus ignored. For example, the first test on HTTP had 60% fewer bytes per packet than the other ten tests, whereas SPDY used 74% fewer packets in its first test. Without these extraneous results, the data was fairly consistent within each protocol, so the low number of tests would appear not to have had a drastic effect.

4.1.4: Analysis

4.1.4.1: SPDY

Before discussing the results as a whole, the results shown for SPDY should be explained. Whereas the Go standard library was used for HTTP and HTTPS, and I wrote the WP library, finding Go support for SPDY was challenging. I was only able to find two libraries; `jmckaskill's gospdy`^[15], and `shykes' spdy-go`^[16]. Since `spdy-go` is still under active development and is not complete (tests with the Chrome web browser gave errors), I opted to use `gospdy`. This package contained some minor bugs, which I fixed (and later pushed upstream), but functioned correctly; providing a SPDY v3 client and server, which I tested with Chrome. As can be seen in the data, however, this library performed worse than every other protocol on all four metrics.

I haven't been able to gain a full understanding of the code, since it uses a fundamentally different application structure from the other three libraries, so I cannot give a reliable explanation for its results, but it would appear to be sending a large volume of additional data. The reason for this is unclear, since the data received at the application level is identical to the other protocols.

I suspect that the problem may be due to an architectural fault, since the package serves both SPDY and HTTPS, depending on the Next Protocol Navigation (NPN) choices received. It is possible that this additional complexity may have caused the data problem.

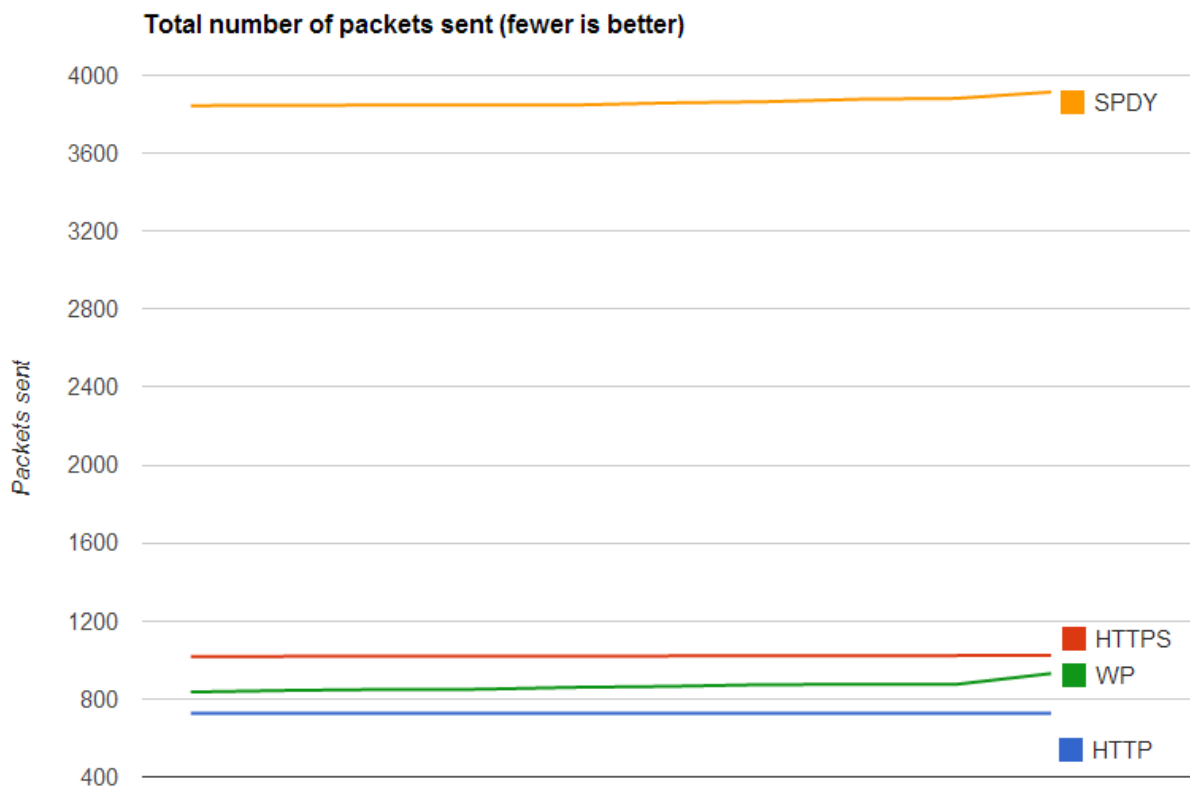
Since completing the tests, I have begun my own implementation of SPDY in Go^[17], although unfortunately, it will not be in a state suitable for performance evaluation until after the completion of the project. If the project was allowed to continue for another two to three months, I would be able to continue the development of this SPDY library to the same extent as the project's WP library, and could implement the library rewrite discussed in Structure Deficiencies (§3.5.2). A further series of tests could then be performed on comparable implementations of SPDY and WP, which would give a realistic comparison of their respective performance capabilities.

Development of WP has provided a far deeper understanding of protocol structure and the implementation of different kinds of protocol. The result of this experience is that the development of my SPDY library has been far quicker than my implementation of WP, particularly when compared to the initial

development in C, and then Go, at the beginning of the project. Using the structure intended for the WP rewrite in SPDY has given particular improvements in reduced coupling and greater cohesion within the library. Although the SPDY library is currently under private development, it will be released publicly under a revised BSD license in the next two months^[17].

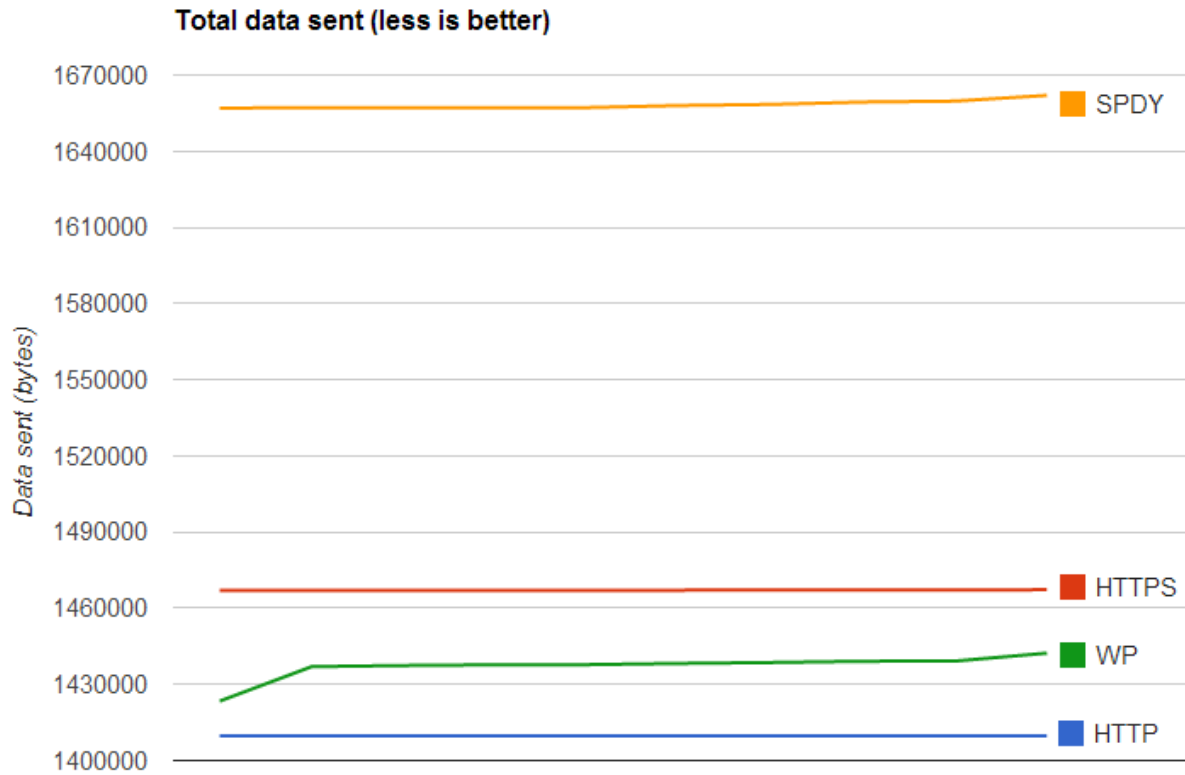
4.1.4.2: Number of Packets Sent

The first metric used is a simple count of the number of packets sent by each protocol. The results immediately show that something is very wrong in the SPDY implementation, as discussed above. A comparison between HTTP and HTTPS shows that encrypted sessions use an average of 20 packets more per domain, which are probably used to complete the TLS connection; such as sending the server's certificate. WP uses slightly fewer packets than HTTPS, which is perhaps due to the reduced data volume requiring fewer TCP packets. WP still requires more packets than HTTP, due to the overhead caused by TLS.



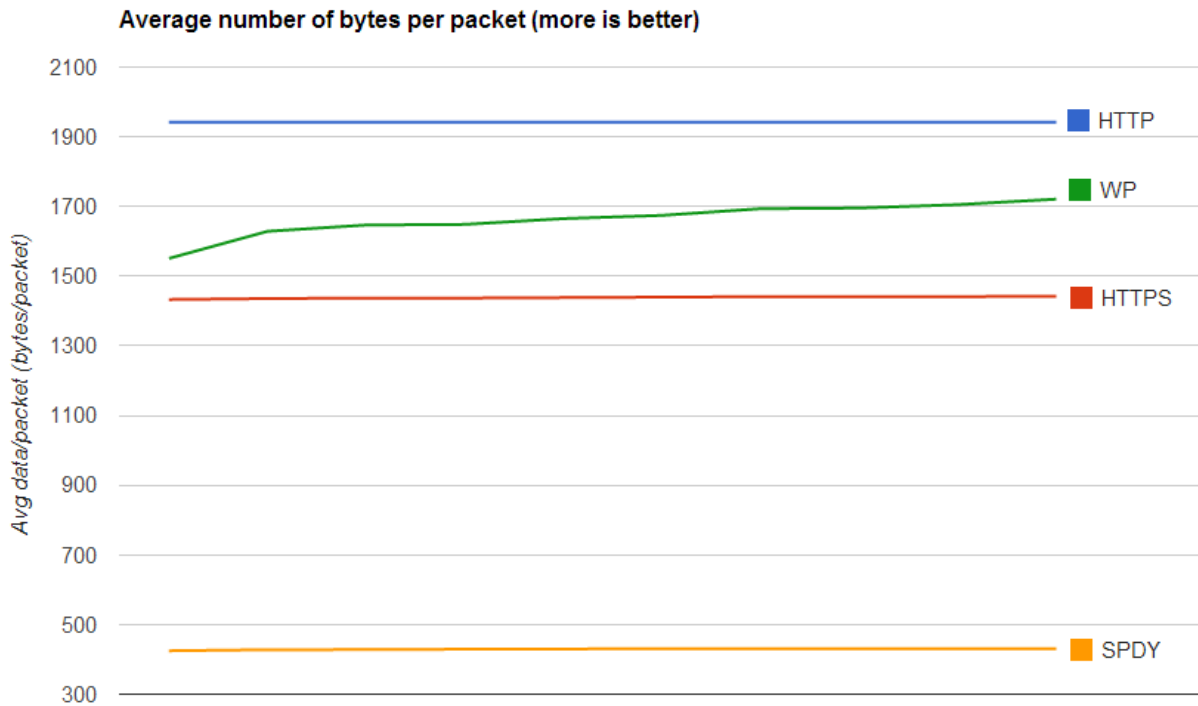
4.1.4.3: Data Transmitted

Since the application-layer data being sent was identical between protocols, the total data volume is an indication of each protocol's sum overhead. Unsurprisingly, this gives very similar data to the number of packets sent, since the results will largely be affected by TLS overhead. WP's slightly lower overhead than HTTPS (and HTTP, once TLS is taken into account) will mostly be due to using fewer TCP sessions, and thus having less overhead from additional handshakes. The savings in header size is unlikely to have reached a single kilobyte, since these requests were not sending many HTTP headers, so its effect on the overall data transmission volume will be minimal.



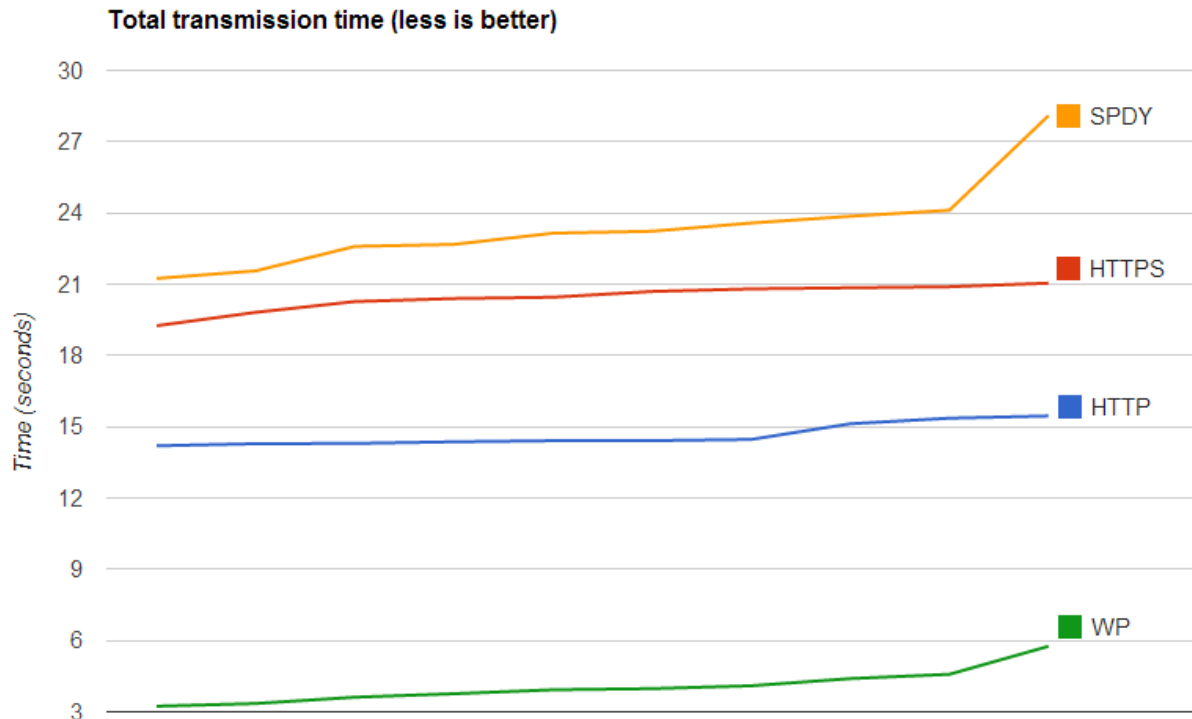
4.1.4.4: Data per Packet

Showing much the same results as the previous two metrics, the number of bytes per packet remained fairly consistent between tests, with WP giving the most variation. SPDY's very poor data per packet rate would suggest that it was sending a huge number of very small packets, since this would reduce the average packet size, while still sending a large data volume, as the previous graphs would suggest. Note that the difference between SPDY and the other protocols is greatest in the packet count results, which would corroborate this further.



4.1.4.5: Time

This graph shows the total amount of time taken to complete the transmission, as explained in section 4.3. The results clearly favour WP over the other protocols, with SPDY performing far better (in comparison to the other protocols) than previously. This would indicate that had SPDY not sent so many small packets, it would likely have had a significantly reduced transmission time. Given its structure, I would expect a time performance somewhere between HTTP and WP, since although it has the same TCP gains as WP, it has a more verbose structure, so is unlikely to be quite as quick as WP.



WP's results are quite staggering: performing between 2.68x and 4.39x as quickly as HTTP, and between 3.66x and 5.95x as quickly as HTTPS. According to the packet capture, the HTTP library decided not to use chunked Transfer-Encoding, so each resource was read into a buffer, then sent as a whole, once the data's length had been read to set the content-length header. It's possible that this will have been slower than WP's streaming approach, where the data will have been sent in a stream of chunks (32 kB in size, due to the use of the `io.Copy` function). If the I/O from disk was slow, sending the data as it is read may have been quicker, since less work would be done once the I/O was finished. Furthermore, the use of fewer TCP sessions, and the resulting drop in the number of TCP handshakes which have to take place (each costing a full 60 ms round-trip), will have accelerated the processing, since the request can be sent sooner, so the response will be processed sooner.

The main factor in WP's temporal performance, I suspect, is WP's circumvention of TCP's Slow-Start mechanism. As mentioned in the interim report, TCP uses a congestion window (CWND) to avoid network congestion. This CWND specifies the number of segments which can be sent between acknowledgements (ACKs) from the recipient. Since each ACK takes a whole round-trip (RTT), the CWND has a large effect on the data transmission rate in high-latency connections. This window is grown gradually with each successful ACK, but this takes time. Since HTTP and HTTPS will be using many more TCP connections, they will spend more time in the early phases of Slow-Start when the CWND is small. While Slow-Start does not have an appreciable effect on overall efficiency in long-term connections, it has a dramatic effect on brief connections which transmit small files, as is often the case in the modern web. According to data from HTTP Archive, the average non-image resource takes up just 15.86 kB, while the initial CWND is between one and six segments (depending on kernel), with each segment containing up to 1,460 bytes^[18]. This gives an initial limit of 8.55 kB per RTT. In recent versions of the Linux kernel, the initial CWND (`initcwnd`) has been increased to 10 segments (14.26 kB per RTT).

With even the Linux kernel's generous initial CWND of 10, the average resource will be split into multiple windows, taking at least 2 RTTs to send. However, if we can use the same TCP session for multiple requests,

as in SPDY and WP, then only the first resource will suffer the effects of Slow-Start, since by the time subsequent requests are made, the CWND will have grown larger, meaning that a large proportion of resources will fit into a single window, taking a single RTT. In high-latency connections, this can have a huge effect on the connection's performance. In the tests run for this experiment, this reduces the time taken to transmit a resource by at least 60 ms, in addition to the extra RTTs used to initiate the TCP handshake. Assuming a single RTT for each, this leaves HTTP taking 120 ms longer than WP to send the same data, without the other improvements provided by frame interleaving and the avoidance of head-of-line issues.

4.1.5: Extrapolation

Since the main savings made over HTTP depend on avoiding the effects of TCP handshakes and slow-start, the performance improvements provided by WP will be at their greatest when requesting many resources from the same domain. Single requests to additional domains will give almost identical performance, since WP will still have to use a new TCP connection. With single-resource requests, the improvements will be limited to the size reductions provided by binary framing. Although this will give slightly higher performance than HTTPS, the overhead induced by TLS means that pages with a single resource per domain will be quicker in unencrypted HTTP. In contrast, pages with many resources per domain will yield large performance benefits, as shown in the test above.

The above applies to 'generic' cases with no server-writer optimisations. Just as time has seen servers 'shard' their domains by creating additional subdomains to allow web browsers to make more simultaneous parallel connections, server-writers aiming to improve their websites' performance with WP or SPDY would need to conduct a process described as 'unsharding', where the subdomains are coalesced into a single domain, to minimise the number of domains used. Furthermore, careful and thorough use of server push/suggest would give additional performance gains, by sending resources without waiting for their request; thus saving another RTT.

4.2: Streaming Resource Testing

4.2.1: Preface

Since the web is now used for far more than serving documents, WP needed to be tested in other ways too. Although a comparison between WP and the Real-time Transport Protocol (RTP) would have been very informative for its effectiveness in true streaming media situations, there is no current support for RTP in Go, and time constraints prevented me from writing my own library for it. However, one streaming data situation for which HTTP is used, and in which it could be compared against WP, is the use of AJAX for continuous data delivery in web pages.

AJAX, Asynchronous JavaScript And XML, is a technology where web application JavaScript can use the browser's networking API to make asynchronous requests over HTTP. These requests are asynchronous in that the application registers a callback function with the request, which is run on the response's arrival (or lack thereof). In the meantime, execution of the application continues. This has had an enormous influence on the web culture, since web pages can keep themselves up-to-date with global change. Where previously pages would need to be refreshed occasionally, we now have rich applications like Twitter and Facebook, which keep up to date constantly, like a traditional desktop application. However, in this circumstance, only HTTP (or HTTPS, where appropriate) is available. Even though the browser's network engine can make use of many other protocols like RTP and WebSockets, these are not made available to the JavaScript API.

4.2.2: Experiment

To demonstrate the difference made by switching to WP, I created a sample application modelled on Twitter's page, where there is a continuous stream of incoming data, arriving at a varying interval. While HTTP's transactional nature can work in this kind of use, WP's streaming responses thrive in it. The structure of the application was fairly simple, and is shown in Figure 1 below. On the server, data becomes available at a random interval, between 1 ms and an upper bound set from a command-line argument. This upper limit was set to 100 ms, 200 ms, 300 ms, 400 ms, 500 ms, 600 ms, 700 ms, 800 ms, 900 ms, 1 s, 2 s, and 5 s, over the course of the tests. Each time, a random amount of data was produced, ranging from 1 B to 1 MB, to simulate the fact that the amount of data available to Twitter users will vary over time. Whenever this data was made available, it was immediately given to an HTTP server and a WP server, simultaneously. Each server then had to give the data to a single client as quickly as possible.

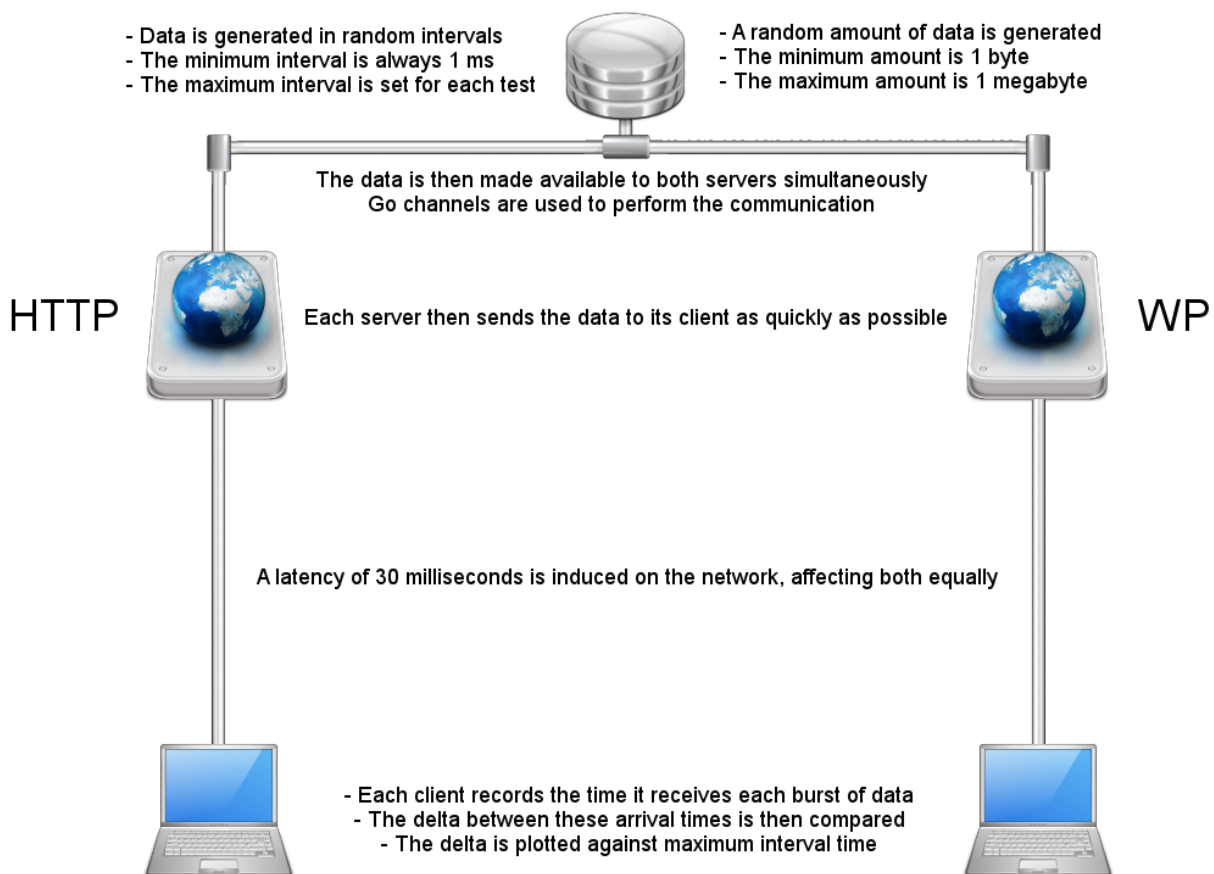


Figure 1: A diagram of the Streaming Resource Test^[19].

Since any good HTTP client in this situation would keep the TCP connection open, and make successive requests, there seemed little point in measuring the number and size of the packets being sent, as they would be fairly similar between the two protocols. Accordingly, the main measurement was of time, since a protocol that brings back the AJAX data sooner will give a more responsive feel to the web application. As the data was made available to both servers simultaneously, the main factor of comparison was the arrival time. As a result, both clients stored the number of bytes received, and the receipt time, for each iteration. The data itself was not stored, to save time and memory use. Once the experiment was complete (after a fixed number of iterations provided by the command-line), the results of each protocol were compared;

making sure that both clients had received the same amount of data each time. All this data (the iteration number, data creation interval, data size, and arrival time for each protocol) was then collected and written to a Comma-Separated Value (CSV) file provided at the command-line.

The WP client was fairly simple to design; the client makes a single request, and the server provides a streaming response until the experiment ends, at which point it sends a final Data Content frame with the 'Finish' flag set. My initial idea for the HTTP client was to use simple polling, where the client would make requests at a set interval, and hope to find the interval with the greatest efficiency. However, it immediately became clear that this would be neither effective nor fair. In the situation of server-initiated data streams with continuous delivery, real-world web applications use Comet, which is a family of technologies used to enable stream-like data in HTTP. The most common and simple form of Comet is a technique called long polling, where the client makes a single request and the server simply doesn't respond until data becomes available. When the client receives the response, it starts to process the data and immediately makes another request. This is fairly efficient and makes best use of HTTP in the situation.

Accordingly, the HTTP client in this experiment used long polling, with the server returning a 204 No Content response at the end of the experiment, to signal the finish. This was very simple to implement, and provided all the functionality needed. As with the previous experiment, a flat latency of 30 ms was induced in the network; giving a web-average RTT of 60 ms. Although encrypted WP was compared against unencrypted HTTP, this did not have a noticeable effect on the experiment, since the additional two RTTs required by the TLS handshake take place only once, and symmetric-key encryption has become incredibly quick.

To illustrate the structure of the test from an implementation perspective, a simplified form of the actual application is included below. The unmodified version is included in the appendices. Where the atomic resource testing used multiple servers in separate applications, to enable testing with many ports per protocol and four protocols, this test only required one port per protocol and two protocols. As a result, the entire test is performed in a single application using six goroutines across four threads on four processors. These six goroutines were the main function, the two clients, two servers, and the data generation function, `SpawnData`.

```
package main

import (
    "bytes"
    "crypto/tls"
    "flag"
    "fmt"
    "io"
    "io/ioutil"
    "math/rand"
    "net/http"
    "net/wp"
    "os"
    "sync"
    "time"
)
```



```

func Random(out []byte) {
    // Random fills 'out' with pseudorandom data.
}

func SpawnData(a, b chan []byte, n int, t time.Duration) {

    // Wait for both servers to connect.

    for i := 0; i < n; i++ {
        // Introduce random wait between 0.001 and t seconds.
        time.Sleep(wait)

        // Generate random amount of random data up to 1 MB.
        data := make([]byte, size)
        Random(data)

        // Send data to both.
    }

    // Signal the end of data.
    close(a)
    close(b)
}

type HTTPServer struct {
    c chan []byte
    s chan struct{}
    o *sync.Once
    f func()
}

```

```

// ServeHTTP is called once per data burst, when the client sends
// its long poll.
func (s *HTTPServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Communicate the connection to SpawnData.

    // Receive data.
    data, ok := <-s.c
    if !ok {
        // Data generation has ended.
        w.WriteHeader(204)
        return
    }

    // Send data back to the client
    _, err := w.Write(data)
    if err != nil {
        // Handle the error.
    }
}

type WPServer struct {
    c chan []byte
    s chan struct{}
    o *sync.Once
    f func()
}

```

```

// ServeWP is called once when the connection is initiated.
func (s *WPServer) ServeWP(w wp.ResponseWriter, r *wp.Request) {
    // Communicate the connection to SpawnData.

    // Receive data.
    for {
        data, ok := <-s.c
        if !ok {
            // Data generation has ended, so the
            // connection ends.
            return
        }

        // Send data back to the client.
        _, err := w.Write(data)
        if err != nil {
            // Handle the error.
        }
    }
}

type Event struct {
    Received time.Time
    Size     int
}

type Results struct {
    Events  []*Event
    Protocol string
}

func (r *Results) Add(e *Event) {
    r.Events = append(r.Events, e)
}

```

```

func (r1 *Results) Compare(r2 *Results) string {
    // Create and initialise the results file.

    for i, ev1 := range r1.Events {
        // Compare the results and write them to the file.
    }
}

type HTTPClient struct {
    done    chan<- struct{}
    client  *http.Client
    Results *Results
}

func (c *HTTPClient) Run() {
    for {
        // Start the long polling.
        res, err := c.client.Get("http://localhost:1337/")
        if err != nil {
            // Handle the error.
        }

        // End when the 204 response is received.
        if res.StatusCode == 204 {
            break
        }

        // Store the amount of data received.
        _, err = io.Copy(counter, res.Body)
        if err != nil {
            // Handle the error.
        }

        // Store the results.
    }

    // Inform main that the HTTP client has finished.
    c.done <- struct{}{}
}

// The WP client uses the new client API, as
// discussed in the code in section 4.1.2.

```

```

type WPClient struct {
    done      chan<- struct{}
    counter   *Counter
    Results   *Results
}

// Init, Suggest, and Close do nothing, since we are
// merely storing the amount of data collected from
// one stream, so they are not shown here.

// Handle is called once per data burst.
func (c *WPClient) Handle(res *wp.Response, _ int) ([]string, error) {

    // Make sure the request was successful.
    if res.StatusCode != 0 {
        // Handle the error.
    }

    // Count the number of bytes received.
    _, err := io.Copy(c.counter, res.Body)
    if err != nil {
        // Handle the error.
    }

    // Store the results

    return nil, nil
}

func (c *WPClient) Run() {
    wp.FetchJust("wp://localhost:1338/", new(tls.Config), c)
}

// These command-line flags are used to set the test parameters.
var n = flag.Int("iterations", 10, "number of times to send data")
var t = flag.Duration("time", 2, "max time between data")

```

```

func main() {

    flag.Parse()

    // Initialise and start data creation.
    go SpawnData(httpc, wpc, starthttp, startwp, *n, *t)

    // Create the HTTP server.

    // Create the WP server.

    // Start both servers.
    go http.ListenAndServe(":1337", httpServer)
    go wp.ListenAndServe(":1338", "cert.pem", "key.pem", wpServer)

    // Create the HTTP client.

    // Create the WP client.

    // Start both clients.
    go httpClient.Run()
    go wpClient.Run()

    // Wait for both clients to finish.
    <-done
    <-done

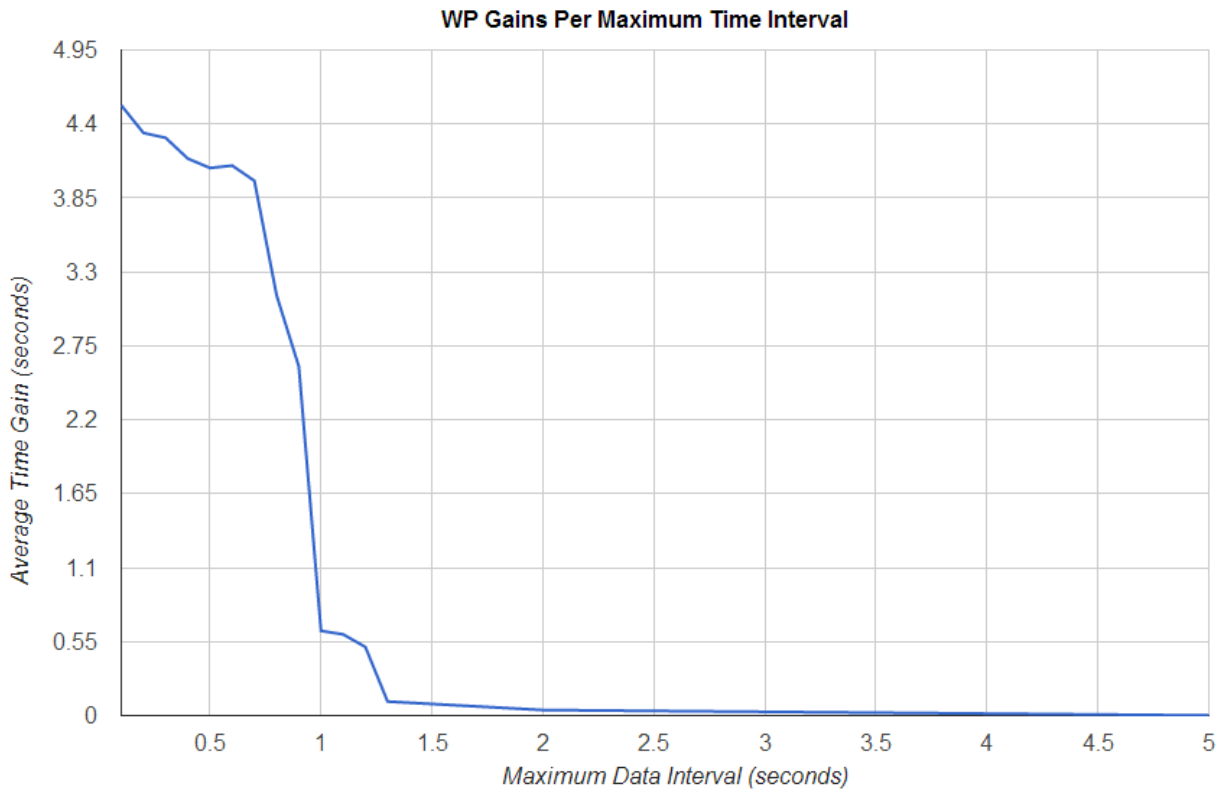
    // Print and write results.

}

```

4.2.3: Results

The full data set is provided in the appendix zip folder in a series of CSV files. For each test, the arrival time at the WP client was subtracted from the time of the HTTP client for each piece of data. This gave a time difference between the protocols, where faster responses by WP gave a positive difference, and faster responses by HTTP gave a negative difference. By averaging this delta, an overall estimation of the protocols' difference can be seen. With fluctuations in the network, and with the varying data payload, the deltas varied by a reasonable amount, but this was mitigated by taking the average over a large data set. For the final experiments, 500 iterations were used for each presentation interval.



As the data shows, once the maximum interval between data presentation reaches about 1.3 seconds, the performance of HTTP and WP become roughly equal. Beyond this point, the improvement provided by WP will likely be little more than the effect of the random interval, since the HTTP request to begin the next long poll will almost always have arrived before data becomes available. As the interval grows, the probability of data being presented before the request arrives drops off to negligible levels. With a network latency of 30 ms, data arriving less than 60 ms after the previous presentation will do so before the request arrives, since the next request will not be sent until the previous response has taken 30 ms to return to the client, and it will then take another 30 ms to reach the server.

When the maximum interval between data presentation is less than about one second, a very clear difference between the two protocols can be seen. While HTTP needs to execute a full request/response cycle for every set of data, WP can simply send the data as quickly as it is presented, allowing very quick response times. Not only does this give great performance, but it was also very easy to implement, with the WP server simply receiving the data, and writing it to the `ResponseWriter` in a loop. This simplicity means that if the server wished to do some additional processing on the data, it would be very clear how to make the additions, particularly if a consistent state needed to be kept, such as compression headers, since the entire process takes place in a single long-running function. By comparison, each iteration in HTTP takes place in a separate call to the function, so any required state must be stored elsewhere.

4.2.4: Analysis

Somewhat unsurprisingly, the driving factor in the difference in performance between the two protocols in this experiment, is the relationship between the average data presentation interval and the latency of the connection. Where HTTP requires at least one full RTT between data bursts, WP requires at most half that, since the TCP connection is likely to be fully-open (where TCP Slow-Start has concluded, and CWND has reached the advertised maximum segment size (MSS)). This will often allow multiple data bursts to be sent within a single ACK window, meaning that successive transmissions can be sent before the previous frame has reached the client. In cases like Twitter, and various other social media (particularly global searches within those media), the intention will be to send almost limitless data as quickly as possible. In situations like these, streaming protocols like WP provide a very clear advantage over transactional protocols like HTTP.

5: Future Work

5.1: Implementation Development

As with the net/http package, each TCP connection received by a WP server is handled in a separate goroutine; a concurrent processing unit far more lightweight than a thread. This allows a server to maintain tens, or even hundreds of thousands of simultaneous connections. One of the early goals of the WP package was to provide further concurrency, to run each separate stream in its own goroutine. This would enable long-running streaming responses to run concurrently with more conventional requests/responses. However, while using net/http as a starting point had allowed a quick development process for the initial implementation, it left substantial complexity and structure appropriate for HTTP, but excessive for WP. This complexity confounded my efforts to implement the inter-stream independence necessary for concurrent handling of each stream, as discussed in section 3.5.2.

To rectify this issue, a substantial rewrite of the library is required. This would involve restructuring the connection object into two separate objects; a single connection construct to manage the coordination of streams and writing to the network, and an additional structure for each stream, which keeps track of the details of that stream and defers to the connection for other tasks. This would allow each stream to run independently and to leave the race-prone managing of the interrelation between streams to the connection object.

Although the library is sufficiently complete for testing and evaluation, further work is required to add final details like caching support, content-type detection, and proper handling of Data Requests with the 'Upload' flag set. None of this is particularly lengthy or challenging work, but it too depends upon the rewrite above, since upload requests and server pushes do not fit neatly into the current structure of single-requests/single-responses created to control HTTP.

The library is publicly available under a revised BSD license at <https://github.com/SlyMarbo/wp>, although further documentation and refinement are scheduled in the coming months to encourage use and development. One particular planned improvement discovered in the development of my SPDY library (discussed in §4.1.4.1) is the use of Google's Next Protocol Negotiation (NPN) extension to TLS, which very simply allows a single application to serve both HTTPS and other protocols like SPDY and WP, with clients announcing other supported protocols, and the server defaulting to HTTPS if no other protocols are declared. This allows the use of advanced protocols with those clients that support them, without abandoning those that do not.

5.2: Adoption Requirements

The first factor in enabling widespread adoption of the protocol is to add the Next Protocol Negotiation TLS extension to the implementation, so that clients which support WP may use it, but those that do not are served by other protocols, such as SPDY or HTTPS. By adding this into the WP package directly, it becomes more useful without requiring additional work by the server-writer. If an efficient SPDY library was added to the Go standard library, then the three protocols (SPDY, HTTPS, WP) could add a mechanism to their `init()` functions to allow registering of the protocols to affect a protocol-agnostic server infrastructure, as is the case with the `Image` package. With this package, the software developer can use a 'null import' of various

image format packages (such as gif, png, and jpeg) to add their decoding/encoding capabilities to the base Image package, so that the Image.Decode function can decode a wide range of image types. A similar infrastructure for web protocols would simplify the server-writer's job substantially, while keeping the implementation of each protocol separate and leaving protocol choice up to the user.

The other main effort required for adoption is the adding of WP to at least one popular web browser. This was outlined in the initial plan as a deliverable for the project, but it quickly became clear that such an effort would be far beyond the scale of a final year project, given the enormous complexity of modern browsers' network engines. Although this was not possible in this project, it is an absolute requirement for the success of the protocol. With SPDY supported in Google Chrome/Chromium, Mozilla Firefox, and Opera, and support rumoured to be coming in Microsoft Internet Explorer 11, SPDY is still only used by a very small proportion of web servers. According to W3Techs, SPDY is supported by roughly 1% of web servers^[20]. WP would require equal support in web browsers, and further support in implementation support to reach the same level of adoption in servers.

6: Conclusions

Both the theory and the experimentation clearly show that multiplexing requests onto a single TCP session provides a substantial performance benefit, with little extra complexity required. The use of streaming data transmission makes packet interleaving more effective, and makes better use of limited-bandwidth connections. With the HTTPbis working group developing HTTP 2.0, using SPDY v2 as the starting point^[7], this kind of protocol will probably underpin the web of the coming years.

There are two main obstacles to the adoption of HTTP 2.0, SPDY, and WP over HTTP 1.1 and 1.0. The first is protocol support in web servers. With SPDY already supported in Apache httpd through `mod_spdy`^[21], and Nginx with the `spdy` patch^[22], it is quickly gaining traction in the packaged web server world. Although less ground has been made in programming language web frameworks, like Django, support in Node.js^[23] and Jetty (Java)^[24] have increased server support substantially. Bringing support to Ruby on Rails, Microsoft's IIS, Go, and Django (Python) would result in a large majority of web servers supporting SPDY, once updated to the latest version.

Implementing client support is the second big task, and arguably the more difficult. While web servers are generally managed by the technologically-adept, web browsers have a long history of problems with updates. While reliable statistics are difficult to find, some data suggests that as much as 6.8% of internet users are browsing with Internet Explorer 6^[25], which was released in 2001. Although some browsers now provide an automatic update system, this doesn't solve the problem caused by older versions, with the data suggesting that only 32.85% of users have the most recent version of their chosen browser^[27]. This means that even bringing support for new protocols (like HTTP 2.0 and WP) to all modern browsers (no small task in itself) would only actually reach roughly a third of web users.

This is a well-known problem for development of the web platform, since new technologies like HTML5 and CSS3 have struggled for widespread adoption when the majority of web users' browsers do not support these new features. While the issue is best-known for its effect on the client-side features provided to the web developer, it has a similar effect on high-performance protocols like WP and SPDY. However, while new additions to the web platform require the developer to modify their application or website to gain their advantages, making use of a new protocol is generally easier. With the use of packages like `net/wp` and `mod_spdy`, the benefits of the protocol can be gained with minimal effort, even if the full advantages made available aren't harnessed.

7: Reflection

This project has taught me a great deal about computer networks; particularly web browser networking. There is an enormous difference between knowing the theory of network protocols and gaining the practical experience of designing and implementing them. Creating WP has bridged that gap for me and has also enabled me to solidify my experience with the Go language. Furthermore, as discussed in the analysis of SPDY's test performance (§4.1.4.1), the process of developing net/wp has given me a far better understanding of protocol implementation structure and the execution styles of different protocols. My own SPDY library has already made use of this understanding by using the sort of structure I envisage for the rewrite of net/wp, which has resulted in a far better implementation of SPDY, with considerably reduced coupling.

This sort of broad experience in protocol implementation is very valuable and has improved my networking skills enormously. It has also taught me that before implementing a network protocol, great care must be taken to identify the fundamental structure of the protocol and to design the implementation structure accordingly. Had I done this at the beginning of the project, I would probably have designed net/wp more effectively, as I have done in the SPDY library I have since started.

In addition to this practical experience, the project has given me the opportunity to improve my efficiency and time management skills. Whereas university assignments are designed to be achievable, research is not always so simple. Often a research project turns out not to be possible, or not to provide definitive results. In research it is important to be able to accept that something may not be feasible within a sensible time frame, and to move on to other work before spending too much time on an unproductive task. Though many students may come across this for the first time in the workplace, this project gave me the opportunity to experience a true research project in a controlled environment, and when progress was declining in the development of stream-level concurrency, I was able to recognise the situation and move on to important testing and evaluation, instead of wasting time trying to add one feature.

Table of Abbreviations

Note: Many terms used in this report are written in block caps, but are not abbreviations, such as SPDY, PING, and C. Accordingly, any terms found in the report which are not listed here should not be considered abbreviations. Furthermore, some units, such as megabyte and gigabyte use full caps to differentiate from other units (like the prefix milli- and the unit bit) which use the lowercase form of the same letter.

ACK	Acknowledgement, used in TCP	MAC	Message Authentication Code
AJAX	Asynchronous JavaScript And XML	MitM	Man in the Middle attack
API	Application Programming Interface	NPN	Next Protocol Negotiation
BSD	Berkeley Software Distribution	RTP	Real-time Transport Protocol
CSS	Cascading Style Sheets	RTT	Round-Trip Time
CSV	Comma Separated Values	SSL	Secure Socket Layer
CWND	Congestion Window	TCP	Transmission Control Protocol
FIFO	First-In, First-Out	TLS	Transport Layer Security
HSTS	HTTP Strict Transport Security	UDP	User Datagram Protocol
HTML	HyperText Markup Language	URI	Uniform Resource Identifier
HTTP	HyperText Transfer Protocol	URL	Uniform Resource Locator
HTTPS	HyperText Transfer Protocol Secure	UTF-8	UCS Transformation Format - 8-bit
IIS	Internet Information Services	WP	Web Protocol
I/O	Input/Output	XML	Extensible Markup Language
LTE	Long Term Evolution		

References

- 1 Grigorik, I. 2012. *SPDY, err... HTTP 2.0* [Online]
Available at: <http://www.igvita.com/slides/2012/http2-spdy-devconf.pdf>
Referencing slide 12: "Mobile, oh Mobile".
[Accessed: 03/05/2013]
- 2 HTTP Archive. *Archive of statistics and trends on popular websites* [Online]
Available at: <http://httparchive.org/>
[Accessed: 03/05/2013]
- 3 Browserscope.org. *A community-driven project for profiling web browsers* [Online]
Available at: <http://www.browserscope.org/?category=network>
[Accessed: 03/05/2013]
- 4 InfoQ. March 2010. *How HTML5 Web Sockets Interact With Proxy Servers* [Online]
Available at: <http://www.infoq.com/articles/Web-Sockets-Proxy-Servers>
[Accessed: 03/05/2013]
- 5 Network Working Group. June 1999. *HyperText Transfer Protocol - HTTP/1.1* [Online]
Available at: <http://tools.ietf.org/html/rfc2616>
Referencing section 8.1.4, which states:
"A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy."
[Accessed: 03/05/2013]
- 6 Langley, A. July 2012. *SSL Stripping Attacks* [Online]
Available at: <http://www.browserscope.org/?category=network>
[Accessed: 03/05/2013]
- 7 HTTPbis working group. June 2013. *SPDY Protocol* [Online]
Available at: <http://tools.ietf.org/html/draft-ietf-httpbis-http2-00>
[Accessed: 03/05/2013]
- 8 Pike, R. et al. *The Go Programming Language* [Online]
Available at: <http://golang.org/>
[Accessed: 03/05/2013]
- 9 Internet Engineering Task Force. November 2012. *HTTP Strict Transport Security* [Online]
Available at: <http://tools.ietf.org/html/rfc6797>
[Accessed: 03/05/2013]
- 10 Grigorik, I. November 2012. *Web Performance Crash Course* [Online]
Available at: <http://www.igvita.com/slides/2012/webperf-crash-course.pdf>
[Accessed: 03/05/2013]
- 11 Virgin Mobile USA. *Sprint Terms of Service: 4G* [Online]
Available at: <http://www.virginmobileusa.com/networkmanagement>
[Accessed: 03/05/2013]
- 12 Virgin Mobile USA. *Sprint Terms of Service: 3G* [Online]
Available at: <http://www.virginmobileusa.com/networkmanagement>
[Accessed: 03/05/2013]

- 13 Herrera, R. November 2011. *Add latency to localhost* [Online]
Available at: <https://gist.github.com/doctyper/1411283>
[Accessed: 03/05/2013]
- 14 Wireshark Foundation. *Wireshark* [Online]
Available at: <http://www.wireshark.org/>
[Accessed: 03/05/2013]
- 15 McKaskill, J. October 2011. *gospdy* [Online]
Available at: <https://github.com/jmckaskill/gospdy>
[Accessed: 03/05/2013]
- 16 Hykes, S. December 2012. *spdy-go* [Online]
Available at: <https://github.com/shykes/spdy-go>
[Accessed: 03/05/2013]
- 17 Hall, J. April 2013. *spdy* [Online]
Will be available at: <https://github.com/SlyMarbo/spdy>
[Accessible: 07/2013]
- 18 Grigorik, I. *High Performance Browser Networking*
Currently in review for publishing.
Available online for proofreading at: <http://chimera.labs.oreilly.com/books/1230000000545/>
Referenced chapter 2, section Slow-Start.
[Accessed: 03/05/2013]
- 19 *IconArchive* [Online]
Available at: <http://www.iconarchive.com/>
Referenced images:
<http://icons.iconarchive.com/icons/oxygen-icons.org/oxygen/128/Devices-computer-laptop-icon.png>
<http://icons.iconarchive.com/icons/rimshotdesign/nod2/128/Hard-Disk-Server-icon.png>
<http://icons.iconarchive.com/icons/oxygen-icons.org/oxygen/128/Places-network-server-database-icon.png>
[Accessed: 03/05/2013]
- 20 W3Techs. *Usage of SPDY for websites* [Online]
Available at: <http://w3techs.com/technologies/details/ce-spdy/all/all>
[Accessed: 03/05/2013]
- 21 *mod_spdy* [Online]
Available at: <https://code.google.com/p/mod-spdy/>
[Accessed: 03/05/2013]
- 22 Bartenev, V. June 2012. *Announcing SPDY draft 2 implementation in nginx* [Online]
Available at: <http://mailman.nginx.org/pipermail/nginx-devel/2012-June/002343.html>
[Accessed: 03/05/2013]
- 23 Indutny, F. March 2013. *node-spdy* [Online]
Available at: <https://github.com/indutny/node-spdy>
[Accessed: 03/05/2013]
- 24 Eclipse Foundation. March 2012. *Jetty/Feature/SPDY* [Online]
Available at: <http://wiki.eclipse.org/Jetty/Feature/SPDY>
[Accessed: 03/05/2013]
- 25 NetMarketShare. *Browser market share* [Online]
Available at: <http://marketshare.hitslink.com/browser-market-share.aspx>
[Accessed: 03/05/2013]

Appendices

Appendix 1: WP Specification.pdf

- The definitive specification for WP.

Appendix 2: c1031815.zip

- Zip folder containing:
 - README.txt, which explains the contents further.
 - Atomic test (test 1)
 - Streaming test (test 2)
 - Go libraries