SPDY: An experimental protocol for a faster web

Executive summary

As part of the "Let's make the web faster" initiative, we are experimenting with alternative protocols to help reduce the latency of web pages. One of these experiments is SPDY (pronounced "SPeeDY"), an application-layer protocol for transporting content over the web, designed specifically for minimal latency. In addition to a specification of the protocol, we have developed a SPDY-enabled Google Chrome browser and open-source web server. In lab tests, we have compared the performance of these applications over HTTP and SPDY, and have observed up to 64% reductions in page load times in SPDY. We hope to engage the open source community to contribute ideas, feedback, code, and test results, to make SPDY the next-generation application protocol for a faster web.

Background: web protocols and web latency

Today, HTTP and TCP are the protocols of the web. TCP is the generic, reliable transport protocol, providing guaranteed delivery, duplicate suppression, in-order delivery, flow control, congestion avoidance and other transport features. HTTP is the application level protocol providing basic request/response semantics. While we believe that there may be opportunities to improve latency at the transport layer, our initial investigations have focussed on the application layer, HTTP.

Unfortunately, HTTP was not particularly designed for latency. Furthermore, the web pages transmitted today are significantly different from web pages 10 years ago and demand improvements to HTTP that could not have been anticipated when HTTP was developed. The following are some of the features of HTTP that inhibit optimal performance:

- Single request per connection. Because HTTP can only fetch one resource at a time (HTTP pipelining helps, but still enforces only a FIFO queue), a server delay of 500 ms prevents reuse of the TCP channel for additional requests. Browsers work around this problem by using multiple connections. Since 2008, most browsers have finally moved from 2 connections per domain to 6.
- Exclusively client-initiated requests. In HTTP, only the client can initiate a request. Even if the server knows the client needs a resource, it has no mechanism to inform the client and must instead wait to receive a request for the resource from the client.
- Uncompressed request and response headers. Request headers today vary in size from ~200 bytes to over 2KB. As applications use more cookies and user agents expand features, typical header sizes of 700-800 bytes is common. For modems or ADSL connections, in which the uplink bandwidth is fairly low, this latency can be significant. Reducing the data in headers could directly improve the

serialization latency to send requests.
- Redundant headers. In addition, several headers are repeatedly sent across requests on the same channel. However, headers such as the User-Agent, Host, and Accept* are generally static and do not need to be resent.
- Optional data compression. HTTP uses optional compression encodings for data. Content should always be sent in a compressed format.

Previous approaches

SPDY is not the only research to make HTTP faster. There have been other proposed solutions to web latency, mostly at the level of the transport or session layer:

- Stream Control Transmission Protocol (SCTP) -- a transport-layer protocol to replace TCP, which provides multiplexed streams and stream-aware congestion control.
- HTTP over SCTP -- a proposal for running HTTP over SCTP. Comparison of HTTP Over SCTP and TCP in High Delay Networks describes a research study comparing the performance over both transport protocols.
- Structured Stream Transport (SST) -- a protocol which invents "structured streams": lightweight, independent streams to be carried over a common transport. It replaces TCP or runs on top of UDP.
- MUX and SMUX -- intermediate-layer protocols (in between the transport and application layers) that provide multiplexing of streams. They were proposed years ago at the same time as HTTP/1.1.

These proposals offer solutions to some of the web's latency problems, but not all. The problems inherent in HTTP (compression, prioritization, etc.) should still be fixed, regardless of the underlying transport protocol. In any case, in practical terms, changing the transport is very difficult to deploy. Instead, we believe that there is much low-hanging fruit to be gotten by addressing the shortcomings at the application layer. Such an approach requires minimal changes to existing infrastructure, and (we think) can yield significant performance gains.

Goals for SPDY

The SPDY project defines and implements an application-layer protocol for the web which greatly reduces latency. The high-level goals for SPDY are:

- To target a 50% reduction in page load time. Our preliminary results have come close to this target (see below).
- To minimize deployment complexity. SPDY uses TCP as the underlying transport layer, so requires no changes to existing networking infrastructure.
- To avoid the need for any changes to content by website authors. The only changes required to support SPDY are in the client user agent and web server applications.
- To bring together like-minded parties interested in exploring protocols as a way of solving the latency problem. We hope to develop this new protocol in partnership with the open-source community and industry specialists.

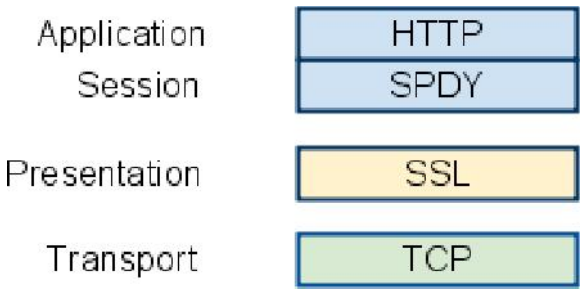
Some specific technical goals are:

- To allow many concurrent HTTP requests to run across a single TCP session.
- To reduce the bandwidth currently used by HTTP by compressing headers and eliminating unnecessary headers.
- To define a protocol that is easy to implement and server-efficient. We hope to reduce the complexity of HTTP by cutting down on edge cases and defining easily parsed message formats.
- To make SSL the underlying transport protocol, for better security and compatibility with existing network infrastructure. Although SSL does introduce a latency penalty, we believe that the long-term future of the web depends on a secure network connection. In addition, the use of SSL is necessary to ensure that communication across existing proxies is not broken.
- To enable the server to initiate communications with the client and push data to the client whenever possible.

SPDY design and features

SPDY adds a session layer atop of SSL that allows for multiple concurrent, interleaved streams over a single TCP connection.

The usual HTTP GET and POST message formats remain the same; however, SPDY specifies a new framing format for encoding and transmitting the data over the wire.



Streams are bi-directional, i.e. can be initiated by the client and server.

SPDY aims to achieve lower latency through basic (always enabled) and advanced (optionally enabled) features.

Basic features

- ***Multiplexed streams***

  SPDY allows for unlimited concurrent streams over a single TCP connection. Because requests are interleaved on a single channel, the efficiency of TCP is much higher: fewer network connections need to be made, and fewer, but more densely packed, packets are issued.

- ***Request prioritization***

  Although unlimited parallel streams solve the serialization problem, they introduce another one: if bandwidth on the channel is constrained, the client may block requests for fear of clogging the channel. To overcome this problem, SPDY implements request

priorities: the client can request as many items as it wants from the server, and assign a priority to each request. This prevents the network channel from being congested with non-critical resources when a high priority request is pending.

- ***HTTP header compression***

  SPDY compresses request and response HTTP headers, resulting in fewer packets and fewer bytes transmitted.

Advanced features

In addition, SPDY provides an advanced feature, *server-initiated streams*. Server-initiated streams can be used to deliver content to the client without the client needing to ask for it. This option is configurable by the web developer in two ways:

- ***Server push*.**

  SPDY experiments with an option for servers to push data to clients via the X-Associated-Content header. This header informs the client that the server is pushing a resource to the client before the client has asked for it. For initial-page downloads (e.g. the first time a user visits a site), this can vastly enhance the user experience.

- ***Server hint***.

  Rather than automatically pushing resources to the client, the server uses the X-Subresources header to *suggest* to the client that it should ask for specific resources, in cases where the server knows in advance of the client that those resources will be needed. However, the server will still wait for the client request before sending the content. Over slow links, this option can reduce the time it takes for a client to discover it needs a resource by hundreds of milliseconds, and may be better for non-initial page loads.

For technical details, see the SPDY draft protocol specification.

SPDY implementation: what we've built

This is what we have built:

- A high-speed, in-memory server which can serve both HTTP and SPDY responses efficiently, over TCP and SSL. We will be releasing this code as open source in the near future.
- A modified Google Chrome client which can use HTTP or SPDY, over TCP and SSL. The source code is at http://src.chromium.org/viewvc/chrome/trunk/src/net/spdy/. (Note that code currently uses the internal code name of "flip"; this will change in the near future.)
- A testing and benchmarking infrastructure that verifies pages are replicated with high fidelity. In particular, we ensure that SPDY preserves origin server headers, content encodings, URLs, etc. We will be releasing our testing tools, and instructions for reproducing our results, in the near future.

Preliminary results

With the prototype Google Chrome client and web server that we developed, we ran a number of lab tests to benchmark SPDY performance against that of HTTP.

We downloaded 25 of the "top 100" websites over simulated home network connections, with 1% packet loss. We ran the downloads 10 times for each site, and calculated the average page load time for each site, and across all sites. The results show a speedup over HTTP of 27% - 60% in page load time over plain TCP (without SSL), and 39% - 55% over SSL.

Table 1: Average page load times for top 25 websites

	DSL 2 Mbps downlink, 375 kbps uplink	
	Average ms	**Speedup**
HTTP	3111.916	
SPDY basic multi-domain* connection / TCP	2242.756	27.93%
SPDY basic single-domain* connection / TCP	1695.72	45.51%
SPDY single-domain + server push / TCP	1671.28	46.29%
SPDY single-domain + server hint / TCP	1608.928	48.30%
SPDY basic single-domain / SSL	1899.744	38.95%
SPDY single-domain + client prefetch / SSL	1781.864	42.74%

* In many cases, SPDY can stream all requests over a single connection, regardless of the number of different domains from which requested resources originate. This allows for full parallelization of all downloads. However, in some cases, it is not possible to collapse all domains into a single domain. In this case, SPDY must still open a connection for each domain, incurring some initial RTT overhead for each new connection setup. We ran the tests in both modes: collapsing all domains into a single domain (i.e. one TCP connection); and respecting the actual partitioning of the resources according to the original multiple domains (= one TCP connection per domain). We include the results for both the strict "single-domain" and "multi-domain" tests; we expect real-world results to lie somewhere in the middle.

The role of header compression

Header compression resulted in an ~88% reduction in the size of request headers and an ~85% reduction in the size of response headers. On the lower-bandwidth DSL link, in which the upload link is only 375 Kbps, request header compression in particular, led to significant page load time improvements for certain sites (i.e. those that issued large number of resource requests). We found a reduction of 45 - 1142 ms in page load

time simply due to header compression.

The role of packet loss and round-trip time (RTT)

We did a second test run to determine if packet loss rates and round-trip times (RTTs) had an effect on the results. For these tests, we measured only the cable link, but simulated variances in packet loss and RTT.

We discovered that SPDY's latency savings increased proportionally with increases in packet loss rates, up to a 48% speedup at 2%. (The increases tapered off above the 2% loss rate, and completely disappeared above 2.5%. In the real world, packets loss rates are typically 1-2%, and RTTs average 50-100 ms in the U.S.) The reasons that SPDY does better as packet loss rates increase are several:

- SPDY sends ~40% fewer packets than HTTP, which means fewer packets affected by loss.
- SPDY uses fewer TCP connections, which means fewer chances to lose the SYN packet. In many TCP implementations, this delay is disproportionately expensive (up to 3 s).
- SPDY's more efficient use of TCP usually triggers TCP's fast retransmit instead of using retransmit timers.

We discovered that SPDY's latency savings also increased proportionally with increases in RTTs, up to a 27% speedup at 200 ms. The The reason that SPDY does better as RTT goes up is because SPDY fetches all requests in parallel. If an HTTP client has 4 connections per domain, and 20 resources to fetch, it would take roughly 5 RTs to fetch all 20 items. SPDY fetches all 20 resources in one RT.

Table 2: Average page load times for top 25 websites by packet loss rate

	Average ms		Speedup
Packet loss rate	HTTP	SPDY basic (TCP)	
0%	1152	1016	11.81%
0.5%	1638	1105	32.54%
1%	2060	1200	41.75%
1.5%	2372	1394	41.23%
2%	2904	1537	47.7%
2.5%	3028	1707	43.63%

Table 3: Average page load times for top 25 websites by RTT

	Average ms		**Speedup**
RTT in ms	**HTTP**	**SPDY basic (TCP)**	
20	1240	1087	12.34%
40	1571	1279	18.59%
60	1909	1526	20.06%
80	2268	1727	23.85%
120	2927	2240	23.47%
160	3650	2772	24.05%

200	4498	3293	26.79%

SPDY next steps: how you can help

Our initial results are promising, but we don't know how well they represent the real world. In addition, there are still areas in which SPDY could improve. In particular:

- Bandwidth efficiency is still low. Although dialup bandwidth efficiency rate is close to 90%, for high-speed connections efficiency is only about ~32%.
- SSL poses other latency and deployment challenges. Among these are: the additional RTTs for the SSL handshake; encryption; difficulty of caching for some proxies. We need to do more SSL tuning.
- Our packet loss results are not conclusive. Although much research on packet-loss has been done, we don't have enough data to build a realistic model model for packet loss on the Web. We need to gather this data to be able to provide more accurate packet loss simulations.
- SPDY single connection loss recovery sometimes underperforms multiple connections. That is, opening multiple connections is still faster than losing a single connection when the RTT is very high. We need to figure out when it is appropriate for the SPDY client to make a new connection or close an old connection and what effect this may have on servers.
- The server can implement more intelligence than we have built in so far. We need more research in the areas of server-initiated streams, obtaining client network information for prefetching suggestions, and so on.

To help with these challenges, we encourage you to get involved:

- Send feedback, comments, suggestions, ideas to the chromium-discuss discussion group.
- Download, build, run, and test the Google Chrome client code.
- Contribute improvements to the code base.

SPDY frequently asked questions

Q: Doesn't HTTP pipelining already solve the latency problem?

A: No. While pipelining does allow for multiple requests to be sent in parallel over a single TCP stream, it is still but a single stream. Any delays in the processing of anything in the stream (either a long request at the head-of-line or packet loss) will delay the entire stream. Pipelining has proven difficult to deploy, and because of this remains disabled by default in all of the major browsers.

Q: Is SPDY a replacement for HTTP?

A: No. SPDY replaces some parts of HTTP, but mostly augments it. At the highest level of the application layer, the request-response protocol remains the same. SPDY still uses HTTP methods, headers, and other semantics. But SPDY overrides other parts of the protocol, such as connection management and data transfer formats.

Q: Why did you choose this name?

A: We wanted a name that captures speed. SPDY, pronounced "SPeeDY", captures this and also shows how compression can help improve speed.

Q: Should SPDY change the transport layer?

A: More research should be done to determine if an alternate transport could reduce latency. However, replacing the transport is a complicated endeavor, and if we can overcome the inefficiencies of TCP and HTTP at the application layer, it is simpler to deploy.

Q: TCP has been time-tested to avoid congestion and network collapse. Will SPDY break the Internet?

A: No. SPDY runs atop TCP, and benefits from all of TCP's congestion control algorithms. Further, HTTP has already changed the way congestion control works on the Internet. For example, HTTP clients today open up to 6 concurrent connections to a single server; at the same time, some HTTP servers have increased the initial congestion window to 4 packets. Because TCP independently throttles each connection, servers are effectively sending up to 24 packets in an initial burst. The multiple connections side-step TCP's slow-start. SPDY, by contrast, implements multiple streams over a single connection.

Q: What about SCTP?

A: SCTP is an interesting potential alternate transport, which offers multiple streams over a single connection. However, again, it requires changing the transport stack, which will make it very difficult to deploy across existing home routers. Also, SCTP alone isn't the silver bullet; application-layer changes still need to be made to efficiently use the channel between the server and client.

Q: What about BEEP?

A: While BEEP is an interesting protocol which offers a similar grab-bag of features, it doesn't focus on reducing the page load time. It is missing a few features that make this possible. Additionally, it uses text-based framing for parts of the protocol instead of binary framing. This is wonderful for a protocol which strives to be as extensible as possible, but offers some interesting security problems as it is more difficult to parse correctly.

Comments