SECURITY NOW!

Transcript of Episode #343

HTTP & SPDY

Description: This week, after catching up with the week's security and privacy news, Steve and Leo take a detailed look at the World Wide Web's current HTTP protocol and examine the significant work that's been done by the Chromium Project on "SPDY," a next-generation web protocol for dramatically decreasing page load times and latency and improving performance and interactivity.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-343.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-343-lq.mp3

SHOW TEASE: Time for Security Now!. Steve Gibson joins us for security news, of course, and updates. And then we'll talk about a new initiative from the Chromium project to speed up the Internet by, like, a factor of three: SPDY. Next.

Leo Laporte: This is Security Now! with Steve Gibson, Episode 343, recorded Tuesday, March 6, 2012: HTTP & SPDY.

It's time for Security Now!, the show that protects your security and privacy online. And boy, I've learned a lesson: That's something people care about. Steve Gibson is here - the man, the myth, the legend. Oh, no, I'm supposed to call you the Explainer in Chief. And you are going to explain today.

Steve Gibson: Ah, there we go.

Leo: You're going to explain spidey sense.

Steve: Oh, yes, it's something that, the more I've looked at it, the more I hope this thing gets traction because it's a project that the Chromium group - actually one main guy within the Chromium group - have been working on to figure out a way to improve the user's experience surfing the 'Net. And I'll give you a benchmark which I also cover at the end of this explanation: They've got it running so that the average of the top 25 websites load in one-third the time. So, I mean, that's a huge...

Leo: Wow. That's not the websites making a change. That's the browser.

Steve: Well, it's looking at the inefficiencies which no one really worried about 10 years ago. Today's Internet is not our grandfathers' Internet.

Leo: No, no, no, no.

Steve: And pages are much more sophisticated, much more complex. They're pulling stuff from all over the place. It's not just a page of text. And so what Google has done is they've carefully looked at how time is spent. There have been other efforts, also. What we'll talk about is a solution to a problem that's been understood for at least a decade. There is an RFC that was written in 2002, so 10 years ago, talking about a next-generation sort of multiplex streaming protocol to replace TCP. The problem is we can't replace TCP. It's just too ubiquitous. All of our little NAT routers would break in our homes, to say nothing of the Internet's routers.

So what Google has done is they've figured out how to stick a shim in between TCP and HTTP so that nothing really has to change, but if you have a SPDY - SPDY is sort of their acronym, like HTTP, this is SPDY. If you have a server that is able to serve HTTP for non-SPDY-aware browsers, but also able to support the SPDY enhancement, then if you're using a browser that also understands this next-generation protocol, your pages come up in one - your pages finish loading in one-third the time.

Leo: Wow.

Steve: I mean, that's a huge, huge improvement. And so we're going to talk about what HTTP does, what it doesn't do right, look briefly at what's come before, and then plow into how to make it faster, what the Chromium guys have done. And they've built a server. They've got a server running; they've got a version of Chrome running. All of this works. And I just hope it gets some traction because it would be great to have. And at no cost. Basically it much better utilizes Internet connections than we are doing now. And we'll talk about how.

Leo: Neat. Yeah.

Steve: So not a ton of news. I did want to just mention something that was in the news, I think it was sort of toward the end of maybe middle or late last week about this Bodog.com site that got - it's a Canadian-based gambling site....

Leo: Oh, yeah, yeah, yeah.

Steve: ...that is based in Vancouver. Their domain, Bodog.com, is registered with a registrar, Canadian registrar DomainClip. And there's no strong affiliation of any sort with the U.S. It's not in the U.S. The registrar's not in the U.S. But despite that fact, the state of Maryland, prosecutors in the state of Maryland were able to obtain a warrant ordering

VeriSign - which is a U.S.-based company, we understand, VeriSign, which manages the dot-com domain name registry, that is, it runs the dot-com top-level domain, and so the other registrars feed their stuff through VeriSign to the master dot-com server records - the state of Maryland ordered VeriSign to redirect the Bodog.com website to a warning page advising that it has been seized by the U.S. Department of Homeland Security.

And so there was a lot of coverage about this. One blogger, Michael Geist wrote, he said, "The message from the case is clear: all dot-com, dot-net, and dot-org domain names are subject to U.S. jurisdiction regardless of where they operate or where they were registered. This grants the U.S. a form of 'super-jurisdiction' over Internet activities since most other countries are limited to jurisdiction with a real and substantial connection. For the U.S., the location of the domain name registry is good enough." And dot-com, dot-net, and dot-org are located here in the U.S.

So he goes on and says, "The aggressive assertion of Internet jurisdiction was one of the key concerns with the Stop Online Privacy Act (SOPA), the [controversial] bill that died following a massive online protest in January. It simply defined any domain name with a registrar or registry in the U.S. as 'domestic' for U.S. law purposes. The Bodog.com case suggests that the provision was not changing the law as much as restating it, since U.S. prosecutors and courts follow much the same approach."

And so he finishes, saying, "In an era when governments are becoming increasingly active in regulating online activities, the Bodog.com case provides a warning that by using popular dot-com domain names, companies and registrants are effectively opting-in to U.S. law and courts as part of the package." And I did see, in reaction to this, a number of people talking about, well, that means we just can't - we can't be under dot-com, dot-org, and dot-net because it's subject to being confiscated.

Leo: Well, and that was the whole point of SOPA was to add these other domains to the ICE takedown capability; right?

Steve: Right, right. It was foreign domains.

Leo: It wasn't for dot-coms, yeah, yeah.

Steve: Correct, correct, correct. Well, and the other big news, I mean, this is hot news from this morning, it turns out that a guy named Hector Xavier Monsegur, who lived in New York's Lower East Side in a housing project, was the head of LulzSec.

Leo: What?

Steve: Yes.

Leo: They caught him?

Steve: Yes.

Leo: Oh, boy.

Steve: They actually caught him last June, and they kept it secret.

Leo: Whoa. No wonder it's been so quiet lately. Anonymous has done everything.

Steve: He pleaded guilty - uh-huh. He pleaded guilty to the charges, 12 hacking-related charges on August 15th of last year, then turned state's evidence.

Leo: Oh, boy.

Steve: And he turned in the other five top-ranking members of LulzSec, who were arrested en masse in a synchronized raid across the globe this morning. Ryan Ackroyd, aka "Kayla," was one of them; Jake Davis, aka "Topiary," both in London. Darren Martyn, whose handle is "pwnsauce," and Donncha O'Cearrbhail, whose handle is "palladium," both in Ireland. And, finally, Jeremy Hammond, who went by "Anarchaos," in Chicago. And Ryan Ackroyd, who I mentioned, is believed to be the No. 2 guy, that is, Monsegur's top deputy. And Jeremy Hammond, the guy in Chicago, is believed to be - and there's a separate indictment for him because he's the person believed to be behind the WikiLeaks email breach, who hacked the U.S. security company, Stratfor, got all of those emails, and sent them off to Assange and company. So this was a big deal this morning.

Leo: Wow.

Steve: It turns out Hector, who they caught in June and who pleaded guilty two months later in August, he goes by Sabu, Xavier DeLeon, and just Leon. He's an unemployed 28-year-old father of two, living in a public housing project in New York's Lower East Side. And he became a "cooperating witness," as they termed it, in June. I put up a - oh, no I didn't. It's for a different story. Gizmodo has - it's just Gizmodo.com/5890886. So that's pretty easy to get in: Gizmodo.com/5890886. On that page is all the legal documentation, all of the collective and individual indictments which are hosted over on Scribd, but you're able to download them.

Leo: Good-looking guy. Oh, boy.

Steve: Yeah, so that's LulzSec. And they have, of course, ties to Anonymous. There was some over-reporting done, talking about, oh, well, this ends all of Anonymous. It's like, well, no. This is six guys who were at the top of LulzSec, and certainly very active in hacking. But this doesn't end the careers of Anonymous.

Leo: I have heard, and I get the sense that the truth about these groups, though, are there are a few, a small number of elite or talented hackers, and the rest are script kiddies or people running the low earth...

Steve: Ion cannon.

Leo: ...orbit cannon, yeah.

Steve: Yeah. And in fact, even in reading some of the stories and looking at some of these depositions, or rather indictments, they describe the roles these people played. And in many cases they say, well, this guy was a kernel hacker who found the exploits and then turned them over to somebody else.

Leo: Yeah, yeah. That makes sense. These people are fairly skilled. Fairly? Significantly skilled, I think.

Steve: Yeah.

Leo: Wow. So it's interesting, though, that he turned state's evidence because there's no honor among hackers.

Steve: Yeah, I was going to say...

Leo: And they kept this secret for so long. That net must have extended to a pretty broad number of people. Are we going to see more indictments? I would guess we will.

Steve: Well, those five people arrested this morning, following up on Hector's arrest back in June. So now you can imagine they will be squeezed.

Leo: There'll be more, yeah.

Steve: And, yeah. Wow. So I ran across an interesting page that I thought our listeners would find interesting. Under the topic of Privacy Watch, this is a page you can go to. I created for this one a Security Now! bit.ly shortcut, that is to say, this is Security Now! Episode 343. So if you put in bit.ly/SN343, that will redirect you to Tom Anthony's page, where he's able to tell which social networks you're logged into.

Leo: Oh, boy. I'm logged into everything. Except Twitter. I use a Twitter client, so I am on Twitter in fact, but it only shows what you're logged into via your browser. I mean, I've got Twitter running right here, but...

Steve: You can see you're logged into Facebook and Google and Google Plus.

Leo: Isn't that interesting, wow.

Steve: Yup. It turns out that - and he says, "This is a demonstration of how a website can detect which social networks a user is logged into when they visit. In my tests," he says, "it seems to work in all the major browsers: Firefox, Chrome, IE7 and on, Safari, and Opera." So what that means is that using this technology, which he explains and is essentially publishing, any website that someone goes to is able to detect whether they are currently logged into Facebook, Twitter, Google, and Google Plus, individually. And so this little page, this bit.ly/SN343, will take you over to Tom's page, where you can see for yourself. So it's like, whoops, just something to be aware of. I mean, it's not a big deal. But it's like, well, it's not a secret.

A couple weeks ago, might have been last week, when we did a Q&A, Leo, I read a question that had some fun math in it. And this was a guy who decided to see how long it would take to crack 256-bit symmetric encryption. And reading from the transcript of the podcast, we said, "So let's say the tricky government" - and this is reading his question. His question was, paraphrasing, so let's say the tricky government has a secret algorithm that somehow allows them to weaken the strength of brute-forcing a 256-bit symmetric encryption key to one-trillionth of the original strength. So we're just going to imagine that there's some way that the government has to do that. And let's say they had a computer that can try one hundred trillion guesses per second. And let's say this computer was one cubic millimeter in size. And let's say they build a cracking complex the size of the entire Earth, made out of these one cubic millimeter crypto-cracking computers.

And he says, in his question, he said, "If I did my math right, it would still take 34 trillion years to crack." And he says, "I like that." And so I think that might have been you, Leo, reading the question, saying "I like that." I responded, "I like that, too." And then Leo, you said, "Did you check his math?" Well, I didn't. But Jason Bache, who has a website, JasonBache.com, did the math. He tweeted me. He's @NerdsLimited, and it's also NerdsLimited.com is his site. He sat down with a copy of MatLab, did the math. And get this, Leo: 33.8802 trillion years.

Leo: That's right on.

Steve: So our original questioner was right, 34 trillion years. But if we want to get a little more precise, 33.8802 trillion years.

Leo: It's a rounding error.

Steve: But again, that really helps to put into perspective that, I mean, we've made a series of assumptions which are worst, worst, worst, worst, worst case, that there's a way of doing it in a trillionth of the original. We just threw away that many bits, essentially, from the key. And that it's going to do a hundred trillion guesses per second, which would be really impressive; and that it's a millimeter cube in size; and that we have an Earth-size worth of them. Given all that - and I don't know how we'd power it or cool it, of course, those little details. But still, 34 trillion years to crack. So 256-bit symmetric encryption looks like it's good to go, given that the only attack we know on it is brute force.

Again, an interesting piece of feedback about SpinRite that I thought was very clever from a listener of ours named Mike Whalen, sent it to me on the 18th of February. He said, "Hi, Steve. I've been a SpinRite user for many years. It has served me well, and I consider it an indispensible tool in my IT arsenal. Having said that, I haven't had that much occasion to use it. I happen to work for a company and with a colleague who's very focused on backups. We do a combination of local and cloud backups with a service called eFolder, which we resell. We manage anywhere from 200 to 250 desktops, and we do see dead drives. But since we have backups all over the place, we rarely see a need to fix a dead drive. We just toss it out and restore.

"Nevertheless, I like to play with SpinRite and have run it on a number of dead drives that we encounter. I've been using it fairly simply. I'd run it as Level 2 or higher, and if it worked, yay. If it didn't, well, as I said, we have lots of backups. On a recent Security Now! you mentioned Level 1 and how it can force a drive to recognize bad spots on the drive. That gave me an idea." And actually I was talking about how that is the case, but also that's a use case for running SpinRite on thumb drives, that is, on flash drives, because they're using error correction a lot. Their cells are soft, especially as they keep cranking the density up. These multilevel cells have problems and rely on error correction, very much in the same way that bits are soft in the magnetic storage of a hard drive. So algorithms are used to sort of forgive the fact that we don't have exactly what we wrote, but it's close enough that we can figure out what we meant, which is what the error correction technology in the drive does.

So anyway, continuing, he says, "That gave me an idea. I recently took possession of a Western Digital MyBook external USB drive that was throwing itself offline. When Windows or anything accessed a certain area of the disk, the drive would become unresponsive. You'd hear the drive spin down. It would spin back up and repeat that problem whenever the same area was touched. I replaced the drive with a NAS. We were due to add one at the site anyway. Meanwhile, I took the disk back to my office to see what SpinRite could do.

"I disassembled the MyBook and connected the SATA drive inside to a desktop machine. The results were not good. Basically, in levels 2 or above, SpinRite would start processing the disk, and the drive would spin itself offline, just like when it was in the case. SpinRite would freeze and couldn't continue. I never got through one successful run at levels 2 through 5. I even tried Level 2 a couple of times, thinking maybe I'd made progress with each run, but no go.

"Your statement about Level 1, though, got me to thinking. Could I run the drive at Level 1 and force the drive to make a decision on those possible bad spots? I did that. I ran SpinRite on the drive at Level 1. Success. SpinRite was able to process the entire drive. I then ran Level 2, which had already failed multiple times before, and success again. I then ran Level 3 for good measure.

"The end result, the drive is completely repaired and working perfectly, all data on it fully recovered. I've used this method on two drives that would not get through levels 2 through 5: Level 1 first, then run subsequent levels. I'm a terrible documentation reader. Maybe SpinRite's documentation points out this in bold HTML blink tag red. Well, I didn't see it. Thank you..."

Leo: He knows you too well.

Steve: "Thank you for mentioning Level 1. Perhaps you could suggest Level 1 as a way to ease a problem drive that won't get through 2 through 5." Well, that's brilliant, and it's not written down anywhere, except now it'll be in the transcript. And all of our listeners just got a new tip for running SpinRite, if you have a drive which, like this - the problem is that all of the other levels are writing something. Level 1 is a read-only pass. And that's why it's safe to run on thumb drives, because it doesn't write anything, absolutely nothing. It only reads.

But the beauty of that is that, as we were saying before, the act of reading shows the drive it has a problem. And clearly this, whatever was going wacky with this and a couple other drives that Mike found, writing gave the drive fits, but reading was okay. So reading was sort of eased into it more gently and allowed the drive to fix the problems so that then writing to them was writing to different areas because the bad spots had been relocated to good areas on the drive. So that's a great tip. It'll definitely make it into our notes for the future. So thank you, Mike.

Leo: Blinking red tag. Red blinking...

Steve: Danger, Will Robinson.

Leo: Danger, danger, danger. All right. Let's talk - now, I have a couple questions. First of all, is Chromium the same as Google?

Steve: Well, Chromium...

Leo: Are these guys Google employees?

Steve: The guy who's doing it I think works for Google. And it's all open source. And the spec is available. And it's still - they're working it out. I checked to see the project began a couple years ago. And it was last touched in January, I think.

Leo: You're talking about SPDY.

Steve: SPDY, yes.

Leo: Chromium is always updated, but SPDY, got it, yeah, yeah.

Steve: Yeah. So in order to...

Leo: And it should be, by the way, "Spidey," for Spidey sense. Okay, go ahead.

Steve: Don't listen to him, kids. Okay. So because I wanted to talk about this protocol, we discussed before TCP. And I explained when I wanted to do a podcast on TCP that I needed to do that so you could understand some of the problems which the SPDY protocol fixed, and which the HTTP protocol has because it runs atop TCP. And so just to refresh briefly, we'll remember that when we're connecting to a remote machine, we're doing that over this Internet, this packet-switched network. And in the old days, when we literally had a modem with tones that was connecting to the other end, the bandwidth was fixed and known, as was the roundtrip time. It was truly a connection between those two endpoints.

What we have today is known as a "virtual connection" because it's essentially the agreement between the endpoints that they're connected, and then the data, rather than being a physical wire, as we know, are individual autonomous packets which are addressed and sort of aimed at each end by the other, and they hop from one router to the next until they get to their destination.

So one of the problems that the designers of the TCP protocol recognized was there was no way for them to know what the bandwidth was, that is, how often can we send packets? How much can we send? What is the roundtrip delay? How long will it take to get acknowledgments back from the other end that they received the packet, so that we know whether to send it again if it got lost or not, and how do we deal with this.

So there's something called the "TCP slow start." And the idea is that TCP starts off cautiously. When you initiate a connection, neither end knows anything about the nature of the speed of the connection it has to the other. So it's built so that it starts off cautiously. And as it continues to succeed, as its packets that it's sending are being correctly acknowledged by the other end, that acknowledgment that comes back encourages it to go faster and faster and faster, and it literally goes as fast as it can until it starts having problems. Then it backs off a little bit and then comes up again and backs off a little bit. And so it's always sort of bumping its head against the ceiling of how fast it's able to run. But this all means that there's the so-called "slow start." There's a ramp-up.

Well, now, think in terms of our web browsers and what that means because the way the web browser works is we put in a URL and hit Enter. And the browser looks up the IP address for the domain name in the URL that we entered. Then it establishes a TCP connection to that IP address and sends its request. The other end, the server, receives the request. It also has this brand new connection. So it starts to send a page. But it can't send it fast because that's not the way TCP works. TCP has to be cautious. It has to sort of seek out the ceiling of where, anywhere between the two endpoints, is there a spot of congestion.

See, because that's the cool thing is, the way the system works, you could have a very fast client connection to the Internet, a very fast server connection to the Internet, but there could be some problem somewhere in between. Could be a flaky router or an overloaded router. And these packets have to get through there. So TCP works very cleverly to just do the best job it can. But it means it has to start slow. So that means that that page is not going to come as fast to us as it technically could if this were on a mature TCP connection that had already learned how fast it was able to go.

Now the problem gets even worse because that page comes to the browser. And as we know, pages are composed of a whole plethora of additional assets, pictures, I mean, especially contemporary websites full of little social networking buttons and icons. And so all of this comes in on the main page, that'll then have tons of URLs for other stuff. It'll have script resources back to the same server. It'll basically spray queries out all over the 'Net to all the other services, advertising services that we've been talking about recently, anything that provides content for that page. And it's just bewildering now how much that is. Well, every single one of those is a new TCP connection that has to start out slow and ramp itself up. So that slows things down.

And there was some work, when we went from HTTP/1.0 to HTTP/1.1. The problem was that originally a browser would make a connection for a single query. It would send the URL, for example, to the server, get the response, and by mutual agreement they would both terminate the connection. Then, if the browser saw, oh, look, I've got 26 things, I've got seven different JavaScript files, and I've got a bunch of icons and navigation stuff and menu resources, all this stuff I need from that same server. It would, for every single one of those resources, initiate a TCP connection, as for that thing, wait till it got it, break the connection. And in TCP there is no memory from one connection to the next. Neither is there memory, even simultaneous connections to the same server, there's no interconnection sharing in the protocol.

So what engineers realized - and this is 10 years ago. We've had the web for a while now. They were realizing this is just not very efficient. TCP is a fantastic solution for end-to-end communication, but it's just not good for short-lived requests. It doesn't make much sense. In fact, that's one of the reasons that DNS, for example, uses UDP, is that there is no setup and speeding and timing and sizing everything. A DNS query is a single query packet that goes out, and a single one that comes back, and you're done.

So they kept, in DNS, they kept the overhead low, but they don't have any of the benefits in the UDP protocol that we get with TCP, which is all of this work being done for us. If we send a big file over, it's possible because of the packet routing that the packets are going to arrive in a different order than they were sent. So they're all, as we know from talking about TCP in the past, they're all serialized, and the receiver is able to sort of check them in on the way in. And if it sees one that it's missing, it's able to wait for that to catch up, and then it drops it in the right place.

So the TCP protocol does all this great stuff. But there are some costs to it. And one is it's just not great for short-lived connections. It just barely gets going, and the connection gets dropped. So when they went to HTTP/1.1 they introduced - the designers of HTTP evolved the protocol to introduce the notion of a persistent connection where there would be a header - actually there was a "keep-alive header," it was called, in HTTP/1.0, but wasn't widely supported. They did a better job in HTTP/1.1. And the idea would be that the browser and server would agree not to drop the connection, that they would hold the connection up so there would be a query and then a response, and then a query over the same connection and then a response, and so forth. And for a while browsers were constrained to only having two connections per domain.

The idea was we didn't want to flood a server with 20 connections between two points because, if you think about it, it's really not efficient. It's not efficient in terms of the resources to manage the connections, both at the client's machine and the server, because you have 20 sets of data, all these individual connections. It's inefficient because of TCP's slow start problem. You don't want to launch 20 connections. They're all going to start slow. So, and you ultimately have some fixed amount of bandwidth between these two points. So if you've got 20 connections, all going at once, you're not going to actually end up finishing any sooner. You've going to end up finishing later because you've got 20 slow starts and some sort of bandwidth limitation between there.

So it makes much more sense to bring up a limited number of connections, and for quite a while it was two, and then let TCP learn what the bandwidth is and then send, instead of having 20 connections with one query each, just have two connections and send 10 queries each. And those later queries and responses will be much faster because they're running over a more mature TCP connection. So there was this two-connection-per-domain limit that browsers were enforcing. The problem is, as web pages got just bigger and heavier, many browsers have now extended that to I've seen a number six is used often, just because you just want more resources from the remote server. Still, it's not as efficient as being a little more patient.

So there was this notion introduced in HTTP/1.1 called "pipelining." Pipelining, the concept there was instead of the client issuing a request and waiting for the reply, the client could - and "pipelining" is a term we've talked about back years ago when we were talking about how to speed up processors, the idea being that you can have multiple things in the pipeline at once. So a browser could make a query, receive the page, holding the TCP connection up, and then look at the page, and then issue a series of requests in the connection to the server, and then get back a series of responses.

It turns out that there were too many proxies involved on the 'Net which were not bug-proof in the face of pipelining. The proxies want to work on a single-transaction basis. And as a consequence, pipelining, although it's technically in the HTTP/1.1 spec, it's disabled by default today in all major browsers. They don't do it. They wait for the response to come back, then they issue another query. That's the only way to reliably get HTTP to work at this point.

Now, the problems with a limited number of connections, which we want to have in order to get performance, but one problem is that that inherently serializes everything. It means that any resource that was, for example, stalled for some reason, it might be big and low priority, but the browser asked for it, and other, smaller resources that could be served more quickly, they're waiting behind the big one. So there's a problem with the serialization, even when we've got a mature, statically held TCP connection to make these HTTP queries over. So that's a problem.

That also means, because we're serializing everything, there's no ability to simultaneously render objects. We've seen, for example, on web pages where JPGs fill in, and they fill in one, then the next, then this one, then that one, then that one. And if your bandwidth is sufficiently low, you can actually see the picture sort of render in. Well, it might be nice if we were able to do more things in parallel. So that was one of the things that the Chromium guys looked at.

Now, we take it for granted that clients ask servers, that is, web clients ask web servers for everything they need. But it's worth questioning why that is so. It's the way it's always been. But think about it. The server that is sending the client the page, it has the page before the client has it. And that means, if it looked at the page, it knows what other things it has that the client is going to ask for, if they're not in the client's cache. So one of the problems with HTTP, which we didn't regard as a problem in the beginning because it was always designed to be a query-and-reply-based protocol. But there is no concept of what's called "Server Push," the idea of the server pushing things to the client on its own because it has reason to believe the client is going to be getting around to ask for it at some point. So that would save the time of the page getting there, the client processing it, figuring out what's there, and then issuing requests that have to go back out.

Essentially, the server is sitting around with its server-to-client bandwidth unused, sitting there just being wasted, time is being wasted, while the page is getting to the client so that the client can look at it and then begin sending requests back. So if we really look at how the time is spent, we can see that there are ways, all kinds of clever things we can do to compress this whole transaction between the client and the server.

Now, another problem is that the payload itself, the thing that's being requested, a page of text, for example, can be compressed. Compression, both a compression technology called "deflate" and also more popularly and better, gzip, are now supported by all browsers and servers, the idea being that the client is able to say, in its query to the server, hey, I know about compression, so feel free to save time and speed. Compress this on the wire before you send it. So even though the server may not have it stored in a compressed fashion, it can use the gzip protocol to do essentially a stream compression, to compress the stream so that it is much smaller. Web pages have a huge amount of redundancy in them with all of the tags that they contain, let alone just often English compresses really well because of common words. But web pages in general, just because they've got so much structure which is highly compressible, that's a win.

So payload can be compressed. But there has never been any way to compress the query and the query headers or the response headers. That is, the headers themselves have never been subject to compression. That just wasn't part of the spec. It turns out headers have been getting bigger. Everybody, as we've been talking about, is going crazy with cookies. Cookies are growing. It turns out that the headers can vary from 200 to as much as 2K bytes, and they're typically between 7 and 800 bytes in size. But that can be squeezed way down.

And in the case of, for example, an ADSL line, where you're highly asymmetric in bandwidth, you've got much lower bandwidth upbound than you have downstream, that kind of just the size of the query headers is slowing down every single query because there's no way to compress them. The other thing that is slowing down every single query is a huge amount of redundancy. There are headers that never change between a given transaction. For example, the user-agent. The user-agent identifies your browser.

Well, if you're sitting there, and the browser is sending off a whole bunch of queries to the same server, it's not changing. So the user-agent data is never changing. Yet every single query has a user-agent header which is highly redundant. The host header is also not going to change, where it describes - it says www.google.com, that's going out. Even though we have a connection to Google's IP, we're still declaring this is going to www.google.com, and again is highly redundant, as is the accept header that tells the server what formats the client is able to accept the responses in, very redundant and uncompressed. So there's just a lot that can be done to fix the protocol.

Now, again, as I said, this has been known for about a decade. There was some work done on a next-generation TCP. It wouldn't replace TCP, but it was called SCTP, Stream Control Transmission Protocol. And the idea would be that, if this had ever happened - and it died on the vine. There's been no progress on it in 10 years, since 2002. There's an introduction to SCTP which did make an RFC, it's RFC 3286, which describes what the original goals of the system were. And the idea would be that, in the same way that UDP - I don't remember the number. I think it's hex 11 and maybe decimal 17. That would make - well, anyway. UDP has a protocol number. TCP has a protocol number. One is 6. Anyway, it's been a long time since I looked at these. But the idea would be...

Leo: That's what reference books are for. You don't need to know that.

Steve: Yes. They would decide - I'm annoyed that I don't remember, but, yeah, I have enough in my head.

Leo: That's exactly right.

Steve: So this would be assigned a new protocol number. And so the browser would send off, on top of the IP protocol, it would embed in the IP packet this SCTP packet, much as right now we embed a TCP packet within the IP packet, and it would define a new protocol. And one of the things that this protocol would offer is the concept of multiplexed streams and stream-aware congestion control. So the idea would be, you'd establish a connection from the endpoints. And then the protocol itself would understand the idea of independent streams of data sharing that same connection. We have nothing like that now in our protocols. We've got - they're very simple - UDP and TCP. And if somebody wanted to do that, they would either have to come up with a next-generation TCP or add something on top. And that's what SPDY does. We'll talk about it in some detail in a second.

But the problem, of course, with coming up with a brand new protocol is nobody else knows about it. None of the routers know about it. Now, you can argue that a router probably doesn't care. It's routing IP packets and is largely agnostic to the contents of the IP packet. So it could contain any - the IP packet could be the envelope containing any interior protocol. But our home routers do care. NAT routers are protocol-aware. I mean, that's what they're doing. They're needing to understand ports in order to handle the whole NAT routing job. And IP packets have IP addresses, but they don't have ports. Ports is an abstraction that lives inside that packet, if it's carrying a UDP or TCP or some other oriented protocol. So it would hugely impact our existing infrastructure if someone tried to just float some brand new protocol because it would make browsers faster. And thus this thing died, even 10 years ago.

Then there was some work done on how HTTP would run on top of this. There was a different thing called SST, which is Structured Stream Transport. It had some of the same problems. And then there was something called MUX and SMUX, which they stopped working on that about in '07. Nothing's really happened since then, as I recall.

So there has been an awareness that we need something better. There is clearly pressure with the increasing size and complexity and, well, and just the fact that we'd like our Internet experience to be much more active, much more interactive. We're sort of getting to the point where we're seeing storage in the cloud, services in the cloud, applications running in the cloud, applications also being spontaneously downloaded in script into our browsers. But these are wanting to be very interactive. So we really want the best utilization of the bandwidth between us and the servers that we're connecting to as possible.

So SPDY comes along. SPDY has the goals of achieving a large reduction in page load time, that is, latency. "Latency" is the word the guys on the team use because what they want is just, if you're using a browser, for example, that doesn't understand SPDY, with a server that does, they would like there to be a dramatic difference in experience with a SPDY-aware browser. And frankly, if this can - they're going to make the server open source. One can imagine maybe Apache will grab some of the code and upgrade themselves. And then that'll get incorporated into some of the Unixes and Linuxes, and that'll put pressure on Microsoft to support the SPDY protocol on IIS over on the Windows-based servers. And when that happens, Google will have Chrome.

Well, that means all the other browsers are going to immediately have to support SPDY because, given what I am about to tell you, this thing really does work. And so imagine going to a website, you go to Facebook, and in one-third the time the page has snapped up, and it feels different. It feels dramatically faster using a browser that knows SPDY. Well, all the other browsers are going to have to support it in order to compete. So the goal is a large reduction in page load time. They wanted, in the design of this, to minimize deployment complexity, which is to say just run on top of TCP. Don't reinvent TCP. We can't at this point. It would break all of our NAT routers. It would break switches at the other end that are running the big server farms. The routers in between don't care, but we still couldn't get our data back and forth.

So use TCP. We have a huge investment in it. It works. I mean, and it works beautifully. Once it gets started, it's very efficient. It just isn't efficient to be used for little tiny requests and queries that are then dropped. And they wanted no changes required from the website. That is, they wanted to be able to introduce this protocol support in the server, and have a browser that was aware, and all the magic happens between the two of them and on the line in between. But the same page ends up being rendered at the client. The same content is sourced at the server. Which is beautiful because, again, this really does ease adoption.

So, more specifically, they've come up with a protocol which allows concurrent HTTP requests to run across a single TCP connection. And I'm going to explain how they've done that. But so what SPDY gives us, and, I mean, this is working now and has been benchmarked, is concurrent requests across a single connection. It absolutely always compresses headers, the so-called "metadata," the stuff that isn't actually [indiscernible] cookies and expires headers and the user-agent and the host headers and all that. That's always compressed in SPDY protocol. The redundant headers are not part of the protocol. They don't appear. They're eliminated. And the unnecessary ones, also. There are some headers which were needed for the non-SPDY protocol which SPDY's operation inherently eliminates.

And SSL is always going to be the underlying transport protocol. So they're just assuming that - they recognize that we're headed to a world where we're going to have point-to-point security, so let's strongly enforce SSL connections so that we just know that if we've got a browser with a SPDY connection, it will automatically be using SSL. And that's also important because, if you had some proxies that were trying to parse your traffic - remember, for example, that ISPs are often running caching proxies, the idea being that they're able to serve common website pieces to their own users more quickly so the query doesn't have to go out on the Internet, off to a remote server, if it's in the ISP's cache. And that lowers their costs because they're able to use their internal bandwidth to satisfy their clients, not have to send stuff out on the Internet. And that's bandwidth they pay for.

The point, though, is that that means that there is, even though we don't see them, we don't see their IP addresses, we're not aware of them, they're there, caching what we do. That's, for example, the reason that I have ShieldsUP! initiate an SSL connection in order to bypass the proxy to get the user's IP address. I don't want to test the security of the proxy. I'm being asked to test the IP of the user. So I do that specifically for proxy avoidance. And SPDY will run over SSL similarly for proxy avoidance so that it blinds proxies that otherwise would sort of see something that looks like an HTTP query but not quite. I mean, it would really foul things up. So this is a good thing for them, too.

The other thing that SPDY does which is huge is it enables the server to initiate communications with the client, that is, to push data to the client whenever possible, not force the server to sit there, knowing what the client is going to get around to asking for as soon as the query gets back to it, but to be preemptive, which is a huge win because obviously the best response we're going to get is if we can get the TCP connection ramped up, keep it running at that speed, and keep it full because time that isn't being spent sending data is just time that we have to wait for the page to end up rendering.

So conceptually we understand how we have a layered set of protocols. We've talked about just now how IP packets are envelopes containing TCP packets, and within the TCP protocol is SSL, which is an application protocol that runs on top of TCP. Well, SPDY will run on top of SSL, and HTTP runs on top of SPDY. So essentially what we've done is we've sort of - we've shimmed in between SSL and HTTP, we've shimmed another layer of protocol stack. What that means is that nothing below needs to change, and nothing above needs to change, as long as the client and server are aware, they've come to an agreement that they both understand SPDY version whatever, and negotiated that, and then are able to take advantage of its features.

So specific features are it supports the multiplexing of streams. The idea is that once a TCP connection is established, the protocol is frame-based. So we're used to thinking in terms of packets. We have to sort of disabuse ourselves of that. Think in terms of TCP just being a stream. Now, the fact that TCP breaks that stream into packets to get where it's going, we understand, but we're going to ignore that for the moment. Now we're seeing at our end a TCP connection, and we're just feeding it data. So we break that data into frames ourselves.

Every frame has an eight-byte header which describes the balance of that frame. Frames can either be control frames or data frames. In the case of a control frame, the very first bit is a one. Data frames have their very first bit as a zero. Then for a control frame there's a 15-bit version number, which ought to be enough, and then a 16-bit type which identifies the type of frame. Then there's eight bits of flags in the second block of 32 bits, and 24 bits of length. So that specifies the length of the payload, followed by the payload. Data frames, as I said, have that first bit is a zero for data. Then they have a 31-bit stream ID. So that means we know that 32 bits is 4GB. So 31 bits is 2GB. So we have two billion possible streams which can be individually tagged and flow over the connection.

However, even streams always go from client to server, and odd streams always go from server to client. And the stream IDs always increase monotonically in each direction. So as the client is initiating new streams, it's numbering them 2, 4, 6, 8, 10, 12, and so forth. As the server is creating streams, it's 1, 3, 5, 7, 9, and so forth, both in upwards directions with the proper evenness and oddness. And so the concept is that we have a single TCP connection. Now, it's freaky when you say, wait a minute, but wouldn't more connections give me more bandwidth? But really that is not the case. We're used to thinking that if we make more connections, we're going to get more bandwidth. But if we make more connections to the same place, those connections absolutely are going to compete with each other. So all you do is lose, if you make more connections.

So this system is one connection between client and server. But because it lays on top of TCP, it lays this frame abstraction, and frames have stream IDs, and so obviously the client is able, over this single TCP connection, it's no longer just sending queries directly on the wire. Instead, it's packaging them in frames. So a single query lives in a single frame. By convention of SPDY, the query headers are always compressed, so that's going to be much smaller. But think about it. Now what that means is that it's able to create a whole batch of very small queries, stacking them in TCP, not thinking about packets. We're not dealing with packets now. These are frames which may well cross packet boundaries. But it means that we're able to pack the TCP packets with much more.

So what we get is a much higher level of utilization of the bandwidth we have in both directions. The server is similarly unconstrained. If it receives a burst of queries, it's able simultaneously to answer them all, and it does. It's able to pull them and as quickly as it can to begin multiplex feeding them back on the line, which means every single outgoing packet is going to be filled. Even if these things are small, and normally you'd have short packets, and then you'd be waiting for acknowledgment and so forth, here, because this multiplexing is such a huge win, because we're able to essentially fill every single packet to the brim, TCP works much more efficiently that way, and the server is able to just send all this stuff at once.

Now, they solve the problem of some things being more important than the other by having two bits assigned for priority. So the client is able to set a priority on its query: 0, 1, 2, or 3, 0 being the lowest priority, 3 being the highest. It's not something that has to be honored by the server. But it typically would be. And the idea would be that, because now we have this multiplex, it might very well be that the client says, oh, I need the scripts for my page more importantly than I need pictures that are going to be loaded because we need the scripts in order to get things running, or I need other important pieces.

So the client is able to prioritize its queries, even while sending them all in a big burst along with the lower priority queries. And the advantage of that is that it has to manually hold the lower priority queries back, knowing that it needs the higher priority ones first. It's able to just dump them all in this multiplex connection, knowing that the server will receive them and will use the available bandwidth to send all the highest priority queries back, the responses to those queries back before it gets around to sending the lower priority ones.

So it's a relatively straightforward, simple addition to the existing TCP protocol. It essentially does not hugely change HTTP at all. It compresses the headers. It eliminates some, makes some slight rearrangements so that a SPDY-aware browser knows how to issue these queries and of course knows how to use its one connection between it and the server in order to saturate. And in the benchmarks which Google did, they took the content of the top 25 benchmarks and simulated a cable modem that had a 4Mb downstream bandwidth from the server to the client and a 1Mb bandwidth from the client upstream to the server. The average load time of these 25 websites, the top 25 on the Internet, was 2.348 seconds over HTTP. And if you think about it, that's like, okay, you go to a big page with that kind of bandwidth, less than 2.5 seconds, 2.348 seconds. Switch to the SPDY protocol, 0.856. So from 2.34, so that's like 2.4, to 0.8. So it's about a factor of 3:1. It takes, I mean, it's dramatically faster. And you're getting maximum bang for your bandwidth, setting up a single connection and just being smarter about the way we use the packets in order to adapt it to the nature of today's pages.

So I've got all my fingers and toes crossed for this. I hope that they get this thing nailed down. I hope that their open source server, as soon as it's ready, gets adopted. We need it somehow to start getting deployed. I mean, I'm sure Google's servers will switch to it. And suddenly browsers that are SPDY-aware will be able to run all Google services much faster. And as soon as other sites see that, they'll say, well, we need servers that allow our site to be running three times faster, one-third the latency to bring the page up. So, I mean, they've demonstrated it. It works. The protocol's here. There's really no downside, no new hardware needed anywhere. We just need to use SPDY-aware browsers. And I'll bet you we've got SPDY in our future.

Leo: SPDY in our future. That's what we should have called this show. So just to reiterate for people who tune in late, Chrome does this automatically. Anything based on Chromium would, as well. So if you're on Linux and use Chromium, I presume that would do it.

Steve: Well, when they incorporate it. Right now it is still in test mode.

Leo: Oh, okay. So you're using SPDY 1, the earlier SPDY. Or no.

Steve: SPDY has not been deployed yet.

Leo: Oh, so we haven't seen this at all.

Steve: No, it's in the Chromium galaxy somewhere, down off in some project, but is not part of the main production code.

Leo: Got it, got it. And then somebody was saying in the chatroom that, if you have Firefox, you can enable SPDY.

Steve: What?

Leo: But I think that must be wrong.

Steve: That would be wonderful, but I'm suspect. I would think someone could come up with their own SPDY converter. No, no, no. The browser needs to be aware of the things it can do, that it can send - oh, and I forgot to mention, also, that - do I not have it in my notes? Did I skip those notes?

Leo: In Firefox 11 you can enable it in about:config, they say.

Steve: Oh, yay.

Leo: It's not a full, maybe not a full implementation. And of course, as you mentioned, the server has to support it. It has to be at both ends, obviously.

Steve: Right. So "Multiplexed Streams" we get. "Request Prioritization" we get. "Header Compression" we get. We also get "Server Push" and "Server Hint." The server is able to push whatever it wants to. And there's a new header. It's "X-Associated-Content." So it tags the thing it's sending, without even being asked for it, as content associated with the following query. So that allows the client that suddenly receives something it didn't ask for to know why it got it. And then there's - so that's called "Server Push," where the server is able to just proactively, preemptively send something, which to me makes so much sense because the server does have the page. It's waiting to be asked for all these other things. Why not just go ahead and get going?

Leo: Right.

Steve: And so this allows that. And the other thing is, because there's some concern that maybe that's a little too aggressive, then there's a little sort of a softer version which is "Server Hint." And so that uses a header called "X-Subresources." And that enumerates all of the resources that the page is known to contain. So the server is sort of - it's suggesting to the client that it should ask for the following resources in cases where the server believes that the client is going to need it.

Leo: So I've entered this Chrome URL that they put in the chatroom, chrome://net-internals/#spdy. And it says SPDY Enabled: true; Use Alternate Protocol: true; Force SPDY Always: false; Force SPDY Over SSL: true. So this is kind of a temp - this is kind of an early release version of this.

Steve: So it's kind of in there.

Leo: It's kind of in there.

Steve: But it's in a production Chrome browser.

Leo: Yeah, this is in my regular Chrome. Maybe you have to hit this to turn it on? They say you could turn it on in Firefox 11. It will be on by default in 13. SPDY's coming.

Steve: Wow.

Leo: Clearly. And right now it looks like only Google, of course, is the only server supporting it. But so you can actually view live SPDY sessions here like this in the browser. This is in Chrome. And so as you browse around in Chrome, you'll be able to see what else is using it, I guess. Pretty cool.

Steve: Very cool.

Leo: Yeah.

Steve: Well, that's our future. And boy, it's fast.

Leo: Yeah. That's exciting.

Steve: Our future is speed.

Leo: Our future is fast. Yeah, you know, it's funny because Google has always said - this is their philosophy. They said people - in fact, they did a study that came out last week that said people will only wait, like, 25 milliseconds before a site - a quarter of a second, 250 milliseconds, a quarter of a second before they go, nah, it's too slow.

Steve: To switch away. I'm going to go somewhere else.

Leo: So it really does become important. And it's something we've been battling with on the new TWiT site is speeding it up a little bit like that. Maybe we'll implement SPDY. Steve Gibson is at GRC.com. That's his website, not yet SPDY-fied, but it will soon. Right? No.

Steve: I don't know. Would like to.

Leo: You're using IIS; right? You're still using IIS.

Steve: Yeah, I am. It'll be when Microsoft - I've got such a huge investment in my own assembly language glue all over IIS.

Leo: Right, right. You can find him there, though. At GRC.com is where SpinRite is, the world's finest hard drive maintenance and recovery utility; all those free utilities he gives out, as well. And this podcast, 16Kb versions for the bandwidth-impaired; transcripts, as well. GRC, Gibson Research Corporation, GRC.com. His Twitter handle - if you follow him you will get good stuff, I promise you - @SGgrc. And you monitor people who are sending you stuff via the "@" sign.

Steve: Yeah. If they mention @SGgrc, that's how I get all of these neat little tweets that are incoming, and that's how we found out about the guy that did the math in the 34 trillion years to crack 256. Yeah. I do keep an eye on that, so I'm accessible that way to our listeners.

Leo: Yeah. And then of course we have all of the audio and video versions except that 16Kb version and the transcript at TWiT.tv. And you should watch live. Now, people are tuning in, saying, well, wait a minute, wait a minute, this is Tuesday. So normally we do this on Wednesday, 11:00 a.m. Pacific, 2:00 p.m. Eastern time, 1900 UTC, at TWiT.tv. But because of some, I don't know, some shindig Apple's throwing tomorrow, we flip-flopped you.

Steve: Eh, what's up?

Leo: Something going on tomorrow. So tomorrow, in the normal Steve Gibson time, actually starting at 9:30 Pacific, 12:30 Eastern, we'll be doing coverage of the Apple iPad announcement.

Steve: Ooh, good to know.

Leo: Yeah, 9:30 a.m. Doors open and the speech begins at 10:00 a.m., so we thought we'd start a little bit early. Tom Merritt, me, Sarah Lane of iPad Today, and Alex Lindsay from MacBreak Weekly will be there. Andy Ihnatko will be on via Skype. And then we do have a friend, I don't even want to say his name in case Apple uninvites him...

Steve: Don't.

Leo: ...will be there. I don't think they'll uninvite him. He's the editor in chief of a major Macintosh magazine. Anyway, he'll come out after and talk to us and maybe, if he can, get us into the demo room because they often have a demo room. So all of that tomorrow instead of Security Now!. Thank you, Steve. I appreciate your letting us move.

Steve: My pleasure, Leo. Always a pleasure. And we'll be back to Wednesday next week for a Q&A, so GRC.com/feedback. Send me your thoughts and comments, and we'll get to them next week.

Leo: Absolutely. Thank you, Steve Gibson. Take care.

Steve: Thanks, Leo.