http2 explained

Daniel Stenberg

Innehållsförteckning

http2 förklarat

Det här är ett detaljerat dokument som beskriver HTTP/2 (RFC 7540), dess bakgrund, koncept, protokollen och lite om existerande implementationer och vad framtiden kanske erbjuder.

Se https://daniel.haxx.se/http2/ för projektets hemsida.

Se https://github.com/bagder/http2-explained för källkoden till allt innehåll i boken.

BIDRA

Jag uppmuntrar och välkomnar all hjälp och alla bidrag från de som har förbättringar att erbjuda! Vi accepterar [pull-requests] (https://github.com/bagder/http2-explained/pulls), men du kan också posta dina problem eller skicka email till daniel-http2@haxx.se med dina förslag!

/ Daniel Stenberg

1. Bakgrund

Det här dokumentet beskriver http2 från en teknisk och protokollnivå. Det började som en presentation Daniel gjorde April 2014 i Stockholm som sedemera gjordes om och breddades till ett fullt utvecklat dokument med alla detaljer och riktiga förklaringar.

RFC 7540 är det officiella namnet på den slutliga http2-specifikationen och den blev publicerad den 15:e maj 2015: https://www.rfc-editor.org/rfc/rfc7540.txt

Alla misstag i det är dokumentet är mina egna och resultatet av mina fel. Peka gärna ut dem så åtgärdar vi dem till en kommande uppdatering.

I det här dokumentet har jag försökt att konsekvent använda ordet "http2" för att bekskriva det nya protokollet, medan det ju rent tekniskt och korrekt faktiskt heter HTTP/2. Jag har gjort det valet för läslighetens skull och för att få ett bättre flöde i texten.

1.1 Författaren

Mitt namn är Daniel Stenberg och jag jobbar för Mozilla. Jag har arbetat med open source och nätverk i över tjugo år i otaliga projekt. Möjligvis är jag mest känd för att jag är huvudutvecklaren av curl och libcurl. Jag har varit involverad i IETF och dess HTTPbis arbetsgrupp under flera år och där har jag hållit mig uppdaterad med uppdateringen av HTTP 1.1 samt varit inblandad i arbetet med standardisering av http2.

Email: daniel@haxx.se

Twitter: @bagder

Web: daniel.haxx.se

Blog: daniel.haxx.se/blog

1.2 Hjälp!

Om du hittar misstag, utelämnade detaljer, fel eller rent av lögner i det här dokumentet, skicka mig gärna en rättad version av det aktuella avsnittet så kommer jag snart publicera en uppdated version. Jag ger ordentliga omnämnanden och tack till alla som hjälper till! Jag ämnar göra dokumentet bättre över tid.

Det här dokumentet finns tillgängligt på https://daniel.haxx.se/http2

1.3 Licens

1.4 Dokumenthistoria

Den första versionen av det här dokumentet publicerades 25:e April 2014. Här följer de större förändringarna i de senaste versionerna.

Version 1.13

- Converted the master version of this document to Markdown syntax
- 13: Mention more resources, updated links and descriptions
- 12: Updated the QUIC description with reference to draft
- 8.5: Refreshed with current numbers
- 3.4: The average is now 40 TCP connections
- 6.4: Updated to reflect what the spec says

Version 1.12

- 1.1: HTTP/2 is now in an official RFC
- 6.5.1: Link to the HPACK RFC
- 9.1: Mention the Firefox 36+ config switch for http2
- 12.1: Added section about QUIC

Version 1.11

- Lots of language improvements mostly pointed out by friendly contributors
- 8.3.1: Mention nginx and Apache httpd specific acitivities

Version 1.10

- 1: The protocol has been “okayed”
- 4.1: Refreshed the wording since 2014 is last year
- Front: Added image and call it “http2 explained” there, fixed link
- 1.4: Added document history section
- Many spelling and grammar mistakes corrected
- 14: Added thanks to bug reporters
- 2.4: Better labels for the HTTP growth graph
- 6.3: Corrected the wagon order in the multiplexed train
- 6.5.1: HPACK draft-12

Version 1.9

- Updated to HTTP/2 draft-17 and HPACK draft-11
- Added section "10. http2 in Chromium" (== one page longer now)
- Lots of spell fixes
- At 30 implementations now
- 8.5: Added some current usage numbers
- 8.3: Mention internet explorer too
- 8.3.1 Added "missing implementations"
- 8.4.3: Mention that TLS also increases success rate

2. HTTP idag

HTTP 1.1 har förvandlats till ett protokoll som används för i princip allting på Internet. Stora investeringar har gjorts i protokoll och infrastruktur som utnyttjar detta. Det är draget så långt att det idag oftast är lättare att få saker att köra över HTTP hellre än att bygga något som är helt nytt och eget.

2.1 HTTP 1.1 är stort

När HTTP skapades och slängdes ut i världen ansågs det antagligen vara ett ganska enkelt och rakt-fram protokoll, men tiden har bevisat att detta inte är sant. HTTP 1.0 är RFC 1945 som är en 60 sidors specifikation släppt 1996. RFC 2616, som beskriver HTTP 1.1, släpptes bara tre år senare 1999, och växte avsevärt till sina 176 sidor. Trots att den redan växt så, delade IETF senare upp specifikationen i sex delar under arbetet med att uppdatera den, med ett mycket högre totalt sidantal. Det som blev RFC 7230 med familj. Hur man än räknar är HTTP 1.1 stort och inkluderar myriader av detaljer, nyanser och inte minst valbara delar.

2.2 En värld av alternativ

HTTP 1.1:s natur med massor av små detaljer och alternativ tillgängliga för senare utökningar har lett till ett ekosystem av programvara där nästan ingen implementation någonsin implementerar allting - och det är inte ens möjligt att riktigt säga vad "allting" är. Det ledde till en situation där funktioner i protokollet som utnyttjades väldigt lite ofta inte alls implementerades - i början - och de fall när någon faktiskt implementerade dem såg de väldigt liten användning.

Senare skapade detta interoperabilitetsproblem när klienter och servrar väl började använda såna funktioner. HTTP pipelining är kanske det främsta exempel på en sådan funktion.

2.3 Otillräckligt nyttjande av TCP

HTTP 1.1 har svårt att verkligen till fullo nyttja all kraft och den prestanda som TCP erbjuder. HTTP-klienter och webbläsare måste vara väldigt kreativa för att hitta lösningar som minskar laddningstider.

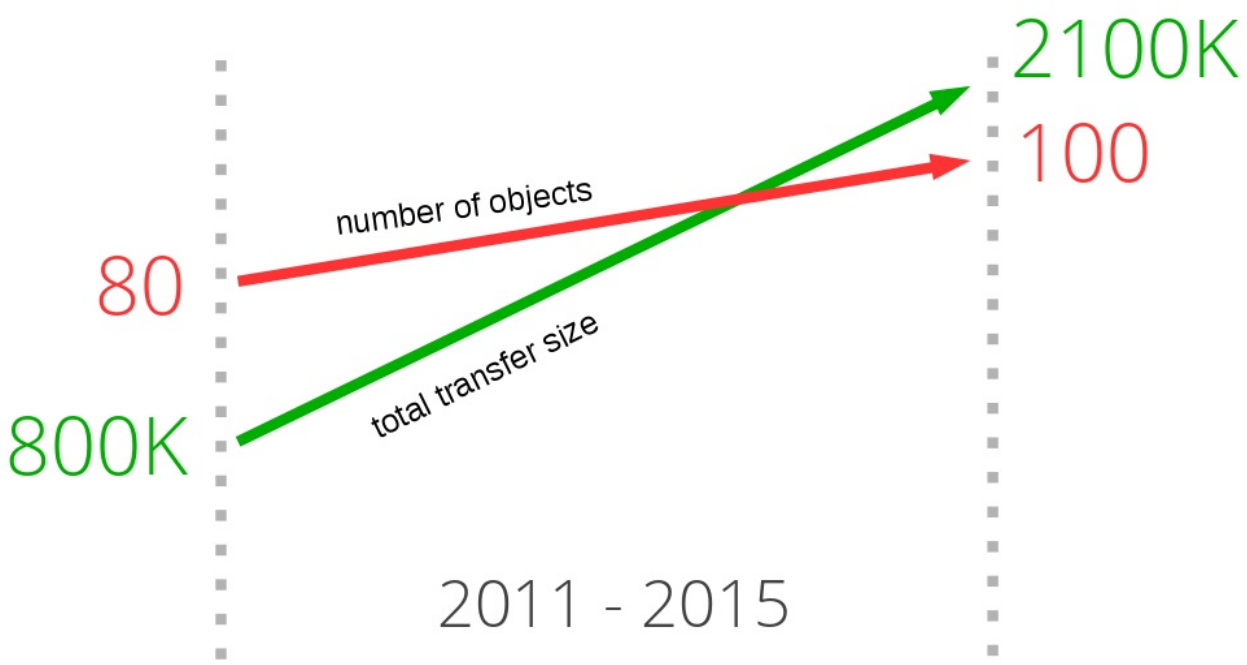
Andra försök som pågått under åren har också bekräftat att TCP inte är så enkelt att ersätta och därför fortsätter vi att både förbättra TCP och de protokoll som vi kör ovanpå det.

Enkelt uttryckt, TCP kan användas bättre för att undvika pauser och ögonblick i tiden som kunde användas till att skicka eller ta emot mer data. De följande avsnitten belyser några av de existerande bristerna.

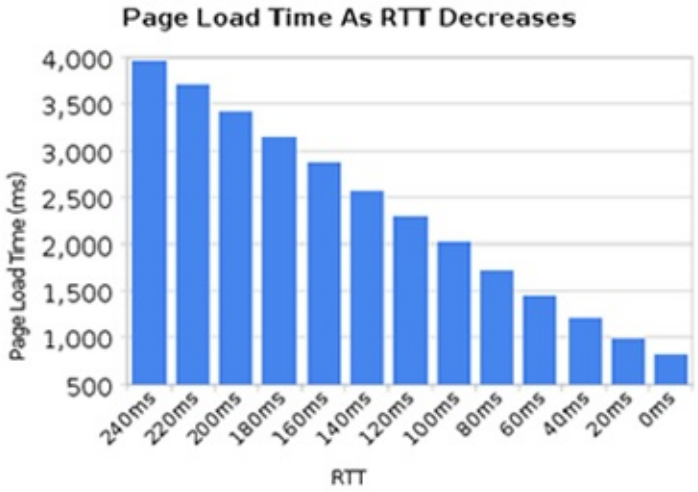
2.4 Överföringsstorlekar och antal objekt

När man tittar på trenden för några av de mest populära sajterna på webben idag och vad det krävs för att ladda ner deras framsidor, så ser man ett tydligt mönster. Genom åren har mängden data som behöver laddas ner gradvis ökat till och förbi 2.1MB. Vad som är än viktigare i det här sammanhanget är att i genomsnitt så behövs det över ett hundra enskilda objekt för att visa varje sida.

Som diagramet nedan visar, så har trenden pågått ett tag och det finns inga indikationer på att den kommer ändras inom kort. Den visar tillväxten av överföringsstorlek (i grönt) och det totala antalet förfrågningar som använts i genomsnitt (i rött) för att visa de mest populära webbsajterna i världen, och hur det har förändrats de senaste fyra åren.



2.5 Fördröjning dödar



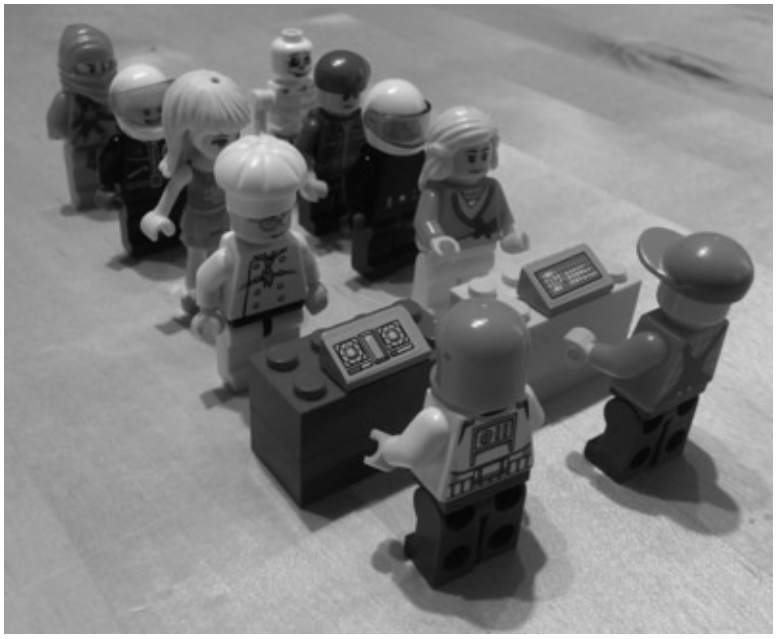
HTTP 1.1 är väldigt känsligt för fördröjningar (latency), delvis för att HTTP Pipelining fortfarande är fyllt med problem och är avslaget per default hos en stor andel av användarna.

Medan vi sett en rejäl ökning i tillgänglig bandbredd hos människor över de senaste åren, så har vi inte sett samma förbättring i att minska fördröjningar. Länkar med stora fördröjningar, som många nuvarande mobila teknologier, gör det riktigt svårt att få en bra och snabb upplevelse av webben, även om du har en riktigt hög bandbredd på uppkopplingen.

Ett annat användningsfall som verkligen behöver låga fördröjningar är vissa former av video, som till exempel videokonferenser, spel och liknande, där det inte bara är en på förhand genererad ström att skicka ut.

2.6. Först-i-kön-blockering

HTTP Pipelining är ett sätt att skicka en till förfrågan till servern medan klienten fortfarande väntar på svaret på den förra frågan. Det är väldigt likt en kö på banken eller till kassan i en mataffär. Du vet helt enkelt inte om personen framför dig i kön är en snabb kund eller den där irriterande personen som kommer ta en evighet innan hon/han är klar: först-i-kön-blockering.



Visst, du kan välja kö omsorgsfullt så att du ökar chansen att verkligen ta rätt kö, och ibland kan du till och med starta en ny, egen kö, men hur du än gör så kan du inte undvika att ta ett beslut och när det väl är taget kan du inte byta kö.

Skapa nya köer är också belagt med prestanda- och resurs-förluster så det skalar inte upp bra till mer än ett ganska lågt antal köer. Det finns helt enkelt ingen perfekt lösning för detta.

Även idag, 2015, så har de flesta desktop-webbläsare HTTP pipelining avslaget per default.

Vidare fördjupning på det här ämnet kan man hitta i till exempel Firefox bugzilla 264354.

3. Tricks för att komma över fördröjningssmärtor

Som alltid när problem dyker upp så har folk samlats och uppfunnit tricks för att komma runt dem. Några tricks är smarta och användbara, några av dem bara hemska fulhack.

3.1 Spriting

Spriting är termen för att beskriva vad man gör när man stoppar massor av små bilder ihop till en enda stor bild. Sen använder man javascript och CSS för att "skära ut" bitar ur den stora bilden för att visa de små individuella bilderna.

En sajt använder det här tricker för hastighet. Att hämta en enda stor bild är mycket snabbare i HTTP 1.1 än att hämta 100 små bilder.

Självklart har det sina nackdelar för de sidor sidor på sajten som bara vill visa en eller två små bilder och liknande. Det gör också att alla bilder rensas från cachen på samma gång istället för låta de mest använda ligga kvar där.

3.2 Inlining

Inlining är ett annat trick där man underviker att skicka enskilda bilder, och det gör man genom att använda data: URLer inbäddade i CSS-filen. Det har liknande nackdelar som i spriting-fallet ovan.

```
.icon1 {
    background: url(<data>) no-repeat;
}

.icon2 {
    background: url(<data>) no-repeat;
}
```

3.3 Concatenation

En stor sajt kan lätt hamna i en situation med väldigt många olika javascriptfiler. Frontendverktyg kan hjälpa utvecklarna att slå ihop varenda en av dem till en enda stor klump så att webbläsaren hämtar en enda stor fil istället för dussintals mindre filer. För mycket data skickas därmed när enbart lite behövs. För mycket data laddas om när en enda ändring behövs.

Den här övningen är förstås mest besvärlig för de involverade utvecklarna.

3.4 Sharding

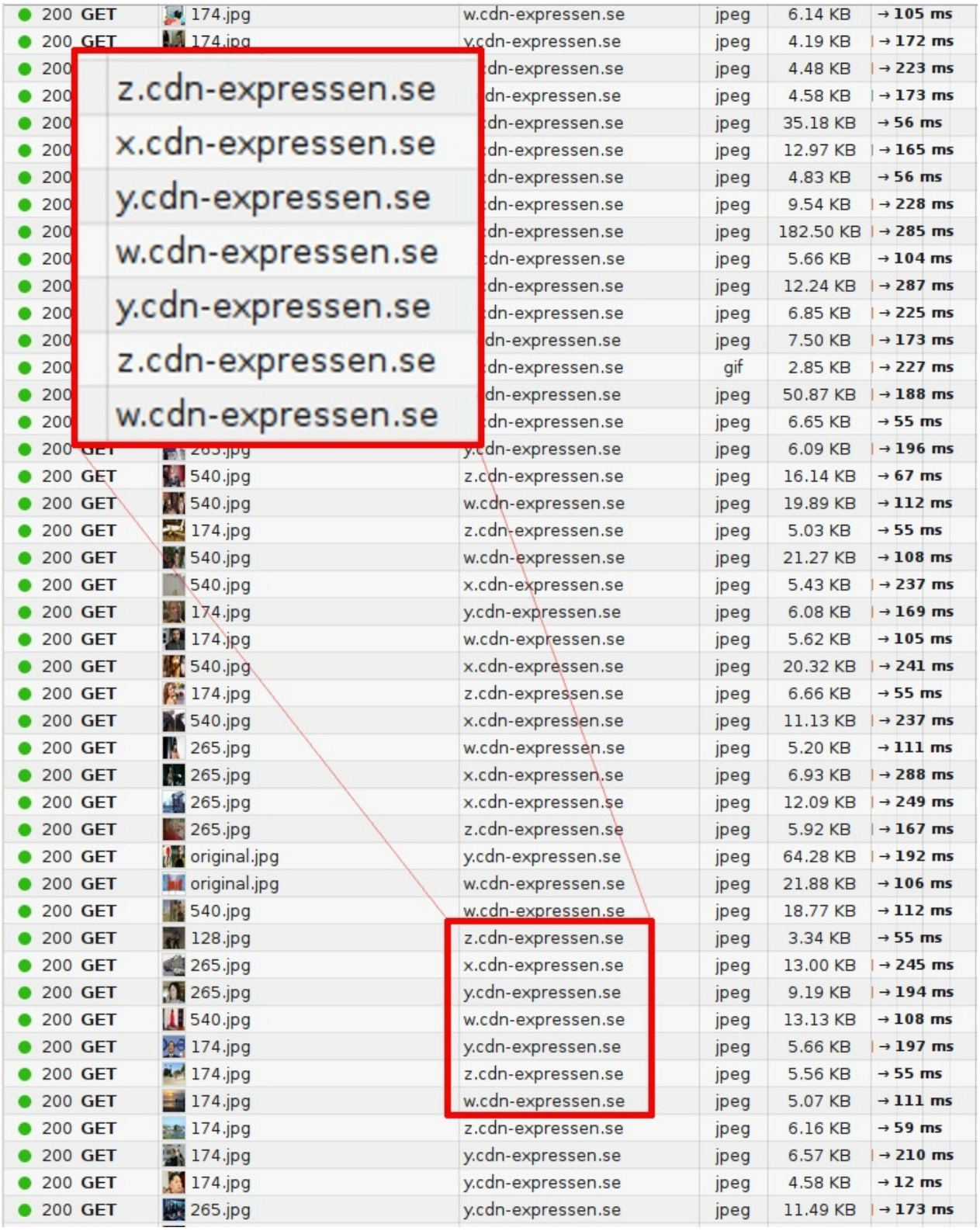
Det sista prestandatricket jag ska nämna kallas ofta för "sharding". Det är principen att hosta olika delar av din sajt från så många olika hostar som möjligt. Det kan förfalla konstigt vid en första anblick men det finns ett logiskt resonemang bakom.

Från början sade HTTP 1.1-specifikationen att en klient endast var tillåten att använda maximalt två TCP-koppel till varje host. Så, för att inte bryta mot specen uppfann smarta sajter nya host namn och - voilá - så kunde du få många fler koppel till din sajt och minska sidladdningstider.

Över tid har den gränsen tagits bort och dagens klienter använder lätt 6-8 koppel per hostnamn, men de behöver fortfarande ha någon gräns så sajter fortsätter att använda den här tekniken för att öka antalet koppel. Med ett ständigt ökande antal objekt (som jag visade tidigare) så måste ett än större antal koppel användas för att få HTTP att prestera bra och göra din sajt snabb. Det är inte ovanligt att enskilda sajter använder långt över 50 eller upp och förbi 100 koppel tack vare den här tekniken. Färsk statistik från httparchive.org visar att av de 300 000 mest populära URLerna i världen så behövs i genomsnitt 40(!) TCP-koppel för att visa sajten, och trendkurvan säger att det fortsätter växa.

En annan anledning att också lägga bilder och liknande resurser på ett separat hostnamn som inte använder cookies, är att storleken på cookies idag kan bli betydande. Genom att använda cookie-lösa bild-värdar så kan du ibland öka prestandan enbart genom att HTTP-förfrågningarna blir så mycket mindre!

Bilden nedan visar paket-spårning och hur det ser ut när en webbläsare besöker en av Sveriges toppsajter, och hur förfrågningarna är distribuerade över flera olika hostnamn.


z.cdn-expressen.se
x.cdn-expressen.se
y.cdn-expressen.se
w.cdn-expressen.se
y.cdn-expressen.se
z.cdn-expressen.se
w.cdn-expressen.se

4. Uppdatera HTTP

Skulle det inte vara trevlig att göra ett förbättrat protokoll? Det skulle inkludera...

1. Göra protokollet mindre fördröjningskänsligt
2. Fixa pipelining- och först-i-kön-blockeringsproblemen
3. Ta bort behovet och önskan att fortsätta öka antalet koppel per värddator
4. Behåll alla existerande gränssnitt, allt innehåll, URI-format och dess scheman
5. Gör det inom IETF:s HTTPbis-arbetsgrupp

4.1. IETF och HTTPbis-arbetsgruppen

Internet Engineering Task Force (IETF) är en organisation som utvecklar och främjar nya internetstandarder. Mest på protokollnivå. De är vitt kända för serien av RFC-dokument som täcker allt från TCP, DNS, FTP till "best practices", HTTP och mängder med protokollvarianter som aldrig kom någon vart.

Inom IETF formas arbetetsgrupper (working groups) med ett avgränsat arbetsområde som jobbar mot ett mål. De etablerar en stadga med ett antal riktlinjer och begränsningar för vad de ska åstadkomma. Alla som vill är tillåtna att delta i diskussionerna och utvecklingen. Alla som deltar och säger något har samma vikt och chans att påverka utgången, och alla är där som enskilda individer. Det läggs väldigt liten vikt vid vilka företag individerna jobbar för.

HTTPbis-gruppen (se nedan för en utvikning runt namnet) startades under sommaren 2007 för att jobba med en uppdatering av HTTP 1.1-specifikationen. Diskussionen om en nästa versions HTTP startade inom gruppen under slutet av 2012. Uppdateringen av HTTP 1.1 avslutades tidigt 2014 och resulterade i RFC 7320-serien.

Det sista interop-mötet för HTTPbis-gruppen hölls i New York City i början av juni 2014. De kvarvarande diskussionerna och avslutande IETF-procedurerna för att verkligen få till en officiell RFC skulle visa sig fortsätta in på det följande året.

Några av de allra största spelarna inom HTTP har saknats inom arbetsgruppens diskussioner och möten. Jag vill inte nämna eller peka ut något särskilt företags eller produktnamn här, men helt klart så verkar vissa aktörer på Internet vara väldigt säkra på att IETF kommer göra rätt även utan att de är inblandade.

4.1.1. Om "bis"-delen i namnet

Gruppen heter HTTPbis där "bis"-delen kommer från det latinska adverbet för två. Bis används ofta som suffix eller del av namnet inom IETF för en uppdatering en andra version av en spec. Precis som i fallet för HTTP 1.1.

4.2. http2 började från SPDY

SPDY är ett protokoll som utvecklades och togs fram av Google. De utvecklade det visserligen öppet och bjöd in alla som ville att delta, men det var tydligt att de utnyttjade sin situation med att kontrollera både en populär webbläsare och ett signifikant server-bestånd med välanvända tjänster.

När HTTPbis-arbetsgruppen bestämde att det var dags att börja jobba på http2 hade SPDY redan bevisats vara ett fungerande koncept. Det hade visat att det var möjligt att driftsätta på Internet och det presenterade siffror som bevisade hur bra det presterade. http2-arbetet började därmed från SPDY/3-utkastet som helt enkelt gjordes om till http2 draft-00 med lite sök och ersätt.

http2-koncept

Så vad skulle http2 åstadkomma? Vad var gränserna för vad HTTPbis-gruppen satte sig för att göra?

De var faktiskt ganska strikta och satte ett antal begränsningar för gruppens möjligheter att förnya protokollet.

- Det måste behålla HTTP:s paradigmer. Det måste fortfarande ett protokoll där klienten skickar förfrågningar till en server över TCP.
- http://- och https://-URLer kan inte ändras. Det kan inte göras ett nytt schema. Mängden innehåll som redan använder sådana URLer är helt enkelt för stor för att tro att de kan ändras.
- HTTP1-servrar och klienter kommer finnas kvar i decennier, vi måste kunna erbjuda proxys för http2-servrar.
- Därför måste proxys kunna mappa http2-funktioner till HTTP 1.1-klienter, ett till ett.
- Ta bort eller minska antalet valbara delar i protokollet. Det var inte riktigt ett krav men mer som ett mantra som följde med från SPDY och Google-teamet. Genom att se till att allting var obligatoriskt så finns det inte något sätt som man kan undvika att implementera allt nu och sedan upptäcka problemen i framtiden.
- Ingen mera fraktionsdel i versionsnumret. Det beslöts att klienter och servrar är antingen kompatibla med http2 eller så är de det inte. Om det kommer ett behov att utöka protokollet eller ändra saker så kommer http3 att skapas. Det finns inte något "minor version" i http2.

5.1. http2 för existerande URI-scheman

Som tidigare nämnts så kan inte existerande URI-scheman ändras, så http2 var tvunget att göras med de redan existerande. Eftersom de används för HTTP 1.x idag så behövde vi förstås ett sätt att uppgradera protokollet till http2 eller på något vis be servern använda http2 istället för äldre protokoll.

HTTP 1.1 har ett definierat sätt att göra detta på, nämligen genom Upgrade:-headern, som tillåter att server skickar tillbaks ett svar som använder det nya protokollet när den den fått en sådan förfrågan över det gamla. Till priset av en tur-och-retur runda.

Det priset av en tur-och-retur runda var något som SPDY-teamet inte kunde acceptera och eftersom de också implementerade SPDY över TLS utvecklade de ett nytt TLS-tillägg som används som en genväg för att förkorta förhandlingen ganska rejält. Genom att nyttja det tillägget, kallat NPN för Next Protocol Negotiation, kan servern berätta för klienten vilka protokoll den kan och klienten kan fortsätta med det protokoll den föredrar.

5.2. http2 för https://

En stor del av fokus för http2 har lags på att få det att bete sig ordentligt över TLS. SPDY används bara över TLS och det har varit en kraftigt tryck för att göra TLS obligatoriskt för http2, men det fick aldrig konsensus varvid http2 skeppas med TLS valbart. Hursomhelst, två prominenta implementatörer har tydligt sagt att de bara kommer implementera http2 över TLS: Mozilla Firefox-ledaren och ledaren för Googles team. Två av de ledande webbläsarna idag.

Anledningar att välja endast TLS inkluderar respekt för användarnas integritet samt att tidiga mätningar visar att nya protokoll har en mycket högre chans till att fungera när de görs över TLS. Det är på grund av den utspridda uppfattningen att allt som går över port 80 är HTTP 1.1 och det får en del mellan-boxar att blanda sig i och förstöra trafik när det faktiskt är något annat protokoll som pratas där.

Obligatorisk TLS är ett ämne som orsakat mycket handviftande och upprörda röster på mailinglistor och möten - är det bra eller är det ondska? Det är ett infekterat ämne - var medveten om detta när du kastar den här frågan i ansiktet på en HTTPbis-deltagare!

Likaså var det en hård och lång debatt om huruvida http2 skulle diktera en lista med chiffer som skulle vara obligatoriska när man använder TLS, eller om det kanske skulle vara en svartlista eller om det inte skulle kräva nånting alls från TLS-"lagret" utan lämna det till TLS-arbetsgruppen. Det som slutligen hamnade i specen är att TLS måste vara minst version 1.1 och det finns krav på vilka chiffer som måste användas.

5.3. http2-förhandling över TLS

Next Protocol Negotiation (NPN), är tillägget som användes i SPDY för att förhandla med TLS-servrar. Eftersom det inte var en riktig standard så togs det till IETF och igenom och det som kom ut blev ALPN: Application Layer Protocol Negotiation. ALPN är det som nu lyfts upp för att användas i http2, medan SPDY-klienter och -servrar fortsätter använda NPN.

Det faktum att NPN fanns först och att ALPN tog ett stund att gå igenom standardiseringsprocessen har lett till att flera tidiga http2-klienter och servrar är gjorda att använda båda dessa tillägg när de förhandlar http2. Vidare, eftersom NPN används för SPDY och många servrar ju stöder både SPDY och http2 så är det vettigt att stödja både NPN och ALPN på sådana servrar.

ALPN skiljer sig i huvudsak från NPN genom vet det är som bestämmer vilket protokoll som pratas. Med ALPN är det klienten som ger servern en lista med protokoll i den ordningen den föredrar att använda dem, och servern väljer det protokoll den vill ha, medan för NPN så är det klienten som gör det slutliga valet.

5.4. http2 för http://

Som tidigare nämnts i förbifarten, för klartext-HTTP 1.1 så är mekanismen att förhandla http2 att fråga servern med en Upgrade:-header. Om servern då pratar http2 svarar den med en "101 Byter" status och från då och framöver pratar den istället http2 på det kopplet. Du inser förstås att den uppgraderingsproceduren kostar en hel nätverks fram-och-tillbaka-tur men en fördel är att ett http2-koppel bör vara möjligt att hålla levande och återanvända i mycket högre grad än HTTP1-koppel generellt är.

Medan vissa webbläsares talespersoner har sagt att de inte kommer implementera det här sättet att prata http2, så sade Internet Explorer-teamet en gång i tiden att de skulle göra det - även om de sedan aldrig leverat det. curl och en del andra icke-webbläsarklienter stöder http2 i klartext.

Idag stöder ingen av de stora webbläsarna http2 utan TLS.

6. http2-protokollet

Nog om bakgrunden, historien och politiken bakom det som tog oss hit. Låt oss dyka ner i detaljerna i protokollet. De bitarna och koncepten som gör http2.

6.1. Binärt

http2 är ett binärt protokoll.

Bara låt det sjunka in en liten stund. Om du är en person som varit involverad i Internetprotokoll förr så är chansen stor att du helt instinktivt reagerar negativt mot det valet och plockar fram dina argument om hur protokoll baserade på text/ascii är överlägsna för att människor kan köra dem manuellt över telnet och så vidare.

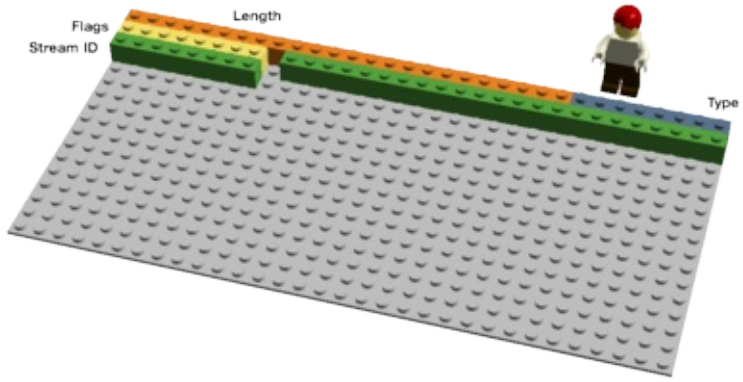
http2 är binärt för att göra inramningen av paket mycket enklare. Att lista ut början och slutet av paket är en av de riktigt komplicerade sakerna i HTTP 1.1 och faktiskt i text-baserade protokoll i allmänhet. Genom att plocka bort valbart antal white space och andra sätt att skriva samma sak så blir implementationer mycket enklare.

Dessutom gör det det mycket lättare att separera själva protokolldelarna från inramningen - vilket var förvirrande blandat i HTTP1.

Det faktum att protokollet har komprimering och ofta kommer köras över TLS minskar också värdet av text eftersom man inte skulle se text över kabeln i alla fall. Vi måste helt enkelt vänja oss vid tanken på att använda en Wireshark-inspector eller liknande för att lista ut exakt vad som pågår på protokollnivån i http2.

Debugging av det här protokollet kommer istället antagligen göras med verktyg som curl eller analyser av nätverksströmmen med Wiresharks http2-support och liknande.

6.2. Det binära formatet



http2 skickar binära paket ("frames"). Det finns olika paket-typer som kan skickas och de har allihop samma upplägg:

Längd, typ, flaggor, ström-id och paket-data ("payload").

Det finns tio olika paket-typer definierade i http2-specen och de två kanske mest fundamentala som direkt mappar HTTP 1.1-funktioner är DATA och HEADERS. Jag kommer beskriva några av paket-typerna i närmare detaljer nedan.

6.3. Multiplexade strömmar

Ström-id som nämndes i stycket ovan om det binära formatet, gör att varje paket som kommer över http2 är associerat med en "ström". En ström är en logisk förbindelse. En oberoende, bi-direktionell sekvens av paket som utbytes mellan klienten och servern inom ett http2-koppel.

Ett enda http2-koppel kan överföra många samtidiga strömmar mellan ändpunkterna genom att skicka paket från olika strömmar över kopplet. Strömmar kan etableras och användas ensidigt eller delas av både klienten och servern och de kan stängas av endera sidan. Ordningen som paketen skickas inom strömmen behålls, så mottagaren tar emot de i samma ordning som de skickades.

Multiplexande strömmar betyder att paket från många strömmar blandas och skickas över samma koppel. Två (eller fler) individuella tåg med data trycks ihop över ett enda koppel och delas upp igen på andra sidan. Här är två tåg:

De två tågen multiplexade över samma koppel:

6.4. Prioriteter och beroenden

Varje ström har också en prioritet (även känd som "vikt"), vilken används för att berätta för andra sidan vilka strömmar som skall anses viktigast ifall resursbrist tvingar servern att välja vilka strömmar som ska skickas först.

Genom att använda ett PRIORITY-paket kan en klient också berätta för servern vilken annan ström den beror på. Detta möjliggör för en klient att bygga ett "träd" med prioriteter där flera "barn-strömmar" kan bero att "föräldra-strömmar" först avslutas.

Prioritetsvikter och beroenden kan ändras dynamiskt under körning, vilket bör tillåta webbläsare att se till att när användare scrollar ner en sida full med bilder så kan den berätta vilka bilder som är viktigast, eller om du byter tabbar så kan den prioritera en ny uppsättning strömmar som då plötsligt kommer i fokus.

6.5. Header-komprimering

HTTP är ett state-löst protokoll. I korthet betyder det att varje förfrågan måste ta med sig precis så mycket detaljer som servern behöver för att serva den frågan utan att servern ska behöva mellanlagra info eller meta-data från tidigare förfrågningar. Eftersom http2 inte ändra några sådana paradigmer så måste det också fungera så.

Detta gör HTTP repetativt. När en klient frågar efter många resurser från samma server, typ bilder på en webbsida, så kommer det göras en lång serie med förfrågningar som ser nästan identiska ut. En serie med nästan identiska någonting ber om komprimering.

Allt medan antalet objekt per webbsida ökat som jag nämnt tidigare, har användandet av cookies och storleken på förfrågningarna också gjort det. Cookies behöver också inkluderas i alla förfrågningar, oftast helt identiska för många förfrågningar.

Storleken på HTTP 1.1-förfrågningar har faktiskt blivit så stora över tid att de ibland överstiger det initiala TCP-fönstret, vilket gör dem väldigt långsamma att skicka eftersom de då behöver en hel rund-tur för att få en ACK tillbaks från servern innan hela förfrågan har skickats. Ytterligare ett argument för komprimering.

6.5.1. Komprimering är ett lurigt ämne

HTTPS- och SPDY-komprimering befanns vara känsliga för BREACH- och CRIME-attackerna. Genom att infoga känd text i strömmarna och se hur det förändrade utdatan, kunde en attackerare lista ut vad som skickades.

Att komprimera dynamiskt innehåll för ett protokoll utan att bli sårbar för dessa attacker kräver lite omtanke och försiktiga övervägningnar. Det är vad HTTPbis-teamet försökte sig på.

In kommer då HPACK, Header- komprimering för http2, vilket – som namnet lämpligt indikerar - är ett komprimeringsformat speciellt framtaget för http2-headers och det är strikt talat specificerat i en egen specifikation. Det nya formatet, tillsammans med andra motåtgärder, som en bit som ber mellanhänder att inte komprimera en viss header och valbar utfyllnad av paket är tänkt att göra det svårare att exploatera den här komprimeringen.

Som Roberto Peons sade (en av skaparna av HPACK):

> "HPACK utformades för att göra det svårt för en korrekt implementation att läcka information, att koda och dekoda väldigt snabbt/billigt, att erbjuda kontroll över storleken på komprimerings-kontexten, att tillåta proxys att om-indexera (dvs ett delat tillstånd mellan frontend och backend inom en proxy), och för snabba jämförelser av huffman-kodade strängar."

6.6. Reset - ångra dig

En av nackdelarna med HTTP 1.1 är att när ett HTTP-meddelande har skickats iväg med en Content-Length av en viss storlek, så kan man inte enkelt bara sluta skicka det. Visst kan du ofta (men inte alltid) stänga TCP-kopplet men det kommer till priset av att du måste förhandla upp ett nytt TCP-koppel igen.

En bättre lösning skulle vara att bara stoppa meddelandet och skicka ett nytt. Det kan göras med http2:s RST_STREAM-paket som därmed hjälper till att undvika slösa på bandbredd och onödiga nedrivningar av koppel.

6.7. Server push

Detta är en funktion också kallad "cache push". Idén här är att ifall en klient ber om resurs X så kanske servern vet att klienten då troligen också kommer vilja ha resurs Z och skickar den till klienten utan att den har frågat efter den. Den hjälper klienten att stoppa in Z i sin cache så att den finns där direkt när klienten vill ha den.

Server push är något en klient explicit måste tillåta servern att skicka och även om en klient gör det, så kan den på eget beslut snabbt stänga ner en pushad ström med RST_STREAM ifall den inte vill ha den.

6.8. Flödeskontroll

Varje individuell ström över http2 har sin eget annonserade flödesfönster som den andra sidan är tillåten att sända data för. Det är väldigt likt hur SSH fungerar i stil och koncept, om du råkar veta hur det fungerar.

För varje ström måste båda sidor berätta för den andra hur mycket utrymme det finns att lagra inkommande data i, och den andra änden har bara tillåtelse att skicka så mycket data tills dess att fönstret utökas. Bara DATA-paket är flödeskontrollerade.

7. Utökningar

Protokollet kräver att en mottagare måste ta emot och ignorera alla paket som innehåller okända paket-typer. Två parter kan därmed förhandla om en användning av nya paket-typer på hop-by-hop basis (dvs överenskommelsen gäller endast mellan dessa två ändpunkter och inte längre än så), och de paket kan då inte ändra state och de kan inte flödeskontrolleras.

Frågan huruvida http2 skulle tillåta utökningar eller inte debatterades länge medan protokollet utvecklades med åsikter som svängde åt båda hållen. Efter draft-12 svängde pendeln tillbaks en sista gång och utökningar tilläts igen.

Utökningar är därmed inte del av det egentliga protokollen utan kommer dokumenteras utanför huvudspecen. Redan nu finns det två paket-typer som har diskuterats för att inkluderas i protokollet och som troligen tillhör de första typerna att skickas som utökningar. Jag beskriver dem här bara på grund deras popularitet och deras tidigare roll som "inhemska" typer.

7.1. Alternativa tjänster

När http2 antas, finns det anledning att misstänka att TCP-koppel kommer bli mycket långvariga och hållas levande mycket längre än vad HTTP 1.x-kopper någonsin hållits. En klient bör kunna göra mycket av vad den vill över ett enda koppel per varje host/sajt och det enda kopplet kan då potentiellt hållas uppe riktigt länge.

Detta påverkar hur HTTP-loadbalancerare arbetar och det kan komma situationer där en sajt vill annonsera och föreslå att klienten kopplar upp sig mot en annan host. Det kan vara för prestandans skull men även om en sajt håller på att tas ner för underhåll och liknande.

Servern kommer då skicka Alt-Svc:-headern (eller ALTSVC-paketet över http2) och berätta för klienten om en alternativ tjänst. En annan rutt till samma innehåll erbjudet av en annan tjänst, host och portnummer.

En klient är då tänkt att försöka koppla upp sig mot den tjänsten asynkront och bara använda alternativet ifall det fungerar bra.

7.1.1. Opportunistisk TLS

Alt-Svc-headern tillåter en server som tillhandahåller innehåll över http:// att informera klienten att samma innehåll även finns tillgängligt över ett TLS-koppel.

Detta är en någon omdiskuterad funktion. Sådana koppel använder o-autentiserad TLS och kommer inte visas som "säkra" någonstans, kommer inte använda något hänglås i gränssnittet eller faktiskt inte på något vis berätta för användaren att det inte är gammal hederlig HTTP. Men det är fortfarande opportunistisk TLS och en del människor är emot det konceptet väldigt starkt.

7.2. Blockad

Ett paket av den här typen är tänkt att skickas exakt en gång av en http2-part när denne har data att skicka men flödeskontroll förbjuder den att skicka data. Tanken är att ifall din implementation tar emot ett sådant paket så vet du att din implementation har strulat till någonting och/eller du får mindre än optimal överföringshastighet på grund av det.

Ett citat från draft-12, innan det här paketet togs ut och blev en utökning:

> “Paket-typen BLOCKED är med i den här draft-versionen för att erbjuda experimentering. Om resultaten av experimenten inte resulterar i positiv feedback kommer den tas bort.

8. En http2-värld

Så hur kommer saker fungera när http2 blir använt? Kommer det att användas?

8.1. Hur kommer http2 påverka vanliga människor?

http2 har ännu inte utbredd användning. Vi kan inte säga med säkerhet exakt hur saker kommer utvecklas. Vi har sett hur SPDY har använts och vi kan gissa och räkna baserat på det och andra experiment som gjorts tidigare.

http2 minskar antalet turer fram och tillbaks över nätverket och det undviker först-i-kön-blockeringsproblemen genom att multiplexa och att kunna avsluta oönskad strömmar snabbt.

Det tillåter ett stort antal parallella strömmar som i antal vida överstiger antal koppel host även de mest shardade sajterna av idag.

Med prioriteter använda ordentligt på strömmarna finns chansen att klient mycket bättre kommer kunna få den viktiga datan före den mindre viktiga datan. Allt sammantaget, skulle jag säga att chanserna är väldigt goda att det kommer leda till snabbare sidladdning och mer responsiva webbsajter. Kort sagt: en bättre webbupplevelse.

Hur mycket snabbare och hur mycket förbättringar vi kommer se tror jag inte vi kan säga ännu. Först och främst är ju teknologin fortfarnade väldigt ung och sen har vi knappt ens börjat se klienter och servrar trimma sina implementationer till att verkligen dra nytta av alla de krafter detta nya protokoll erbjuder.

8.2. Hur kommer http2 påverka webbutveckling?

Genom åren har webbutvecklare och webbutvecklarmiljöer samlat på sig verktygslådor fulla med trick och verktyg för att jobba runt problem med HTTP 1.1, precis som jag beskrev i början av det här dokumentet som en förklaring till varför http2 togs fram.

Många av de trick som verktyg och utvecklare nu använder per default och utan att tänka efter kommer troligen skada http2-prestanda eller åtminstone inte riktigt nyttja den fulla potentialen som http2 erbjuder. Spriting och inlining ska med största säkerhet inte användas med http2. Sharding kommer troligen att vara direkt skadligt för http2 som kommer tjäna på att ha färre koppel.

Ett problem här är ju att webbsajter och webbutvecklare behöver utveckla och drifta för en värld där det åtminstone på kort sikt kommer finnas både HTTP1.1- och http2-klienter som användare och att nå maximal prestanda för alla användare kan bli utmanande utan att drifta två olika front-ends.

Av enbart dessa anledningar tror jag det kommer ta en tid innan vi kommer nå http2s fulla potential.

8.3. http2-implementationer

Att försöka dokumentera specifika implementationer i ett dokument som det här är förstås helt futilt och dömt at misslyckas och kommer endast kännas gammalt redan inom kort. Istället kommer jag förklara situationen i bredare termer och hänvisa läsare till listan med implementationer på http2-sajten.

Det fanns ett stort antal implementationer redan tidigt och antalet har ökat under tiden vi jobbade med http2. När jag skriver detta finns det över 40 implementationer listade, och de flesta av dem implementerar den slutliga versionen.

Firefox har varit webbläsaren som varit först med support för de allra senaste versionerna av specen. Twitter har hängt med och erbjuder sina tjänster över http2. Google började under april 2014 att erbjudera http2-support på en del test servrar som kör deras tjänster och sedan maj 2014 har de erbjudit http2 support för deras utvecklingsversion av Chrome.

Microsoft har visat en "tech preview" med http2 support i deras nästa Internet Explorer-version. Safari och Opera har båda sagt att de kommer stöda http2.

curl och libcurl stöder osäker http2 likväl som TLS-baserad, användades en av flera olika TLS-bibliotek.

H2O, Apache Traffic Server och nghttp2 har alla släppt http2-kapabla open source-servrar.

8.3.1. Saknade implementationer

De två otroligt populära servrarna Apache HTTPD och Nginx har båda erbjudit SPDY support. Den 22:a september 2015 kom så Nginx med sin första version med officiell support för http2. Nginx har släppt "nginx-1.9.5" och http2-modulen för Apache heter mod_h2 och är på väg att släppas i en publik version "snart".

8.4. Vanlig kritik av http2

Under utvecklingen av det här protokollet så har debatten böljat fram och tillbaka och det är förstås ett visst antal människor som anser att protokollet till slut hamnade helt fel. Jag vill adressera några av de vanligaste klagomålen och nämna några argument mot dem.

8.4.1. "Protokollet är designat och gjort av Google"

Det har också varianter som implicerar att världen därmed blir ännu mer beroende och kontrollerad av Google genom detta. Det är inte sant. Protokollet har utvecklats inom IETF på samma sätt som protokoll har utvecklats i över 30 år. Men, vi alla erkänner och kan ju bara bekräfta Googles imponerande arbete med SPDY som inte enbart visade att det går att driftsätta ett nytt protokoll på det här sättet utan också tillhandahöll siffror som visade vilka vinster som kunde göras.

Google har annonserat att de kommer ta bort support för SPDY och NPN i Chrome under 2016 och de uppmanar servrar att migrera till http2 istället.

8.4.2. “Protokollet är endast användbart för webbläsare"

Det här är typ sant. En av de primära drivarna bakom utvecklingen av http2 var att fixa HTTP pipeliningen. Om ditt användningsfall inte har något behov av pipeliningen så finns det en sannolikhet att http2 inte kommer gör mycket bra för dig. Det är inte den enda förbättringen i protokollet men en stor sådan.

Så snart tjänster börjar använda den fulla kraften och möjligheterna med multiplexade strömmar över ett enda koppel så misstänker jag att vi kommer se fler applikationer använda http2.

Små REST-APIer och enklare programatiska användningar av HTTP 1.1 kommer kanske inte se att steget till http2 erbjuder väldigt stora fördelar. Men också att det är väldigt få nackdelar med http2 för de flesta användarna.

8.4.3. “Protokollet är bara bra för stora sajter"

Inte alls. Multiplexande strömmar kommer förbättra upplevelsen rejält för höglatensförbindelser som ofta mindre sajter utan bred geografisk distribution erbjuder. Stora sajter är redan väldigt ofta snabbare och mer distribuerade med kortare tur-och-returtider till användarna.

8.4.4. “Dess användning av TLS gör den långsammare"

Det kan vara sant till viss utsträckning. TLS-handskakningen lägger på lite extra men det finns redan pågående ansträngningar att reducera antalet tur-och-retur varv ännu mer för TLS. Den extra kostnaden för att använda TLS över kabeln istället för klartext är inte osignifikant och tydligt noterbar så mer CPU och kraft kommer användas för samma trafik mönster som ett osäkert protokoll. Hur mycket och vilken effect det får är ett ämne för tyckande och mätningar. Se till exempel istlsfastyet.com för en källa till sådan info.

Telecom och andra nätverskoperatörer, till exempel inom ATIS Open Web Alliance, säger att de behöver okrypterad trafik för att erbjuder cache, komprimering och andra tekniker som är nödvändiga för att tillhandahålla en snabb webbupplevelse över satelit, i flygplan och liknande. http2 gör inte TLS obligatoriskt så vi ska inte blanda ihop termerna.

Många Internet-användare har uttalat sina preferenser för att TLS ska användas mer utbrett och vi borde hjälpa till att skydda användarnas integritet.

Experiment har visat att genom att använda TLS så får man en högre grad av framgång än när man implementerar ny klartext-protokoll över port 80, eftersom det finns lite för många mellan-boxar där ute i världen som ingriper i det som de tror är HTTP 1.1 om det kommer över port och ibland kan se ut som HTTP.

Till slut, tack vare https multiplexande strömmar över ett fysikt koppel, så kommer vanliga användningsfall med webbläsare ändå göra väsentligt färre TLS-handskakningar och därmed prestera bättre än HTTPS skulle göra för HTTP 1.1.

8.4.5. “Att det inte är ASCII, förstör affären"

Ja, vi gillar att se protokoll i klartext eftersom det gör debuggning och spårning enklare. Men text-baserade protokoll är också mer felbenägna och öppnar upp för mycket mer parsning och fler parsningsproblem.

Om du verkligen inte kan hantera ett binärt protokoll, så kan du inte hantera TLS och komprimering i HTTP 1.1 heller och det har funnits där och används under en väldigt lång tid.

8.4.6. “Det är inte snabbare än HTTP/1.1”

Det är förstås ett ämne för debatt och diskussioner om hur man mäter och vad snabbare betyder, men redan under SPDY-dagarna gjordes många tester och som bevisade snabbare sidladdningar (som "How Speedy is SPDY?" av folk vid University of Washington och "Evaluating the Performance of SPDY-enabled Web Servers" av Hervé Servy) och såna experiment har repeterats med http2 också. Jag ser fram emot att få se fler såna tester och experiment publicerade. Ett grundläggande första test av httpwatch.com kan indikera att http2 håller sina löften.

8.4.7. “Den bryter mot lager-principer"

Seriöst, är det ditt argument? Lager är inte heliga oberörbara pelare i en global religion och om vi har klivit över inpå en del gråzoner när vi gjorde http2 så har det varit med avsikten att göra ett bra och effektivt protokoll inom de givna ramarna.

8.4.8. “Det fixar inte flera av HTTP 1.1s problem"

Det är sant. Med det specifika målet att behålla HTTP/1.1-paradigmer så vare det flera gamla HTTP funktioner som var tvugna att finnas kvar. Som t.ex headers och därmed också de ofta ogillade cookiesarna, authorization-headrar och mer. Men med fördelen att genom att behålla dessa paradigmer fick vi ett protokoll som det är möjligt att driftssätta utan att kräva en helt otänkbar mängd uppgraderingsarbete där fundamentala delar måste ersättas eller skrivas om. http2 är i grund och botten bara ett ny inramning ("framing layer").

8.5. Kommer http2 att driftssättas vitt och brett?

Det är för tidigt att säga säkert, men jag kan ändå gissa och estimera och det är vad jag kommer göra här.

Nej-sägarna kommer säga "kolla hur bra IPv6 har gjort ifrån sig" som ett exempel på ett protokoll som det tagit decennier bara för att börja bli driftsatt brett. http2 är inte en IPv6 dock. Det här är ett protokoll ovanpå TCP som använder vanliga HTTP-uppgraderingsmekanismer, portnummer och TLS etc. Det kommer inte kräva att de flesta routers eller brandväggar ändras alls.

Google bevisade för världen med deras SPDY-arbete att ett nytt protokoll som det här kan driftsättas och bli använt av webbläsare och tjänster med multipla implementaitoner inom en relativt kort tidsperiod. Medan antalet servrar på Internet idag som erbjuder SPDY är på 1%-nivån, så är mängden data dessa servrar hanterar mycket större. Några av de allra mest populära webbsajterna idag använder SPDY.

http2, baserat på samma principer som SPDY, skulle jag säga kommer sannolikt driftsättas ännu mer eftersom det är ett IETF-protokoll. SPDY-användningen hölls alltid tillbaks en del av dess "det är ett Google-protokoll"-stigma.

Det finns flera stora webbläsare bakom utrullningen. Representanter från Firefox, Chrome, Safari, Internet Explorer och Opera har sagt att de kommer skeppa http2-kapabla webbläsare och de har visat fungerande implementaitoner.

Det finns flera stora server-operatörer som är sannolika att erbjuda http2 snart, inklusive Google, Twitter och Facebook och vi hoppas få se att http2-support snart läggs till i populära server-implementationer som Apache HTTPD och Nginx. H2o är en ny vrålsnabb HTTP-server med http2-stöd som visar potential.

Några av de största proxy-tillverkarna, inklusive HAProxy, Squid och Varnish har uttalat sina intentioner att stödja http2.

Allt igenom 2015 har mängden http2-trafik ökat. I början av September visade användningen av Firefox 40 http2 i 13% av all HTTP-trafik och 27% av all HTTPS-trafik, medan Google ser http2 i ungefär 18% av sin inkommande trafik. Det kan noteras att Google driver andra experiment med nya protokoll (Se QUIC i 12.1) vilket ger lägre http2-nivåer än det annars kunde vara.

9. http2 i Firefox

Firefox har varit hack-i-häl med draftarna och har erbjudit http2-testimplemenationer under många månader. Under utvecklingen av http2-protkollet har klienter och servrar måste komma överens om vilken draft-version av protokollet de implementerat vilket har gjort det lite lätt irriterande att köra tester. Var bara medveten och kontrollera att din klient och server är överens om vilken protokoll-draft de implementerar.

9.1. Först, se till att det är påslaget

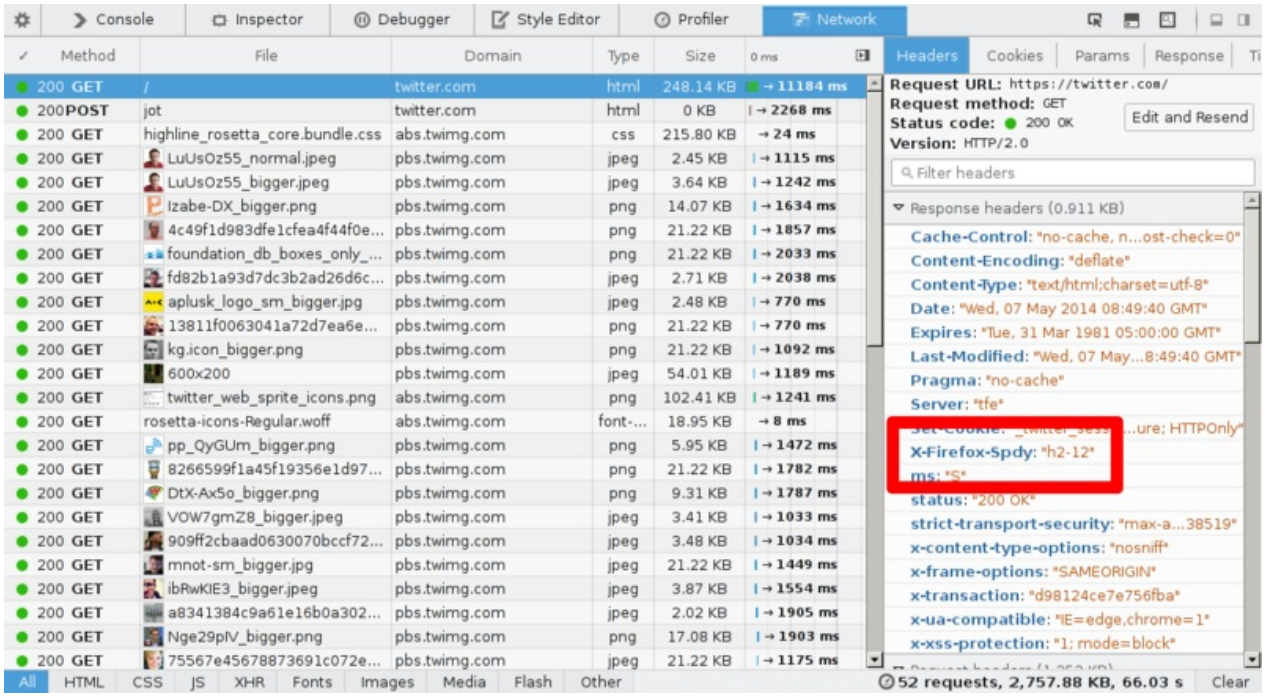
I alla Firefox-versioner sedan version 35, släppt 13:e januari 2015, är http2- support påslaget per default.

Skriv in 'about:config' i adress-fältet och sök efter ett alternativ som heter “network.http.spdy.enabled.http2draft”. Se till att den är att till *true* Firefox 36 lade till en annan config-variabel med namnet “network.http.spdy.enabled.http2” vilken är *true* per default. Den senare kontroller den "vanliga" http2-versionen medan den första slår på och av förhandling av http2-draftversioner. Båda är true per default sedan Firefox 36.

9.2. Bara TLS

Kom ihåg att Firefox bara implementerar http2 över TLS. Du kommer bara se http2 i aktion när du går till https://-sajter som erbjuder http2.

9.3. Transparant!



Det finns inget gränssnittselement någonstans som berättar att du pratar http2. Du kan helt enkelt inte enkelt se det. Ett sätt att lista ut det är att välja “Web developer->Network” och kontrollera svars-headrar och se vad du fick tillbaks från servern. Svaret är “HTTP/2.0”-någonting och Firefox stoppar in sin egen header som heter “X-Firefox-Spdy:” som kan ses på skärmdumpen ovan.

Headrarna du ser i nätverks-verktyget när du pratar http2 har konverterats från https binära format till gammaldags HTTP-1.x-liknande headers.

9.4. Se http2-användning

Det finns Firefox-tillägg som hjälper till att visualisera om en sajt använder http2. En av dem är “SPDY Indicator”.

10. http2 i Chromium

Chromium-teamet har implementerat http2 och stött det i dev- och beta-kanaler under en lång tid. Med början i Chrome 40, släppt den 27:e januari 2015, så är http2 påslaget per default för en viss mängd användare. Det började med ett fåtal och har sedan gradvis ökats.

SPDY-support kommer tas bort framöver enligt en bloggpost från projektet postat i februari 2015:

> "Chrome har stött SPDY sedan Chrome 6, men eftersom de flesta av fördelarna finns i HTTP/2, är det dags att säga adjö. Vi planerar att ta bort stödet för SPDY i början av 2016.

10.1. Först, se till att det är påslaget

Skriv in “chrome://flags/#enable-spdy4" i din webbläsares adressfält och klick “enable” om det inte redan säger att det är påslaget.

10.2. Endast TLS

Kom ihåg at Chrom bara implementerat http2 över TLS. Du kommer endast see http2 i aktion i Chrome när du går til https://-sajter som erbjuder http2-support.

10.3. Se HTTP/2-anvädning

Det finns Chrome-tillägg tillgängliga som hjälper att visualisera om en sajt använder http2. En av dem är “SPDY Indicator”.

10.4. QUIC

Chromes nuvarande experiment med QUIC (se sektion 12.1) späder ut http2-siffrorna något.

11. http2 i curl

curl-projektet har tillhandahållit experimentell support av http2 sedan september 2013.

I curls anda ämnar vi supporta varje aspekt av http2 som vi bara kan. curl används ofta som ett testverktyg och en utforskares sätt att peka på webbsajter och vi tänker fortsätta med det för http2 också.

curl använder det separata biblioteket nghttp2 för all funktionalitet i http2-lagret. curl kräver nghttp2 1.0 eller senare.

Notera att just nu skippas curl på linux inte alltid med http2-support påslaget.

11.1. HTTP 1.x-liknande

curl konverterar inkommande http2-headrar till HTTP 1.x-liknande headers och skickar dem till användaren, så att de kommer vara väldigt lika de i existerande HTTP. Det skapar en enkel övergång för vadsomhelst som använder curl och HTTP idag. På sammas sätt konverterar curl utgående headrar. Ge dem till curl i HTTP 1.x-stil och den gör om dem automatiskt när den pratar med http2-servrar. Det låter också användare att inte behöva bry sig så mycket om vilken specifik HTTP version som faktiskt används över kabeln.

11.2. Klartext, osäkert

curl stöder http2 över vanlig TCP mha Upgrade:-headern. Om du gör en HTTP-request och ber om http2, kommer curl be servern att uppdatera kopplet till http2 om det är möjligt.

11.3. TLS och vilka bibliotek

curl supportar en bred samling olika TLS-bibliotek för sin TLS-funktion, och det gäller även http2-stödet. Utmaningen med TLS för http2 är ALPN-stödet och i viss utsträckning stödet för NPN.

Bygg curl med en modern version av OpenSSL eller NSS för att få både ALPN- och NPN-stöd. Använder du GnuTLS eller PolarSSL får du ALPN-stöd men inte NPN.

11.4. Användning på kommandorad

För att säga åt curl att använda http2, antingen i klartext eller över TLS, så använder du dess `--http2` flagga (det är “minus minus http2”). curl använder fortfarande per default HTTP/1.1 så den extra optionen behövs när du vill ha http2.

11.5. libcurl-optioner

11.5.1 Slå på HTTP/2

Din applikation använder https:// eller http:// URLer precis som vanligt, men du sätter curl_easy_setopt-optionen `CURLOPT_HTTP_VERSION` till `CURL_HTTP_VERSION_2` för att få libcurl att försöka använda http2. Den kommer då göra sitt bästa för att använda http2, men annars fortsätta använda HTTP 1.1.

11.5.2 Multiplexande

Eftersom libcurl försöker behålla nuvarande beteende så mycket som möjligt måste du slå på http2 multiplexing för din applikation med CURLMOPT_PIPELINING-optionen. Annars kommer den fortsätta använda en request i taget per koppel.

En annan liten detalj att ha i åtanke är att ifall du ber om flera överföringar samtidigt med libcurl, mha dess multi-interface, är att en applikation kan mycket väl starta ett antal överföringar samtidigt och ifall du då hellre vill att libcurl ska vänta lite för att köra alla över samma koppel istället för att starta nya koppel för alla, så använder du CURLOPT_PIPEWAIT-optionen för varje individuell överföring du hellre vill ska vänta.

11.5.3 Server push

libcurl 7.44.0 och senare stöder HTTP/2 server push. Du kan dra nytta av den funktionen genom att sätta upp en push callback med CURLMOPT_PUSHFUNCTION-optionen. Om "pushen" accepteras av applikationen kommer den skapa en ny överföring som en curl easy handle och leverera data över den, precis som vilken annan överföring som helst.

12. Efter http2

En hel del tuffa beslut och kompromisser gjordes för http2. När http2 driftsätts finns det ett etablerat sätt att uppgradera till andra protokoll-versioner och det arbetet är ett fundament för att göra fler protokollrevisioner framöver. Det tar också in ett medvetande och en infrastruktur som kan hantera flera olika versioner parallellt. Kanske behöver vi inte fasa ut det gamla helt när vi introducerar nytt?

http2 har fortfarnade en massa gammal HTTP 1-"legacy" med sig i bagaget som det tar med sig in i framtiden pga sin önskan att göra det möjligt att köra proxies mellan HTTP1 och http2. En del av det legacyt begränsar framtida utveckling och innovationer. Kanske kan http3 klippa bort några av dem?

Vad tycker du fortfarande saknas i http?

12.1. QUIC

Googles QUIC (Quick UDP Internet Connections)-protokoll är ett intressant experiment, utfört mycket i samma stil och anda som de gjorde med SPDY. QUIC är en TCP + TLS + HTTP/2-ersättare, implementerat över UDP.

I QUIC skapas ny koppel med mycket mindre fördröjning, det löser paket-loss problemet så att det endast blockar individuella strömmar istället för alla som det gör för http2 och det gör det möjligt för koppel att lätt hållas över olika nätverksinterface - därmed täcker det också områden som MPTCP var menat att lösa.

QUIC är än så länge bara implementerat av Google i Chrome och deras servrar, och den koden är inte lätt att återanvända på andra ställen, även om det finns ett libquic som försöker precis det. Protokollet har skrivits ner som en Internet draft inom IETFs transportarbetsgrupp.

13. Fortsatt läsning

Om du tycker att det här dokumentet var lite tunt på innehåll eller tekniska detaljer så finner du ytterligare resurser att främja din nyfikenhet här nedan:

- HTTPbis-arbetsgruppens e-postlista samt dess arkiv: https://lists.w3.org/Archives/Public/ietf-http-wg/
- Den faktiska http2-specifikationen i HTML-version: https://httpwg.github.io/specs/rfc7540.html
- Firefox http2-nätverksdetaljer: https://wiki.mozilla.org/Networking/http2
- curls http2 implementationsdetaljer: https://curl.haxx.se/docs/http2.html
- http2s webbplats: https://http2.github.io/ samt möjligtvis just denna specifika FAQ-fråga: https://http2.github.io/faq/
- Ilya Grigoriks HTTP/2-kapitel i hans bok "High Performance Browser Networking": https://hpbn.co/http2/

14. Tack

Inspiration och paketformatbilden i Lego kommer från Mark Nottingham.

HTTP trenddata kommer från https://httparchive.org/.

RTT-graferna kommer från presentationer av Mike Belshe.

Mina barn Agnes och Rex för att dom låtit mig låna deras Legofigurer för "head of line"-bilden.

Tack till följande vänner för granskning och kommentarer: Kjell Ericson, Bjorn Reese, Linus Swälas och Anthony Bryan. Er hjälp har varit mycket uppskattad och har verkligen förbättrat dokumentet!

Under olika iterationer har följande vänliga människor rapporterat diverse felaktigheter och gjort förbättringar av dokumentet: Mikael Olsson, Remi Gacogne, Benjamin Kircher, saivlis, florin-andrei-tp, Brett Anthoine, Nick Parlante, Matthew King, Nicolas Peels, Jon Forrest, sbrickey, Marcin Olak, Gary Rowe, Ben Frain, Mats Linander, Raul Siles, Alex Lee, Richard Moore.