Das HTTP/2-Protokoll[1]

Prof. Dr. Stefan Zander

16. Februar 2018

[1] Ausgesuchte Einheit der Vorlesung "Entwicklung Webbasierter Anwendungen"

Lernziele:

- *Kennenlernen der Unterschiede zwischen HTTP/1.1 und HTTP/2*
- *Kennenlernen der mit HTP/2 eingeführten Neuerungen*
- *Kennenlernen der grundlegenden Konzepte von HTTP/2*

Inhaltsverzeichnis

---

1 Einführung

Die Ladezeit einer Webseite ist eines wenn nicht das wichtigste Kriterium einer guten Web Usability. Eine Ladezeit von < 1 Sekunde wird als Heiliger Gral angesehen, nach dem alle News-Sites, Onlineshops, Portale, Blogs und Landingpages suchen. Denn an der Ladezeit hängen sowohl Nutzerzufriedenheit als auch Konversionsraten und deshalb letztlich Traffic und Umsatz einer Seite. Der Einfluss ist so immens, dass beispielsweise Amazon bei einem Anstieg der Ladezeit um 0,1 Sekunden bereits ein Prozent an Umsatz verlieren würde [Witt, 2018].

Das Problem besteht insbesondere auch bei mobilen Webseiten, wie nachfolgendes Zitat verdeutlicht [Anderson, 2018]:

> *"The average time it takes to fully load a mobile landing page is 22 seconds, according to a new analysis.* ***Yet 53% of visits are abandoned if a mobile site takes longer than three seconds to load.*** *That's a big problem."*
>
> —Google

Google Research hat hierzu interessante Studien veröffentlich [An and Meenan, 2016]:

> *"Mobile sites lag behind desktop sites in key engagement metrics such as average time on site, pages per visit, and bounce rate. For retailers, this can be especially costly since 30% of all online shopping purchases now happen on mobile phones. The average U.S. retail mobile site loaded in 6.9 seconds in July 2016, but, according to the most recent data,* ***40% of consumers will leave a page that takes longer than three seconds to load****. And* ***79% of shoppers who are dissatisfied with site performance*** *say they're less likely to purchase from the same site again."*

Mit Hilfe eines neuronalen Netzwerks fand man heraus, dass die Hauptfaktoren, die sich für die Dauer des Seitenaufbaus verantwortlich zeichnen die

- Anzahl der Seitenelemente, sowie
- die Anzahl der enthaltenen Bilder

sind. Bei komplexen Seiten mit vielen Elementen dauert das Parsen des Seitencodes sowie der Aufbau des DOMs bedeutend länger. Bei Grafiken sollte man auflösungsoptimierte JPEGs anstatt PNG Formate verwenden.

Eine weitere Studie, welche die **Bounce rate** – also das Verlassen einer Seite ohne deren Inhalt genauer in Augenschein zu nehmen – untersucht hat, kommt zu folgendem Ergebnis [An, 2017]:

Die *Bounce rate* misst den prozentualen Anteil der Nutzer, die eine Web-Präsenz nach der Startseite wieder verlassen ohne die weiteren Inhalte in Augenschein genommen zu haben (*"[...]without exploring beyond the landing page."*).

> *"Recently, we trained a deep neural network—a computer system modeled on the human brain and nervous system—with a large set of bounce rate and conversions data. The neural net, which had a 90% prediction accuracy, found that as page load time goes from one second to seven seconds, the probability of a* ***mobile site visitor bouncing increases 113%****. Similarly, as the number of elements—text, titles, images—on a page goes from 400 to 6,000, the* ***probability of conversion drops 95%****."*

Weitere Interessante Einblicke einschließlich Maßnahmen zur Behebung von Geschwindigkeitsproblemen bietet die folgende Seite[2].

[2] `https://www.thinkwithgoogle.com/marketing-resources/experience-design/mobile-page-speed-load-time/`

Aufgrund solcher Ergebnisse ist es nicht verwunderlich, dass Unternehmen wie Google seit Jahren das Thema Webperformance vorantreiben und sich die **Ladezeit einer Seite sogar auf ihre Platzierung in den Google-Suchergebnissen auswirkt**.

Technisch gesehen hängt die Ladezeit einer Seite an drei zentralen Faktoren:

1. an der Verarbeitung im Server,

2. an der Übertragung über das Netzwerk, und
3. an der Darstellung im Browser.

Alle drei können einen starken Einfluss auf die Performance haben und die Ladezeitoptimierung zu einem langen und komplexen Prozess machen.

Einen dieser drei Aspekte – die Übertragung der Webseite vom Server zum Browser – konnten Entwickler bislang kaum beeinflussen. Seit 1999 ist das Hypertext Transfer Protocol HTTP in der Version 1.1 der Standard zur Datenübermittlung im Web. Nach über 15 Jahren gibt es nun mit Version 2 (auch HTTP/2 oder kurz h2 genannt) den langersehnten Nachfolger.

Nur zur Erinnerung: im Jahre 1999 war der bedeutendste Hersteller für Mobiltelefone ein Unternehmen aus Finnland mit dem Namen Nokia. Der Präsident der USA hieß Bill Clinton. Apple lieferte seine Rechner mit fest verbauten Bildröhren aus, und deutsche Kunden bezahlten dafür damals nicht in Euro, sondern in D-Mark. Quelle: [Weinschenkler, 2017]

2 *HTTP/2*

HTTP/2 ist ein Binärprotokoll, setzt einen klaren Fokus auf Performance und gibt Entwicklern verschiedene Möglichkeiten zur Optimierung der Seitenladezeit an die Hand. Version 2 des Hypertext Transfer Protokolls wurde im Mai 2015 als Standard verabschiedet. Das Ziel von HTTP/2 ist, die Bandbreite im World Wide Web besser auszunutzen und die Netzwerk-Latenz zu verringern. Hierzu beinhaltet HTTP/2 einige zentrale Optimierungen gegenüber HTTP/1.1, wie insbesondere **Request Multiplexing**, **Header Compression**, **Resource Prioritization**, und **Server Push**[3]. Request Multiplexing und die Kompression der Headerinformationen verbessern die Ladezeit einer Seite unmittelbar. Um die anderen beiden Optimierungen zu nutzen, ist ein gewisser Entwicklungsaufwand erforderlich.

[3] Siehe Kapitel 2.1

2.1 *Semantik und Bestandteile*

Die Semantik des HTTP-Protokolls bleibt mit Version 2 unberührt; die bekannten HTTP-Schlüsselwörter wie `GET`, `POST`, `PUT`, `DELETE` oder `OPTIONS` bleiben erhalten. Auf der Anwendungsschicht ändert sich nichts. Neu ist die Abwicklung des Transports der Header und Nutzdaten. HTTP-Requests und -Responses sind in **Streams**, **Messages** und **Frames** kodiert. Das HTTP/2-Protokoll besteht aus folgenden Bausteinen:

- **Stream**: Ein bidirektionaler Austausch von Daten zwischen Server und Client, der aus einer oder mehreren Messages besteht. Die Übertragung eines Streams erfolgt innerhalb einer TCP-Verbindung. Mehrere Streams lassen sich simultan über dieselbe TCP-Verbindung übertragen.
- **Message**: Eine komplette Abfolge von Frames, die zu einer logischen Message gehören. Eine Message gehört zu einem Stream.
- **Frame**: Die kleinste Kommunikationseinheit innerhalb von HTTP/2. Sie enthält binär kodierte Header- oder Nutzdaten.

Das Zusammenspiel dieser Bausteine kann wie folgt beschrieben werden:

- Die gesamte Kommunikation zwischen einem Client und dem Server wird über eine TCP-Verbindung abgehandelt, welche eine nahezu beliebige Anzahl an bidirektionalen Streams beinhalten kann.
- Jeder Stream besitzt einen eindeutigen Identifier und optionale Prioritätsinformationen
- Jede Message ist eine logische HTTP-Request oder Response Nachricht, welche aus einem oder mehreren Frames besteht.
- Ein Frame ist die kleinste Kommunikationseinheit über die Daten wie HTTP Header, Message Payload (Nachrichteninhalt) etc. transportiert werden.

Zusammengefasst unterteilt HTTP/2 die HTTP Protokollkommunikation in einen Austausch binär-kodierter Frames, welche Nachrichten zugeordnet sind die zu einem bestimmten Stream gehören.

Abbildung 1 zeigt eine beispielhafte Client-Server-Kommunikation (übernommen von [Weinschenkler, 2017]). Der komplette Nachrichtenaustausch läuft über genau eine TCP-Verbindung ab. Die dabei versendeten sechs Frames werden in nachfolgender Auflistung erläutert:

1. Der Client beginnt, indem er einen Request für die Datei `index.html` per `HEADERS`-Frame an den Server schickt. Dafür eröffnet er einen neuen Stream – einen in sich abgeschlossenen Nachrichtenaustausch mit dem Server –, der die `ID 41` erhält. Der Client setzt weiterhin zwei Flags in dem Frame. `+END_HEADERS` bedeutet, dass der Client im aktuellen Stream keine weiteren `HEADERS`-Frames mehr versenden wird. `+END_STREAM` bedeutet, dass der Client nicht beabsichtigt, weitere Frames im aktuellen Stream zu versenden.
2. Der Server kann die Datei `index.html` ausliefern und schickt innerhalb desselben Streams `41` ebenfalls einen `HEADERS`-Frame mit dem Statuscode `200` an den Client zurück. Der Code ist bereits aus HTTP/1.1 bekannt und besagt, dass die Datei verfügbar ist und der Client sie demnächst erhalten wird. Neu ist, dass der Server die eigentliche Nutzlast in Form der Datei `index.html` mit einem separaten `DATA`-Frame verschickt.
3. Der Server schickt einen `PUSH_PROMISE`-Frame, eine Neuerung, mit der er von sich aus eine Datenübertragung an den Client initiiert. Im Fall des Beispiels von Abbildung 1 antizipiert der Server, dass der Client direkt nach seinem Request für die Datei `index.html` die Datei `style.css` benötigen wird. Deshalb bietet er an, diese Datei über den neuen Stream `42` an den Client zu versenden.

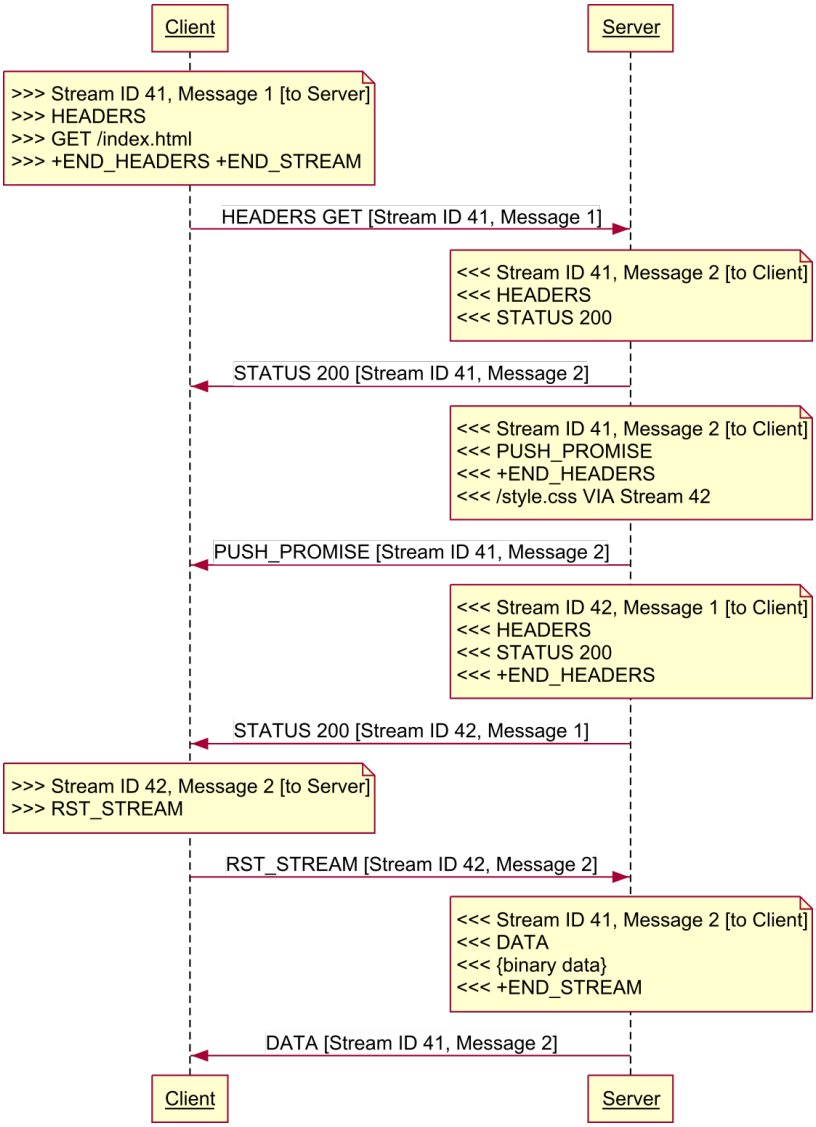


Abbildung 1: Beispielhafte Darstellung eines HTTP/2-Nachrichtenaustauschs zwischen Browser und Server [Weinschenkler, 2017].

4. Wie bei Punkt 2 schickt der Server nun im Stream `42` einen `HEADERS`-Frame mit dem Statuscode `200` bezogen auf die Datei `style.css` an den Client. Letzterer kündigt die baldige Übertragung der Datei an. Ohne weiteren Widerspruch des Clients würde als nächstes im Stream `42` die Übertragung eines `DATA`-Frame mit der `style.css` erfolgen.

5. Im Beispiel benötigt der Client die Datei `style.css` nicht, weil er noch eine Kopie davon in seinem Cache zur Verfügung hat. Er sendet deshalb im Stream `42` einen `RST_STREAM`-Frame. Damit schließt er den Stream `42`, und der Server wird darüber keinerlei Daten mehr versenden.

6. Bleibt noch die eingangs durch den Client im Stream `41` angefragte Datei `index.html`, die der Server nun in einem `DATA`-Frame an den Client versendet. Damit beendet der Server seinerseits die Kommunikation im Stream `41`.

2.2 *Übersicht der Neuerungen*

Wie eingangs beschrieben nutzt HTTP/2 die bestehende Bandbreite bei der Datenübertragung besser aus als seine Vorgänger. Das erreicht der neue Standard durch vier Neuerungen.

1. **Request Multiplexing**
   Das Request Multiplexing sorgt dafür, dass mehrere Request-/Response-Konversationen gleichzeitig über ein und dieselbe TCP-Verbindung erfolgen können.

2. **Server Push**
   Das Konzept des Server Push ermöglicht, dass ein HTTP-Client einen HTTP Request in Richtung Server absetzen und Letzterer darauf mit mehr als einer Response antworten kann.

3. **Header Compression**
   HTTP/2 verwendet eine verbesserte Implementierung zur Datenkompression speziell für die Protokoll-Metadaten – den HTTP Header.

4. **Resource Priorization**
   Beschreibung folgt...

Diese Vorlesungseinheit gibt einen Überblick über die zentralen Ideen und Techniken von HTTP/2 und setzt sie in Bezug zu anderen Techniken zur Steigerung der Webperformance.

2.3 *Request Multiplexing*

HTTP/1.1 bietet keine direkte Möglichkeit, Requests zu parallelisieren. Über eine TCP-Verbindung kann jeweils nur ein HTTP-Request

HTTP/1.1 erlaubt jeweils nur eine Request-Response-Konversation pro TCP-Verbindung

und daraufhin nur eine HTTP-Response fließen. Das ist für zeitgemäße Webseiten problematisch. Die Startseiten großer Online-Portale erfordern nicht selten den Transfer mehrerer Megabyte an Daten vom Server an den Client. Um diesen Download in Gang zu setzen und abzuschließen, sind mitunter hunderte von HTTP-Requests notwendig.

HTTP/1.1 erlaubt max. 6 parallele Verbindungen zum Server

Um diese Limitierung zu umgehen öffnen Browser üblicherweise sechs parallele Verbindungen, sodass **maximal sechs Ressourcen gleichzeitig übertragen** werden können. Alle übrigen Requests müssen auf eine freie Verbindung warten; dieser Flaschenhals wird **Head of Line Blocking** genannt und ist in Abbildung 3 (linke Hälfte) dargestellt.

Head of Line Blocking

Workaround 1: Aufbau mehrerer paralleler Verbindungen zum Server

Typische Workarounds waren bisher, dass Webbrowser **mehrere TCP-Verbindungen** zum Server aufbauen, über die sich parallel mehrere HTTP-Verbindungen abwickeln lassen. Der dadurch entstehende Protokoll-Overhead fällt jedoch vergleichsweise stark ins Gewicht, da jede TCP-Verbindung einen initialen Aufbau erfordert. Dieser notwendige Nachrichtenaustausch für den Verbindungsaufbau zwischen Client, DNS-Service und Server ist in Abbildung xxx dargestellt

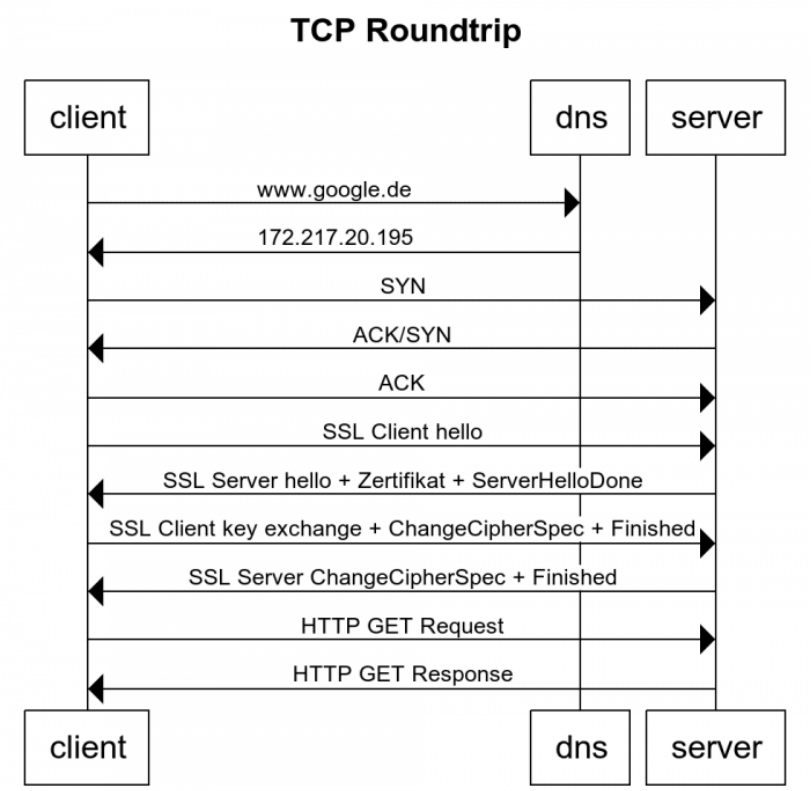


Abbildung 2: Notwendiger Nachrichtenaustausch für den Aufbau einer TCP-Verbindung [Weinschenkler, 2017].

Workaround 2: Reduktion der HTTP/1.1-Requests durch Zusammenfassung von Ressourcen

Ein weiterer Workaround war das Verringern von HTTP-Requests durch das **Zusammenfassen von Ressourcen**. Entwickler kombinierten viele Bilder zu einer großen Grafik und zeigten über *CSS-Spriting* jeweils einzelne Ausschnitte an. Einzelne CSS- und Javascript-Dateien

fassten sie ebenfalls zu größeren Dateien zusammen. Der Nachteil dieser Verfahren ist, dass die einzelnen Ressourcen, also die Bilder, CSS- und Javascript-Dateien, sich nicht mehr separat im Cache des Clients verwahren lassen. Eine Änderung an einer einzelnen kleinen Bilddatei erfordert einen kompletten Neubau eines Sprites auf der Serverseite. Der Server muss das Sprite anschließend komplett neu an den Client übertragen, obwohl sich nur ein kleiner Teil verändert hat. Dasselbe gilt für zusammengefasste CSS- und JavaScript-Dateien.

Workaround 3: Verteilung von Ressourcen auf mehrere Hostnamen

Wie eingangs erwähnt erlauben Webbrowser-Implementierungen nur eine begrenzte Anzahl paralleler TCP-Verbindungen zum selben Hostnamen. Je nach Browser ist diese Höchstzahl unterschiedlich, aber meist sind es weniger als zehn. Ist also der Download einer HTML-Datei, mehrere Bilder und weiterer Ressourcen notwendig, umging man die Begrenzung durch das Verteilen der Ressourcen auf mehrere Hostnamen:

- Die HTML-Datei liegt auf `www.site.de`. Das ist der Hostname, den der Besucher in der Adresszeile des Browsers eingibt.
- Bilder bietet der Server `img.site.de` an. Bei einer großen Zahl an Bildern ist es üblich, weitere Hosts wie `img2.site.de` und `img3.site.de` einzusetzen.
- CSS- und Javascript-Dateien liegen auf `static.site.de`.

Entwickler und Administratoren betreiben damit einen gewaltigen Aufwand, um Einschränkungen eines Protokolls zu umgehen, das aus dem Jahre 1999 stammt.

HTTP Pipelining

Hinzu kommt ein weiteres Problem, das man durchaus als Konstruktionsfehler bezeichnen kann: In HTTP/1.1 lassen sich zwar mehrere Request-Response-Konversation über eine TCP-Verbindung abwickeln, aber nur sequenziell. Die Bezeichnung für dieses Vorgehen lautet **HTTP Pipelining**. Es erzwingt, dass der Server über eine TCP-Verbindung zu einem Zeitpunkt nur eine Response an einen Client übertragen kann.

Sendet der Client mehrere `GET`-Requests über die Pipeline zum Server, kann er die zugehörigen Responses nur sequenziell in der Reihenfolge der ursprünglichen `GET`-Requests empfangen und verarbeiten. Wenn die Response auf den ersten `GET`-Request viel Zeit benötigt, blockiert sie alle nachfolgenden Responses, die nicht vor der ersten Response beim Client ankommen können. Das Problem heißt auch **Head-of-Line-Blocking**.

3 HTTP/2 Features

3.1 Request Multiplexing

HTTP/2 bringt zunächst eine harte Einschränkung bezüglich der Verbindungsressourcen. Pro Gegenstelle, also pro Server, darf der Client **genau eine TCP-Verbindung** öffnen. Das klingt zunächst nach einer Verschlechterung gegenüber HTTP/1.1, bei dem das Öffnen mehrerer TCP-Verbindungen die Grundlage der parallelisierten Datenübertragung war. Das neue Protokoll bringt jedoch die entscheidende Verbesserung des Multiplexing. Über die eine zulässige TCP-Verbindung lassen sich simultan beinahe beliebig viele HTTP-Konversationen, sogenannte *Streams*, zwischen Client und Server abwickeln. RFC 7540 empfiehlt, die Anzahl simultaner Streams pro TCP-Verbindung nicht unter einen Wert von 100 zu konfigurieren, um den Parallelisierungsgrad nicht zu sehr einzuschränken [Weinschenkler, 2017].

In der Konsequenz sind alle zuvor genannten Workarounds hinfällig, die mit HTTP/1.1 nötig waren. Das Verwenden mehrerer TCP-Verbindungen ist nicht mehr möglich und das Zusammenfassen von Bild-, Skript- und CSS-Dateien nicht mehr nötig. Ebenso sind unterschiedliche Hostnamen auf der Serverseite zum Ermöglichen der Parallelisierung überflüssig.

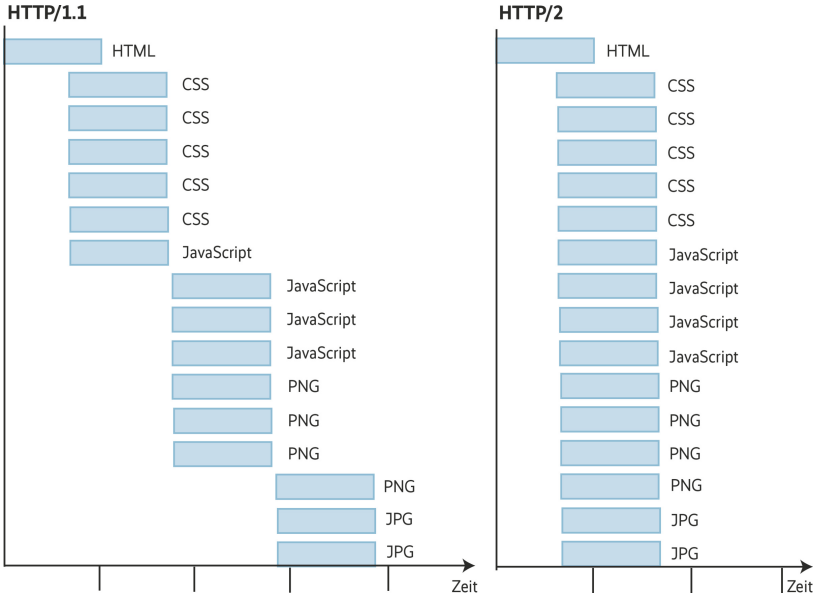


Abbildung 3: Das Zeitverlaufsdiagramm zeigt, wie der Browser bei HTTP/1.1 nach dem Download der HTML-Seite sechs Requests parallel stellt. Die weiteren Requests werden so lange geblockt, bis diese beantwortet sind (*Head of Line Blocking*). Mit HTTP/2 kann der Browser alle Ressourcen gleichzeitig anfragen, sodass sich die Ladezeit verringert [Witt, 2018].

Bei HTTP/2 gibt es diese Limitierung nicht mehr. Alle Ressourcen werden über eine einzige Verbindung geladen und der Browser kann beliebig viele Requests parallel stellen. Dieses Multiplexing umgeht mehrere Probleme:

1. Erstens wird die anfängliche Durchsatzbegrenzung einer neuen TCP-Verbindung durch den Slow-Start-Algorithmus schneller überwunden.

2. Zweitens fällt bei nur einer Verbindung der langsame Verbindungsaufbau per SSL weniger ins Gewicht. Indem HTTP/2 alle Requests in einer Verbindung bündelt, wird so schnell die maximale Bandbreite des Netzwerks genutzt.

Im Zeitverlaufsdiagramm einer Webseite (siehe Abbildung 3) ist der Unterschied zwischen HTTP/1.1 und HTTP/2 dadurch besonders gut zu erkennen. Der Ladezeitrückgang durch Multiplexing variiert von Seite zu Seite, ist aber in vielen Fällen substanziell [Witt, 2018].

—-START—-

—-END—-

3.2 Header Compression

Bei jedem Request des Browsers werden Metainformationen wie Version und Größe der Ressource als HTTP-Header übertragen. Besonders viele dieser Metainformationen senden Seiten, auf denen Cookies zum Einsatz kommen.

Bei HTTP/1.1 werden diese Header bei jedem Request vollständig und unkomprimiert übertragen[4]. Insbesondere bei Cookies ist die Datenübertragung massiv redundant. Mithilfe des *HPACK-Algorithmus*[5] komprimiert HTTP/2 die übertragenen Header durch ein einfaches Wörterbuchverfahren im Schnitt um 88% [Witt, 2018]. Der Dienst `cloudflare.com` beispielsweise berichtet von durchschnittlich 30% reduzierten Headergrößen durch den Einsatz von HTTP/2 Header Compression [Krasnov, 2016]. Der Einfluss auf die Ladezeit macht sich vor allem bei geringer Bandbreite bemerkbar, sodass mobile Nutzer besonders stark profitieren.

[4] Siehe Kapitel zu Cookies und Sessions

[5] Siehe `https://http2.github.io/http2-spec/compression.html`

Mobile NutzerInnen profitieren besonders von Header Compression

Der HPACK-Algorithmus arbeitet zweistufig:

1. Für 61 bekannte Header existiert ein fest definiertes, statisches Kompressionswörterbuch. Header-Namen, teilweise sogar in Kombination mit häufig vorkommenden Werten, werden als eine einfache Zahl kodiert – beispielsweise der Header :method:GET[i] auf den Wert 2, die Kombination [i]:method:POST auf 3. Der Header-Name :referer: entspricht dem Wert 51. Auf die Weise lassen sich HTTP-Header auf bis zu ein Byte reduzieren.

2. Für Header, die nicht Teil des statischen Kompressionswörterbuchs sind, handeln Client und Server dynamisch zum Zeitpunkt der Verbindungsaufnahme ein weiteres, dynamisches Wörterbuch aus, das nur für die aktuelle Verbindung gültig ist.

HPACK verringert den Umfang der Header-Daten dramatisch. Die Standard-Header zeichnen sich von Natur aus durch eine hohe Redundanz aus. Die Header der überwiegenden Mehrheit aller HTTP-Verbindungen beginnt mit :method:GET[i]. Weiterhin sind die

Header [i]:content-type:application/html sowie :accept:application/html Teil nahezu jeder Kommunikation zwischen Webbrowser und -server. HPACK verkürzt die zu übertragende Datenmenge auf jeweils ein Byte.

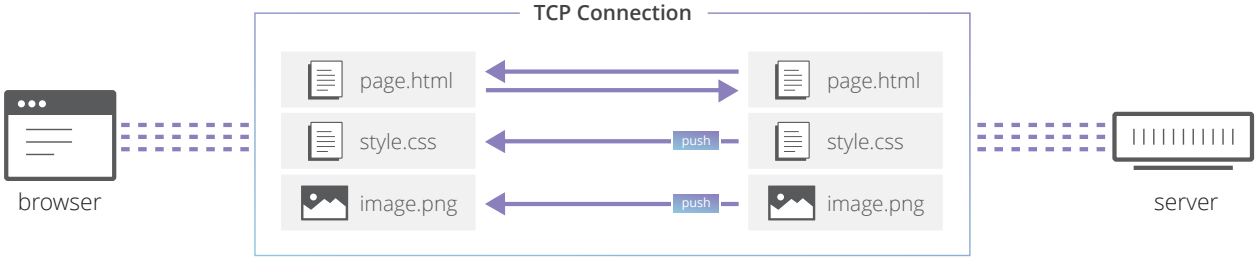
3.3 *Resource Prioritization*

Manche Ressourcen sind wichtiger für die Darstellung einer Webseite als andere. So müssen Style-Informationen in Form von CSS-Dateien schneller geladen werden als Bilder, die sich weit unten auf der Seite befinden. Die Priorisierung von Ressourcen hat daher einen erheblichen Einfluss darauf, wie schnell der Browser beginnen kann, die Seite anzuzeigen.

Browser sind bereits sehr gut darin, HTML-Dateien zu analysieren und die dort referenzierten Ressourcen zu priorisieren. Zusätzlich kann der Entwickler beispielsweise mit Preload-Tags dem Browser eine Reihenfolge zum Laden der Ressourcen vorgeben. Allerdings stellt sich dabei die Frage: Wie kann der Browser sicherstellen, dass die Ressourcen auch in der priorisierten Reihenfolge bei ihm ankommen?

3.4 *Server Push*

Server Push erlaubt Servern benötigte Ressourcen proaktiv und opportunistisch auszuliefern.

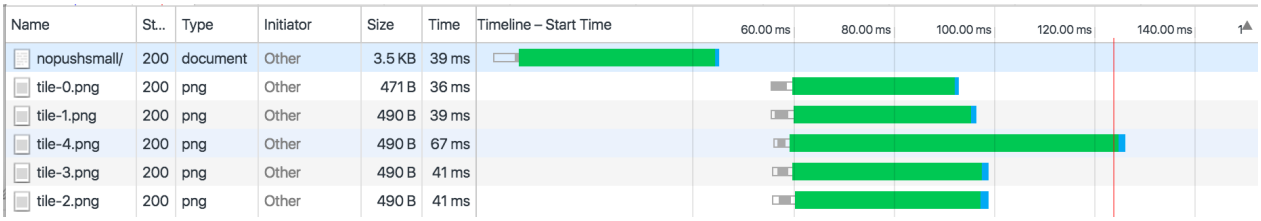


Single TCP Connection, Single HTTP Request

Abbildung 4: Schematische Darstellung der Verarbeitung von HTTP/2 Server Push [Krasnov, 2016].

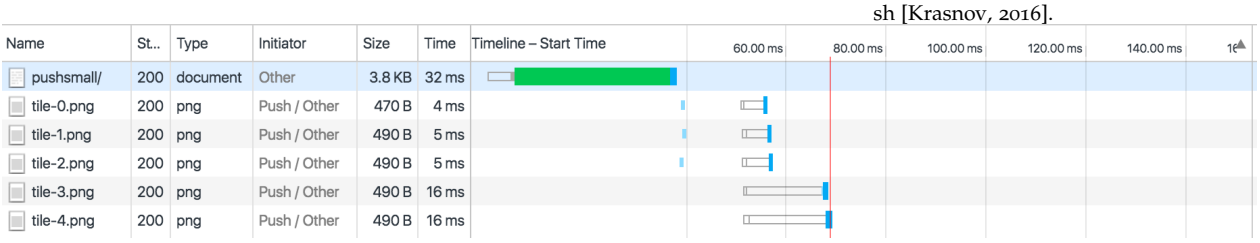
Abbildung xxx zeigt die Browser-seitige Verarbeitung einer Demonstrationswebseite auf der neben der Hauptseite fünf Grafiken enthalten sind – je einmal ohne Server Push (Abbildung xxx) und einmal mit (Abbildung xxx). Mittels sog. Profiling Tools lässt sich der Geschwindigkeitszuwachs sehr schön visualisieren und benchmarken.

Wie in Abbildung 5 zu sehen ist, initiiert der Browser im Anschluss an das Laden der Hauptseite einen weiteren Request für die fünf enthaltenen Grafiken. Nachdem der Server den erneuten Request verarbeitet hat werden diese anschließend an den Browser ausgeliefert und von diesem angezeigt.



Name	St...	Type	Initiator	Size	Time	Timeline – Start Time
nopushsmall/	200	document	Other	3.5 KB	39 ms	
tile-0.png	200	png	Other	471 B	36 ms	
tile-1.png	200	png	Other	490 B	39 ms	
tile-4.png	200	png	Other	490 B	67 ms	
tile-3.png	200	png	Other	490 B	41 ms	
tile-2.png	200	png	Other	490 B	41 ms	

Abbildung 5: Verarbeitung einer Seite mit fünf Grafiken ohne Server Push [Krasnov, 2016].



Name	St...	Type	Initiator	Size	Time	Timeline – Start Time
pushsmall/	200	document	Other	3.8 KB	32 ms	
tile-0.png	200	png	Push / Other	470 B	4 ms	
tile-1.png	200	png	Push / Other	490 B	5 ms	
tile-2.png	200	png	Push / Other	490 B	5 ms	
tile-3.png	200	png	Push / Other	490 B	16 ms	
tile-4.png	200	png	Push / Other	490 B	16 ms	

Abbildung 6: Verarbeitung der Beispielseite mit Server Push [Krasnov, 2016].

Mit aktiviertem Server Push werden die Grafiken noch während der Client-seitigen Verarbeitung der Seite automatisch durhc den Server ausgeliefert; eine zusätzliche Anfrage ist demnach nicht notwendig und spart Zeit. Zusammen mit der automatischen Auslieferung der Grafiken schickt der Server ein `PUSH_PROMISE` Frame[6] an den Browser. Sobald der Browser diese benötigt bzw. verarbeiten kann gleicht er diese mit dem `PUSH_PROMISE` Frame ab und kann diese sofort ohne einen weiteren Request nutzen.

[6] Durch **Push Promisies** signalisiert der Server dem Browser einen `PUSH` von zu einer Seite gehörenden Ressourcen. Dieses Frame wird immer automatisch am Beginn eines `PUSH` gesendet.

—- START —-

Der beispielhafte Ablauf aus dem letzten Abschnitt zeigt die simultane Übertragung von zwei Streams innerhalb einer TCP-Verbindung. Im dritten Frame versendet der Server einen PUSH_PROMISE-Frame innerhalb des Streams mit der ID 41. Das passiert, weil der Server den nächsten Request des Clients vorhersagt und daraufhin eine Response ankündigt. Im genannten Beispiel sieht der Server eine Anfrage für eine Datei ßtyle.css"kommen und kündigt deren Übertragung an den Client mit einem HEADERS-Frame an, den er im neuen Stream mit der ID 42 versendet.

Der Client kann nun zwei Dinge tun. Um die angebotene Datei zu akzeptieren, muss der Client aktiv nichts weiter tun. Nach dem HEADERS-Frame wird der Server im Stream 42 einen DATA-Frame mit besagter Datei versenden. So erhält der Client die Datei ßtyle.cssöhne sie jemals aktiv angefordert zu haben.

Alternativ kann der Client zu der Entscheidung kommen, dass er die Datei ßtyle.css"nicht benötigt. In diesem Fall schließt der Client den Stream 42, indem er einen RST_STREAM-Frame an den Server schickt. Das Protokoll verbietet es dem Server, über Streams zu kommunizieren, für die der Client einen solchen Frame gesendet hat.

Server-Push verringert die Latenz somit durch das Einsparen von Client-Requests. Dafür muss der Server jedoch in der Lage sein, die Anfragen des Clients zuverlässig zu antizipieren. Er muss den aus-

zuliefernden Content und das Verhalten seiner Clients gut kennen, um nachfolgende Requests nach weiteren Inhalten vorherzusehen.

Es ist Aufgabe der Webserver-Administratoren oder der Webentwickler, die Push-Funktionalität durch Konfiguration und Implementierung zu gewährleisten.

—- END —-

Literatur

Daniel An. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 02 2017.

Daniel An and Pat Meenan. Why marketers should care about mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/experience-design/mobile-page-speed-load-time/, 07 2016.

Shaun Anderson. How fast should a website load in 2018?, 2 2018. URL `https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/`.

Vlad Krasnov. Announcing support for http/2 server push. https://blog.cloudflare.com/announcing-support-for-http-2-server-push-2/, 04 2016.

Jan Weinschenkler. Mit java auf dem http/2-zug. https://www.heise.de/developer/artikel/Mit-Java-auf-dem-HTTP-2-Zug-3918097.html, 12 2017.

Erik Witt. Schnellere websites mit http/2, 02 2018. URL `https://www.heise.de/ix/heft/WWW-Beschleuniger-3948333.html`.