

Das HTTP/2-Protokoll¹

Prof. Dr. Stefan Zander

7. Juni 2018

¹ Ausgesuchte Einheit der Vorlesung
"Entwicklung Webbasierter Anwendungen"

Lernziele:

- Kennenlernen der Unterschiede zwischen HTTP/1.1 und HTTP/2
- Kennenlernen der mit HTTP/2 eingeführten Neuerungen
- Kennenlernen der technischen Beschränkungen von HTTP/1.1

Inhaltsverzeichnis

1 Einführung	1
2 HTTP/2	3
2.1 Semantik und Bestandteile	5
2.2 Übersicht der Neuerungen	8
3 HTTP/1.1	8
3.1 Beschränkungen	8
3.2 Workaround 1: Nutzung paralleler Verbindungen . . .	9
3.3 Workaround 2: Zusammenfassen von Ressourcen . . .	10
3.4 Workaround 3: Ressourcenverteilung auf mehrere Hostnamen	10
4 HTTP/2	10
4.1 Request Multiplexing	10
4.2 Header Compression	12
4.3 Resource Prioritization	12
4.4 Server Push	13
4.5 Browserunterstützung	15
5 Zusammenfassung	15

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 nachfolgender Ausschnitt aus einer 2018 veröffentlichten Studie 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 weitere Studien durchgeführt (vgl. [An and Meenan, 2016]) und kommt zu folgenden Ergebnissen:

"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 folgende Hauptfaktoren für die Dauer des Seitenaufbaus verantwortlich sind:

- (i) Anzahl der Seitenelemente, sowie die
- (ii) Anzahl der enthaltenen Bilder

Bei komplexen Seiten mit vielen Elementen dauert das Parsen des Seitencodes sowie der Aufbau des DOMs bedeutend länger. Bei Grafiken empfiehlt es sich, auflösungsoptimierte JPEGs anstatt PNG Formate zu verwenden.

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

"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%."

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

Bei einer Verlangsamung der Seitenladezeit von 1 Sec. auf 7 Sec. verdoppelt sich die Wahrscheinlichkeit, dass ein mobiler Surfer die eigene Seite nach der Startseite wieder verlässt. Das selbe Verhalten ist zu beobachten, wenn sich die Anzahl der Seitelemente von 600 auf 4.000 erhöht.

Weitere interessante Einblicke einschließlich Maßnahmen zur Behebung von Geschwindigkeitsproblemen bietet die Seite

<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** (vgl. ²).

²

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

1. der Verarbeitung im Server,
2. der Übertragung über das Netzwerk, und
3. der Darstellung im Browser.

Je nach den vorherrschenden Gegebenheit können alle drei Faktoren einen starken Einfluss auf die Performance haben und die Ladezeit-optimierung 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.

2 HTTP/2

HTTP/2 stellt den Nachfolger von HTTP/1.1 dar und basiert auf dem von Google entwickelten *SPDY (Speedy) Protokoll*³. Im Gegensatz zu SPDY benötigt HTTP/2 keine Verschlüsselung mittels TLS (Transport Layer Security) und kann auch für HTTP Seiten verwendet werden [Mueller, 2015]. Langfristig soll HTTP/1.1 durch HTTP/2 abgelöst werden und sich als Standard etablieren. Vorerst stellt die neue Protokollversion jedoch nur eine Alternative dar und gewährleistet durch Abwärtskompatibilität, dass auch Browser, die dieses Protokoll nicht unterstützen, Webseiten über HTTP/1.1 laden.

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 Protokolls⁴ wurde im Mai 2015 von der Internet Enginee-

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]

³ SPDY Whitepaper des Chromium Projekts: <https://www.chromium.org/spdy/spdy-whitepaper>

⁴ Spezifikation: <https://tools.ietf.org/html/rfc7540>

ring Task Force (IETF) als Standard verabschiedet. 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**⁵. Request Multiplexing und die Kompression der Headerinformationen verbessern die Ladezeit einer Seite unmittelbar. Um die anderen beiden Optimierungen zu nutzen, ist jedoch ein gewisser Entwicklungsaufwand erforderlich (siehe Kapitel xxx).

⁵ Siehe Kapitel 2.1

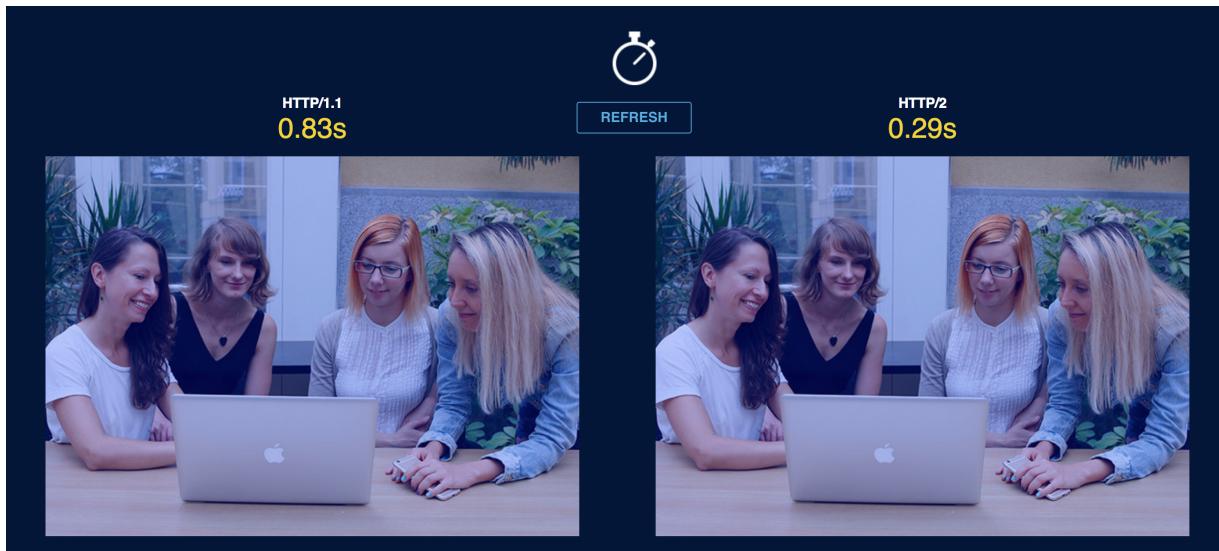


Abbildung 1: Vergleich der Ladezeiten zwischen HTTP/1.1 (linke Hälfte) und HTTP/2 (rechte Bildhälfte) am Beispiel einer großen Bilddatei (Screenshot aus dem Bild 1)

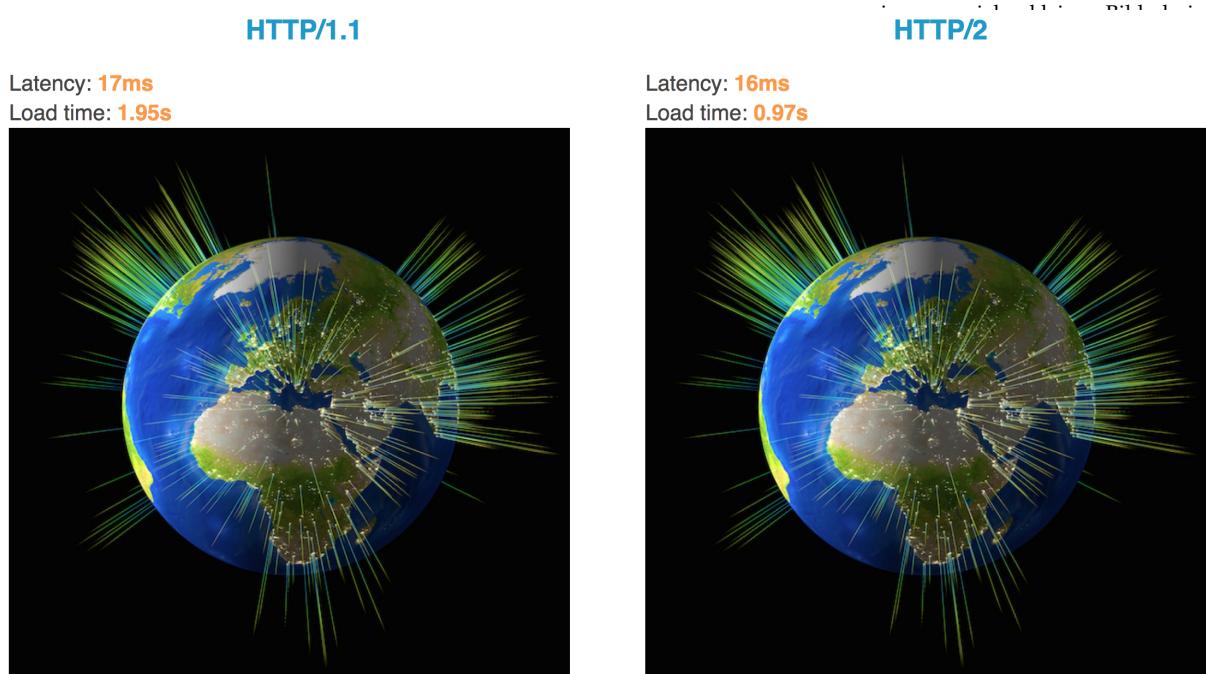


Abbildung 1 und 2 zeigen die Ladezeiten einer großen Bilddatei,

Abbildung 2: Vergleich der Ladezeiten zwischen HTTP/1.1 (linke Hälfte) und HTTP/2 (rechte Bildhälfte) am Beispiel einer aus vielen kleinen Bildschichten zusammengesetzten großen Bilddatei (Screenshot aus dem Bild 2 auf <https://http2.akamai.com/demo>)

die sich aus vielen kleinen Bildschnipseln (*sprites*) zusammensetzt. In der linken Bildhälfte ist jeweils die Ladezeit mittels HTTP/1.1 zu sehen, die rechte Bildhälfte zeigt die Ladezeit mit dem HTTP/2 Protokoll. Sowohl das erste, als auch das zweite Beispiel zeigen eine deutliche Verbesserung der Ladezeiten unter HTTP/2. Zum selber testen finden sich die beiden Demos unter <http://www.http2demo.io/> sowie <https://http2.akamai.com/demo>.

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 weiterhin erhalten; auf der Anwendungsschicht ändert sich nichts. Neu ist hingegen die Abwicklung des Transports der Header und Nutzdaten: HTTP-Requests und -Responses sind in **Streams**, **Messages** und **Frames** kodiert.

- **Stream:** Ein bidirektonaler 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 bidirektonalen 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.

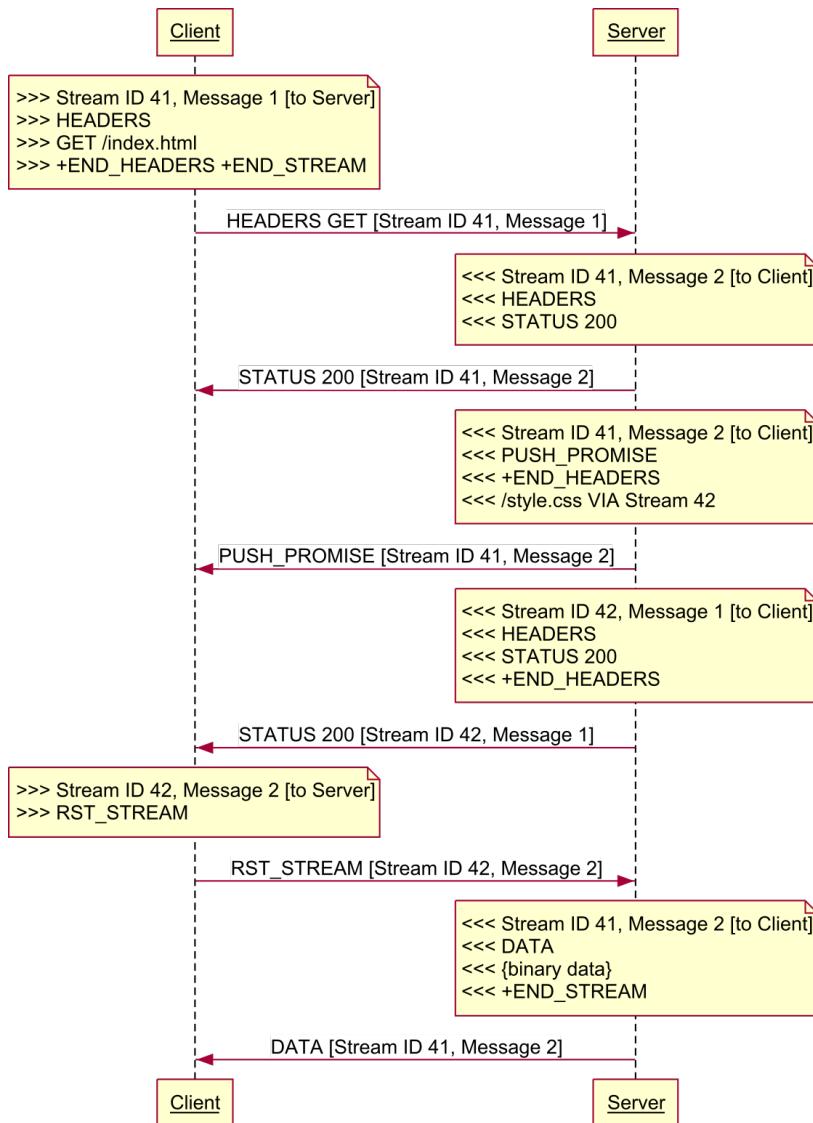


Abbildung 3: Beispielhafte Darstellung eines HTTP/2-Nachrichtenaustauschs zwischen Browser und Server [Weinschenkler, 2017].

Abbildung 3 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 absichtigt, 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 3 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.
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

Datenpakete werden nach Wichtigkeit sortiert und in entsprechender Reihenfolge übermittelt, wodurch zuerst die Dateien an den Browser übermittelt werden, die für einen schnellen Seitenaufbau wichtig sind (vgl. ⁶).

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.

⁶ Patrick Mueller. Http/2 und seo: Welche vorteile hat das neue protokoll? <https://www.seonative.de/http2-und-seo-welche-vorteile-hat-das-neue-protokoll/>, December 2015

3 HTTP/1.1

3.1 Beschränkungen

Eine wesentliche Einschränkung von HTTP/1.1, das man auch als Konstruktionsfehler [Witt, 2018] bezeichnen kann ist, dass HTTP keine direkte Möglichkeit bietet, Requests zu parallelisieren. In HTTP/1.1 lassen sich zwar mehrere Request-Response-Konversation über eine TCP-Verbindung abwickeln, aber nur *sequenziell*. Dieses Vorgehen bezeichnet man auch als **HTTP Pipelining**. Es erzwingt, dass der Server über eine TCP-Verbindung zu einem Zeitpunkt nur eine Response an einen Client übertragen kann.

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

HTTP Pipelining

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

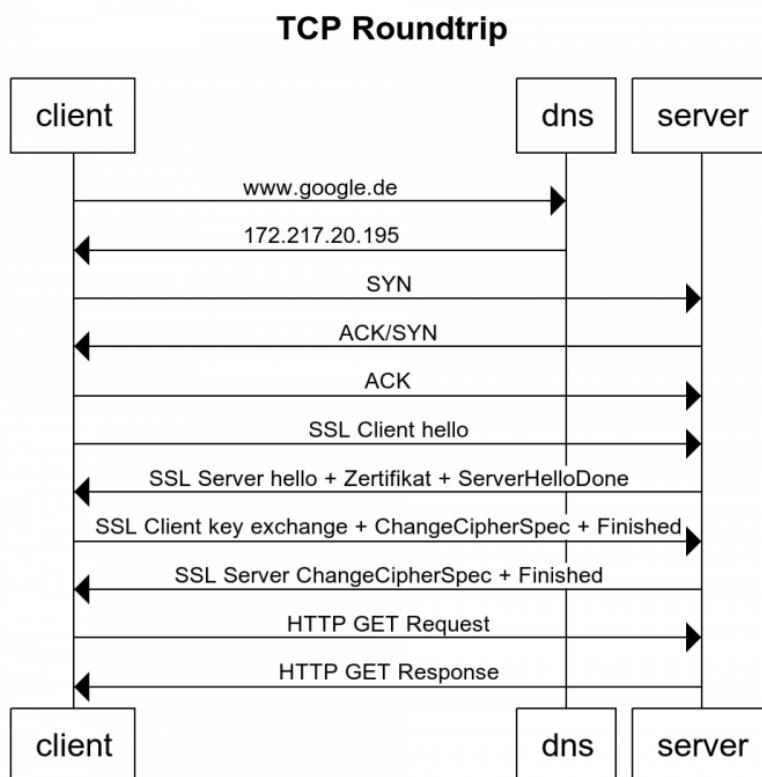
der ersten Response beim Client ankommen können. Diese Problematik bezeichnet man auch als **Head of Line Blocking** (vgl. linke Hälfte von Abbildung 5).

Dieses Verhalten 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.

3.2 Workaround 1: Nutzung paralleler Verbindungen

Um diese Limitierung zu umgehen öffnen Browser üblicherweise **sechs parallele Verbindungen**, so dass maximal sechs Ressourcen gleichzeitig übertragen werden können. Alle übrigen Requests müssen auf eine freie Verbindung warten (vgl. Head of Line Blocking).

Der Aufbau mehrerer paralleler Verbindungen zum Server hat jedoch den Nachteil, dass der dadurch entstehende *Protokoll-Overhead* vergleichsweise stark ins Gewicht fällt, da jede TCP-Verbindung einen initialen Aufbau erfordert. Dieser notwendige Nachrichtenaustausch für den Verbindungsaufbau zwischen Client, DNS-Service und Server ist in Abbildung 4 dargestellt.



Workaround 1: Aufbau mehrerer paralleler Verbindungen zum Server

Browser erlaubt max. 6 parallele Verbindungen zum Server

Abbildung 4: Notwendiger Nachrichtenaustausch für den Aufbau einer TCP-Verbindung [Weinschenkler, 2017].

3.3 Workaround 2: Zusammenfassen von Ressourcen

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

3.4 Workaround 3: Ressourcenverteilung 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`.

Workaround 3: Verteilung von Ressourcen auf mehrere Hostnamen

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

4 HTTP/2

4.1 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 bahezu beliebig viele HTTP-Konversationen,

sogenannte *Streams*, zwischen Client und Server abwickeln, d.h., der Browser kann beliebig viele Requests parallel stellen. 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].

Das 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 Verbindungsauftakt 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 5) 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 substantiell [Witt, 2018].

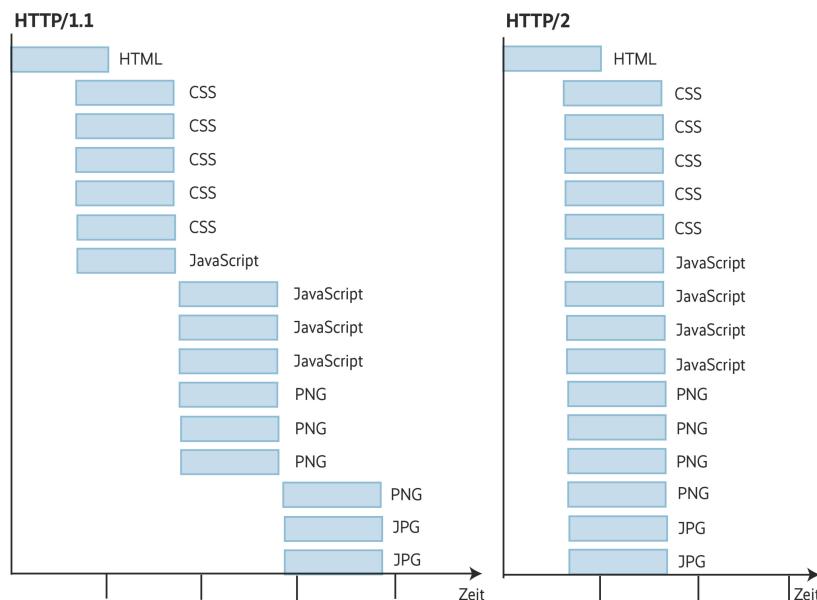


Abbildung 5: Das Zeitverlaufsdiagramm zeigt, wie der Browser bei HTTP/1.1 nach dem Download der HTML-Seite sechs Requests parallel stellt. Die weiteren Requests werden so lange geblockt, bis diese beantwortet sind (*Head of Line Blocking*). Mit HTTP/2 kann der Browser alle Ressourcen gleichzeitig anfragen, sodass sich die Ladezeit verringert [Witt, 2018].

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.

4.2 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 übertragen⁷. Insbesondere bei Cookies ist die Datenübertragung massiv redundant. Mithilfe des *HPACK-Algorithmus*⁸ 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.

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

- (a) [i]:content-type:application/html
- (b) :accept:application/html

Teil nahezu jeder Kommunikation zwischen Webbrowser und -server. HPACK verkürzt die zu übertragende Datenmenge auf jeweils ein Byte.

⁷ Siehe Kapitel zu Cookies und Sessions

⁸ Siehe <https://http2.github.io/http2-spec/compression.html>

Mobile NutzerInnen profitieren besonders von Header Compression

4.3 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?

4.4 Server Push

Server Push erlaubt Servern benötigte Ressourcen proaktiv und opportunistisch auszuliefern (siehe Abbildung 6).

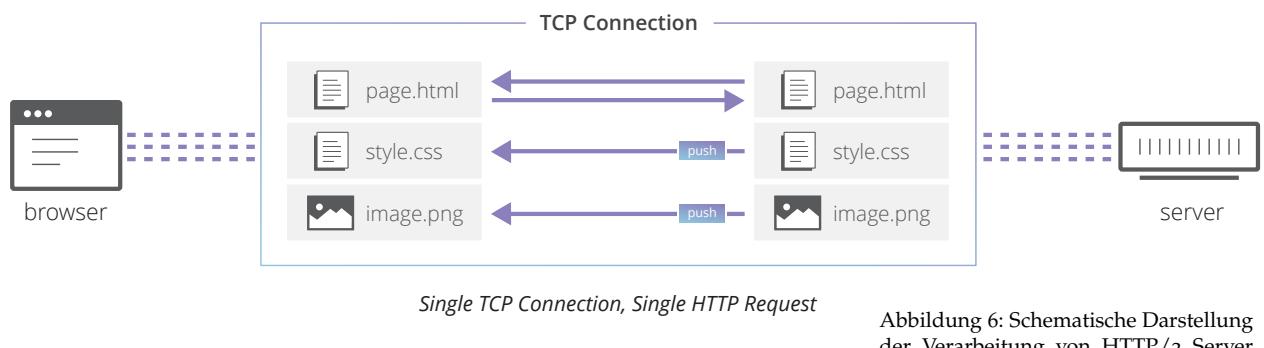


Abbildung 6: Schematische Darstellung der Verarbeitung von HTTP/2 Server Push [Krasnov, 2016].

Abbildung 7 & 8 zeigt die Browser-seitige Verarbeitung einer Demonstrations Webseite auf der neben der Hauptseite fünf Grafiken enthalten sind – je einmal ohne Server Push (Abbildung 7) und einmal mit (Abbildung 8). Mittels sog. Profiling Tools lässt sich der Geschwindigkeitszuwachs sehr schön visualisieren und benchmarken.

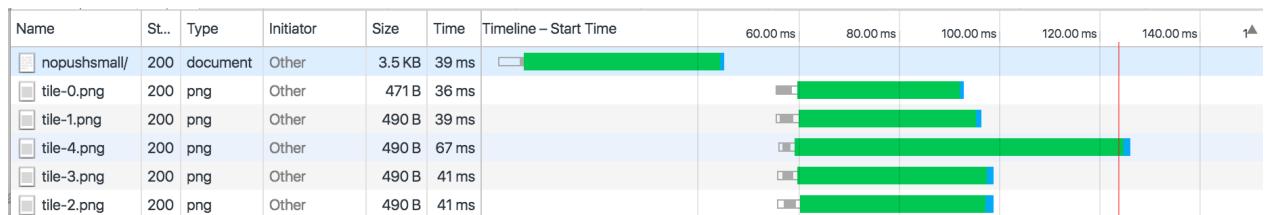


Abbildung 7: Verarbeitung einer Seite mit fünf Grafiken ohne Server Push [Krasnov, 2016].

Wie in Abbildung 7 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.

Mit aktiviertem Server Push werden die Grafiken noch während der Client-seitigen Verarbeitung der Seite automatisch durch den

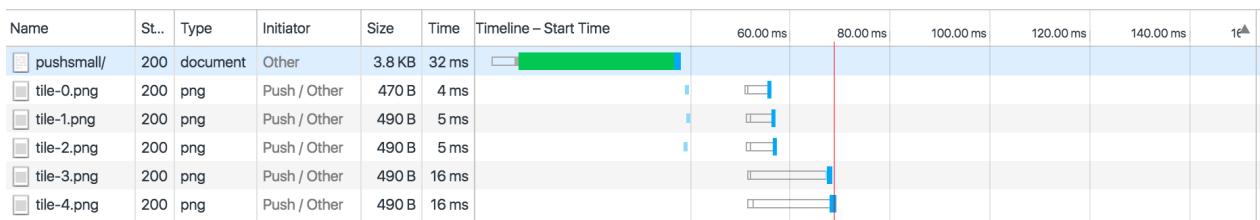


Abbildung 8: Verarbeitung der Beispielseite mit Server Push [Krasnov, 2016].

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 Frame⁹ 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.

— START —

Der beispielhafte Ablauf aus Abbildung 3 (vgl. Kapitel 2.1) 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 `style.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 `style.css` ohne sie jemals aktiv angefordert zu haben.

Alternativ kann der Client zu der Entscheidung kommen, dass er die Datei `style.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 auszuliefernden 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 —

⁹ Durch **Push Promises** signalisiert der Server dem Browser einen PUSH von zu einer Seite gehörenden Ressourcen. Dieses Frame wird immer automatisch am Beginn eines PUSH gesendet.

4.5 Browserunterstützung

Laut der Website "Can I use"¹⁰ wird HTTP/2 bereits von 87,59% der Internet Browser in Deutschland und 79,57% der Browser weltweit unterstützt (Stand 2. März 2018). Mittlerweile unterstützen auch die meisten mobilen Browser HTTP/2. Microsoft's Internet Explorer 11 unterstützt das Protokoll, jedoch nur unter Windows 10. Einschränkungen gibt es auch bei Firefox und Chrome, welche HTTP/2 nur unterstützen, falls der Server die TLS-Protokollerweiterung **Application-Layer Protocol Negotiation (ALPN)**¹¹ unterstützt. Generell (mit Ausnahme des IE11) muss das TLS-Protokoll Server-seitig implementiert und unterstützt werden. Aufgrund der Abwärtskompatibilität werden Webseiten Webseiten trotzdem ohne Probleme übertragen, wenn auch nur über HTTP/1.1. Weitere Details finden sich auf den Seiten von "Can I use".

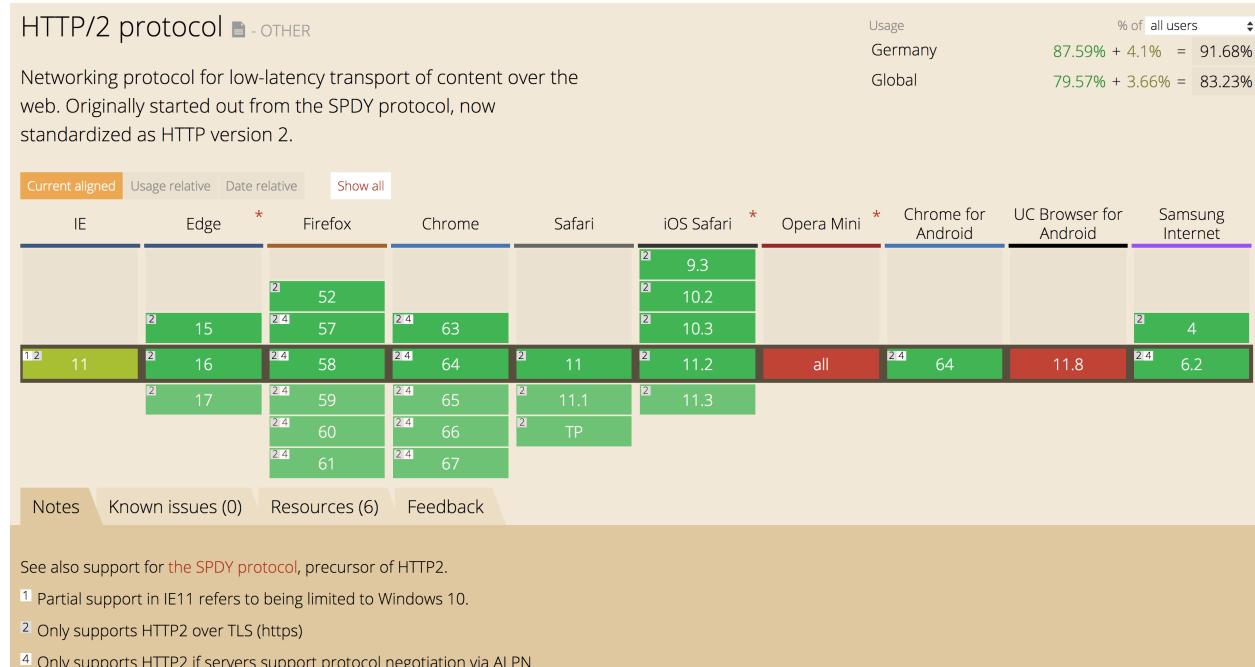


Abbildung 9: Übersicht der Browserunterstützung von HTTP/2

5 Zusammenfassung

Durch die Protokollversion HTTP/2 wird die Geschwindigkeit, Effizienz und Sicherheit der Datenübertragung verbessert. HTTP/1.1 verwendet zum Laden von unterschiedlichen Seitenelementen wie JS-, CSS- und Bilddateien mehrere TCP Verbindungen. Im Vergleich dazu werden bei http/2 mehrere Daten parallel über nur eine Verbindung übertragen, was die Ladezeiten deutlich beschleunigt. Die Technik, die hierfür verwendet wird, nennt sich Multiplexing. Auch die Art und Weise wie Header Daten übertragen werden, unterscheiden sich bei beiden Protokollen. Im Gegensatz zur HTTP 1.1 Verbindung, bei der die Daten unkomprimiert verschickt werden, übertragt

¹⁰ <https://caniuse.com/#search=http%2F2>

¹¹ https://en.wikipedia.org/wiki/Application-Layer_Protocol_Negotiation

HTTP/2 die Informationen komprimiert im Binärcode. Dadurch soll zusätzlich die Verarbeitung der Daten beschleunigt werden. Weitere Vorteile von HTTP/2 im Vergleich zu HTTP/1.1 sind die Priorisierung von Datenpaketen, der Server Push und der Wegfall von Head-of-Line-Blocking. Datenpakete werden nach Wichtigkeit sortiert und in entsprechender Reihenfolge übermittelt, wodurch zuerst die Dateien an den Browser übermittelt werden, die für einen schnellen Seitenaufbau wichtig sind. Über den Server Push können JS, CSS und andere Dateiformate dabei ohne vorherige Anfrage des Clients übermittelt werden. Durch den Wegfall von Head-of-Line-Blocking kann es in HTTP/2 nicht mehr zu dem Problem kommen, dass durch die Verzögerung bei der Übertragung eines Datenpakets alle folgenden Datenpakete blockiert werden. Dieses Problem besteht bisher noch in HTTP/1.1.

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.

Patrick Mueller. Http/2 und seo: Welche vorteile hat das neue protokoll? <https://www.seonative.de/http2-und-seo-welche-vorteile-hat-das-neue-protokoll/>, December 2015.

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>.