Hochschule für Technik, Wirtschaft und Kultur Leipzig

Fakultät Informatik und Medien

Bachelorstudiengang Informatik

Bachelorarbeit

zur Erlangung der akademischen Grades

Bachelor of Science (B.Sc.)

Untersuchung der Performance-Steigerung beim Umstieg von HTTP/1.1 zu den Versionen 2.0 und 3.0

Eingereicht von: Alexander Krahl

Matrikelnummer: 70365

Leipzig Oktober 2020

Erstprüfer: Prof. Dr. rer. nat. Thomas Riechert

Zweitprüfer: B.Sc. Alexander Lisnitzki

Inhaltsverzeichnis

1 Motivation 1

2 Grundlagen 2
- 2.1 HTTP/0.9, 1.0, 1.1 . . . 2
  - 2.1.1 HTTP/0.9 . . . 2
  - 2.1.2 HTTP/1.0 . . . 3
  - 2.1.3 HTTP/1.1 . . . 4
- 2.2 HTTP/2 und SPDY . . . 5
  - 2.2.1 SPDY . . . 6
  - 2.2.2 HTTP/2 . . . 7
- 2.3 HTTP/3 und QUIC . . . 7
- 2.4 Bisherige Tools zur Messung der Performance . . . 11

3 Vorstellung des Untersuchungsobjekts 13

4 Untersuchung der Performance 16
- 4.1 Untersuchungskriterien . . . 16
- 4.2 Aufbau der Untersuchung . . . 17
  - 4.2.1 Server . . . 17
  - 4.2.2 Client . . . 19
  - 4.2.3 HTTP/2 Server Push . . . 19
  - 4.2.4 Durchführung der Messung . . . 21
  - 4.2.5 Probleme beim Aufbau . . . 22
- 4.3 Auswertung der Ergebnisse . . . 23
  - 4.3.1 Umgebung: keine Beschränkung . . . 24
  - 4.3.2 Umgebung: Beschränkt auf 4 Mbit Bandbreite . . . 26

4.3.3 Umgebung: Paketverlust 28
4.3.4 HTTP/2 Server Push 31

5 Fazit 32

Abbildungsverzeichnis I

Tabellenverzeichnis II

Literaturverzeichnis III

1 Motivation

Ziel dieser Arbeit ist, das Messen und Vergleichen der Performance der Transportprotokolle HTTP/1 2 und 3. Dafür wird ein Aufbau auf Basis von Docker Containern vorgestellt, an dem die Messungen durchgeführt werden. Ziel der Messung ist dabei den Ressourcenverbrauch der verschieden Versionen festzustellen. Besonders wird dabei auf die Latenz, die Zeit welche zum laden der Website benötigt wird, geachtet, da diese sich auf das Google Search Ranking auswirkt(vgl. [WP18]) und ein besseres Ranking in der Suche zu mehr Website-Aufrufen führt, welche Monetarisiert werden können. Im Diagramm 1.1 sieht man, dass besonders nach 2010 das Web stark wächst, außerdem sieht man die Verbreitung der hier untersuchten Technologien.

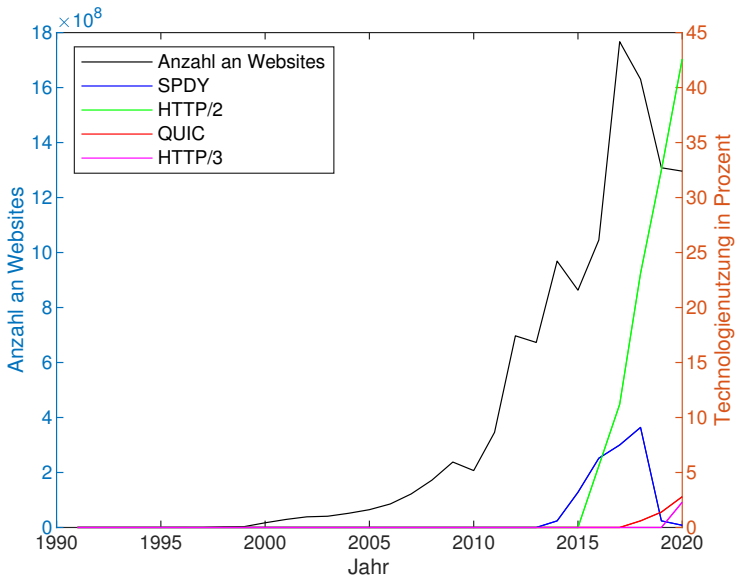

Abbildung 1.1: Anzahl and Webseiten und Verbreitung verschiedener Technologien. [Net20, Sta, W3T20]

2 Grundlagen

Um einen Überblick über die HTTP Versionen zu verschaffen werden diese vorgestellt und deren wichtigsten Neuerungen und Verbesserungen aufgezeigt.

2.1 HTTP/0.9, 1.0, 1.1

HTTP entstand 1990 während der Implementierung der Idee World Wide Webs von Tim Berners Lee. Es ist einer von 4 Bausteinen des World Wide Webs:

- HyperText Markup Language(HTML) als Textformat der Dokumente
- HyperText Transfer Protocol(HTTP) als Übertragungsprotokoll zwischen Client und Server
- Ein Client für Anzeigen und Ändern von Dokumenten(Browser)
- Ein Server um die Dokumente bereitzustellen

(vgl. [Con20])

2.1.1 HTTP/0.9

HTTP 0.9 ist die erste Version von HTTP, welche von der World Wide Web Initiative als Prototyp in 1991 definiert wurde. Es handelt sich um eine Untermenge der Funktionen des später standardisierten HTTP/1.0 Protokolls. HTTP/0.9 ist ein Client-Server Protokoll. Der Client eröffnet eine TCP-Verbindung zum Server

und sendet seine Anfrage. Der Server antwortet und schließt danach die Verbindung. Ein HTTP/0.9 Request beginnt mit `GET`, gefolgt von einem Leerzeichen und der Dokument-Adresse. HTTP 0.9 unterstützt query Parameter, diese erlauben es Schlüssel-Wert-Paare durch Anhängen an die URL an den Server zu übergeben. Alle Daten von Anfrage und Antwort werden ASCII-kodiert übertragen. Alle Anfragen sind voneinander unabhängig. Es gibt keine HTTP Header, Status und Fehler Codes (vgl. [BL91, Con20]).

2.1.2 HTTP/1.0

Nach der Veröffentlichung von HTTP 0.9 wurde dieser Standard von einer Vielzahl an Clients und Servern implementiert. HTTP/0.9 hatte Limitierungen, unter anderem unterstützte es nur HTML Dokumente, Antworten waren nicht einfach maschinenlesbar und Daten konnten nur durch ein Query an den Server gesendet werden. Aus diesen Gründen haben viele Entwickler von Client und Server Software selbst weitere Funktionen zu ihren HTTP Implementierungen hinzugefügt. Dies würde zu einer immer größeren Aufspaltung des HTTP Ökosystems führen, da die meisten Programme nicht mehr untereinander kompatibel währen. Die Funktionen, welche von den meisten Clients/Servern implementiert wurden, sind im Mai 1996 als HTTP/1.0 standardisiert wurden, um Interoperabilität zwischen den Systemen beizubehalten (vgl. [NFBL96, S. 4]). Außerdem muss jeder HTTP/1.0 Server auch auf HTTP/0.9 Anfragen antworten können.

Die Standards für HTTP/1 und folgende Versionen werden von der Internet Engineering Task Force(IETF) in der Publikation Request for Comments(RFC) veröffentlicht.

Zu HTTP/0.9 wurden folgende Funktionen hinzugefügt:

- Spezifizierung der HTTP Version in der Anfrage.
- Hinzufügen von Statuscodes zu Antworten, welche maschinenlesbar sind. Bei HTTP/0.9 musste auch im Fehlerfall mit einem HTML-Dokument geantwortet werden, welches dann vom Client interpretiert werden muss.

- Hinzufügen von Metadaten durch HTTP Header. Durch den MIME Standard können auch andere Inhalte, welche keine HTML-Dokumente sind, übertragen werden. Header vereinfachen die Interpretation der gesendeten Daten durch einen Computer, da der Client keinen Algorithmus mehr benötigt, welcher Datenformat und Codierung anhand der gesendeten Daten kalkuliert.
- Hinzufügen anderen Anfragemethoden (`HEAD`, `POST`).
- Ein einfaches Authentifizierungssystem auf der Protokollebene durch hinzufügen des `Authenticantion` Headers, die Übertragung ist trotzdem in Klartext.

(vgl. [NFBL96, S. 12,21,22,24,26 ff.])

2.1.3 HTTP/1.1

Bereits vor der Veröffentlichung des HTTP/1.0 Standards wurde an einer Standardisierung weiterer Funktionen gearbeitet. Dieser neue Standard wurde im Januar 1997 veröffentlicht. Eine Änderung der Version wurde nötig, da es viele Implementationen gab, welche vorgegeben haben, HTTP/1.0 zu sein, aber den Standard nicht komplett implementiert haben (vgl. [FNM+97, S. 7] [Con20]).

HTTP/1.1 nach RFC 2068 weist folgende größer Änderungen zu HTTP/1.0 auf:

- Persistente Verbindungen verringern die Zugriffszeiten, da nicht mehr für jede Anfrage eine TCP Verbindung aufgebaut und wieder abgebaut werden muss. Stattdessen wird die Verbindung standardmäßig offengehalten, um sie für weitere Anfragen zu nutzten. Client und Server können über den `Connection` Header der Gegenseite mitteilen das die Verbindung geschlossen werden soll oder gar nicht erst persistent Aufgebaut werden soll.
- Durch Pipelining in persistenten Verbindungen können Anfragen hintereinander gesendet werden, ohne auf eine Antwort zu warten. Antworten werden in der Reihenfolge der Anfragen gegeben (vgl. [FNM+97, S. 44, 45]).

- Content Negotiation: abhängig vom HTTP Header (`language`, `content-encoding`, etc.) kann vom Server oder Client die "beste" Antwort ausgewählt werden, wenn von der angefragten Ressource mehrere Varianten vorliegen.
- Hinzufügen der neuen Zugriffsmethoden `OPTIONS`, `PUT`, `DELETE`, `TRACE`.
- Hinzufügen von Caching Semantik, welches es erlaubt, die Antworten auf Anfragen am Client oder einem Proxy zwischenzuspeichern, um die benötigte Bandbreite vom Quellserver zu verringern, oder Anfragen direkt zu vermeiden.
- Unterstützung von mehreren virtuellen Hosts auf einer IP durch das `host` Header Feld. Dies ermöglicht mehrere HTTP Server auf einer IP gleichzeitig bereitzustellen, solange die Server für unterschiedliche Domains Webseiten bereitstellen.
- Hinzufügen von verschieden Kodierungsverfahren zur Übertragung, darunter Komprimierung und das Aufteilen der Daten auf mehrere Antworten durch eine Teilung in Chuncks.

(vgl. [FNM+97, S. 24, 25, 43-45, 50 ff., 67 ff., 70 ff., 119])

RFC 2068 wurde 1999 durch RFC 2616 abgelöst, welche wiederum 2014 durch die RFCs 7230 bis 7235 abgelöst wurde. In diesem Update des Standards wurden diese klarer geschrieben und einige Anforderungen wurden angepasst (vgl. [FR14b, S. 91-93]). Zwei große Änderungen waren, dass die Rückwärtskompatibilität zu HTTP/0.9 kein Muss mehr ist und das Limit von 2 persistenten Verbindungen von Client zum Server aufgehoben wurde (vgl. [FR14a, S. 81, 82] [NMM+99, S. 47, 170]).

2.2 HTTP/2 und SPDY

HTTP/2 entstammt im Gegensatz zu HTTP/1 nicht dem akademischen Bereich, sondern den Unternehmerischen. In diesem Abschnitt wird HTTP/2 und dessen Vorgänger SPDY, vorgestellt.

2.2.1 SPDY

Als Teil seiner "Let's make the web faster" Initiative hat die Firma Google im Februar 2012 einen Internet Draft für ein neues Transportprotokoll namens SPDY vorgelegt (vgl. [BP12, S. 4] [Pro]). Ziel von SPDY ist es, die Latenz um 50% zu reduzieren, ohne das Webseitenbetreiber einen großen Aufwand für den Umstieg haben. SPDY baut dafür auf die SSL/TLS-Schicht auf, die HTTP-Semantik und -Syntax von Anfragen und Antworten wird aber beibehalten(vgl. [Pro, S. 12]). HTTP/1 ist ein textbasiertes Protokoll, d. h. die Daten werden in menschenlesbarer Form übertragen, SPDY hingegen ist ein binäres Protokoll, die Daten werden in einem Format übertragen, welches einfacher vom Computer interpretiert werden kann.

Dies wird durch folgende Funktionen erreicht:

- Es wird nur noch eine TCP Verbindung aufgebaut, über diese können mehrere unabhängige Datenströme gleichzeitig übertragen werden. Client und Server kommunizieren mit Frames, dies sind Basisblöcke des Protokolls. Verschiedene Arten von Frames wie z.B.: `HEADERS` und `DATA` sind standardisiert (vgl. [BPT15, S. 4]).
- Anfragen-Priorisierung erlaubt es dem Client, Anfragen zu priorisieren, was verhindern soll, dass die Verbindung überlastet wird. Außerdem ermöglicht dies die Übertragung von mehreren Dateien in einer Reihenfolge, welche es dem Browser ermöglicht, die Scripts so Zeitig wie möglich zu verarbeiten, weshalb diese zu einen früheren Zeitpunkt fertig abgearbeitet sind.
- Die HTTP-Header werden komprimiert, wodurch weniger Daten übertragen werden müssen.
- Server push erlaubt es dem Server, dem Client Daten zu übertragen, bevor der Client diese angefragt hat.
- Server hint erlaubt es dem Server, dem Client vorzuschlagen, bestimmte Daten anzufragen, von denen der Server weiß, dass sie noch benötigt werden.

Nach Messungen von Google hat SPDY eine Reduzierung der Latenz von 27% bis 63% erreicht. (vgl. [Pro])

2.2.2 HTTP/2

Als Reaktion auf die Implementierung von SPDY in Chrome und Firefox wurde Anfang 2012 in der IETF darüber diskutiert, einen HTTP/2 Standard zu entwickeln (vgl. [Not12]). Als Ausgangspunkt wurde im November 2012 direkt der Draft des SPDY Protokolls übernommen (vgl. [BPTM12]). HTTP/2 baut nicht mehr ausschließlich auf SSL/TLS auf, sondern ist auch unverschlüsselt nutzbar. Außerdem hat HTTP/2 einen extra Standard zur Komprimierung der Header: HPACK. Dieser war nötig, da es in den TLS Versionen 1.2 und darunter eine Lücke gab, welche es erlaubte, den Klartext aus den verschlüsselten Headern auszulesen (vgl. [Cor12]). Der Standardisierungsprozess wurde im Mai 2015 abgeschlossen, woraufhin Google angekündigt hat, die Entwicklung von SPDY einzustellen und die Implementierung aus ihrem Browser Chrome zu entfernen (vgl. [BPT15, BB15]).

Da über das Protokoll HTTP oder HTTPS in der URL nicht kommuniziert werden kann, ob HTTP/1 oder HTTP/2 verwendet werden soll, wurde 2016 der `alt-svc` HTTP Header standardisiert(vgl. [NMR16, S. 2,3]). Aus diesem kann ein Client auslesen, dass es die gleiche Ressource auch mit einem anderen Protokoll verfügbar ist und wie lang diese Information gültig ist. Der Client kann dann gegebenenfalls zu diesem Protokoll wechseln. Die Protokoll Id von HTTP/2 ist h2.

2.3 HTTP/3 und QUIC

Neben SPDY arbeitet Google auch an anderen Initiativen um die Latenz zu reduzieren, darunter auch die Entwicklung das Transportprotokoll QUIC. Dabei handelt es sich im ein Transportprotokoll, welches auf UDP basiert. QUIC wurde dabei ursprünglich als Übertragungsschicht für SDPY entwickelt (vgl. [Ros13, Abschnitt

MOTIVATIONS]). HTTP/3 ist dabei eine Übertragung der HTTP Semantik auf QUIC als Transportprotokoll, deshalb ist HTTP/3 ähnlich zu HTTP/2, wobei einige Aufgaben von HTTP/2 nun von QUIC übernommen wurden (vgl. [Bis20, S. 5]). QUIC ist dabei nicht an HTTP/3 gebunden, sondern ist auch mit HTTP/2 oder als Übertragungsschicht für andere Protokolle nutzbar. QUIC ist als eine Verbesserung zu TCP und TLS ausgelegt, da bei der Entwicklung von SPDY folgende Probleme aufgetreten sind:

- Die Verzögerung eines Paketes verzögert auch alle anderen Pakete, unabhängig vom Datenstrom, da nur eine TCP-Verbindung verwendet wird.
- Weil nur eine TCP Verbindung verwendet wird, werden alle Datenströme der Verbindung drastisch reduziert, wenn das TCP Fenster gesenkt wird. Davor wurde von den k-Verbindungen nur die beeinflusst, bei der die Pakete verloren gegangen sind(meist k = 6, Bandbreite auf 11/12 reduziert). HTTP/2 hat in einer Umgebung, wo das TCP Fenster häufig reduziert werden muss, durch z.B.: andere Anfragen, eine höhere Latenz als HTTP/1, wobei dieser Effekt umso größer ist, je mehr Verbindungen die HTTP/1 Implementierung gleichzeitig nutzen würden.
- Um eine TLS Sitzung wieder aufzunehmen, wird mindestens ein zusätzlicher Roundtrip benötigt. Ein Roundtrip(RT) ist das Übertragen von Daten von A nach B und die Rückübertragung von B zu A, die Round-Trip-Time(RTT) ist die Zeit, welche dafür benötigt wird. Dies liegt an der Implementierung von TLS, nicht an den Sicherheitsanforderungen.
- TLS Version 1.0 und deren Vorgänger benötigen die Pakte in ihrer Sendereihenfolge, um sie zu entschlüsseln. Diese Vorgehensweise wurde mit TLS 1.1 geändert, dadurch ist die Paketgröße um einige Bytes gestiegen.

(vgl. [Ros13, S. 5,6 Abschnitt SPDY SUPPORT MOTIVATIONS])

Daraus leiten sich die Designziele von QUIC ab:

- Breite Verfügbarkeit: Das Protokoll muss auf der derzeitigen Netzwerkinfrastruktur verwendet werden können, muss also auf TCP oder UDP aufbauen, da nicht TCP- oder UDP-Verkehr von middle-boxes geblockt wird.
- Head of Line blocking durch Paketverlust soll verringert werden. Head of Line Blocking bezeichnet hier das Phänomen, bei dem im Fall eines Paketverlustes in HTTP/2, da nur eine TCP Verbindung verwendet wird, alle Übertragungen über diese Verbindungen warten müssen, bis das verlorene Paket wieder übertragen wurde.
- Latenz verringern durch Verringerung der Roundtips beim Verbindungsaufbau
- Latenz verringern durch Nutzung von Forward Error Correction, welches die erneute Übertragung von Daten bei Übertragungsfehler verringern soll.
- Verbesserte Performance für Netzwerkwechsel(z.B.: von WiFi auf Mobilfunk).
- Überlaststeuerung, welche vergleichbar zu TCP ist, jedoch auch Streamebene statt auf Verbindungsebene.
- Reduzierte Bandbreitennutzung und Reduzierung der Paketanzahl.
- Gleiche Sicherheits- und Privatsphäre- Zusicherungen wie TLS
- Skalierbarkeit auf Client und Server.
- Simulation von TCP.

(vgl. [Ros13, S. 6 ff. Abschnitt GOALS]) (vgl. [LRW+17, S. 183, 184])

Es wurde entschieden, ein neues Protokoll zu schreiben, statt TCP zu ändern, da eine Änderung am TCP Standard lange dauern würde und noch länger bräuchte, bis sie von der Mehrheit der Infrastruktur unterstützt würde (vgl. [Ros13, S. 7]). QUIC ist ähnlich zu SDPY dahingehend, dass über Frames kommuniziert wird (vgl. [IT20, S. 7]). Statt wie TCP, die Tupel (Quell IP, Quell Port, Ziel IP, Ziel

Port, Protokoll) zur Identifizierung der Verbindung zu nutzen, hat jeder Stream eine 64-Bit Id (vgl. [LRW+17, S. 187]).

Da der Client nicht vorher weiß, ob ein Server HTTP/3 unterstützt, teilt der Server dies dem Client durch das Setzen des alt-svc Headers mit. Da HTTP/3 noch nicht standardisiert ist gibt es für jeden Draft eine Protokoll Id, nach dem Schema h3-<Draft Version>. Die HTTP/3 Versionen, welche auf Google QUIC basieren, haben die Protokoll Id h3-Q<Version>. Bei späteren Verbindungen wird eine QUIC und eine TLS-/TCP-Verbindung aufgebaut und die Verbindung, welche als erstes aufgebaut ist, wird verwendet. Um QUIC zu priorisieren wird die TCP-Verbindung um bis zu 300 ms verzögert, so dass meist QUIC verwendet wird und die TCP-Verbindung nur als Fallback genutzt wird, falls die QUIC-Verbindung fehlschlägt (vgl. [LRW+17, S. 187]).

2015 begann die Standardisierung von QUIC unter der IETF (vgl. [IS15]). In der IETF-Version wird zum Aufbau sicherer Verbindungen TLS 1.3 verwendet, dieses unterstützt 1-Roundtrip-Verbindungen oder auch 0-Roundtrip-Verbindungen. TLS-Informationen werden in QUIC über den CRYPTO Frame übertragen (vgl. [TT20, S. 8,9]). Da QUIC einige Aufgaben, welche vorher HTTP/2 hatte übernimmt, wie das Multiplexen der Verbindungen, wird der HTTP over QUIC Standard entwickelt, welcher dann in HTTP/3 umbenannt wurde (vgl. [Bis20, S. 5]).

Eine 0-RTT-Verbindung wird dadurch erreicht, dass der Client, wenn dieser sich das erste mal mit dem Server verbindet, die statischen Daten, welche für Authentifizierung des Servers und die weitere Verschlüsselung notwendig sind, von Server gesendet bekommt und diese gespeichert werden. Bei weiteren Verbindungen wird die erste Anfrage mit diesen Daten verschlüsselt, so dass es bei erneuten Verbindungen keinen Roundtrip zum Aufbau der Verschlüsselung benötigt wird. Der erste Roundtrip ist aus diesem Grund allerdings gegen Replay Angriffe anfällig (vgl. [LRW+17, S. 185]).

Um QUIC plan gerecht zu standardisieren wurde entschieden, einige Funktionen von QUIC v1 nach QUIC v2 zu verschieben. Diese sind:

- Forward Error Correction. Dies wurde von Google getestet hatte aber keinen signifikanten Einfluss auf die Latenz (vgl. [HISW16] [LRW+17, S. 193,194]).
- Multipath. Durch Multipath können mehrere Netzwerkwege gleichzeitig verwendet werden, um die Redundanz und Bandbreite der Verbindung zu erhöhen.
- Unzuverlässige Datenströme. Mit diesen soll UDP simuliert werden könnten.
- Die Unterstützung für nicht HTTP-Protokolle.

(vgl. [Ste20])

Im Oktober 2020 hat Google bekanntgegeben, HTTP/3 im Chrome Browser standardmäßig zu aktivieren. Dafür wird die Protokoll Id h3-29 verwendet welche aber auch für Draft 30 und 31 genutzt wird. Google hat bei der Nutzung von IETF QUIC eine Reduzierung der Google Suche Latenz von 2% gemessen (vgl. [SYS20]).

2.4 Bisherige Tools zur Messung der Performance

Untersuchungen zur Performance von HTTP wurden bereits während des Standardisierungsprozesses von SDPY durchgeführt. Im Paper "A comparison of SPDY and HTTP performance" [PN12] wird eine Chrome Installation ferngesteuert und es wird die Latenz der Webseiten mit einer bestimmten Bandbreite gemessen, diese werden ohne und mit SSL oder Pipelining durchgeführt. Es wurden allerdings Microsoft interne Tools verwendet. Im Paper "Can SPDY really make the web faster?" [ETW14] wird auch Chrome ferngesteuert und Wireshark wird verwendet, um die Zeit des ersten und letzten Paketes zu messen und daraus die Latenz zu berechnen. Es wurden live Tests gegen reale Webseiten und emulierte Tests gegen einen lokalen Server durchgeführt. Bei den emulierten Tests wurde die Bandbreite und der Paketverlust mit dem Programm `tc`[1] verwaltet. Für das Paper "How Speedy is SPDY?" [WBKW14] wurde ein Programm entwickelt, welches das Laden einer Website simulieren kann. Dafür muss kein Browser verwendet werden,

[1] Unter Debian Buster im `iproute2` Paket

welcher selbst Rechnung durchführt, wodurch die Zeit zwischen Anfragen nicht deterministisch ist. Dies hat jedoch den Nachteil, das das verwendete Programm für jede neue HTTP Version angepasst werden muss, was bei dem im Paper genutzten Programm nicht für HTTP/3 geschehen ist.

3 Vorstellung des Untersuchungsobjekts

Untersucht wird eine Entwicklungsversion der Webseite matchbus.tours, welche in React neu geschrieben wurde. Diese Version wurde vom Unternehmen für diese Bachelorarbeit zur Verfügung gestellt. Es handelt sich um eine Single-Page Application d.h. als HTML Datei existiert nur die Indexdatei, alle Unterseiten werden dynamisch durch Javascript erzeugt. Die Website verwendet Code-Spliting, das durch webpack erzeugte Javascript Bundle wird in mehrere Dateien geteilt, dies erlaubt das Codeteile lazy nachgeladen werden können, sobald sie benötigt werden und nicht früher (vgl. [Fac20]). Bei Matchbus wird der Code anhand der Subseite, welche angezeigt werden soll, geteilt(Route-Splitting). Wie man an Tabelle 3.1 sehen kann, wird dadurch die Downloadgröße für normale Nutzer verringert, da die Chunks für das Administratoren Interface und Unternehmer Interface nicht heruntergeladen und verarbeitet werden müssen. Welche Route welchen Chunck benötigt, ist aus Tabelle 3.2 zu entnehmen. Das Wissen, welche Chuncks welche anderen Chuncks benötigen, hilft dabei, HTTP/2 Server Push zu nutzen. Ob die Nutzung dieser Technologie sinnvoll ist, wird in Kapitel 4.3.4 untersucht. Als Server wird das Paket `net/http` der Go Standartbibliothek der Version 1.13 verwendet.

Chunck	Größe in kB(nicht kompri-miert)	Beschreibung	Abhängigkeiten
main.js	420	Ansichten für nicht eingeloggete Nutzer	6
legal.js	20	Ansichten für Impressung, Datenschutz	main
loggedIn.js	98	Ansichten für eingeloggete Nutzer	main, 9
companyProfile.js	6	Ansicht für die Unternehmenspräsentation	main, 7
adminPanel.js	55	Ansicht für Administratortools	main, 8
6.js	743	NPM Abhängigkeiten, unter anderen React, @react-google-maps, react-datepicker	
7.js	244	NPM Abhängigkeiten, unter anderen pdfjs	
8.js	813	NPM Abhängigkeiten, unter anderen ace editor	
9.js	72	NPM Abhängigkeiten, unter anderen qr.js, react-calendar	
main.css	29	Alles Selbstgeschriebe CSS	
6.css	23	NPM Abhängigkeiten	
9.css	3	react-calendar	

Tabelle 3.1: Alle Erzeugte Javascript Chunks [1]

[1] Für Commit 555dfd6253664f0f2502a8e1897d1126c5c2c9d0

Route	Chunck
/, /busRequest, /news, /travelService, /register, /login, /travelCompany, /forgotPassword, /landing/bus, /landing/marketing, /landing/busrequest	main.js, main.css, 6.js, 6.css
/profile, /management, /fleet	loggedIn.js, main.js, main.css, 6.js, 6.css, 9.js, 9.css
/imprint, /dataprotection, /contact	legal.js, main.js, main.css, 6.js , 6.css
/companyprofile/*	companyProfile.js, main.js, main.css, 6.js, 6.css, 7.js
/adminPanel/*	adminPanel.js, main.js, main.css, 6.js, 6.css, 8.js

Tabelle 3.2: Welche Route braucht welchen Chunck, nicht transitiv

4 Untersuchung der Performance

In diesem Abschnitt werden die Untersuchungskriterien definiert und ein Aufbau um diese zu messen vorgestellt.

4.1 Untersuchungskriterien

Die Untersuchungskriterien leiten sich aus den Neuerungen des HTTP-Standards ab. Diese Kriterien wurden gewählt, um zu überprüfen, ob die neuen Versionen des HTTP Protokolls ihr Ziel erreicht haben und ob sich deshalb ein Wechsel aus Ressourcensicht und für die Search Engine Rankings, bei dessen Berechnung die Latenz mit gewertet wird, für das Unternehmen lohnt.

Folgende Kriterien werden betrachtet betrachte:

Latenz: Es wird die Zeit, bis alle Dateien, welche für die angefragte Seite benötigt werden, vom Server zum Client übertragen wurden, betrachtet. Dies ist wichtig, da dadurch die Website schneller angezeigt wird, was sich positiv auf das Google Search Ranking auswirkt(vgl. [WP 18]). Also umso geringer die Latenz, umso höher das Ranking.

Benutzte Bandbreite: Je geringer die genutzte Bandbreite ist, umso mehr Anfragen können von einem Server gleichzeitig behandelt werden.

Genutzte CPU Zeit: Umso weniger CPU-Zeit genutzt wird, umso mehr Anfragen können von einem Server behandelt werden.

Verhalten bei Wechsel des Netzwerkes: Es wird die Latenz untersucht, wenn mitten in der Anfrage das Netzwerk gewechselt wird. Bei QUIC wird der Stream über seine Id statt über IP's und Port's identifiziert, dadurch soll eine Verbindung nach einem Verbindungsabbruch schneller wieder aufgebaut werden können.

4.2 Aufbau der Untersuchung

In diesem Abschnitt wird der Aufbau der Messungen beschrieben und die Auswahl von Nginx und Google Chrome als Software für Server und Client begründet. Weiterhin werden die Probleme, welche im Zusammenhang mit HTTP/3 und QUIC aufgetreten sind, beschrieben. Die Quellen für den Server-Aufbau sowie weitere Tools sind online verfügbar [1].

4.2.1 Server

Als Server wird Nginx 1.16.1 mit BoringSSL als Implementierung der Verschlüsselung verwendet. Diese Version von Nginx wurde ausgewählt, da es für diese Version einen Patch von Cloudflare's quiche Projekt gibt, welcher HTTP/3 über QUIC hinzufügt. Damit der Aufbau vergleichbar bleibt, wird auch für HTTP/1 und HTTP/2 BoringSSL als OpenSSL Ersatz verwendet. Alle HTTP-Varianten wurden mit HTTPS über TLS1.3 getestet, da HTTP/3 TLS1.3 als Voraussetzung hat und eine Änderung der Verschlüsselungsmethode Auswirkungen auf die Messung der CPU-Zeit haben könnte. Der Server wird auf einem Raspberry Pi 4GB in lokalen Netzwerk gehostet, es wird das Betriebssystem Raspian verwendet, der Server läuft also in einer 32 Bit Umgebung. Es wird ein selbst erstelltes SSL-Zertifikat verwendet, welches auf dem Client in den vertrauenswürdigen Zertifikatspeicher hinzugefügt wurde.

Der Server läuft in einem Docker Container, welcher genutzt wird, um die CPU-Zeit zu messen, außerdem ermöglicht dies ein einfaches Hinzufügen der Netzwerkma-

[1] https://gitlab.imn.htwk-leipzig.de/akrahl/bachelorarbeit

nipulationsschicht. Die Containerisierung erlaubt es den Testaufbau schnell und einfach auszutauschen, sowie die Netzwerkmanipulationsschicht auch auf anderen Webserver Containern zur Messung auszusetzen. Um die Verfügbarkeit von HTTP/3 zu kommunizieren wird der gleiche `alt-svc` Header wie bei maps.google.com verwendet, nur mit verkürzter Gültigkeit, da die HTTP Version beim Durchführen der Messungen häufig gewechselt wird. Dies war nötig, da der Google Chrome Browser eine Verbindung über Google QUIC priorisiert und Google QUIC eine andere Protokoll Id als die HTTP/3 Draft Implementierung, welche den IETF QUIC Standard nutzt, besitzt.

Zur Simulation von Paketverlust wird das Programm `tc` verwendet, dafür wird ein Docker Image erstellt, welches auf dem Server Image basiert und das Programm hinzufügt und konfiguriert. Die Werte für `BURST`, `RATE`, `LATENCY` und `LOSS` werden über Umgebungsvariablen gesetzt, welche dem Container übergeben wurden. Wird ein Paketverlust und eine Limitierung der Bandbreite konfiguriert, so werden Pakete zuerst verworfen und dann, wenn sie nicht verworfen wurden, in die Warteschlange zur Limitierung der Bandbreite eingereiht.

Zur Begrenzung der Bandbreite mit `tc` wird ein Token Bucket Filter genutzt. Bei einem Token Bucket Filter benötigen ausgehende Pakete Tokens, welche zeitabhängig erzeugt werden, um versendet zu werden. Dabei können maximal `BURST` Token im Bucket gehalten werden. Sind keine Token vorhanden, wird das Paket in eine Warteschlange eingereiht. Sollte die Warteschlange voll sein, wird das Paket verworfen. Das Paket in der Warteschlange wird erst versendet, wenn genug Token vorhanden sind. Die Minimale Größe von `BURST` ist $gewuenschteBandbreite/HZ$ wobei HZ der Wert der Kernel Konfiguration `CONFIG_HZ` ist, es handelt sich bei HZ um einen Wert zu Konfiguration des Linux Kernel Timer Systems. Bei kleineren Werten als den errechneten, können mehr Token pro Zeitschritt erzeugt werden als der Bucket halten kann. `LATENCY` bestimmt die maximale Zeit, welche ein Paket in der Warteschlange verweilen kann bevor es verworfen wird. Dieser Wert sollte angemessen groß sein, da sonst eine erhöhte Anzahl an Paketverlusten von der TCP Implementierung gemessen wird, was sich auf die Größe des TCP Fensters auswirkt (vgl. [HK]). Messungen wurden mit `LATENCY` Werten von 1 ms und 1000 ms

durchgeführt. Bei dem höheren Wert hatten alle HTTP Versionen 1-2% geringere Latenz.

4.2.2 Client

Als Client wird Google Chrome Dev(Version 86) verwendet, Cache wurde deaktiviert, QUIC aktiviert, Remote debugging ist auf Port 9222 geöffnet(siehe [2]). Der Test läuft auf dem Betriebssystem Fedora 32, welches den Linux Kernel 5.7 verwendet. Der Client ist per Ethernet direkt mit dem selben Router verbunden wie der Server. Der Router hat eine maximale Interface Bandbreite von 1Gbit/s.

4.2.3 HTTP/2 Server Push

Mit HTTP/2 wurde die Server Push Funktion hinzugefügt. Diese ermöglicht es dem Server auf eine Anfrage hin noch zusätzliche Ressourcen zu übertragen, statt nur der angeforderten Ressourcen. Damit es nicht dazu kommt, dass der Server etwas sendet, der Client die gesendete Ressource jedoch angefragt hat bevor die Übertragung des Servers eintrifft (vgl. Zeichnung 4.1), kann der Server eine `PUSH_PROMISE` senden (vgl. Zeichnung 4.2), welche die Anfrage Header für die zu pushende Datei enthält (vgl. [BPT15, S. 40]).

[2] https://gitlab.imn.htwk-leipzig.de/akrahl/bachelorarbeit/-/blob/master/tools/launch-chromium.sh

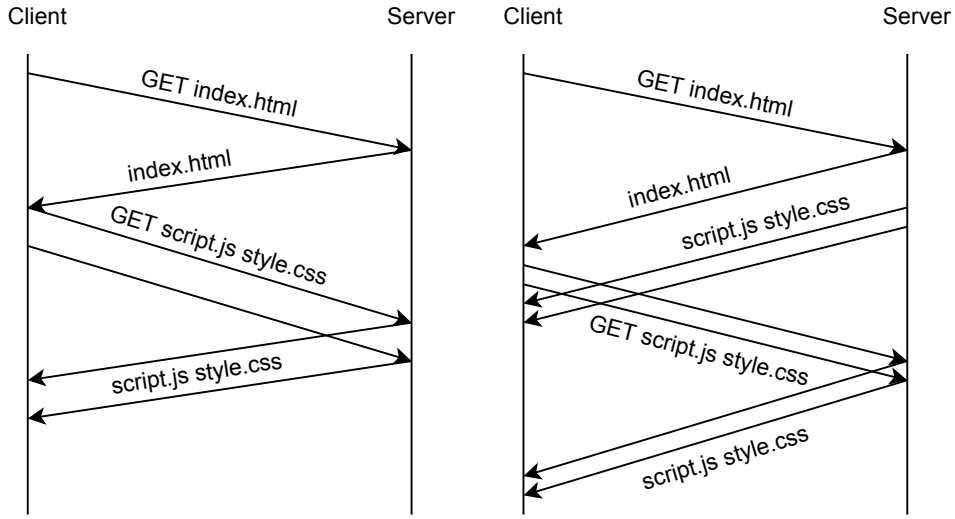

Abbildung 4.1: HTTP Anfrage und HTTP Anfrage mit Push aber ohne `PUSH_PROMISE`

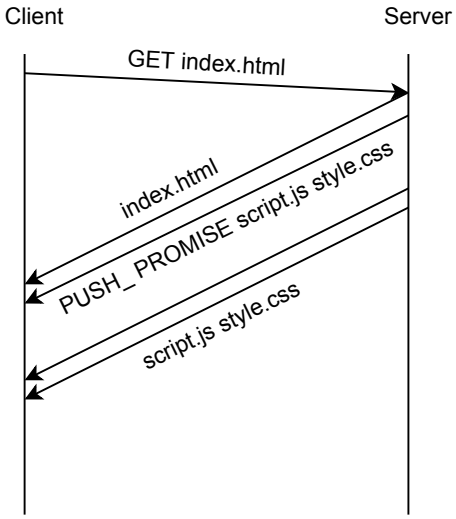

Abbildung 4.2: HTTP Anfrage mit `PUSH_PROMISE`

Da durch Code-Splitting(Kapitel 3) bereits bekannt ist welche Dateien von welcher Route benötigt werden, werden die beschriebenen Messungen auch an einen Aufbau mit HTTP/2 Server Push durchgeführt. Um die Konfigurationsdateien, welche NGINX für HTTP/2 Server Push benötigt zu erzeugen, wurde ein Tool entwickelt, welches die Konfigurationsdateien anhand eines Aufgezeichneten Websitenaufrufes in HAR Format erzeugt.

4.2.4 Durchführung der Messung

Zur Messung der Latenz wird `chrome-har-capturer`[3] verwendet, dies ist ein Tool, welches Google Chrome über das Devtools Protokoll fernsteuert und als Ausgabe eine Datei generiert, welche HTTP-Anfragen und deren Antworten enthält. Der Browser Cache ist dabei deaktiviert. Diese Dateien enthalten Daten im HTTP Archive(HAR) Format. In dieser Datei sind alle HTTP Anfragen und Antworten aufgezeichnet sowie deren Laufzeit. Da die Website Inhalte von anderen Anbietern herunterlädt, auf deren Serverkonfiguration man keinen Einfluss hat, wurde für diese Arbeit das `har-filter` Tool entwickelt. Dies ermöglicht es die HTTP Anfragen anhand einer Domain Whitelist zu filtern. Das Tool berechnet auch die Zeit, welche von den Anfragen benötigt wurde. Es werden 10 Messungen durchgeführt, die statistischen Ausreißer werden entfernt. Dann wird der Durchschnitt der restlichen Zeiten gebildet. Um eine größere Anzahl an Anfragen automatisiert durchführen zu können, wurden einige Scripts entwickelt.

Zur Messung der CPU-Zeit wird der Docker Container genutzt. Diese Container nutzten die `cgroups` Funktion des Linux Kernels zur Verwaltung ihrer Ressourcen. Auf Raspian kann man unter

```
/sys/fs/cgroup/cpu,cpuacct/docker/<container id>/cpuacct.usage
```

die verwendete CPU-Zeit des Containers in Nanosekunden auslesen. Diese Messung wird jeweils für 100, 50 und 10 Anfragen ausgeführt und es wird jeweils der Mittelwert von Zeit pro Anfrage gebildet.

Zur Messung der Bandbreite werden die gesendeten und empfangen Pakete mit `tcpdump`[4], durch den Befehl `sudo tcpdump host <Server IP> -w <Ausgabedatei>`, und `udpdump`, mit dem Wireshark Programm aufgezeichnet. Die Aufzeichnung wird dann in Wireshark geöffnet, welches ermöglicht, alle Pakete, welche nicht mit der Server IP in Verbindung stehen, herauszufiltern. Über Statistics > Conversations wird dann die Größe des Traffics der IPv4 Pakete zwischen Server und Client ausgelesen.

[3] https://github.com/cyrus-and/chrome-har-capturer
[4] Unter Debian Buster im `tcpdump` Paket

Der Server wird unter verschieden Netzwerkbedingungen getestet. Die erste Umgebung ist eine ohne künstliche Veränderungen, diese ist nur durch die verwendete Hardware beschränkt. Mit Blick auf die immer größere weltweite Verbreitung von Smartphones werden die Kriterien auch unter der Wirkung von zwei weiteren Einflüssen getestet: Paketverlust und Limitierung der verfügbaren Bandbreite. Gemessen wird in einer Umgebung mit 5% sowie 10% zufälligen Paketverlust und/oder einer Limitierung der Bandbreite auf 4 Mbit. 4 Mbit entspricht dabei dem Download einer LTE-Verbindung (vgl. [Con19]). LTE ist in Deutschland bei der Telekom am weitesten verbreitet mit einer Abdeckung von $> 80\%$ in ländlichen Regionen und bei O2 am geringsten mit einer von $> 50\%$ (vgl. [Fen20]). Dazu wird das Programm `tc` auf dem Server verwendet, um den Datenfluss zu beeinflussen. Zu beachten ist dabei, dass dieses Programm nur den Datenfluss vom Server zum Client manipulieren kann. In dem nachfolgenden Test ist Burst auf 50000 und Latency auf 1000ms gesetzt. Das Beschränken der Bandbreite mit dieser Methode verhindert für Latency Werte $>= 50$ ms das Aufbauen einer QUIC Verbindung, weshalb für HTTP/3 Latency auf 50 ms reduziert wurde. Dieser Wert wurde Experimentell ermittelt indem der Latency Wert verkleinert wurde bis eine Verbindung aufgebaut werden konnte.

4.2.5 Probleme beim Aufbau

Ursprünglich sollte der HTTP Server, welcher bei Matchbus bereits verwendet wird, getestet werden. Dieser nutzt die HTTP Implementierung der Go Standartbibliothek und leitet je nach angefragter URL an einen Fileserver oder ein Proxy zur Datenbank weiter. HTTP/1 und HTTP/2 werden von der GO Standardbibliothek unterstützt, als HTTP/3 Implementation wurde quic-go ausgetestet, es konnte jedoch keine QUIC Verbindung aufgebaut werden. Da der Server hier nur als Reverse-Proxy zur Datenbank und Fileserver für die Website agiert, die Nutzung von Go also nicht zwingend ist, wurde zu NGINX als Server Software gewechselt, da die Dokumentation des Aufbaus eines HTTP/3 Servers dafür ausführlicher ist.

Auf Windows gab es das Problem, dass weder Firefox Nightly noch Google Chrome Canary eine HTTP/3 Verbindung zum Server aufbauen konnten. Daraufhin wurde zu Firefox Nightly auf Fedora 32 gewechselt, wo eine HTTP/3 Verbindung aufgebaut werden konnte. Das DevTool Protokoll, welches von `chrome-har-capturer` zur Automatisierung der Anfragen genutzt wird, ist in Firefox noch nicht vollständig implementiert, weshalb dieses Tool nicht funktioniert und schließlich Chrome Unstable genutzt werden musste.

4.3 Auswertung der Ergebnisse

Es ist zu erwarten, dass HTTP/3 in den gemessenen Metriken ein geringeres Ergebnis hat als HTTP/2, da möglicherweise noch nicht die optimalsten Anforderungen Standardisiert sind. Die Implementation ist noch nicht so optimiert, wie die der älteren HTTP Versionen, da HTTP/1 vor 23 Jahren und HTTP/2 vor 4 Jahren veröffentlicht wurden und deshalb mehr Zeit in die Optimierung der Performance von Implementierungen des Standards investiert werden konnte. Da es sich bei HTTP/3 noch um eine Draft Version handelt, ist zu erwarten, dass sich HTTP/3 in den gemessen Metriken weiter verbessern wird. Zum Beispiel wurde die TCP Implementation im Linux Kernel über viele Jahrzehnte optimiert, weil TCP mehr als UDP verwendet wird.

Die hier folgenden visualisierten Daten sind auch Online verfügbar: HTWK Gitlab.

4.3.1 Umgebung: keine Beschränkung

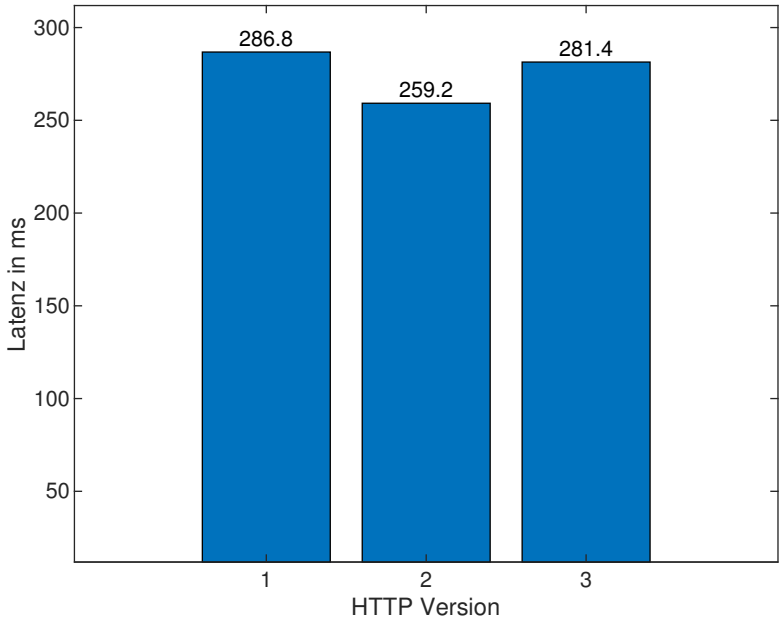

Abbildung 4.3: Latenz für HTTP Version

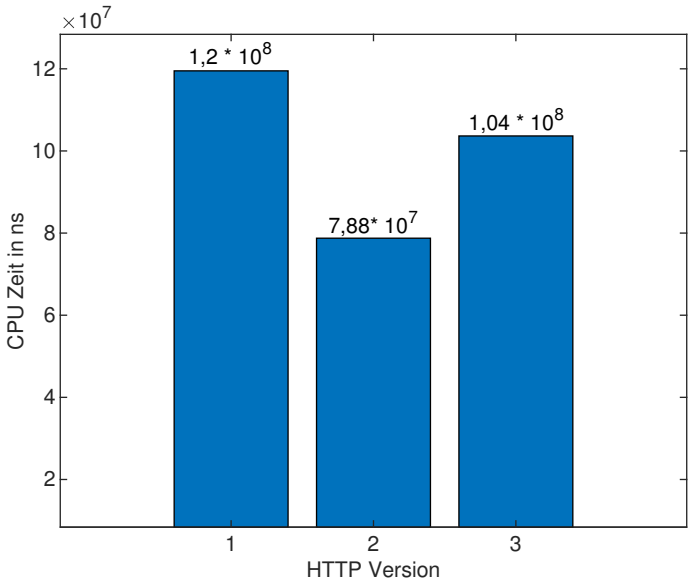

Abbildung 4.4: CPU Zeit pro Anfrage für HTTP Version

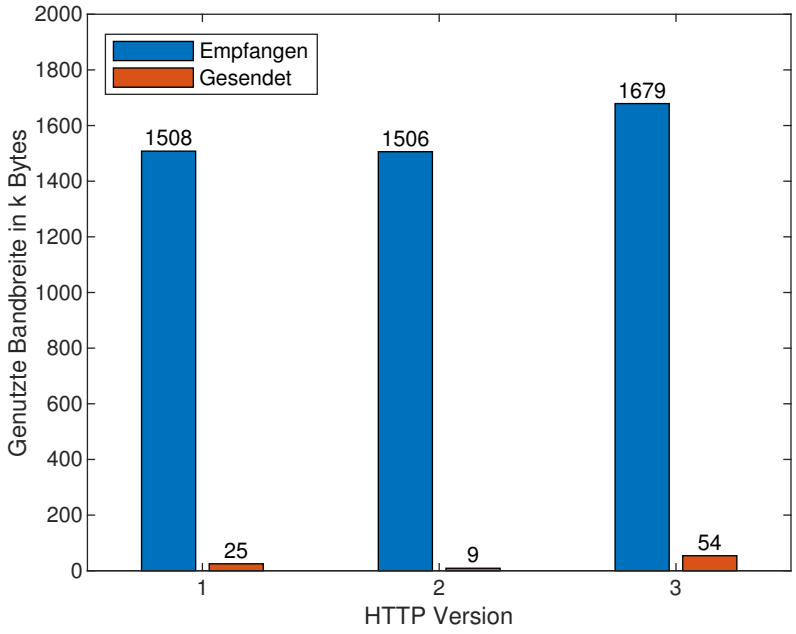

Abbildung 4.5: Bandbreite für HTTP Version, gerundet auf volle k byte

Latenz: Wie in dem Diagramm 4.3 zu sehen, hat HTTP/2 eine geringere Latenz als HTTP/1. HTTP/3 ist bereits jetzt schneller als HTTP/1, liegt aber noch 20 ms hinter HTTP/2 zurück. Bei diesen Messdaten gab es statistische Ausreißer mit 1185 ms bei HTTP/1 und 2034 ms bei HTTP/3. Diese wurden nicht zur Durchschnittsbildung für Diagramm 4.3 verwendet.

CPU Zeit: Erwarten würde man in Diagramm 4.4 ein Ansteigen der benötigten CPU-Zeit mit jeder HTTP Version, um den erweiterten Funktionsumfang zu unterstützen. Dies ist jedoch nicht der Fall, HTTP/2 und 3 benötigen weniger CPU-Zeit als HTTP/1. Das ist möglicherweise darauf zurückzuführen, dass HTTP/2 und 3 Binärprotokolle sind und deshalb keine Stringmanipulation durchgeführt werden muss. In [LRW+17, S. 192] wurde eine 3.5 fache CPU Nutzung von QUIC im Vergleich zu TCP/TLS gemessen, es wurde im Vergleich zu dieser Messung die CPU-Last reduziert.

Genutzte Bandbreite: Unter Anderem sind im Diagramm 4.5 bei HTTP/2 im Vergleich zu HTTP/1 Reduzierungen bei der genutzten Bandbreite im Upload

sichtbar. Die Ergebnisse der Messungen schwanken, da die Größe der IPv4 Pakete gemessen wird und die Anzahl der Pakete in den Messungen um einen Wert schwankt. Der Unterschied an genutzter Bandbreite zwischen HTTP/1 und HTTP/2 ist deshalb hier insignifikant.

4.3.2 Umgebung: Beschränkt auf 4 Mbit Bandbreite

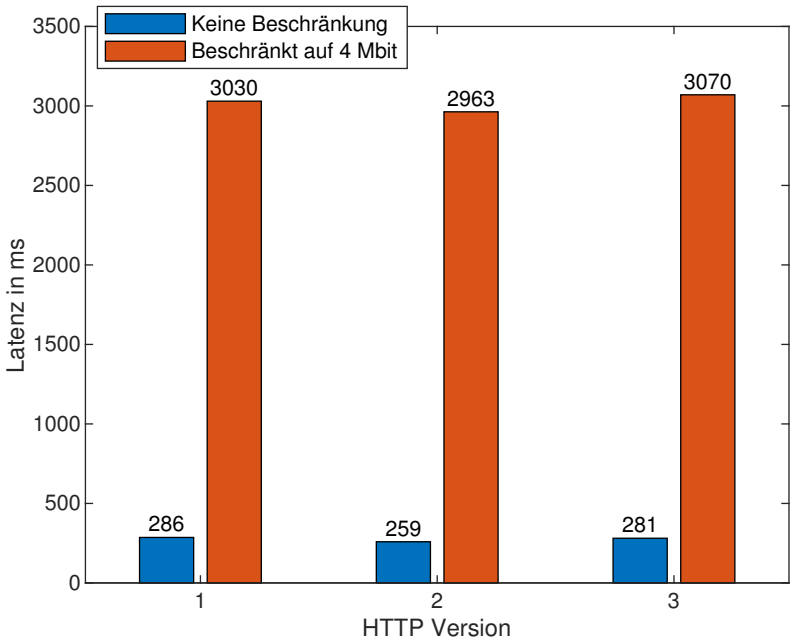

Abbildung 4.6: Latenz für HTTP mit einer Drosselung auf 4 Mbit in Vergleich zur Probe

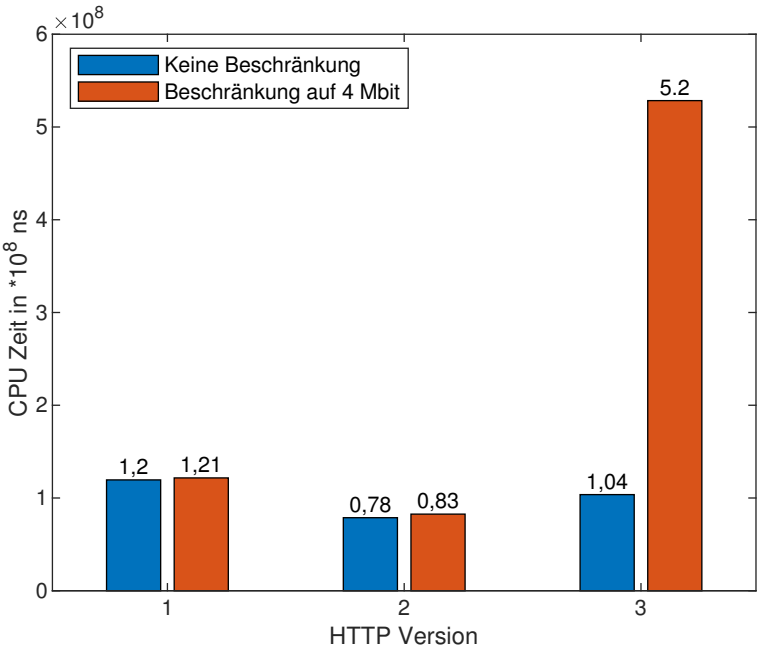

Abbildung 4.7: CPU Zeit für HTTP mit einer Drosselung auf 4 Mbit in Vergleich zur Probe

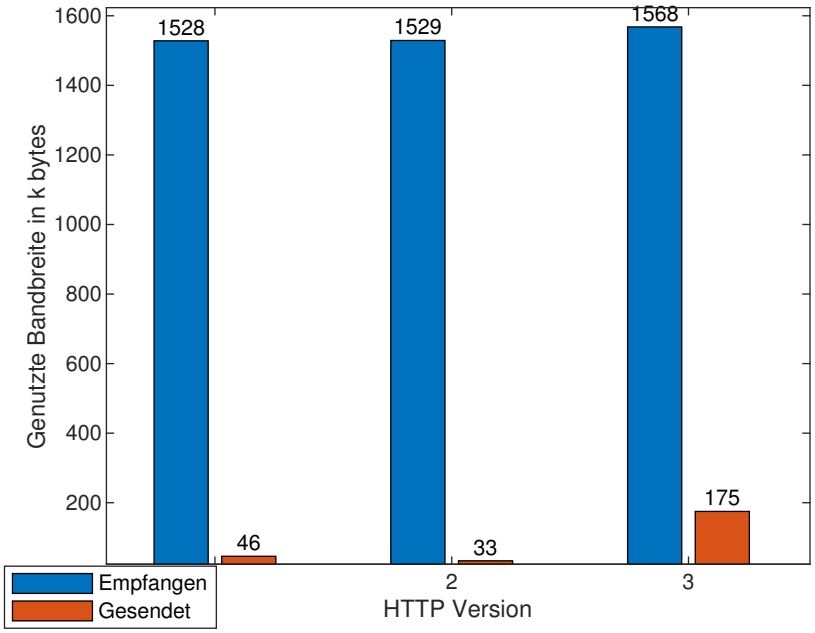

Abbildung 4.8: Bandbreite für HTTP Version, gerundet auf volle k Byte

Latenz: Bei einer Drosselung auf 4 Mbit zeigen sich in Diagramm 4.6 die gleichen Unterschiede auf, wie bei der nicht beschränkten Messung in Diagramm 4.3. Durch die Drosselung sind diese Unterschiede jedoch vernachlässigbar geworden, da alle Versionen eine hohe Latenz haben.

CPU Zeit: Aus Diagramm 4.6 kann man entnehmen, dass es bei einer Drosselung auf 4 Mbit nur einen geringen Anstieg der CPU-Zeit im Vergleich zwischen HTTP/1 und HTTP/2 gibt.

Genutzte Bandbreite: Aus den gemessenen Daten in Diagramm 4.8 lässt sich schließen, dass eine Drosselung der Bandbreite keinen Einfluss auf die benutzte Bandbreite besitzt.

4.3.3 Umgebung: Paketverlust

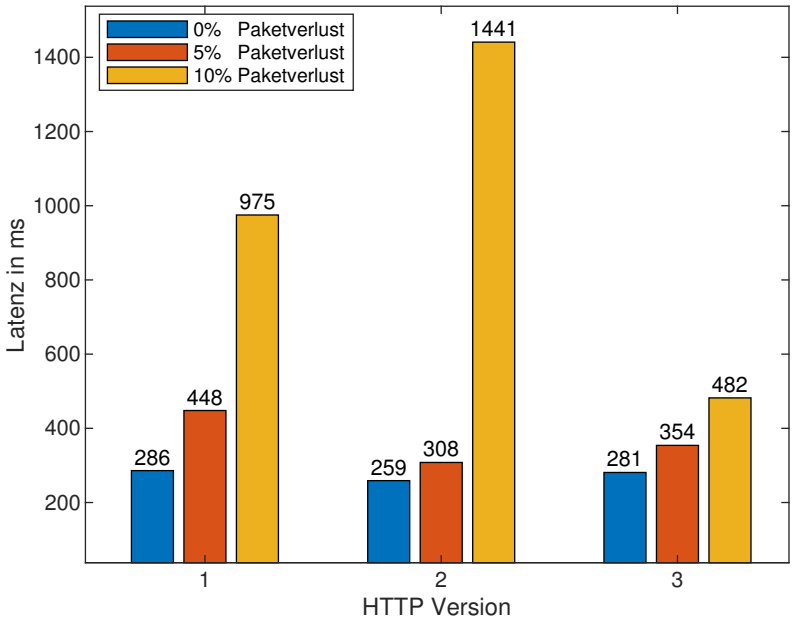

Abbildung 4.9: Latenz für HTTP mit einem Paketverlust von 5 und 10 Prozent

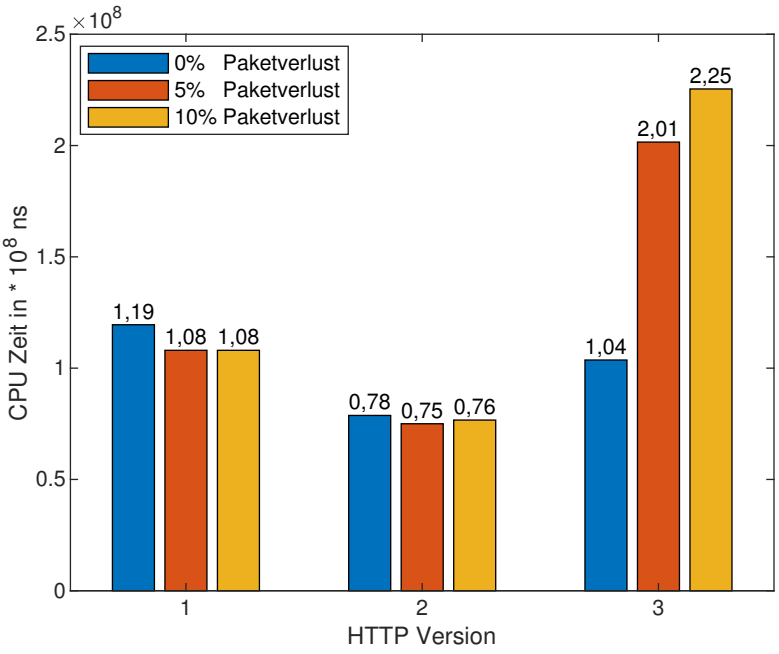

Abbildung 4.10: CPU Zeit für HTTP mit einem Paketverlust von 5 und 10 Prozent

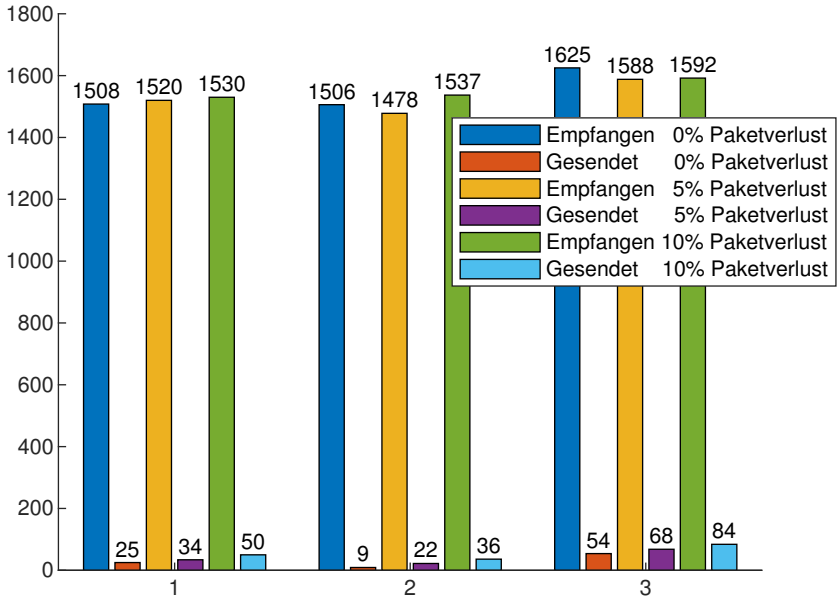

Abbildung 4.11: Genutzte Bandbreite für HTTP mit einem Paketverlust von 5 und 10 Prozent

Latenz: In Diagramm 4.9 ist zu beobachten, dass die in der unbeschränkten Umgebung festgestellten Verhältnisse zwischen den HTTP Versionen für 5% Paketverlust weiterhin gültig sind. Für einen Paketverlust von 10% hat HTTP/1 jedoch eine geringere Latenz als HTTP/2, dies ist darauf zurückzuführen, dass HTTP/2 nur eine TCP Verbindung aufbaut, bei der das Übertragungsfenster durch den hohen Paketverlust häufig verkleinert wird, da in den Algorithmen von einer Überlast im Netzwerk als Grund für Paketverlust ausgegangen wird. HTTP/3 hingegen zeigt im Vergleich den geringsten Anstieg der Latenz, es ist 50% schneller als HTTP/1, welches wiederum nur 66% der Latenz von HTTP/2 hatte.

CPU Zeit: In Diagramm 4.10 zeigt sich bei der genutzten CPU-Zeit ein ähnliches Verhalten wie bei Diagramm 4.7. Die benötigte CPU-Zeit bleibt bei HTTP/1 und HTTP/2 ungefähr gleich, Abweichungen sind so gering, dass diese vom Aufbau stammen könnten. Die für HTTP/3 benötigte CPU-Zeit verdoppelt sich hingegen.

Genutzte Bandbreite: Aus dem Diagramm 4.11 lässt sich nur ein geringer Anstieg der genutzten Bandbreite auslesen.

4.3.4 HTTP/2 Server Push

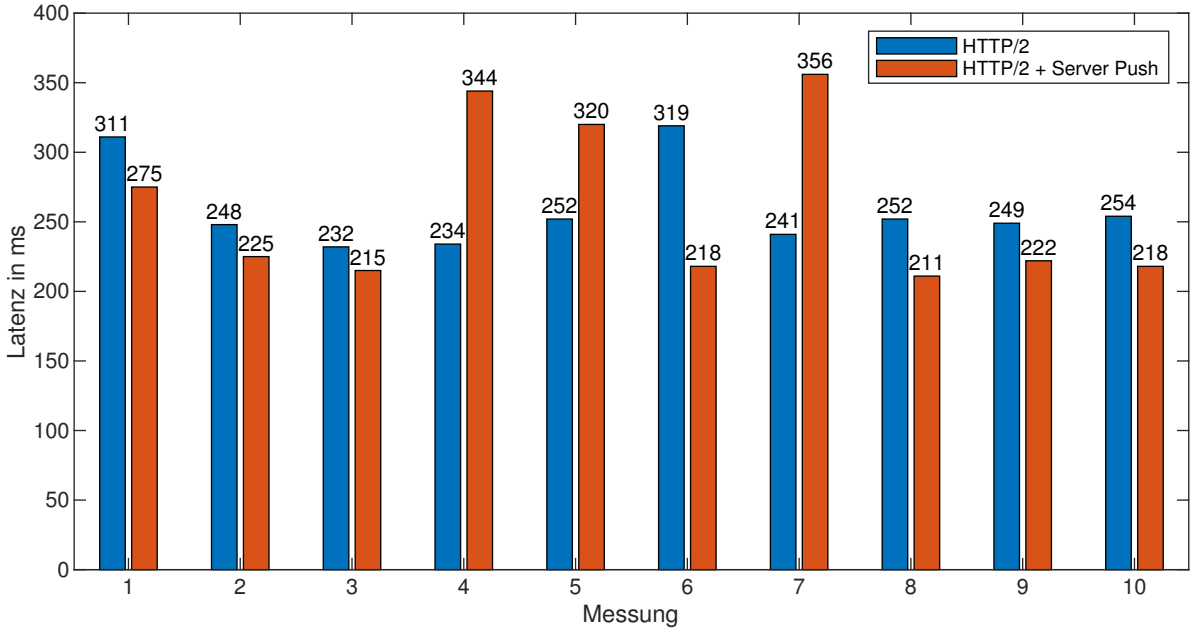

Abbildung 4.12: Latenz für HTTP/2 mit und ohne Server Push

Latenz: In Diagramm 4.12 ist erkennbar, dass mit Server Push eine geringere Latenz erreicht werden kann, jedoch ist es bei Server Push zu Ausreißern in der Zeit gekommen, wodurch der Durchschnitt für beide Werte ungefähr gleich ist. HTTP/2 kommt ohne Server Push auf eine Durchschnittliche Latenz von 259.2 ms und mit Server Push auf 260.4 ms.

CPU Zeit: HTTP/2 Server Push hat im Durchschnitt 90.25 ms an CPU Zeit zum Senden der Ressourcen benötigt, im Vergleich zu 78 ms, wenn es nicht benutzt wird.

Bei einer Limitierung der Bandbreite auf 4 Mbit hat HTTP/2 Server Push einige Prozent geringere Latenz im Vergleich zu HTTP/2. Bei dem Hinzufügen von Paketverlust hat HTTP/2 Server Push eine bessere Latenz als HTTP/2. Im Fall von 5% Paketverlust wurde eine 16% und bei 10% Paketverlust eine 47% bessere Latenz gemessen.

5 Fazit

Die Messungen konnten erfolgreich durchgeführt werden. Anhand der Messungen ist zu sehen, dass HTTP/2 zurzeit am besten bei den Kriterien Latenz und CPU Zeit abschneidet. HTTP/2 sollte bevorzugt werden, da es von fast allen Browsern unterstützt wird, wie man auf der Webseite [DS20] sehen kann. HTTP/1 sollte nur als Fallback für Internet Explorer bereitgestellt werden. Server Push sollte, wenn möglich und die Datei Abhängigkeiten der Anfragen bekannt sind, verwendet werden. Die Performance von HTTP/3 ist zu dieser Zeit schlecht einzuordnen, da der Standard noch nicht fertigstellt ist. HTTP/3 sollte deshalb nach einer Standardisierung und nach Optimierung der genutzten Server Software oder des Linux Kernels nochmals untersucht werden.

Eine Beschränkung des Aufbaues ist die Nutzung des Programms `tc` zur Manipulation des Traffics. Dass Programm beeinflusst nur Traffic vom Server zum Client. Es sollte jedoch wenig Einfluss auf die Messung haben, da der Server weitaus mehr Pakete und Daten an die Client überträgt als andersherum. Es gibt eine Lösung, damit dies auch beidseitig funktioniert. Dafür wird allerdings ein weiteres Netzwerkinterface benötigt, welches mithilfe eines Kernel Modules erstellt werden muss. Dies ist in einem Docker Container zwar möglich, dafür muss jedoch das Modul auf dem Docker Host installiert sein oder muss zum Start des Containers kompiliert werden. Dafür müsste der Docker Container Zugriff auf die Kernel Header haben entweder durch mounten dieser als Volume oder durch installieren der Header mithilfe des Paketmanagers der Distribution des Containers. Diese Einschränkungen machen das Dockerfile weniger portabel.

Mit `tc` kann nur zufälliger Paketverlust erzeugt werden, was einem Paketverlust durch Überlast simuliert. Es kann keine Simulation des Paketverlustes bei Netz-

werkwechsel durchgeführt werden, dies ist aber eine Metrik, welche bei QUIC verbessert werden sollte. Für eine solche Simulation benötigt man aber spezielle Software, welche zu einem bestimmten Zeitpunkt in der Messung einen Paketverlust hervorruft, wobei selbst in diesem Fall IP und Port gleich bleiben und keine neue TCP Verbindung etabliert werden muss, wie es bei einem Wechsel von WiFi auf Mobilfunk der Fall wäre. Bei einem zufälligen Paketverlust schwanken die gemessenen Werte für die verwendete Bandbreite sehr stark.

Die Messung der Latenz muss mehrfach durchgeführt werden, da die Werte der Messung nicht nur vom Server, sondern auch vom genutzten Client abhängen. Das ist kein Problem, da durch die genutzten Tools viele Messungen schnell hintereinander angefertigt werden können. Auch ermöglicht es die Varianz in der Latenz zu ermitteln, welche im Realgebrauch auftaucht.

Die Messung der CPU Zeit ist nur mit Werten, welche mit der gleichen Methode aufgenommen wurden, vergleichbar. Der Overhead, welcher für die Initialisierung des Programms anfällt, wird durch eine genügend große Anzahl an Anfragen im Mittel irrelevant, da der Ressourcenverbrauch der Anfragen viel größer ist. Die gemessenen Werte weichen jedoch von den Werten, welche von NGINX gemessen werden, ab. In den Logs ist 0 ms Verarbeitungszeit aufgezeichnet, dies ist die größte Auflösung, die NGINX besitzt. Die Anfrage wird in einem Durchlauf des internen Event loops abgearbeitet.

Abbildungsverzeichnis

1.1 Anzahl von Webseiten und Verbreitung verschiedener Technologien 1

4.1 HTTP Anfrage und HTTP Anfrage mit Push aber ohne `PUSH_PROMISE` 20
4.2 HTTP Anfrage mit `PUSH_PROMISE` 20
4.3 Latenz für HTTP Version 24
4.4 CPU Zeit pro Anfrage für HTTP Version 24
4.5 Bandbreite für HTTP Version, gerundet auf volle k byte 25
4.6 Latenz für HTTP mit einer Drosselung auf 4 Mbit in Vergleich zur Probe 26
4.7 CPU Zeit für HTTP mit einer Drosselung auf 4 Mbit in Vergleich zur Probe 27
4.8 Bandbreite für HTTP Version, gerundet auf volle k Byte 27
4.9 Latenz für HTTP mit einem Paketverlust von 5 und 10 Prozent . . 28
4.10 CPU Zeit für HTTP mit einem Paketverlust von 5 und 10 Prozent . 29
4.11 Genutzte Bandbreite für HTTP mit einem Paketverlust von 5 und 10 Prozent 29
4.12 Latenz für HTTP/2 mit und ohne Server Push 31

Tabellenverzeichnis

3.1 Alle Erzeugte Javascript Chunks 14
3.2 Welche Route braucht welchen Chunck, nicht transitiv 15

Literaturverzeichnis

[BB15] Bentzel, Chris ; Béky, Bence: *Hello HTTP/2, Goodbye SPDY*. https://blog.chromium.org/2015/02/hello-http2-goodbye-spdy.html. Version: Februar 2015

[Bis20] Bishop, Mike: Hypertext Transfer Protocol Version 3 (HTTP/3) / Internet Engineering Task Force. Version: Juni 2020. https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-29. Internet Engineering Task Force, Juni 2020 (draft-ietf-quic-http-29). – Internet-Draft. – Work in Progress

[BL91] Berners-Lee, Tim: *The HTTP Protocol As Implemented In W3*. https://www.w3.org/Protocols/HTTP/AsImplemented.html. Version: 1991

[BP12] Belshe, Mike ; Peon, Roberto: SPDY Protocol / Internet Engineering Task Force. Version: Februar 2012. https://datatracker.ietf.org/doc/html/draft-mbelshe-httpbis-spdy-00. Internet Engineering Task Force, Februar 2012 (draft-mbelshe-httpbis-spdy-00). – Internet-Draft. – Work in Progress

[BPT15] Belshe, Mike ; Peon, Roberto ; Thomson, Martin: *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. http://dx.doi.org/10.17487/RFC7540. Version: Mai 2015 (Request for Comments)

[BPTM12] Belshe, Mike ; Peon, Roberto ; Thomson, Martin ; Melnikov, Alexey: SPDY Protocol / Internet Engineering Task Force. Version: Februar 2012. https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-http2-00. Internet Engineering Task Force, Februar 2012 (draft-ietf-httpbis-http2-00). – Internet-Draft. – Work in Progress

[Con19] Contributors, MDN: *Throttling*. https://developer.mozilla.org/en-US/docs/Tools/Network_Monitor/Throttling. Version: Mai 2019

[Con20] Contributors, MDN: *Evolution of HTTP*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP. Version: September 2020. – Library Catalog: developer.mozilla.org

[Cor12] Corporation, MITRE: *CVE - CVE-2012-4929*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929. Version: 2012

[DS20] Deveria, Alexis ; Schoors, Lennart: *Can I use... Support tables for HTML5, CSS3, etc.* https://caniuse.com/http2. Version: September 2020

[ETW14] Elkhatib, Yehia ; Tyson, Gareth ; Welzl, Michael: Can SPDY really make the web faster? In: *2014 IFIP Networking Conference*, 2014, S. 1–9

[Fac20] Facebook: *Code-Splitting - React*. https://reactjs.org/docs/code-splitting.html. Version: Juli 2020. – Library Catalog: reactjs.org

[Fen20] Fenwick, Sam: *German rural 4G users see big differences in 4G Availability and download speeds between operators*. https://www.opensignal.com/2020/03/12/german-rural-

4g-users-see-big-differences-in-4g-availability-and-download-speeds-between-operators. Version: März 2020

[FNM+97] Fielding, Roy T. ; Nielsen, Henrik ; Mogul, Jeffrey ; Gettys, Jim ; Berners-Lee, Tim: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2068. http://dx.doi.org/10.17487/RFC2068. Version: Januar 1997 (Request for Comments)

[FR14a] Fielding, Roy T. ; Reschke, Julian: *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. http://dx.doi.org/10.17487/RFC7230. Version: Juni 2014 (Request for Comments)

[FR14b] Fielding, Roy T. ; Reschke, Julian: *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. http://dx.doi.org/10.17487/RFC7231. Version: Juni 2014 (Request for Comments)

[HISW16] Hamilton, Ryan ; Iyengar, Jana ; Swett, Ian ; Wilk, Alyssa: QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2 / Internet Engineering Task Force. Version: Januar 2016. https://datatracker.ietf.org/doc/html/draft-tsvwg-quic-protocol-02. Internet Engineering Task Force, Januar 2016 (draft-tsvwg-quic-protocol-02). – Internet-Draft. – Work in Progress

[HK] Hubert, Bert ; Kuznetsov, Alexey N.: *tc-tbf(8): Token Bucket Filter - Linux man page*. https://linux.die.net/man/8/tc-tbf

[IS15] Iyengar, Janardhan ; Swett, Ian: *I-D Action: draft-tsvwg-quic-protocol-00.txt*. https://mailarchive.ietf.org/arch/msg/i-d-announce/zSk53ClZRO6eSH4s5a7bQ5Jiuns/. Version: Juni 2015

[IT20] Iyengar, Jana ; Thomson, Martin: QUIC: A UDP-Based Multiplexed and Secure Transport / Internet Engineering Task Force. Version: September 2020. https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-30. Internet Engineering Task

Force, September 2020 (draft-ietf-quic-transport-30). – Internet-Draft. – Work in Progress

[LRW+17] Langley, Adam ; Riddoch, Alistair ; Wilk, Alyssa ; Vicente, Antonio ; Krasic, Charles ; Zhang, Dan ; Yang, Fan ; Kouranov, Fedor ; Swett, Ian ; Iyengar, Janardhan ; Bailey, Jeff ; Dorfman, Jeremy ; Roskind, Jim ; Kulik, Joanna ; Westin, Patrik ; Tenneti, Raman ; Shade, Robbie ; Hamilton, Ryan ; Vasiliev, Victor ; Chang, Wan-Teh ; Shi, Zhongyi: The QUIC Transport Protocol: Design and Internet-Scale Deployment. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA : Association for Computing Machinery, August 2017 (SIGCOMM '17). – ISBN 978–1–4503–4653–5, 183–196

[Net20] Netcraft: *Web Server Survey*. https://news.netcraft.com/archives/category/web-server-survey/. Version: September 2020

[NFBL96] Nielsen, Henrik ; Fielding, Roy T. ; Berners-Lee, Tim: *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. http://dx.doi.org/10.17487/RFC1945. Version: Mai 1996 (Request for Comments)

[NMM+99] Nielsen, Henrik ; Mogul, Jeffrey ; Masinter, Larry M. ; Fielding, Roy T. ; Gettys, Jim ; Leach, Paul J. ; Berners-Lee, Tim: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. http://dx.doi.org/10.17487/RFC2616. Version: Juni 1999 (Request for Comments)

[NMR16] Nottingham, Mark ; McManus, Patrick ; Reschke, Julian: *HTTP Alternative Services*. RFC 7838. http://dx.doi.org/10.17487/RFC7838. Version: April 2016 (Request for Comments)

[Not12] Nottingham, Mark: *Rechartering HTTPbis*. https://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html. Version: Januar 2012

[PN12] Padhye, Jitu ; Nielsen, Henrik F.: A comparison of SPDY and HTTP performance. Version: Juli 2012. https://www.microsoft.com/en-us/research/publication/a-comparison-of-spdy-and-http-performance/. 2012 (MSR-TR-2012-102). – Forschungsbericht

[Pro] Projects, Chromium: *SPDY: An experimental protocol for a faster web - The Chromium Projects*. https://www.chromium.org/spdy/spdy-whitepaper

[Ros13] Roskind, Jim: *QUIC: Design Document and Specification Rationale*. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit. Version: Dezember 2013

[Sta] Stats, Internet L.: *Total number of Websites - Internet Live Stats*. https://www.internetlivestats.com/total-number-of-websites/

[Ste20] Stenberg, Daniel: *QUIC v2*. https://http3-explained.haxx.se/en/quic-v2. Version: 2020

[SYS20] Schinazi, David ; Yang, Fan ; Swett, Ian: *Chrome is deploying HTTP/3 and IETF QUIC*. https://blog.chromium.org/2020/10/chrome-is-deploying-http3-and-ietf-quic.html. Version: Oktober 2020

[TT20] Thomson, Martin ; Turner, Sean: Using TLS to Secure QUIC / Internet Engineering Task Force. Version: Juni 2020. https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-29. Internet Engineering Task Force, Juni 2020 (draft-ietf-quic-tls-29). – Internet-Draft. – Work in Progress

[W3T20] W3Techs: *Historical yearly trends in the usage statistics of site elements for websites, October 2020*. https://w3techs.com/technologies/history_overview/site_element/all/y. Version: 2020

[WBKW14] Wang, Xiao S. ; Balasubramanian, Aruna ; Krishnamurthy, Arvind ; Wetherall, David: How Speedy is SPDY? In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA : USENIX Association, April 2014. – ISBN 978–1–931971–09–6, 387–399

[WP18] Wang, Zhiheng ; Phan, Doantam: *Using page speed in mobile search ranking*. https://webmasters.googleblog.com/2018/01/using-page-speed-in-mobile-search.html. Version: Januar 2018

Eidesstatliche Erklärung

Ich erkläre hiermit, dass ich diese Bachelorarbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Leipzig, 15. Oktober 2020 Unterschrift