Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Vincent Roederer

Peer-Assisted Content Distribution im Browser

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Vincent Roederer

Peer-Assisted Content Distribution im Browser

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Abgegeben am 13.07.2016

Vincent Roederer

Thema der Arbeit

Peer-Assisted Content Distribution im Browser

Stichworte

WebRTC, WebSocket, Peer-to-Peer, Browser, peerunterstützte Verteilung, JavaScript, Signaling, CDN

Kurzzusammenfassung

In dieser Arbeit wird eine Middleware für Browser entwickelt, die Inhalte von Webseiten über Peers statt vom Webserver bezieht. Dadurch werden Verkehrsmuster optimiert. In Tests wird gemessen, inwieweit dieses alternative Verfahren Vorteile bietet.

Vincent Roederer

Title of the paper

Browser-Based Peer-Assisted Content Distribution

Keywords

WebRTC, WebSocket, Peer-to-peer, Browser, peer-assisted Content Distribution, JavaScript, Signaling, CDN

Abstract

In this thesis a browser-based middleware is being developed which obtains website content from peers instead of web servers. Allowing traffic patterns to be optimized. Tests will be evaluated in order to determine the advantages of this procedure.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Viele Webseitenbetreiber haben mit erhöhtem Datenverkehrsaufkommen zu kämpfen. So kann es bei Überschreitung der verfügbaren Bandbreite zu langsam ladende Webseiten kommen, bis hin zur Gefährdung der Verfügbarkeit. Dies ist aufgrund der teuren Skalierbarkeit ein typisches Problem des Client-Server-Modells, das im Web Anwendung findet. Bisher war es in browser-gestützten Systemen nicht nativ möglich, diese Datenverkehrsmuster zwischen Client und Server in Client-Client-Muster zu überführen. Mit WebRTC, einem neuen Standard der in vielen Browsern implementiert ist, ist es möglich Peer-to-Peer-Verbindungen herzustellen, wodurch eine Alternative zur klassischen Stern-Topologie entsteht. Auf diesen Verbindungen lassen sich allgemeine Daten als auch ganze Video- und Audiostreams versenden. Mithilfe der WebRTC-Datenkanäle können so Inhalte einer Webseite an andere Browser gesendet werden, wodurch der Datenverkehr zwischen den Browsern verläuft und Webserver entlastet werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer generischen Middleware zur Verlagerung der Datenverkehrsmuster von Client-Server-Anwendungen im Web. Die Middleware dient dazu Inhalte von Webseiten über andere Peers zu beziehen statt vom ursprünglichem Webserver. Dazu ist eine JavaScript-Anwendung zur Steuerung des Browsers nötig und ein Signaling-Server der den Verbindungsaufbau zwischen Browsern ermöglicht und koordiniert.

Weiter werden Tests durchgeführt, um zu erfahren, inwiefern diese Verlagerung der Datenverkehrsmuster Webserver entlastet, mit welchen Einschränkungen sie verbunden sind und wie schnell Inhalte mit dieser Lösung aufgelöst werden können.

1.3 Gliederung der Arbeit

In Kapitel 2 werden elementare Begriffe und Konzepte erklärt, die zum Verständnis der Funktionsweise der Middleware notwendig sind. Darauf folgt eine Übersicht über

alternative Lösungen in Kapitel 2.6. In Kapitel 3 wird die Architektur der Middleware erklärt, wichtige Entscheidungen beleuchtet und auf Auffälligkeiten bei der Entwicklung eingegangen. Weiter werden in Kapitel 4 die durchgeführten Experimente beschrieben, die Ergebnisse präsentiert und diskutiert. Abgeschlossen wird in Kapitel 5 mit einer Schlussbetrachtung über die gewonnenen Erkenntnisse und einem Ausblick auf neue Lösungsansätze.

2 Grundlagen

Als Basis dieser Arbeit dient das Hypertext Transfer Protocol (HTTP) [1]. Es ist ein Anwendungsprotokoll, welches primär eingesetzt wird, um Hypertext-Dokumente, wie die Seitenbeschreibungssprache Hypertext Markup Language (HTML) [2] von einem Webserver in einen Webbrowser zu laden. Standardisiert wird HTTP von der Internet Engineering Task Force (IETF) [3] und dem World Wide Web Consortium (W3C) [4]. HTTP ist fundamental für die Bereitstellung der zu entwickelnden clientseitigen Middleware, bei der Beschaffung von Inhalten auf Webservern und teilt mit dem in Kapitel 2.1 beschriebenen WebSocket-Protokoll Teile des Verbindungsaufbaus.

In den Kapiteln 2.1, 2.2 und 2.5 werden Browser-Schnittstellen beschrieben, die von einer Webseite benutzt werden können, um bestimmte Aktionen durchzuführen. Dies sind:

- Die bidirektionale Kommunikation mit einem Webserver
- Die Kommunikation mit einem anderen Browser
- Das Abfangen von HTTP-Anfragen

Im Folgenden werden die Begriffe Client und Peer verwendet. Ein Client bezeichnet ein Programm, das auf einem Endpunkt ausgeführt wird und mit einem zentralen Server kommuniziert. Als Peer wird ein Teilnehmer in einem Peer-to-Peer-Netz bezeichnet.

2.1 WebSocket

Das WebSocket-Protokoll [5] ist ein Netzwerkprotokoll basierend auf TCP, welches Kommunikation zwischen einem WebSocket-Server und einem WebSocket-Client bidirektional ermöglicht. Genormt wird die Schnittstelle vom W3C und das Netzwerkprotokoll von der IETF. Es existieren bereits Schnittstellen wie Server-Sent Events [6], XMLHttpRequest [7] und Fetch [8] in Browsern, die Client-Server-Kommunikation ermöglichen, jedoch nicht mit den Eigenschaften von WebSocket. Diese besonderen Eigenschaften von WebSocket werden für die Implementierung dieser Arbeit benötigt, um Server und Clients schnellstmöglich über anstehende

Ereignisse zu informieren. Die Unterschiede der Schnittstellen werden im Folgenden genauer erläutert.

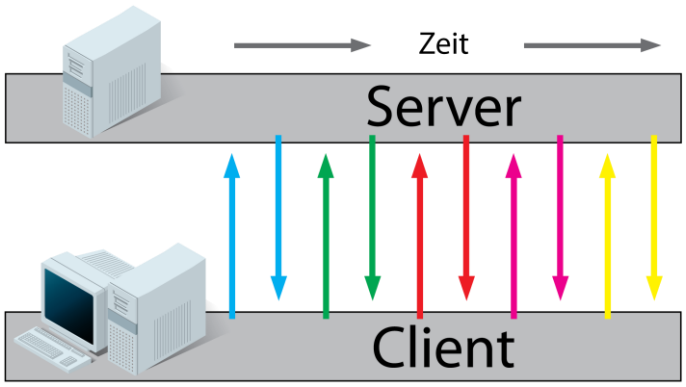

Abbildung 1: Nachrichtenverlauf bei XMLHttpRequest-/ und Fetch-Schnittstellen. Basierend auf [9]

In Abbildung 1 ist ein typischer Nachrichtenverlauf von XMLHttpRequest-/ und Fetch-Schnittstellen auf einer HTTP-Verbindung dargestellt. Die farbigen Pfeile symbolisieren HTTP-Anfragen und Antworten. Bei diesen Schnittstellen kann immer nur eine HTTP-Anfrage mit der dazugehörigen Antwort zurzeit bearbeitet werden. Damit ein Client eine Antwort vom Server bekommen kann, muss er also vorher eine Anfrage stellen. Es ist nicht möglich, dass ein Client eine weitere Anfrage stellt, bevor die Vorherige beantwortet wurde. Dies ist das Anfrage-Antwort-Verfahren. Um nun mehr Anfragen in weniger Zeit zu beantworten, gibt es mehrere Möglichkeiten.

Zum einen existiert die HTTP-Pipelining-Technik [10], die sowohl vom Client als auch vom Server unterstützt werden muss. Bei dieser Technik können mehrere HTTP-Anfragen gestellt werden, bevor eine Antwort empfangen werden muss. Um die einzelnen Antworten im Nachhinein zuordnen zu können, müssen die Antworten in derselben Reihenfolge versendet werden, wie die Anfragen beim Server eingetroffen sind. Diese Technik hat den Nachteil, dass eine längere Antwort alle dahinterstehenden Antworten blockieren kann. Durch diesen Nachteil und darüberhinausgehende Problemen findet diese Technik kaum Einsatz.

Zum anderen kann man mehrere HTTP-Verbindungen zum Server aufbauen auf denen jeweils das Anfrage-Antwort-Verfahren angewendet wird. Nachteil hier ist der zusätzliche Aufwand des Verbindungsaufbaus.

Letztendlich lässt sich sagen, dass diese Schnittstellen im Gegensatz zu WebSockets nicht bidirektional sind.

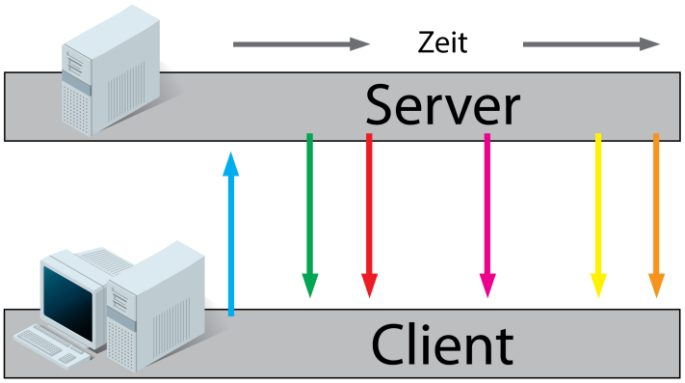

Abbildung 2: Nachrichtenverlauf bei Server-Sent Events. Basierend auf [9]

Abbildung 2 zeigt einen Nachrichtenverlauf bei Verwendung von Server-Sent Events. Anders als bei XMLHttpRequest-/ und Fetch-Schnittstellen kann der Client nach Versenden einer HTTP-Anfrage nur noch Nachrichten vom Server empfangen und selber keine Nachrichten mehr auf dieser HTTP-Verbindung versenden. Server-Sent Events haben den Vorteil, dass der Client nicht erst eine Anfrage stellen muss, um eine Antwort zu bekommen.

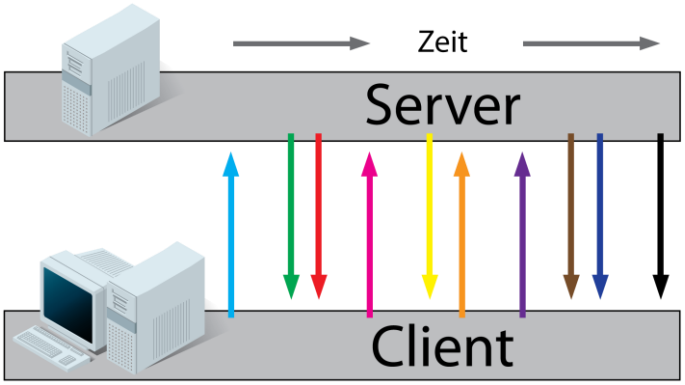

Abbildung 3: Nachrichtenverlauf WebSocket. Basierend auf [9]

Abbildung 3 zeigt nun den Nachrichtenverlauf bei WebSockets. Nachdem der Client die WebSocket-Verbindung initiiert hat, können sowohl Server als auch Client bidirektional Nachrichten versenden. Das Anfrage-Antwort-Verfahren gilt hier nicht mehr. Der Client muss also nicht mehr eine Anfrage senden, um eine Antwort zu erhalten.

Besonders am WebSocket-Protokoll ist der Handschlag. Dieser verwendet bestimmte HTTP-Header, sodass ein Webserver der WebSocket unterstützt sowohl HTTP-Kommunikation als auch WebSocket-Kommunikation auf demselben Port ermöglichen kann. Das hat den Vorteil, dass die WebSocket-Kommunikation auch in einem restriktiven Netzwerk funktionieren kann, falls alle anderen Ports außer des Standard-HTTP-Ports blockiert sind.

2.2 WebRTC

Anders als WebSocket ermöglicht Web Real-Time Communication (WebRTC) [11] [12] die Kommunikation zwischen Browsern. WebRTC ist ein Framework mit einer Reihe von Kommunikationsprotokollen und Schnittstellen, das wie WebSocket auf Schnittstellenebene vom W3C und auf Protokollebene von der IETF standardisiert wird. Auch WebRTC wird von vielen Browsern unterstützt [13], sodass kein zusätzliches Plug-in nötig ist. Neben allgemeinem Datenverkehr zwischen Browsern sind auch komplexe Video- und Audioübertragungen mittels der WebRTC-Schnittstelle einfach möglich.

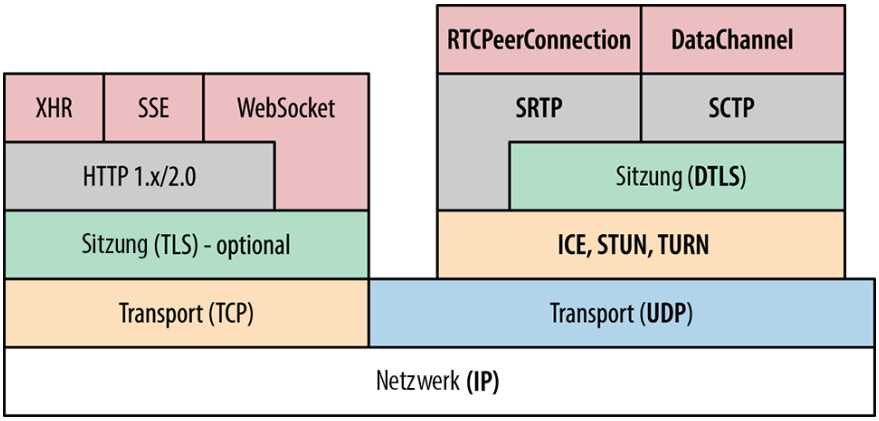

Abbildung 4: WebRTC-Protokollstapel. Basierend auf [14]

Im Rahmen dieser Arbeit kommen nur die Protokolle Interactive Connectivity Establishment (ICE) [15] und Session Traversal Utilities for NAT (STUN) [16] zum Einsatz, die für den Verbindungsaufbau zwischen Browsern benötigt werden. Traversal Using Relays around NAT (TURN) [17] ist auch ein Protokoll, welches beim Verbindungsaufbau zwischen Browsern hilft, jedoch mithilfe eines Relay-Servers, falls keine Direktverbindung möglich ist. Diese Funktionalität wird hier nicht benötigt.

Clients, bei denen keine Direktverbindung möglich ist, nehmen nicht am Peer-to-Peer-Netz teil und lösen die Inhalte normal über HTTP-Anfragen zum Webserver auf.

Als Basis für den WebRTC-Datenverkehr dient Datagram Transport Layer Security (DTLS) [18] über User Datagram Protocol (UDP) [19]. UDP ist ein minimales Netzwerkprotokoll für den Versand von Datagrammen in IP-Netzen, welches in vielen Umgebungen unterstützt wird. DTLS ist ein Verschlüsselungsprotokoll basierend auf Transport Layer Security (TLS) [20] und dient zur Verschlüsselung dieser Datagramme, um Manipulierungen und Ausspähung von Daten zu verhindern.

Auf diesen Protokollen kommen bei WebRTC nun Secure Real-time Transport Protocol (SRTP) [21] und Stream Control Transmission Protocol (SCTP) [22] zum Einsatz. SRTP dient hier hauptsächlich der sicheren Übertragung von Video- und Audiodatenströme; SCTP hingegen zur Übertragung von allgemeinen Daten. Die zu entwickelnde Middleware macht ausschließlich Gebrauch von SCTP und nicht von SRTP, daher wird in Kapitel 2.3 auf die Eigenschaften von SCTP fokussiert.

Die Schnittstelle der Datenübertragung von WebRTC auf SCTP nennt man auch den WebRTC-Datenkanal [23]. Die Nutzung des WebRTC-Datenkanals ist das Herzstück dieser Arbeit. Auf diesem werden Teilstücke von Dateien und Kontrollnachrichten zu anderen Peers gesendet.

2.2.1 Verbindungsaufbau

Der Verbindungsaufbau zwischen Browsern ist schwieriger als zu einem Server. Heutzutage befinden sich viele Browser hinter einer NAT-Middlebox. Eine NAT-Middlebox findet sich oft in Routern und wird heutzutage hauptsächlich eingesetzt, um der Knappheit an IPv4-Adressen entgegenzuwirken, indem es mehrere private IP-Adressen [24] unter einer öffentlichen IP-Adresse [24] abbilden kann. Die Ende-Ende-Beziehung ist nicht mehr eindeutig. Dies hat zur Folge, dass Hosts hinter einer NAT-Middlebox nicht ohne weiteres von außen ansprechbar sind. ICE und STUN versuchen diesem Problem entgegen zu wirken, sodass Hosts doch hinter einer NAT-Middlebox ansprechbar werden. In Kapitel 2.4 wird genauer auf NAT eingegangen.

Für den Verbindungsaufbau zwischen Browsern wird außerdem noch ein Signaling-Server benötigt. Ein Signaling-Server ist für den Austausch von Verbindungsdaten zwischen Browsern zuständig. Browser teilen ihre Datenstromeigenschaften mit dem Session Description Protocol (SDP) [25] dem Signaling-Server mit. SDP beschreibt Eigenschaften von Datenströmen, wie z.B. den zu verwendenden Videocodec. Diese Informationen dienen dazu, dass sich Browser auf Eigenschaften von Datenströme

einigen können. Der Signaling-Server sendet diese SDP-Informationen jeweils dem anderen Browser, die miteinander verbunden werden sollen. Die Kommunikation zwischen Browser und Signaling-Server erfolgt hier über die WebSocket-Schnittstelle.

Abbildung 5: Austausch SDP über Signaling-Server mit *Offer/Answer*-Verfahren [14]

Die SDP-Informationen werden im *Offer/Answer*-Verfahren [26] erstellt. Dazu erstellt ein Browser ein Angebot, welches von einem anderen Browser beantwortet wird.

Neben den SDP-Informationen werden noch *ICE candidates* benötigt. Dies sind Paare aus IP-Adresse und Port, um den jeweiligen Browser adressieren zu können. Die Paare werden durch ICE von den Netzwerkschnittstellen des Endgeräts und durch einen STUN-Server ermittelt (beschrieben in Kapitel 2.4.1).

Wenn all diese Informationen zwischen den Browsern durch den Signaling-Server ausgetauscht worden sind, kann eine Direktverbindung zwischen den Browsern hergestellt werden. Ist dies erfolgreich, sind beide Browser Peers im Peer-to-Peer-Netzwerk.

2.3 Datentransport mit SCTP

Stream Control Transmission Protocol (SCTP) [22] ist ein Transportprotokoll, welches in der Transportschicht des OSI-Modells [27] angesiedelt ist. Verwendet wird SCTP als Basis für die WebRTC-Datenkanäle, daher befindet es sich nicht in der Transportschicht, sondern in der Anwendungsschicht des OSI-Modells. Geschuldet ist dies der fehlenden Unterstützung des Protokolls in sämtlichen Umgebungen, wie Betriebssystemen und Routern.

WebRTC tunnelt SCTP über UDP und erweitert so den Datenstrom um wichtige Eigenschaften:

- Konfigurierbare Reihenfolgeerhaltung

- Konfigurierbare Zuverlässigkeit durch die vorgeschriebene Partial-Reliability-Erweiterung [28] durch WebRTC
- Flusskontrolle
- Staukontrolle
- Multistreaming

Normalerweise ist ein Vier-Wege-Handschlag für den Verbindungsaufbau durch SCTP nötig, wird aber im Rahmen von WebRTC auf einen Zwei-Wege-Handschlag verkürzt [29]. Der Handschlag ist ein Verfahren zum Verbindungsaufbau zwischen zwei Endpunkten. Mehrere Beispiele eines Handschlags werden in Abbildung 18 gezeigt.

2.4 Network Address Translation

Network Address Translation (NAT) [30] ist ein Ersetzungsverfahren von IP-Adressen, häufig in Verwendung mit Network Address Port Translation [31], ein Ersetzungsverfahren von TCP-/ und UDP-Ports. Ports dienen zur Unterscheidung von Verbindungen zu einem Endpunkt. Beide Verfahren zusammen werden gewöhnlich als NAT bezeichnet und werden heutzutage genutzt, um private Netzwerke mit nur einer öffentlichen IP-Adresse abzubilden. Dieses Verfahren wirkt so der Erschöpfung der IPv4-Adressen [32] entgegen, indem mehrere Endpunkte unter einer IP-Adresse adressierbar sind. Jedoch entsteht dadurch das Problem, dass Endpunkte hinter einer NAT-Middlebox nicht mehr ohne weiteres von außen ansprechbar sind. Die Ende-zu-Ende-Beziehung durch die IP-Adresse gilt nicht mehr. Eine NAT-Middlebox weiß nicht, wohin es die Pakete weiterleiten soll, wenn keine weiteren Informationen außer der öffentlichen IP-Adresse und einem Port bekannt sind. Hierzu kann in einer NAT-Middlebox eine Portweiterleitung eingerichtet werden, die Pakete zum gewählten Host auf bestimmten Ports weiterleitet. Die Einrichtung von Portweiterleitungen für den Verbindungsaufbau von außen steht in vielen privaten Netzen aber außer Frage. Daher wurden weitere Techniken entwickelt, um diese Limitierung der NAT-Verfahren zu umgehen. Dieses Problem tritt jedoch nur auf, wenn die Kommunikation von außerhalb des lokalen Netzwerks initiiert wird. Beim Initiieren der Kommunikation innerhalb des lokalen Netzwerks greift das NAT-Verfahren, das in Abbildung 6 dargestellt wird.

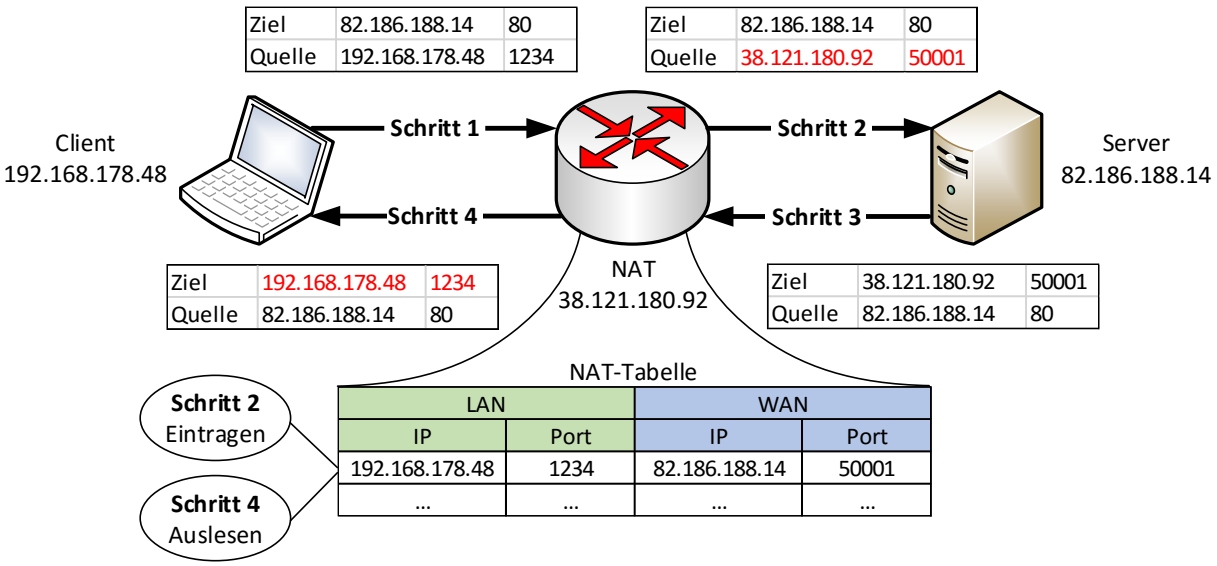

Abbildung 6: Funktionsweise NAT

In diesem Beispiel wird ein Datenpaket von einem Client hinter einer NAT-Middlebox zu einem Server und wieder zurück versendet. Im ersten Schritt sendet der Client ein Datenpaket an den Server mit dem Quellport 1234. Beim Durchlaufen des Pakets durch die NAT-Middlebox wird in der NAT-Tabelle ein Eintrag mit der Quelle des Datenpakets, der Ziel-IP und einen zufälligen freien Port angelegt – hier Port 50001. Außerdem wird die Quelle des Datenpakets mit der öffentlichen IP des NAT und dem ausgewählten Port ersetzt. Der Server empfängt nun dieses Paket und sendet in Schritt 3 eine Antwort. Ziel und Quelle sind also im neuen Datenpaket vertauscht. Trifft das neue Datenpaket nach Schritt 3 wieder in der NAT-Middlebox ein, wird in der NAT-Tabelle nachgeschaut, ob zu diesem Zielport ein Eintrag existiert. Da für Port 50001 ein Eintrag existiert, wird im 4. Schritt das Ziel des Pakets durch die LAN-Daten des Eintrags ersetzt und zum ursprünglichen Client gesendet, der die Anfrage gestellt hat.

Durch dieses Verfahren können nun mehrere Clients mit verschiedenen lokalen IP-Adressen im gleichen Netz durch die Ersetzung des zufälligen freien Ports und der öffentlichen IP-Adresse der NAT-Middlebox Antworten von außen erhalten.

Allerdings funktioniert nicht jede NAT-Middlebox auf dieselbe Weise. Es existieren Unterschiede bei der Wahl des gewählten Ports, zu sehen in Abbildung 6 Schritt 2 und bei der Weiterleitung zum Client in Schritt 4. Das durch RFC 5389 [16] ersetzte RFC 3489 [33] versuchte noch die Unterschiede in mehrere Klassen zu kategorisieren,

was jedoch aufgrund der Vielfalt an Unterschieden aufgegeben wurde. Um diese Unterschiede zu analysieren, wird das STUN-Protokoll [16] verwendet.

2.4.1 NAT-Traversal

Abbildung 6 zeigt die Kommunikation zwischen einem Client hinter einer NAT-Middlebox und einem Server ohne NAT- Middlebox. Möchte man aber zwischen zwei Clients kommunizieren, die sich beide hinter einer NAT- Middlebox befinden, dann benötigt man eine NAT-Traversal-Technik. ICE ist beispielsweise ein Protokoll zur NAT-Durchdringung, welches das STUN-Protokoll zur Analyse der Internetverbindung und des NAT-Typs nutzt. Verwendet wird beides in WebRTC. Es wird mindestens ein öffentlicher STUN-Server benötigt, der von beiden Clients ansprechbar ist. Es existieren noch andere Techniken wie UPnP [34] und PCP [35]. UPnP und PCP kommen ohne öffentlich erreichbare Server aus und können unter anderem Portweiterleitungen in einer NAT-Middlebox einrichten.

In WebRTC wird nur ein Teil des STUN-Protokolls genutzt, da eine vollständige Analyse der Internetverbindung nicht benötigt wird. Um nun eine Verbindung zwischen den zwei Clients aufzubauen, senden beide eine sogenannte „binding request“-Nachricht an einen STUN-Server. Dieser teilt dann den anfragenden Clients ihre öffentliche IP-Adresse und Port mit, mit der die Anfrage gestellt worden ist. Diese Informationen senden die Clients einem Signaling-Server, der auch von beiden Clients erreichbar ist. Der Signaling-Server teilt die Informationen jeweils dem anderen Client mit. Nun wird bei WebRTC die Hole-Punching-Technik [36] angewendet. Beide Clients versuchen eine Verbindung zu der mitgeteilten Adresse des anderen aufzubauen. In beiden NAT- Middlebox wird dadurch in der NAT-Tabelle ein Eintrag angelegt, der den Verbindungsaufbau zur Adresse des anderen protokolliert. Falls beide Clients über kein symmetrisches NAT [37] verfügen, welches andere Ports zu anderen Zielen verwendet, so würde der gleiche Port benutzt werden, wodurch die Nachrichten vom anderen Client auf demselben Port ankommen und die Kommunikation stattfinden kann. Die Verbindung zwischen den Clients ist nun aufgebaut und beide können sich Nachrichten senden, die auch durch die NAT-Middlebox hindurchkommen.

Der Ablauf des Hole-Punching-Prozesses wird in Abbildung 7, Abbildung 8 und Abbildung 9 noch mal verdeutlicht.

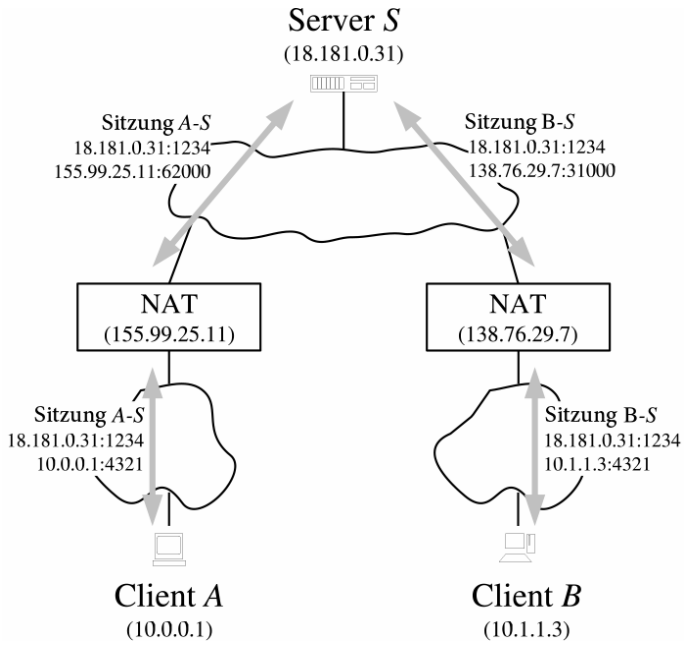

Abbildung 7: Vor dem Hole-Punching-Prozess. Basierend auf [36]

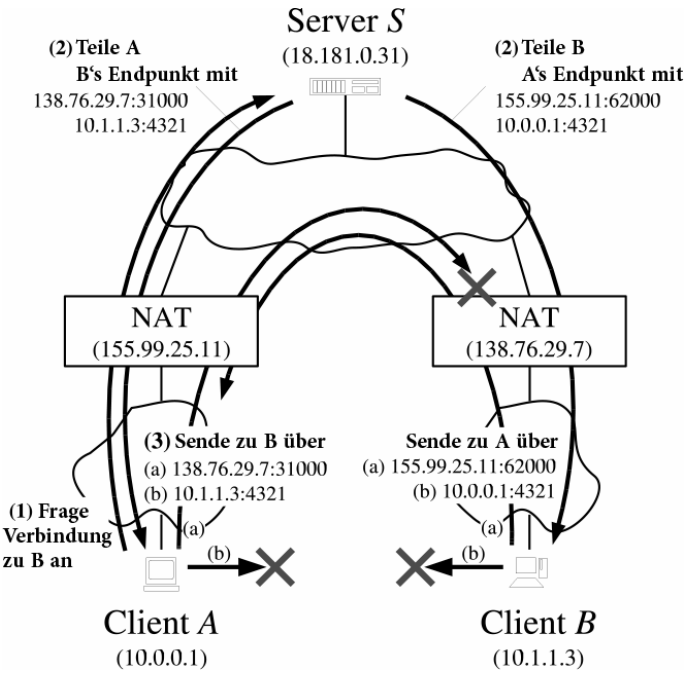

Abbildung 8: Der Hole-Punching-Prozess. Basierend auf [36]

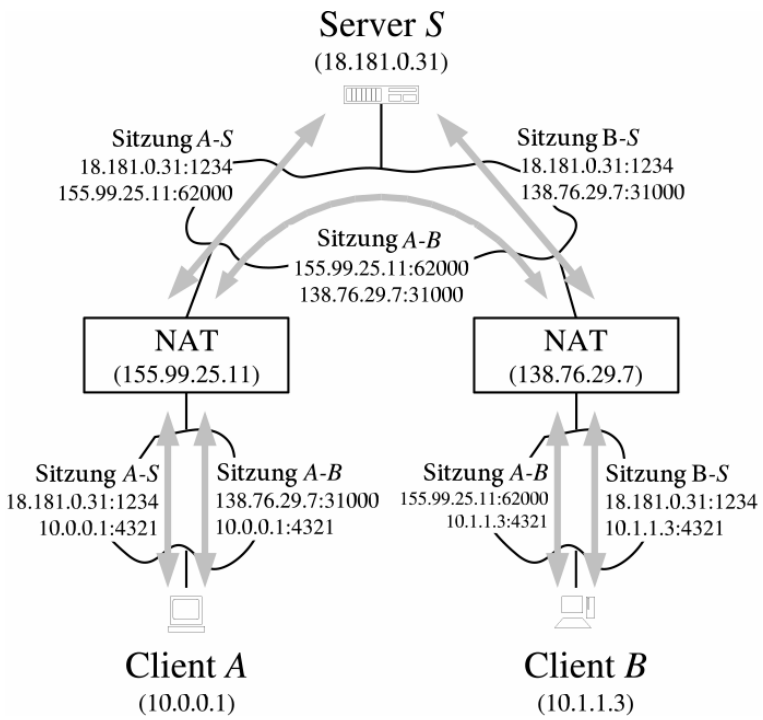

Abbildung 9: Nach dem Hole-Punching-Prozess. Basierend auf [36]

2.5 Service Worker im Browser

Ein Service Worker [38] ist ein Skript, welches im Hintergrund eines Browsers ausgeführt wird und von einer Webseite aus installiert wurde. Es kann Anfragen zu externen Inhalten abfangen und beantworten, die von dieser Webseite aus gestellt wurden. Der Geltungsbereich wird über den Ursprung [39] (bestehend aus Protokoll, Host und Port) und dem Pfad ermittelt, der bei der Installation gewählt wurde.

Dadurch kann man beispielsweise Anfragen umleiten, einen eigenen Cache anlegen und bei einem Internetausfall u. U. eine ältere Version der Anfrage aus dem Cache anzeigen.

Unterstützt und von Haus aus aktiviert sind Service Worker bei Firefox [40] ab Version 44 [41] (veröffentlicht am 26.01.2016 [42]) und in Chrome [43] seit Version 40 (veröffentlicht am 21.01.2015 [44]).

Service Worker sind für die Implementierung dieser Arbeit interessant, um Anfragen umzuleiten und diese externen Inhalte über andere Peers zu beziehen. Leider ist die WebRTC-Schnittstelle (noch) nicht im Service Worker verfügbar [45]. Ein Antrag zur

Bereitstellung wurde bereits eingereicht [46]. Ein weiterer Vorteil ist, dass ein Service Worker im Browser weiterläuft, obwohl die zugehörige Webseite geschlossen wurde. Dadurch kann man die aufgebauten Verbindungen zum Signaling-Server und zu anderen Peers halten. Ohne Service Worker müsste bei jedem Seitenaufruf einer Webseite jede Verbindung zu den Peers neu aufgebaut werden. Abhilfe kann hier eine *Single-page application* [47] schaffen, bei der beim Navigieren nicht die komplette Webseite neu aufgebaut werden würde. Außerdem wird die Möglichkeit im Service Worker Anfragen abzufangen benötigt, da es im normalen JavaScript-Kontext einer Webseite nicht möglich ist und andernfalls der HTML-Quelltext editiert werden müsste, damit der Browser nicht automatisch beginnt externe Inhalte aufzulösen.

2.6 Alternative Lösungen

PeerCDN ist auch eine Lösung zur peerunterstützten Verteilung von Inhalten über WebRTC und wahrscheinlich die erste Bekanntmachung in diesem Umfeld. Vorgestellt wurde sie am 28. März 2013 auf Hacker News [48]. Bis zu einer Veröffentlichung dieser Lösung ist es aber nie gekommen. Bereits am 17. Dezember 2013 wurde PeerCDN von Yahoo erworben [49]. Die Webseite http://peercdn.com ist mittlerweile nicht mehr erreichbar. Ein Screenshot der Webseite vom 30. Juni 2015 konnte noch aufgefunden werden, auf der die wichtigsten Punkte von PeerCDN zu sehen sind [50]. Dies sind:

1. Bandbreitenkosten werden um bis zu 90% reduziert
2. Besteht nur aus JavaScript
3. Kommt bei Bursts von Datenverkehr mit Leichtigkeit klar
4. Breite Browserunterstützung
5. Funktioniert mit gewöhnlichen CDNs [51]
6. Integrierte Sicherheit
7. Zuverlässigkeit
8. Einfach zu installieren

Peer5 [52] ist eine weitere Lösung in diesem Umfeld, die sich auf Videostreams spezialisiert hat. Die Benutzung dieser Lösung ist allerdings nur bis zu einem bestimmten Datenverbrauch kostenlos. Peer5 besitzt auch einen peerunterstützten Downloader für den expliziten Download größerer Dateien, der nicht mehr angeboten zu sein scheint, da kein Link von der Hauptseite zum Downloader führt und die Seite nur über den bekannten Link aufrufbar ist. Über Preise wird nicht informiert und der Link zur Registration ist nicht funktional. Eine Webseite, die

Gebrauch des Downloaders macht wurde aber gefunden. Auf https://basketbuild.com/ werden Downloads zu Android-Betriebssystemen und Apps angeboten, die man wahlweise über den Peer5 Downloader herunterladen kann. Laut Blogartikel wurde der Webseitenbetreiber von Peer5 kontaktiert [53].

Ein weiterer Ansatz zur Verteilung von Inhalten ist BOPlish [54]. In dieser Lösung können sogar Anwender Inhalte veröffentlichen. Dazu ist eine neue Identifikation von Inhalten erforderlich. Peers werden mithilfe einer verteilten Hashtabelle [55] aufgefunden und ein zentraler Server wird nicht benötigt. Anders als in BOPlish wird in dieser Arbeit ein zentraler Server eingesetzt, der die Verwaltung von Peers übernimmt. Dadurch lassen sich kürzere Auflösungszeiten von Inhalten erreichen, die in dieser Arbeit angestrebt werden. Durch die peerunterstützte Auflösung von Inhalten kommen nämlich viele Paketumlaufzeiten im Gegensatz zur normalen HTTP-Auflösung hinzu.

Bis heute konnte keine Lösung gefunden werden, die die Idee von PeerCDN aufgreift. Es scheint sich speziell nur auf Videostreaming konzentriert zu werden. Weitere Dienste in diesem Umfeld sind unter anderem Streamroot [56] und Swarmify [57].

3 Entwicklung der Middleware

Wie in Kapitel 1.2 beschrieben, ist die Entwicklung einer Middleware zur Verlagerung der typischen Datenverkehrsmuster im Web das Ziel. Dazu ist eine Client-Server-Anwendung nötig. In Kapitel 3.1 wird eine Architekturübersicht gegeben, die die verschiedenen Akteure zeigt. Die generelle Funktionsweise wird in Kapitel 3.2 beschrieben. In den darauffolgenden Kapiteln wird auf Besonderheiten eingegangen, die bei der Entwicklung der Middleware beobachtet wurden.

Orientiert wurde sich bei einigen Entscheidungen am etablierten BitTorrent-Protokoll [58]. BitTorrent ist ein Filesharing-Protokoll, bei dem Daten Peer-to-Peer verteilt werden und bekannt für die schnelle Verteilung von großen Datenmengen.

3.1 Architekturübersicht

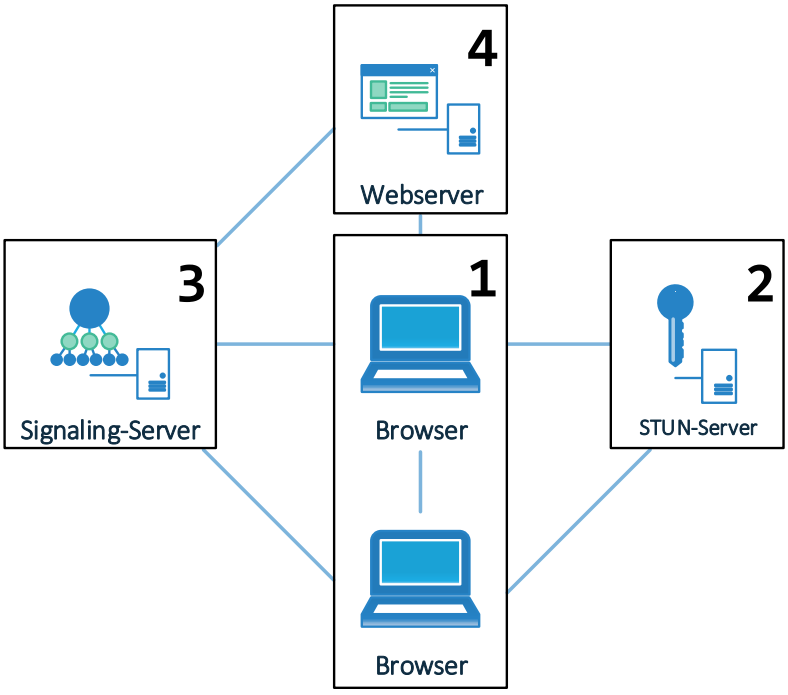

Abbildung 10: Architekturübersicht der Middleware

Grundsätzlich ist die Architektur in vier Akteure zu unterteilen:

1. Der Browser, der die Middleware beanspruchen möchten
2. Der STUN-Server, der Verbindungsinformationen bereitstellt
3. Ein Signaling-Server der die Verbindungsdaten anderen Browsern mitteilt
4. Der eigentliche Webserver, der die Beanspruchung der Middleware einleitet

Die Browseranwendung und der Signaling-Server sind in der Programmiersprache JavaScript [59] programmiert. Auf dem Server kommt Node.js [60] zum Einsatz. Eine Plattform für Netzwerkanwendungen, die in der JavaScript-Laufzeitumgebung V8 [61] ausgeführt wird, die unter anderem in Chrome Anwendung findet. Node.js benutzt eine ereignisgesteuerte Architektur [62], die sich öfters von anderen Serveranwendungen unterscheidet, die pro Verbindung einen eigenen Thread reservieren.

Der verwendete STUN-Server in der Middleware ist einer von vielen öffentlichen STUN-Servern, die ohne Einschränkungen genutzt werden können, um die benötigten Verbindungsinformationen der Browser zu beschaffen. Konkret kommt ein STUN-Server von Google mit der Adresse stun.l.google.com:19302 zum Einsatz. Eine eigene Implementierung wäre durchaus denkbar, ist im Rahmen dieser Arbeit nicht notwendig.

Die Browser, die die Middleware beanspruchen möchten, müssen nicht unbedingt dieselben sein. Optimiert wurde die Middleware jedoch für Firefox, aber eine Kompatibilität mit anderen Browsern ist angestrebt.

Als Webserver kann ein Beliebiger verwendet werden. Es muss jedoch im HTML-Quelltext eine Anweisung zur Ausführung der Middleware hinterlegt werden, falls der Webseitenbetreiber eine peerunterstützte Dateiverteilung wünscht. Dazu ist der HTML-Quelltext entsprechend zu erweitern:

```
<script src="pfad/zur/middleware/peerassist.js"></script>
```

Abbildung 11: Einbindung der Middleware mit HTML-Element

Zusätzlich müssen *src*-Attribute von HTML-Elementen in „data-src“-Attribute umbenannt werden. Dadurch wird dieser Inhalt peerunterstützt heruntergeladen. Diese Anweisung kann in Zukunft durch eine dynamische Anweisung über JavaScript erweitert werden. Das automatische Auflösen von Dateien über Service Worker ist

wie in Kapitel 2.5 beschrieben auch möglich, wird aber aufgrund der noch fehlenden WebRTC-Schnittstelle nicht implementiert.

```
<img src="img.jpg"> <!-- Normale Auflösung -->
<img data-src="img.jpg"> <!-- Auflösung über Middleware -->
```

Abbildung 12: Einbindung der externen Inhalte mit und ohne "data-src"-Attribut am Beispiel eines *img*-HTML-Elements

3.2 Funktionsweise

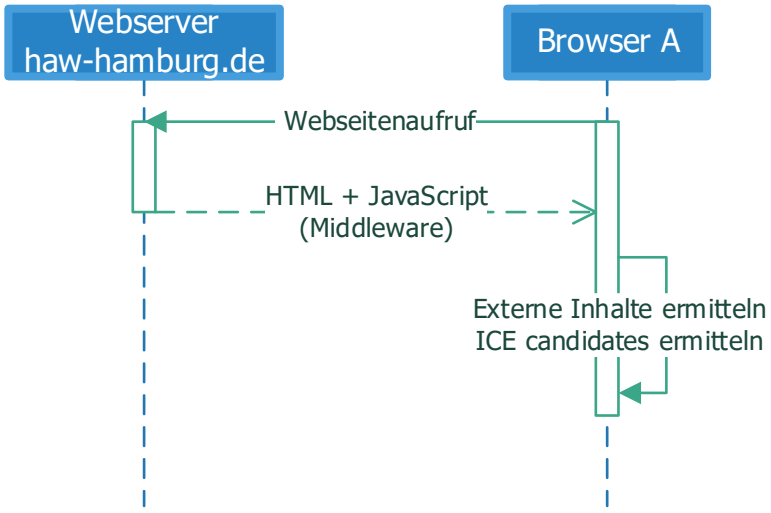

Abbildung 13: Start der Middleware im Browser

Als erstes agiert der Browser, der eine Webseite aufruft und das mitgelieferte clientseitige Middleware-Skript ausführt. Ab diesem Zeitpunkt sammelt das Skript alle HTML-Elemente ein, die ein „data-src“-Attribut besitzen und über die Middleware aufgelöst werden sollen. Zeitgleich wird ein Pool von WebRTC-Instanzen angelegt, wodurch *ICE candidates* (Kapitel 2.2.1) ermittelt werden. Ein Pool hat den Vorteil, dass der Signaling-Server diese Informationen vorrätig hat und dadurch ohne eine extra Paketumlaufzeit ausgewählten Peers die Informationen bereitstellen kann.

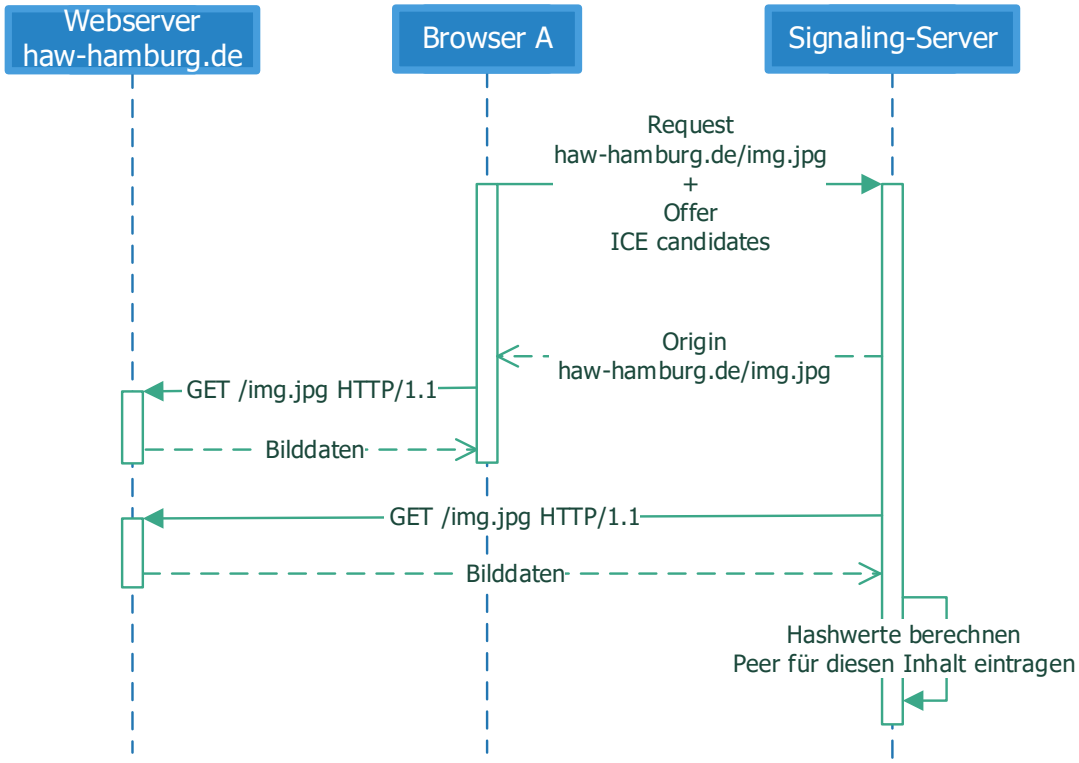

Abbildung 14: Ablauf der Middleware bei keinen Peers

Nachdem die Informationen ermittelt wurden, wird eine Verbindung zum Signaling-Server über WebSocket aufgebaut. Die über die Middleware aufzulösenden Inhalte (hier das Bild /img.jpg auf haw-hamburg.de) werden dem Signaling-Server mitgeteilt, samt *ICE candidates* und Offers, die durch die Instanzen von WebRTC generiert wurden.

Im Szenario abgebildet in Abbildung 14, kennt der Signaling-Server weder Peers die den angeforderten Inhalt besitzen, noch die benötigten Hashwerte zur Integritätsprüfung des Inhalts. Daher lädt der Signaling-Server selbst den Inhalt vom Webserver herunter (hier haw-hamburg.de) und sendet dem anfragenden Browser eine Nachricht, dass der Inhalt normal vom Webserver heruntergeladen werden soll. Daraufhin berechnet der Signaling-Server die Hashwerte und trägt den anfragenden Browser in die Liste der Browser ein, die diesen Inhalt vorrätig haben.

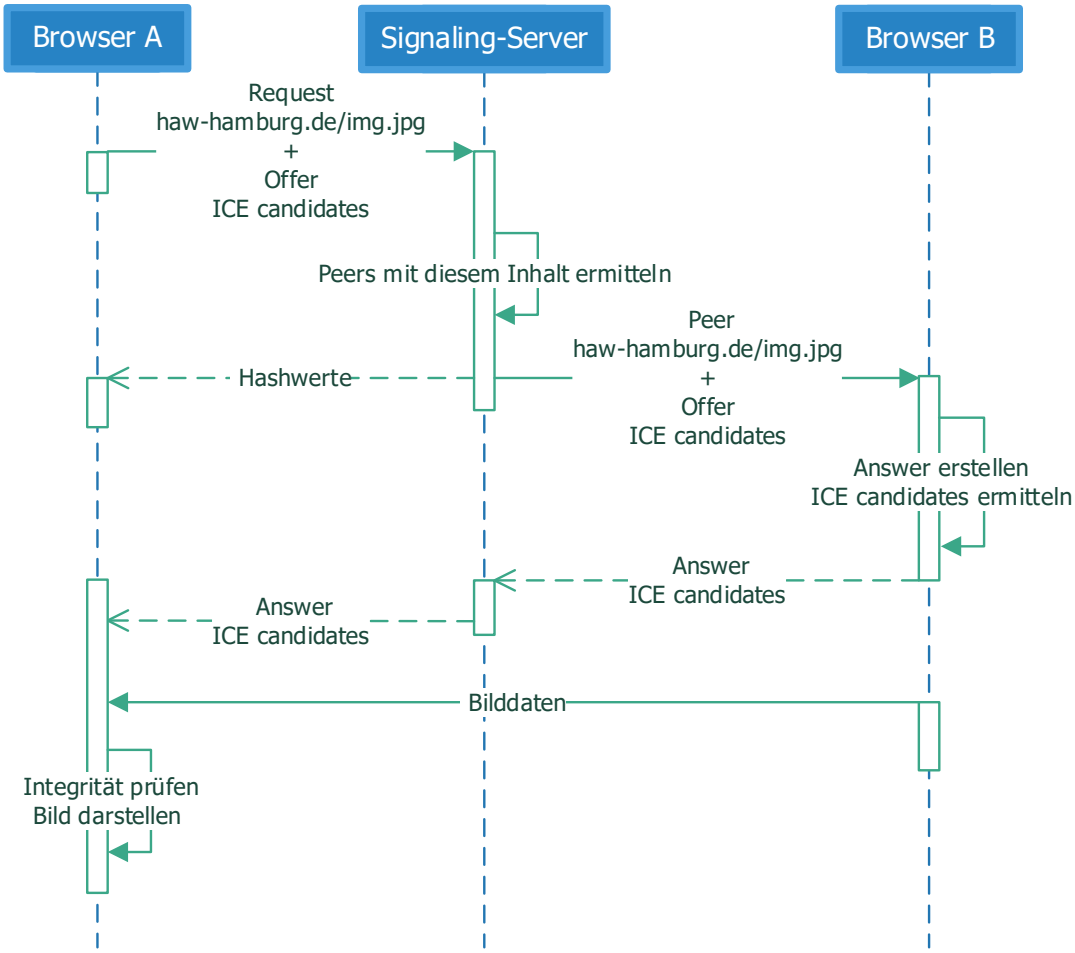

Abbildung 15: Sequenzdiagram der Funktionsweise mit Peers und Hashwerten

In Abbildung 15 ist nun der eigentliche Ablauf zu sehen, bei dem Inhalte peerunterstützt heruntergeladen werden. Nachdem Browser A eine Anfrage zum Signaling-Server gesendet hat und dieser andere Browser als auch Hashwerte des angeforderten Inhalts kennt, werden bestimmte Browser ausgewählt (hier Browser B), benachrichtigt und mit Informationen versorgt, um eine Verbindung der beiden Browser zu ermöglichen. Browser A werden die benötigten Hashwerte zur Integritätsprüfung vom Signaling-Server mitgeteilt. Browser B beantwortet die SDP-Offer und sendet die Antwort samt *ICE candidates* zurück zum Signaling-Server. Dieser leitet die Informationen weiter an den anfragenden Browser A. Beide Browser versuchen mit diesen Informationen nun eine Verbindung aufzubauen. Falls die

Verbindung erfolgreich war, sendet Browser B automatisch die Daten(-stücke), die der Signaling-Server ausgewählt hat an Browser A. Dieser prüft während der Datenübertragung die Integrität der Daten mit den vom Signaling-Server mitgeteilten Hashwerten. Stimmen die Hashwerte alle überein, so kann Browser A das Bild darstellen und ist vor falschen Bildern geschützt.

Die Auflösung von anderen Inhalten wie Videos und Skripte sind mit der Middleware auch möglich. Inhalte können auch von mehreren Peers heruntergeladen werden, sodass eine höhere Datenübertragungsrate erreicht werden kann.

Ob Peers noch für die peerunterstützte Verteilung zur Verfügung stehen und die Webseite noch nicht verlassen haben, wird aufgrund der aktiven WebSocket-Verbindung zum Signaling-Server durch die verwendete WebSocket-Bibliothek ermittelt.

Der Signaling-Server entscheidet, welche Peers welche Datenelemente senden. So wird eine Paketumlaufzeit gespart, in der die Peers sonst untereinander klären müssten, welche Datenelemente zu versenden sind. Diese zentrale Vermittlung hat auch den Vorteil, dass der Signaling-Server Informationen wie die verfügbare Bandbreite und Latenzzeit der Peers im Vorfeld ermitteln kann und so Peers mit viel Bandbreite und wenig Latenzzeit mehr *Pieces* an andere Peers versenden lassen kann. Die Middleware kann in Zukunft um diese Funktionalität erweitert werden.

3.3 Verwaltung der Inhalte

3.3.1 Identifizierung

```
http://www.haw-hamburg.de/bild.jpg
```

Abbildung 16: Beispiel einer URL

Zur exakten Identifizierung von Inhalten wird die URL [63] verwendet. Sie wird vom Signaling-Server zeichengenau überprüft, da jegliche Abweichung zu einem anderen Inhalt führen kann und so die peerunterstützte Dateiverteilung fehlschlägt. Dazu zählt sowohl die Groß- und Kleinschreibung als auch das verwendete Protokoll.

3.3.2 Aufteilung der Daten

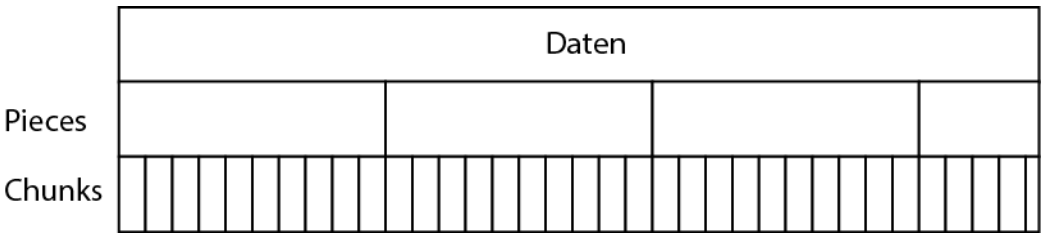

Abbildung 17: Aufteilung der Daten in kleinere Teilstücke

Daten werden in kleinere Teilstücke aufgeteilt, um beim Fehlerfall kleinere Stücke ersetzen zu können. BitTorrent nennt diese *Pieces* und sind standardmäßig 256 KiB groß (2^{18} Byte). Bei der Implementation der Middleware wird die gleiche Größe verwendet. Außerdem wird eine noch kleinere Einheit eingeführt, die *Chunks*. Sie besitzen eine feste Größe und sind die Dateneinheiten, die von Peer zu Peer über den WebRTC-Datenkanal gesendet werden. BitTorrent verfolgt diesen Ansatz nicht und sendet Datenpakete dynamischer Größe. Die jeweils letzten *Chunks* und *Pieces* einer Datei besitzen u.U. eine kleinere Größe, falls die Dateigröße kein vielfaches der *Pieces* ist, so wie in Abbildung 17 dargestellt.

Entschieden wurde sich für die *Chunks*, um die Komplexität der Middleware geringer zu halten. Für spätere Optimierungen kann der dynamische Ansatz von BitTorrent verfolgt werden. Die *Chunks* eignen sich aber zum besseren Testen des Einflusses von bestimmten Nachrichtenlängen in WebRTC-Datenkanälen. Denn diese haben verschiedene Auswirkungen auf die Browser. Bei Firefox wurde ein zufälliges Timeout von ziemlich genau einer Sekunde festgestellt, wenn *Chunks* mit einer Größe von mehr als 8 KiB (2^{13} Byte) über den WebRTC-Datenkanal versendet wurden, obwohl sich dadurch die beanspruchte Bandbreite nicht wirklich vergrößert hat. Der Ursprung dieses Verhaltens konnte im Umfang dieser Arbeit nicht analysiert werden.

3.3.3 Integrität

Die Integrität der externen Inhalte wird über eine Liste von Hashwerten sichergestellt. Zu jedem *Piece* wird ein Hashwert über den Inhalt im jeweiligen *Piece* generiert. Wie bei BitTorrent wurde sich für SHA-1 [64] als Hashfunktion entschieden. Zwar existieren für SHA-1 diverse Kollisionsangriffe, jedoch sind sie für diesen Anwendungsfall nicht relevant. Solange keine Preimage-Angriffe [65] existieren, ist der Einsatz in diesem Umfeld gerechtfertigt.

Zur Generierung der Hashwerte wird die Web Cryptography API [66] genutzt, die von vielen Browsern unterstützt wird [67]. Anders als Firefox verweigert Chrome die Verwendung dieser Schnittstelle, falls die Webseite über das HTTP-Protokoll aufgerufen wurde. Diese Restriktion ist gerechtfertigt, da dem Browser bei einem Man-in-the-Middle-Angriff [68] falsche Hashwerte mitgeteilt werden können und es so im schlimmsten Fall dazu kommen könnte, dass fremder JavaScript-Code auf der Webseite ausgeführt wird. Jedoch ist das Einschleusen von JavaScript-Code über HTTP wesentlich einfacher als das Einschleusen von falschen Hashwerten in die zu entwickelnde Middleware. Daher wird der Grund dieser Restriktion nicht geteilt und die Entscheidung der Firefox-Entwickler unterstützt.

Eine JavaScript-Implementation von SHA-1 könnte daher verwendet werden, falls die Middleware auf HTTP-Webseiten Anwendung findet.

Subresource Integrity [69] (SRI) ist eine weitere Möglichkeit für Browser die benötigten Hashwerte zu berechnen. SRI ist ein Sicherheitsfeature, bei dem man in *script*- und *link*-HTML-Elementen einen Hashwert im *integrity*-Attribut hinterlegen kann. Wenn der Browser diesen Inhalt auflöst, berechnet er selber einen Hashwert und vergleicht ihn mit dem Attribut und verweigert die Ausführung des Inhalts, falls die Hashwerte nicht übereinstimmen. Dieses Feature wird seit einiger Zeit von mehreren Browsern unterstützt [70]. Für kleinere Inhalte mit nur einem Hashwert wäre dieser Ansatz denkbar, wird aber aufgrund der mangelnden Flexibilität von SRI nicht in der Middleware eingesetzt.

Ermittelt werden die Hashwerte vom Signaling-Server. Dieser lädt den angefragten Inhalt vom Webserver herunter und berechnet die Hashwerte der *Pieces* und stellt sie für spätere Anfragen den Browsern zur Verfügung.

3.4 12-Wege-Handschlag

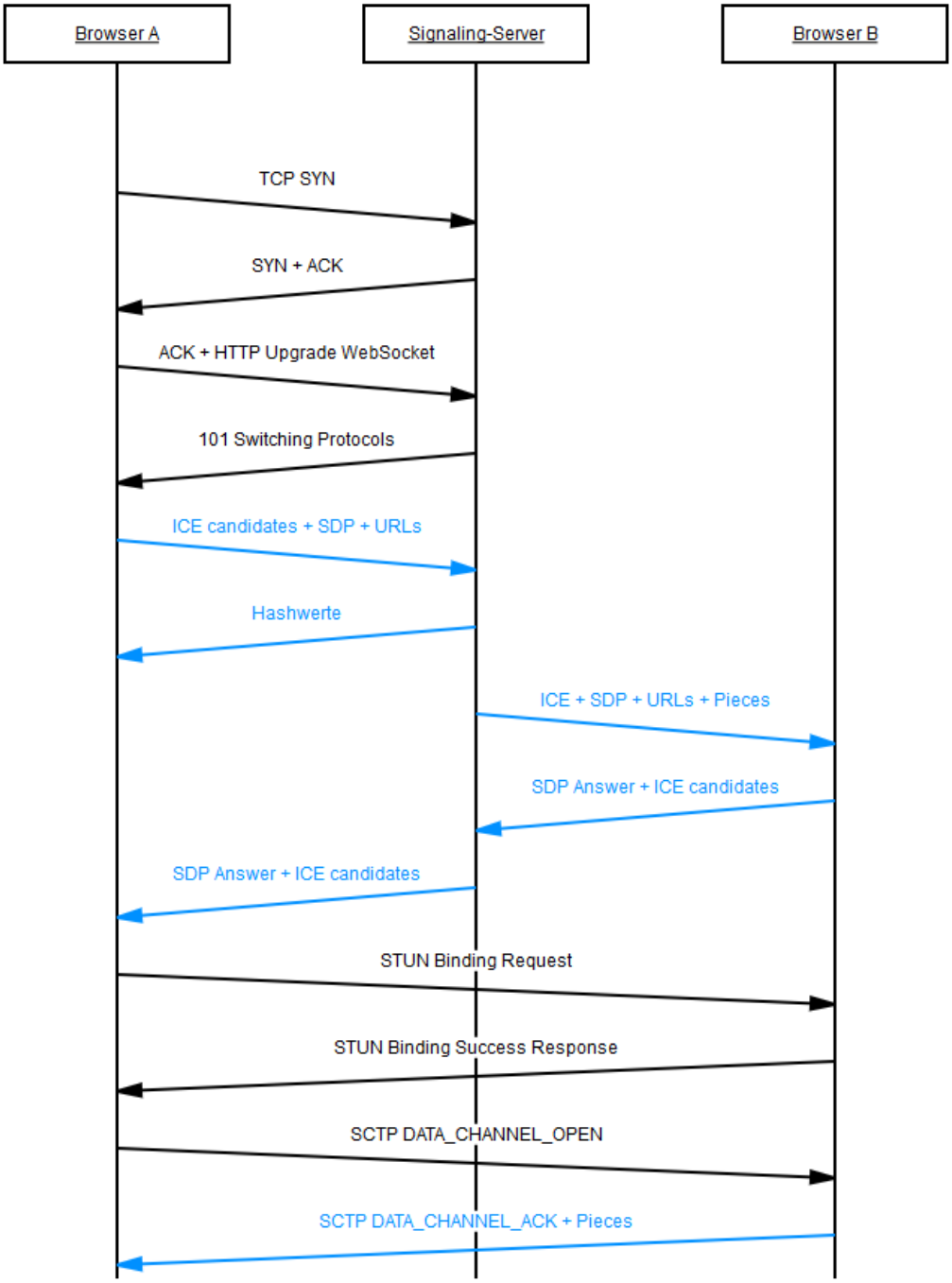

Abbildung 18: Ablauf des Verbindungsaufbaus zweier Peers

In Abbildung 18 sind alle Pakete dargestellt, die versendet werden müssen, damit eine Kommunikation im Rahmen der Middleware zwischen zwei Peers stattfinden kann. Die blauen Pfeile zeigen anwendungsspezifische Datenübertragungen an, die nicht zum Protokolloverhead dazu gehören. Zunächst wird ein Drei-Wege-Handschlag zum Signaling-Server für den Verbindungsaufbau von TCP durchgeführt. Nach dem Handschlag sendet Browser A eine HTTP-Nachricht inklusive bestimmter HTTP-Header für den Aufbau einer WebSocket-Verbindung. Der Server bestätigt dies durch eine Antwort mit dem HTTP-Statuscode 101. Das Austauschen der Informationen zwischen den beiden Peers schlägt mit vier Datenübertragungen zu Buche. Darauf folgt die NAT-Durchdringung (Kapitel 2.4.1) mit mind. zwei Datenübertragungen und zum Schluss der Zwei-Wege-Handschlag zum Verbindungsaufbau der WebRTC-Datenkanäle (Kapitel 2.3).

3.5 Herausforderungen

Herausforderungen für die Middleware können Inhalte stellen, die einen anderen Ursprung [39] besitzen als die aufgerufene Webseite. Aufgrund von Sicherheitsbestimmungen (Cross-Origin Resource Sharing (CORS) [71]) lässt der Browser die Weiterverarbeitung einer HTTP-Anfrage nicht zu, wenn der Ursprung nicht mit dem Ziel der HTTP-Anfrage übereinstimmt. Beispielsweise eine HTTP-Anfrage von der Webseite http://foo zu http://bar. Dieses Problem lässt sich nur umgehen, wenn der Webserver zu http://bar bestimmte HTTP-Header mitsendet, die die Weiterverarbeitung erlauben. Daher müssten die Webserver mit diesen HTTP-Kopfdaten ausgestattet sein, damit die peerunterstützte Verteilung funktioniert.

Alternativ kann der Signaling-Server diese Inhalte bereitstellen und fungiert dadurch als Proxy [72]. Durch diese Lösung benötigt der Signaling-Server allerdings deutlich mehr Bandbreite, da dann nicht mehr ausschließlich Kontrollnachrichten versendet werden, sondern auch Dateninhalte.

Ein weiteres Problem sind mangelnde HTTP-Caching-Header der Inhalte. Sind diese nicht vorhanden, könnte der Signaling-Server für die Verbreitung alter Daten sorgen, da dieser nicht weiß, wann sich Inhalte geändert haben. Verfügen die Inhalte beispielsweise über einen HTTP-Expires-Header, der die maximale Gültigkeit des Inhalts angibt, kann der Signaling-Server ohne Bedenken diese bis zum Ablauf der Gültigkeit verbreiten. Zukünftig müsste eine Strategie entwickelt werden, die diese „dynamischen“ Inhalte ohne Caching-Header mit Kompromissen verwaltet. Eine entsprechende Strategie ist nicht Teil dieser Arbeit.

Die Feststellung der Netzwerkkonnektivität stellt auch noch ein Problem dar. Zwar existiert eine Spezifikation der Network Information API [73], findet aber bisher kaum Unterstützung in Browsern. Beim Testen der Schnittstelle mit dem Standardbrowser in Android 5.1.1 konnten keine verwertbaren Informationen der Netzwerkkonnektivität ermittelt werden. So sollten Mobilgeräte bis zur Lösung dieses Problems nicht an der peerunterstützten Verteilung teilnehmen. Zur Feststellung des Geräts kann die *navigator.userAgent*-Eigenschaft [74] herangezogen werden. Sie verrät u.U. den verwendeten Browser und eine Identifikation des Geräts. Laut [74] wird vor der Benutzung gewarnt, da die Funktion in Zukunft nicht mehr Unterstützt wird. Diese Information konnte aber nicht weiter verifiziert werden.

Leider ändern sich auch die Adressen der aufgelösten Inhalte über die Middleware. Zur Darstellung der Daten benötigt man eine Browserinterne URL, die durch die *URL.createObjectURL*-Funktion [75] erstellt wird. Auf http://haw-hamburg.de angewendet, könnte sie wie in Abbildung 19 lauten.

```
blob:http://haw-hamburg.de/76325657-8d1d-4f61-ae5b-d87c21721357
```

Abbildung 19: Beispielhafte URL durch *URL.createObjectURL*

Wenn nun beispielsweise ein Bild mit der Adresse http://haw-hamburg.de/img.jpg über die Middleware aufgelöst wird und man die URL des Bildes kopiert, so erhält man eine URL in Form von Abbildung 19. Da dies eine Browserinterne URL ist, können andere die Information nicht verwerten.

Ein weiteres Problem ist das lokale Caching. Mit Bekanntgabe des Endes der File API [76] [77] ist ein Grundpfeiler weggefallen, um größere Dateien im Browser zu speichern. Zwar existieren noch andere Möglichkeiten wie über *localStorage* [78], jedoch besteht beispielsweise im Firefox 47 ein Limit von 5 MiB. [1] Abhilfe kann die Cache-Schnittstelle schaffen, die im Rahmen der Spezifikation der Service Worker eingeführt wurde [38]. Obwohl sie in der Service Worker Spezifikation spezifiziert ist, kann sie auch in anderen Browserkontexten verwendet werden. Ohne eine Caching-Lösung müssten alle Inhalte bei einem erneuten Webseitenaufruf erneut geladen werden. Wie in Kapitel 2.5 beschrieben, verhindern *Single-page applications* das komplette Neuladen der Webseiten beim Navigieren.

[1] Zu entnehmen aus der Einstellung *dom.storage.default_quota*.

3.6 Browserunterschiede

Bei der Entwicklung sind einige Unterschiede zwischen den Browsern Firefox und Chrome aufgefallen. Zum einen ist in Chrome die WebRTC-Schnittstelle über *webkitRTCPeerConnection* [79] erreichbar und im Firefox unter *RTCPeerConnection* [79]. Diese Unterschiede werden durch die Bibliothek *WebRTC adapter*, die in Kapitel 3.9.1 beschrieben wird, vereinheitlicht.

Außerdem existiert ein Unterschied beim Senden und Empfangen von Nachrichten auf dem WebRTC-Datenkanal. Chrome kann den Datentyp Blob weder versenden noch empfangen. Daher muss man ggf. den Datentyp Blob in einen ArrayBuffer konvertieren, sodass Chrome die Daten verarbeiten kann. Firefox unterstützt beide Datentypen.

```
function ArrayBuffer2Blob(ab) {
    return new Blob([ab])
}
function Blob2ArrayBuffer(blob) {
    return new Promise(res => {
        let fr = new FileReader()
        fr.addEventListener('loadend', () => res(fr.result))
        fr.readAsArrayBuffer(blob)
    })
}
```

Abbildung 20: Konvertierung von ArrayBuffer in Blob und vice versa

Ein Bug wurde während der Entwicklung in Firefox Version 44 festgestellt. Es ließen sich pro Tab nur 79 WebRTC-Instanzen erstellen, bis keine *ICE candidates* mehr für den Verbindungsaufbau zurückgegeben worden sind. Folgendes Skript wurde dafür verwendet:

```
for (var i = 0; i < 100; i++) {
    setTimeout(function(i) {
        var pc = new RTCPeerConnection({
            iceServers: [{
                urls: 'stun:stun.l.google.com:19302'
            }]
        });
        pc.onicecandidate = function(v) {
            if (v.candidate) console.log(i)
        };
        pc.createDataChannel('test', {});
        pc.createOffer().then(function(v) {
            pc.setLocalDescription(v)
        })
    }.bind(null, i), i * 300)
}
```

Abbildung 21: JavaScript-Code zum Testen der maximalen WebRTC-Instanzen in einem Tab

3.7 Sicherheit

Um die Integrität zu gewährleisten, werden Hashwerte benötigt. Leider liefern die meisten HTTP-Server keine Hashwerte des jeweiligen angefragten Inhalts in den HTTP-Kopfdaten mit. Um dieses Problem zu lösen, lädt der Signaling-Server die angefragten Inhalte vom HTTP-Server herunter und berechnet die Hashwerte, die dann den anfragenden Peers zur Verfügung gestellt werden. Für den Signaling-Server könnte dies ein Problem darstellen, da jeder den Signaling-Server dazu veranlassen kann, willkürliche HTTP-Anfragen zu Inhalten zu stellen. So könnte ein Angreifer sämtliche Bandbreite vom Signaling-Server beanspruchen, indem er ihn stetig dazu veranlasst, große Inhalte herunterzuladen. Ein entsprechendes Konzept ist aber nicht Teil dieser Ausarbeitung.

Des Weiteren könnte aus Anwendersicht ein Anonymitätsproblem entstehen. Wie in Kapitel 2.4 beschrieben, werden häufig mehrere Hosts durch NAT hinter einer einzigen IP-Adresse abgebildet. Surfen nun mehrere Endpunkte im gleichen LAN mit der gleichen öffentlichen IP-Adresse auf derselben Webseite, so würde ein schlauer Signaling-Server diese als Kandidaten für das Peer-to-Peer-Netz vorschlagen, da sie geographisch gesehen sehr günstig beieinanderliegen. Die daraus entstehende LAN-Verbindung grenzt die Endpunkte ein. Durch weitere Analysen der angefragten Inhalte kann dadurch unter Umständen herausgefunden werden, welche Unterseiten

einer Webseite aufgerufen werden, falls die angefragten Inhalte der Unterseite bekannt und eindeutig sind.

3.8 Programmierschnittstellen

Zwei Netzwerkprotokolle wurden im Rahmen dieser Arbeit entworfen:

- Das Signaling-Protokoll zwischen Client und Server
- Das Peer-Protokoll zwischen Peers

Diese werden nun vorgestellt.

3.8.1 Signaling-Protokoll

Das Signaling-Protokoll verläuft zwischen einem Client und einem Signaling-Server. Die Kommunikation basiert auf WebSocket-Nachrichten. Diese Nachrichten bestehen aus dem Datenformat JSON [80]. Entschieden wurde sich für JSON aufgrund der Flexibilität, der einfachen Handhabung in vielen Programmiersprachen und der allgemeinen Bekanntheit. In Zukunft kann auf kompaktere Datenformate wie MessagePack [81] zurückgegriffen werden oder ein eigenes Datenformat entwickelt werden, um die Nachrichtengröße geringer zu halten und so mehr Ressourcen im Netzwerk zu schaffen.

Jede Nachricht besteht an oberster Stelle aus einem JSON-Objekt und beinhaltet immer die Zeichenkette „type“ als Schlüssel und als Wert eine Zeichenkette, die den Typ einer Nachricht definiert. Enkodiert sind die Zeichenketten in UTF-8 [82].

Alle definierten Nachrichtentypen zwischen Signaling-Server und Browser sind aus dem Anhang in A.2 zu entnehmen. Eine Beispielnachricht ist im Folgenden zu sehen:

Typ	**request**
Beschreibung	Der Browser erfragt, wie die URLs in **requests** aufzulösen sind. Ob über Peers oder vom Webserver selbst. Optional werden Offers mitgesendet, um eine zügigere Verbindung zu Peers zu ermöglichen.

Zusätzliche Parameter	**requests**: Array[String] Die URLs, die der Browser auflösen möchte. **offers**: Array[Object] Vom Browser mitgesendete Offers samt *ICE candidates*.

Abbildung 22: Beschreibung der *request*-Nachricht

```
{
    "type": "request",
    "offers": [{
        "sdp": {
            "type": "offer",
            "sdp": "v=0\r\no=mozilla [...] \r\n"
        },
        "candidates": [{
            "candidate": "candidate:0 1 UDP 2122187007 192.168.178.48
53141 typ host",
            "sdpMid": "sdparta_0",
            "sdpMLineIndex": 0
        }]
    }],
    "requests": ["http://haw-hamburg.de/img1.jpg", "http://haw-
hamburg.de/img2.png"]
}
```

Abbildung 23: Beispiel einer gekürzten *request*-Nachricht im JSON-Format

3.8.2 Peer-Protokoll

Das Peer-Protokoll verläuft zwischen den Peers. Es basiert auf jeweils zwei WebRTC-Datenkanälen. Einen Kontrollkanal und einen Datenkanal. Auf dem Kontrollkanal werden JSON-Objekte versendet, die den Datenstrom des Datenkanals beschreiben. Der Datenkanal transferiert ausschließlich Daten und benötigt daher kein Datenformat. Gewählt wurde dieser Ansatz aus zwei Gründen:

- Zum einen müssen so die Daten nicht in ein Datenformat serialisiert werden, um Kontrollinformationen und Daten aus der Nachricht zu parsen

- Zum anderen können während einer Datenübertragung gleichzeitig Kontrollnachrichten versendet werden. Diese Multistreaming-Funktionalität wird in Kapitel 2.3 beschrieben.

Die Kommunikation auf dem Kontrollkanal ähnelt der Kommunikation des Signaling-Protokolls. Auch hier werden JSON-Objekte mit dem Schlüssel „type“ versendet, der den Typ einer Nachricht festlegt. Folgende Nachricht ist für das Peer-Protokoll definiert:

Typ	**resources**
Beschreibung	Beschreibt dem anderen Peer die Zugehörigkeiten der Datenpakete auf dem Datenkanal.
Zusätzliche Parameter	**resources**: Array[Object] Die beschriebenen Inhalte inklusive der genauen *Pieces*.

Abbildung 24: Beschreibung der *resources*-Nachricht

3.9 Verwendete Software und Werkzeuge

Zur Realisierung der Lösung und Experimente wurden diverse Bibliotheken und Werkzeuge hinzugezogen. In diesem Kapitel wird deren Nutzen im Rahmen dieser Arbeit erläutert und gliedert sich in die Bereiche: Frontend, Backend und Testumgebung. Das Frontend beschreibt den Teil der Middleware, der im Browser ausgeführt wird und Backend den anderen Teil, der auf dem Server ausgeführt wird.

Sowohl Frontend als auch Backend sind in JavaScript programmiert und machen Gebrauch vom JavaScript-Compiler Babel [83]. Dieser Compiler dient zur Erweiterung des JavaScript-Sprachkerns, wodurch neue Funktionen ermöglicht werden, die in vielen JavaScript-Engines noch nicht verfügbar sind. Dazu transformiert Babel die JavaScript-Syntax in eine andere, die von der jeweiligen JavaScript-Engine verstanden wird. Im Backend ist dies die V8-JavaScript-Engine, die z.B. in Google Chrome Verwendung findet. Die zugrundeliegende JavaScript-Engine im Frontend ist je nach benutztem Browser eine andere. Das könnte unter anderem Mozillas SpiderMonkey [40] oder Microsofts Chakra [84] sein.

Das (automatische) Kompilieren des JavaScript-Quelltextes mittels Babel übernimmt Gulp [85], ein sogenannter Task Runner. Gulp vereinfacht das Erstellen von Aufgaben, wie in diesem Falle das Kompilieren. Weitere Anwendungsfälle sind beispielsweise

das Starten und Stoppen von Anwendungen, die Kompilierung von CSS, das Zusammenfassen von Quelltexten und die Minimierung von sämtlichen Inhalten. Ohne Plug-ins verfügt Gulp allerdings selbst nicht über viel Funktionalität. Für viele Funktionalitäten ist ein Plug-in erforderlich. Verwendet werden folgende Gulp-Plug-ins: *gulp-babel* [86], *gulp-newer* [87], *gulp-rename* [88] und *gulp-sourcemaps* [89].

3.9.1 Frontend

Das Frontend beschreibt die JavaScript-Anwendung, die den Browser des Benutzers steuert. Hier findet nur eine Bibliothek Verwendung: Der *WebRTC adapter*. Sie dient dazu die browserunterschiedlichen WebRTC-Schnittstellen zu vereinheitlichen, sodass ein einfacher Gebrauch möglich ist. Außerdem erweitert der Adapter die Schnittstelle um eine *Promise*-Unterstützung [90]. *Promises* dienen zur besseren Programmierung mit asynchronen Operationen und wirken der *Callback Hell* [91] entgegen. Zu sehen in folgendem Beispiel:

```
function createOffer(pc) {
    pc.createOffer(offer => {
        pc.setLocalDescription(offer)
    })
}
async function createOffer(pc) {
    pc.setLocalDescription(await pc.createOffer())
}
```

Abbildung 25: Vergleich der WebRTC-Schnittstelle mit und ohne *Promise*-Unterstützung

3.9.2 Backend

Der Signaling-Server macht Gebrauch einer WebSocket-Bibliothek, die im Node Package Manager [60] - ein Paketmanager für JavaScript – unter dem Namen *websocket* [92] zu finden ist. Diese Bibliothek nimmt viel Arbeit ab, indem es eine Implementation des WebSocket-Servers bereitstellt. Neben *websocket* existiert auch eine beliebtere WebSocket-Server-Implementation unter dem Namen *ws* [93]. Auf diese Bibliothek wurde zuerst in der Lösung aufgebaut, aber aufgrund von mangelnder Flexibilität wieder entfernt. Insbesondere im Hinblick auf die Einstellbarkeit der maximalen Nachrichtenlänge einer WebSocket-Nachricht. Denn anders als beispielsweise TCP, arbeitet WebSocket nachrichtenorientiert, d.h. in den

meisten Implementierungen bekommt man als Anwender nur fertige Nachrichten von unbestimmter Länge zu sehen, die sich dann bereits im Speicher befinden und Ressourcen beanspruchen. So kann eine WebSocket-Nachricht bei dieser Implementation die gesamten Ressourcen eines Systems verbrauchen.

3.9.3 Testumgebung

Zum umfangreichen Testen der Lösung werden möglichst viele Browser benötigt, um eine Umgebung zu simulieren bei der sich viele Browser auf einer Webseite befinden, die sich die Inhalte der Webseite untereinander teilen. Selenium [94] ist ein Werkzeug zum Automatisieren von Browsern. Die Automatisierung erfolgt über eine der vielen unterstützen Programmiersprachen. Java, C#, Ruby, Python und JavaScript werden offiziell unterstützt. Im Rahmen dieser Arbeit wurde sich für Python entschieden, da Python bereits in vielen Linux-Distributionen und auf dieser Testplattform enthalten ist. Als Browser wird Firefox in der Version 44 verwendet, da dieser ohne extra Bibliotheken von Selenium automatisiert werden kann. Firefox wird im *headless*-Modus gestartet, d.h. der Browser wird nicht auf einem Monitor dargestellt. So kann der Browser auf Systemen ausgeführt werden, die über keinen Monitor verfügen. Hierzu wird die Software Xvfb eingesetzt, die einen virtuellen Monitor simuliert.

4 Experimente

Im Folgenden werden nun die Auswirkungen durch Verlagerung der Datenverkehrsmuster getestet. Dabei wird die benötigte Zeit der Browser zur Auflösung der Inhalte gemessen und die Auslastung des Webservers bestimmt. Es wird erwartet, dass bei Überlastung des Webservers die peerunterstützte Verteilung sowohl eine kürzere Auflösungszeit der Browser, als auch positive Auswirkungen auf die Serverbandbreite des Webservers bewirkt.

4.1 Experimentaufbau

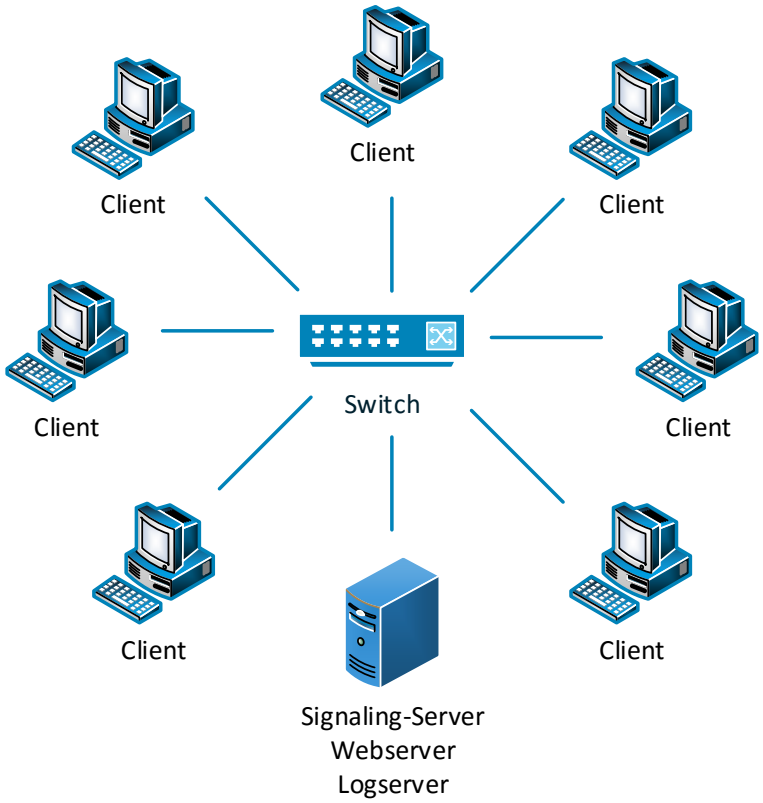

Abbildung 26: Testaufbau in Stern-Topologie

In den Tests werden acht Computer verwendet, die alle über einen Ethernet-Switch verbunden sind. Auf allen Computern wird die Linux-Distribution *openSUSE* [95] und auf dem Server Windows 8.1 verwendet. Die Computer sind alle mit 1 Gbit/s vollduplex zum Switch verbunden. Der Server ist auf 10 Mbit/s vollduplex gedrosselt, um effizienter den relevanten Fall testen zu können, bei der die Serverbandbreite überlastet ist. Als Webserver kommt *nginx* [96] („Engine-X“ ausgesprochen [97]) zum Einsatz. Sämtliche HTTP-Caching-Header wurden in der Konfiguration von *nginx* deaktiviert, damit die Browser bei den Seitenaufrufen jeweils einen neuen Besucher simulieren können.

Die aufzurufende Testwebseite besteht im Grunde nur aus der Middleware und zwei Bildern, um Seiteneffekte zu reduzieren. Ein Bild im JPG-Format mit einer Größe von 41.361 Bytes und eins im PNG-Format [98] mit einer Größe von 398.167 Bytes, mit einer Bildauflösung in der Größenordnung von 640 x 480 Pixeln.

Vom Webserver werden die HTTP-Anfragen protokolliert und so die Anfragen pro Sekunde als auch die beanspruchte Bandbreite ermittelt. Die Zeit, die die Browser zur Auflösung der Inhalte benötigen, wird durch eine eingebettete JavaScript-Datei ermittelt. Dabei wird Gebrauch der Performance-Schnittstelle [99] im Browser gemacht, um viele Zeitstempel in hoher Zeitauflösung zur Verfügung zu haben. Die ermittelten Testwerte des Browsers werden beim Verlassen der Webseite durch die *sendBeacon*-Funktion [100] zum Logserver gesendet. Während des Testaufbaus ist aufgefallen, dass Browsererweiterungen wie *uBlock Origin* [101] - ein Blocker für Inhalte auf Webseiten - in der Standardeinstellung die *sendBeacon*-Funktion im Browser deaktivieren. Die „Begründung“ dieser Entscheidung [102] kann nicht geteilt werden, da *sendBeacon* eine browserfreundliche Alternative zu anderen Methoden darstellt, die u.U. den Browser zum Stocken bringen bis alle Daten übertragen wurden. Im Folgenden ist so eine Methode gezeigt:

```
window.addEventListener('unload', () => {
    let xhr = new XMLHttpRequest()
    xhr.open('POST', logServerURL, false)
    xhr.setRequestHeader('Content-Type', 'text/plain')
    xhr.send(data)
})
```

Abbildung 27: Eine Methode um Daten beim Verlassen einer Webseite zu versenden

4.2 Experimentdurchführung

Per SSH [103] wird sich auf alle Clients verbunden. Auf jedem Client wird ein Skript ausgeführt, das jeweils vier Browser startet, die alle fünf bis sechs Sekunden die Testwebseite aufrufen. Zwei Tests werden durchgeführt. Beim ersten Test wird die Inhaltsauflösung ohne Middleware durchgeführt und beim zweiten Test mit Middleware, wodurch die Bilder peerunterstützt aufgelöst werden.

4.3 Ergebnisse und Diskussion

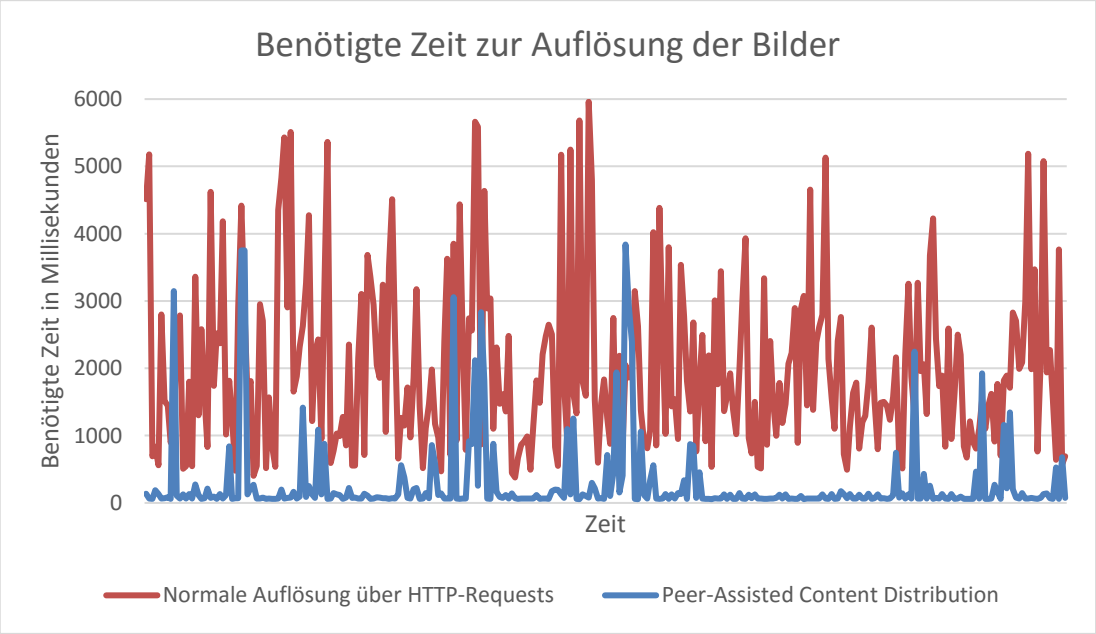

Abbildung 28: Vergleich beider Tests zur benötigten Zeit zur Auflösung der Bilder

Das in Abbildung 28 dargestellte Diagramm zeigt die benötigte Zeit der Browser zur Auflösung der Bilder. Der Durchschnitt der gemessenen Werte liegt beim ersten Test gerundet bei 2.006 ms und beim zweiten Test gerundet bei 277 ms. Es ist zu sehen, dass die Browser beim Test der Middleware deutlich kürzer für die Auflösung der Bilder benötigt haben, als bei der normalen Auflösung. Dieses Ergebnis wurde erwartet. Die höheren Werte beim Test der Middleware jedoch nicht. Diese sind auf Fehlern in der Middleware zurückzuführen. *Race Conditions* [104] wurden beobachtet, wodurch die Auflösung nicht innerhalb einer bestimmten Zeitspanne erfolgte. Nach Ablauf dieser Zeitspanne werden die Inhalte stattdessen ganz vom Webserver heruntergeladen. So entstehen diese hohen Werte. Durch diese Fehler wird das Ergebnis allerdings nicht entscheidend verfälscht.

Unterstützt der Webserver auch HTTP-Range-Anfragen, so können auch einzelne Byteabschnitte heruntergeladen werden, sodass nur fehlende Bytes vom Webserver übertragen werden. Diese Funktion unterstützt die Middleware im Rahmen dieser Arbeit aber jedoch noch nicht.

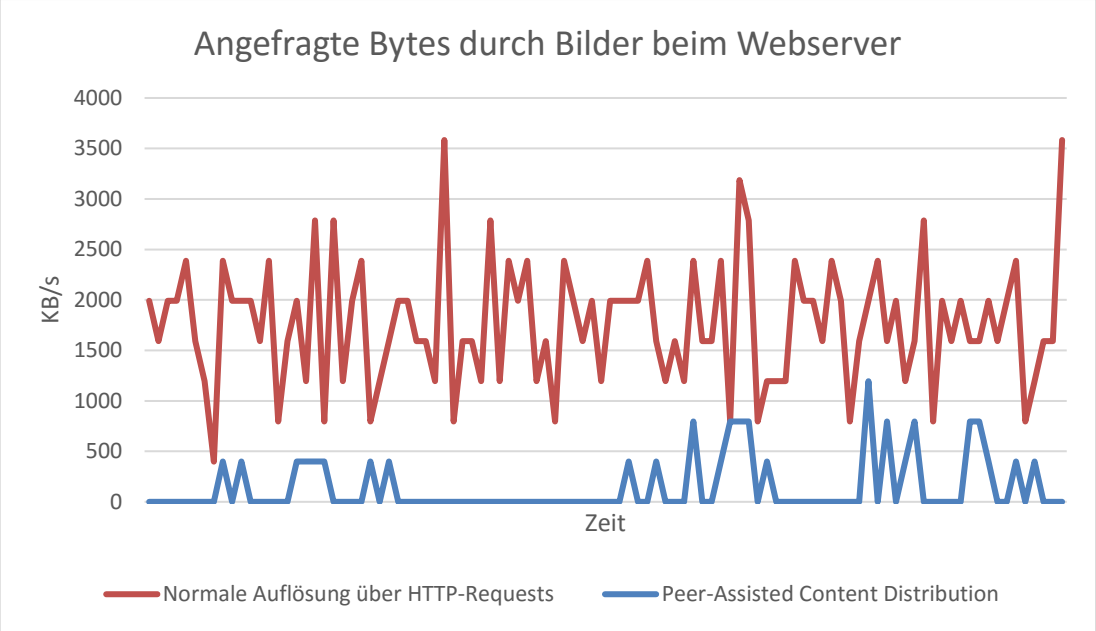

Abbildung 29: Vergleich beider Tests zu den angefragten Bytes beim Webserver

In Abbildung 29 sind die angefragten Bytes der Bilder pro Sekunde zu sehen. Gerundet liegt der Durchschnitt beim ersten Test bei 1.768 KB/s und beim zweiten Test gerundet bei 139 KB/s. Die maximale Übertragungsrate des Servers liegt bei 1.250 KB/s. Einige Werte liegen beim ersten Test über diesem Limit. Das Webserverlog verfügt nur über eine Auflösung im Sekundenbereich und das Diagramm zeigt nur die Menge an angeforderten Bytes, also nicht die wirkliche Übertragungsrate. Dadurch können diese Werte über der maximalen Übertragungsrate erklärt werden.

Beim Test der Middleware ist zu entnehmen, dass zeitweise gar keine Bandbreite des Webservers beansprucht wurde. Wie bereits erwähnt, lädt die Middleware bei Fehlerfällen das gesamte Bild herunter, obwohl unter Umständen nur noch wenige Bytes benötigt werden, da der Rest bereits von anderen Peers bezogen wurde. Dadurch ist die benötigte Bandbreite höher als sie sein könnte. Falls zum Zeitpunkt einer Anfrage nach einem Bild kein Peer verfügbar ist, wird das Bild direkt vom Webserver heruntergeladen, wodurch die angeforderten Bytes nach Bildern beim Webserver steigen.

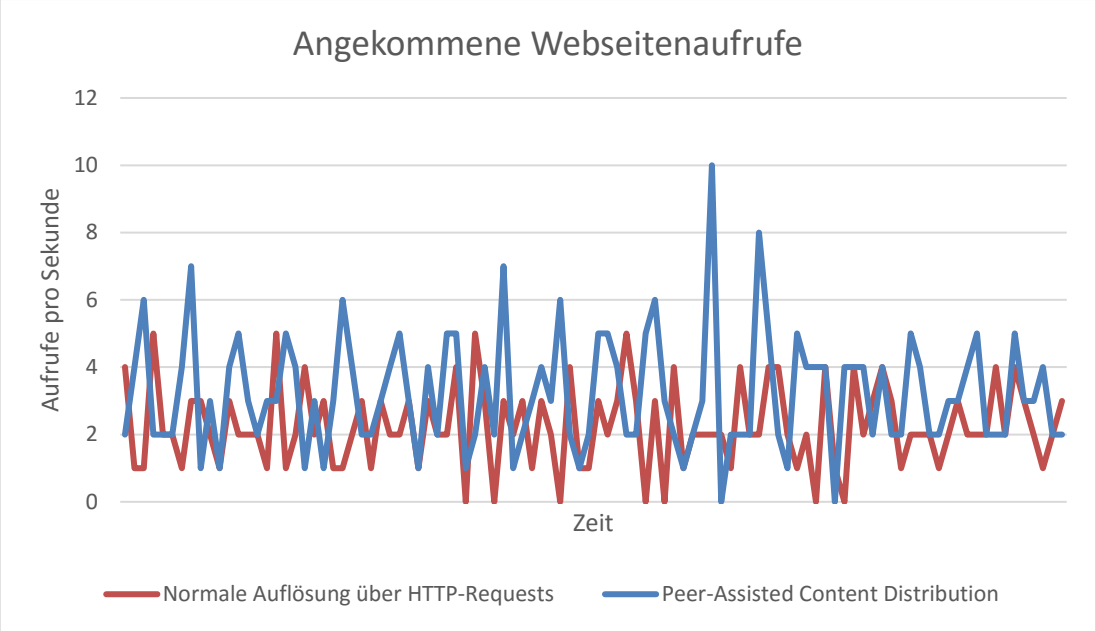

Abbildung 30: Anfragen pro Sekunde beider Tests

Zu sehen sind in Abbildung 30 die angekommenen Webseitenaufrufe beim Webserver. Der durchschnittliche Wert beim ersten Test liegt gerundet bei 3,21 und bei zweiten Test gerundet bei 2,25. Es ist zu erkennen, dass mehr Anfragen pro Sekunde beim Test der Middleware zum Webserver durchgekommen sind, obwohl gleichviele Browser in gleichen Intervallen die Testwebseite aufgerufen haben. Der Grund hierfür ist in Abbildung 29 zu sehen. Der Webserver war damit beschäftigt die Bilder zu übertragen und konnte aufgrund der Überlastung der Bandbreite nicht alle Webseitenaufrufe verarbeiten. Selbst bei mehr Seitenaufrufen fielen die angefragten Bytes beim Webserver geringer aus.

Diese Tests zeigen, dass die Anpassung der Verkehrsmuster erfolgreich ist. Zusätzlich konnten im Falle einer Überbeanspruchung der Bandbreite mehr HTTP-Anfragen bearbeitet werden, die Auflösung der Inhalte schneller erfolgen und die benötigte Bandbreite des Webservers reduziert werden, wenn die Middleware im Einsatz war. Diese Ergebnisse entsprechen dem Erwartungsziel.

5 Schlussbetrachtung

Ziel dieser Arbeit war die Verlagerung der typischen Client-Server-Datenverkehrsmuster im Web. Mithilfe der WebRTC-Schnittstelle und einer Client-Server-Anwendung konnte dies bewerkstelligt werden. Entwickelt wurde ein Signaling-Server, der für den Verbindungsaufbau der Browser sorgt, diese koordiniert und Metainformationen bereitstellt. Für den Browser wurde eine Peer-to-Peer-Anwendung in JavaScript entwickelt. Diese kann Datenelemente mit anderen Peers austauschen. Für die Kommunikation wurden zwei Netzwerkprotokolle entworfen, die jeweils zwischen Peers und zwischen Client und Server verlaufen. Zur Durchführung der Experimente wurde eine Testumgebung konstruiert, in der Browser automatisiert werden können. Eine Infrastruktur wurde geschaffen, bei der Browser eine Testwebseite von einem Webserver aufrufen können. Trotz kleineren Problemen und Einschränkungen der Lösung wurde das Ziel erreicht. Durch die Verlagerung wurde auf eine Entlastung der Webserver und einer größeren Erreichbarkeit gehofft. Mit den durchgeführten Experimenten konnte dies bestätigt werden.

5.1 Ausblick

Zukünftig kann die Middleware in einigen Bereichen verbessert werden. Es gilt herauszufinden, wie die Nachrichtenlänge auf WebRTC-Datenkanälen die Performance der Browser beeinflusst. Weiter kann für eine größere Browserkompatibilität gesorgt werden. Eine JavaScript-Implementation der eingesetzten Hashfunktion kann verwendet werden, um die Limitierung der Crypto-Schnittstelle im Chrome auf ungesicherten Webseiten zu umgehen. Die Anfrage beim Webserver nach einzelnen Byteabschnitten kann auch hinzugefügt werden, um nur Teile eines Inhalts herunterzuladen.

Wenn die WebRTC-Schnittstelle in Service Workers verfügbar wird, lassen sich umfangreichere Middlewares entwickeln, als die im Rahmen dieser Arbeit. Die Verfügbarkeit von Peers wird gesteigert, da die Service Workers selbst nach Schließen der Webseite weiter ausgeführt werden. Inhalte könnten im Cache abgelegt werden und bis auf das Einfügen der Middleware wäre keine Änderung an der Webseite nötig, um die Inhalte über die Middleware aufzulösen.

Der Signaling-Server kann zukünftig weitere Informationen zur Auswahl der Peers hinzuziehen, wie die geographische Position, Latenzzeit und die verfügbare Bandbreite. Die Funktion als Proxy-Server kann auch erweitert werden, um Inhalte von Webseiten mit fehlenden CORS-Header [71] im Peer-to-Peer-Netz verfügbar zu machen. Für dynamische Inhalte muss allerdings noch eine Strategie entwickelt werden.

Vielleicht werden in Zukunft neue HTTP-Header standardisiert, damit Webserver Hashwerte der Inhalte in den Kopfdaten zur Verfügung stellen, wodurch der Signaling-Server diese nicht im Vorfeld selbst ermitteln muss.

A. Anhang

A.1 Quelltext

Der Quelltext der Middleware (Client und Server) befindet sich auf der beigelegten CD im ZIP-Archiv *Quelltext.zip*.

Alternativ befindet sich der Quelltext auf GitHub unter:

https://github.com/Transport-Protocol/HAW-PAC

A.2 Signaling-Protokoll

Typ	**request**
Beschreibung	Der Browser erfragt, wie die URLs in **requests** aufzulösen sind. Ob über Peers oder vom Webserver selbst. Optional werden Offers mitgesendet, um eine zügigere Verbindung zu Peers zu ermöglichen.
Zusätzliche Parameter	**requests**: Array[String] Die URLs, die der Browser auflösen möchte. **offers**: Array[Object] Vom Browser mitgesendete Offers samt *ICE candidates*.

Typ	**candidate**
Beschreibung	Der „ICE agent“ hat neue IP und Portpaare gefunden. Diese werden dem Signaling-Server mitgeteilt.
Zusätzliche Parameter	**candidate**: Object IP-Adresse und Port. **offerId**: Integer Die zugehörige Offer-ID zum Kandidat. - oder -

	answerId: Integer Die zugehörige Answer-ID zum Kandidat.

Typ	**answer**
Beschreibung	Ein Peer beantwortet ein SDP-Offer.
Zusätzliche Parameter	**answerId**: Integer Die zugehörige Answer-ID. **answer**: Object Die SDP-Antwort zur Offer.

Typ	**offer**
Beschreibung	Der Browser hinterlegt beim Signaling-Server SDP-Offers, um schneller mit Peers verbunden werden zu können.
Zusätzliche Parameter	**offers**: Array[Object] Die Offers.

Folgende Nachrichtentypen sind für das Signaling-Protokoll vom Signaling-Server zum Browser definiert:

Typ	**origin**
Beschreibung	Informiert den Browser darüber, die externen Inhalte vom Webserver zu laden. Grund dafür könnten zu wenige Peers sein.
Zusätzliche Parameter	**resources**: Array[String] Die URLs, die vom Browser selbst aufgelöst werden sollen.

Typ	**distributed**
Beschreibung	Liefert dem anfragenden Browser eine Antwort auf die zuvor gesendete Offer, damit sich die Peers verbinden können.

Zusätzliche Parameter	**offerId**: Integer Die zugehörige Offer-ID des anfragenden Browsers. **answer**: Object Die SDP-Antwort zur Offer.

Typ	**candidate**
Beschreibung	Informiert den Browser über neue Kandidaten für den Verbindungsaufbau zum Peer.
Zusätzliche Parameter	**candidate**: Object IP-Adresse und Port. **offerId**: Integer Die zugehörige Offer-ID zum Kandidat. - oder - **answerId**: Integer Die zugehörige Answer-ID zum Kandidat.

Typ	**refill**
Beschreibung	Veranlasst den Browser neue Offers zu erstellen.
Zusätzliche Parameter	**count**: Integer Anzahl der zu erstellenden Offers.

Typ	**metadata**
Beschreibung	Teilt dem Browser Metadaten zu den Ressourcen mit.
Zusätzliche Parameter	**resources**: Array[Object] Jede Ressource beinhaltet die URL, die Hashwerte, die Länge, die Größe eines *Piece*, die Anzahl der *Pieces* und die Größe des letzten *Piece*.

Typ	**seed**

Beschreibung	Fordert den Browser dazu auf, Inhalte an einen bereits verbundenen Peer zu übermitteln.
Zusätzliche Parameter	**resources**: Array[Object] Die zu übermittelnde Inhalte inklusive der genauen *Pieces*. **offerId**: Integer Die zugehörige Offer-ID zum Peer. - oder - **answerId**: Integer Die zugehörige Answer-ID zum Peer.

Typ	**peer**
Beschreibung	Veranlasst die Verbindung zu einem neuen Peer und beinhaltet auch die zu übermittelnde Inhalte an den Peer.
Zusätzliche Parameter	**resources**: Array[Object] Die zu übermittelnde Inhalte inklusive der genauen *Pieces*. **offer**: Object Die SDP-Offer des anfragenden Peers. **candidates**: Array[Object] Mehrere IP-Adresse und Portpaare des Peers. **answerId**: Integer Die zugehörige Answer-ID.

Literaturverzeichnis

[1] M. Belshe, R. Peon und M. Thomson, „Hypertext Transfer Protocol Version 2 (HTTP/2),“ Mai 2015. [Online]. Available: https://tools.ietf.org/html/rfc7540.

[2] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor und S. Pfeiffer, „HTML5,“ 28 Oktober 2014. [Online]. Available: https://www.w3.org/TR/html5/. [Zugriff am 18 April 2016].

[3] „Internet Engineering Task Force (IETF),“ [Online]. Available: http://ietf.org/. [Zugriff am 8 Juli 2016].

[4] „World Wide Web Consortium (W3C),“ [Online]. Available: https://www.w3.org/. [Zugriff am 8 Juli 2016].

[5] I. Fette und A. Melnikov, „The WebSocket Protocol,“ Dezember 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455.

[6] I. Hickson, „Server-Sent Events,“ 03 Februar 2015. [Online]. Available: https://www.w3.org/TR/eventsource/. [Zugriff am 18 April 2016].

[7] A. v. Kesteren, „XMLHttpRequest Standard,“ 25 März 2016. [Online]. Available: https://xhr.spec.whatwg.org/. [Zugriff am 18 April 2016].

[8] A. van Kesteren, „Fetch Standard,“ 18 April 2016. [Online]. Available: https://fetch.spec.whatwg.org/. [Zugriff am 18 April 2016].

[9] S. Schwartz, „WebSockets and Methods for Real-Time Data Streaming,“ 12 Februar 2012. [Online]. Available: https://os.alfajango.com/websockets-slides/. [Zugriff am 18 April 2016].

[10] D. Kaspar, K. Evensen, P. Engelstad und A. F. Hansen, „Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces,“ *Communications (ICC), 2010 IEEE International Conference on,* p. 5, 23 Mai 2010.

[11] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan und B. Aboba, „WebRTC 1.0: Real-time Communication Between Browsers,“ 15 Februar 2016. [Online]. Available: https://w3c.github.io/webrtc-pc/. [Zugriff am 18 April 2016].

[12] „Real-Time Communication in WEB-browsers (rtcweb),“ [Online]. Available: https://datatracker.ietf.org/wg/rtcweb/documents/. [Zugriff am 9 Juli 2016].

[13] „Is WebRTC ready yet?,“ [Online]. Available: http://iswebrtcreadyyet.com/. [Zugriff am 9 Juli 2016].

[14] I. Grigorik, High Performance Browser Networking, September: O'Reilly Media, 2016.

[15] J. Rosenberg, „Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols,“ April 2010. [Online]. Available: https://tools.ietf.org/html/rfc5245.

[16] J. Rosenberg, R. Mahy, P. Matthews und D. Wing, „Session Traversal Utilities for NAT (STUN),“ Oktober 2008. [Online]. Available: https://tools.ietf.org/html/rfc5389.

[17] R. Mahy, P. Matthews und J. Rosenberg, „Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN),“ April 2010. [Online]. Available: https://tools.ietf.org/html/rfc5766.

[18] E. Rescorla und N. Modadugu, „Datagram Transport Layer Security Version 1.2,“ Januar 2012. [Online]. Available: https://tools.ietf.org/html/rfc6347.

[19] J. Postel, „User Datagram Protocol,“ 28 August 1980. [Online]. Available: https://tools.ietf.org/html/rfc768.

[20] T. Dierks und E. Rescorla, „The Transport Layer Security (TLS) Protocol Version 1.2,“ August 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246.

[21] M. Baugher, D. A. McGrew, M. Naslund, E. Carrara und K. Norrman, „The Secure Real-time Transport Protocol (SRTP),“ März 2004. [Online]. Available: https://tools.ietf.org/html/rfc3711.

[22] R. R. Stewart, „Stream Control Transmission Protocol,“ September 2007. [Online]. Available: https://tools.ietf.org/html/rfc4960.

[23] R. Jesup, S. Loreto und M. Tuexen, „WebRTC Data Channels," 4 Januar 2015. [Online]. Available: https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-13. [Zugriff am 18 April 2016].

[24] Y. Rekhter, R. G. Moskowitz, D. Karrenberg, G. Jan de Groot und E. Lear, „Address Allocation for Private Internets," Februar 1996. [Online]. Available: https://tools.ietf.org/html/rfc1918.

[25] M. Handley, V. Jacobson und C. Perkins, „SDP: Session Description Protocol," Juli 2006. [Online]. Available: https://tools.ietf.org/html/rfc4566.

[26] J. Rosenberg und H. Schulzrinne, „An Offer/Answer Model with the Session Description Protocol (SDP)," Juni 2002. [Online]. Available: https://tools.ietf.org/html/rfc3264.

[27] „Information technology – Open Systems Interconnection – Basic Reference Model: The basic model," 01 07 1994. [Online]. Available: http://handle.itu.int/11.1002/1000/2820. [Zugriff am 18 April 2016].

[28] R. R. Stewart, M. A. Ramalho, Q. Xie, M. Tuexen und P. T. Conrad, „Stream Control Transmission Protocol (SCTP) Partial Reliability Extension," Mai 2004. [Online]. Available: https://tools.ietf.org/html/rfc3758.

[29] R. Jesup, S. Loreto und M. Tuexen, „WebRTC Data Channel Establishment Protocol," 4 Januar 2015. [Online]. Available: https://tools.ietf.org/html/draft-ietf-rtcweb-data-protocol-09. [Zugriff am 18 April 2016].

[30] P. Srisuresh und M. Holdrege, „IP Network Address Translator (NAT) Terminology and Considerations," August 1999. [Online]. Available: https://tools.ietf.org/html/rfc2663.

[31] P. Srisuresh und K. B. Egevang, „Traditional IP Network Address Translator (Traditional NAT)," Januar 2001. [Online]. Available: https://tools.ietf.org/html/rfc3022.

[32] G. Huston, „The Changing Foundation of the Internet: Confronting IPv4 Address Exhaustion," Oktober 2008. [Online]. Available: http://wattle.rand.apnic.net/papers/isoc/2008-10/v4depletion.pdf. [Zugriff am 18 April 2016].

[33] J. Rosenberg, J. Weinberger, C. Huitema und R. Mahy, „STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)," März 2003. [Online]. Available: https://tools.ietf.org/html/rfc3489.

[34] „Information technology -- UPnP Device Architecture -- Part 1: UPnP Device Architecture Version 1.0," 1 September 2009. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57195. [Zugriff am 18 April 2016].

[35] D. Wing, S. Cheshire, M. Boucadair, R. Penno und P. Selkirk, „Port Control Protocol (PCP)," April 2013. [Online]. Available: https://tools.ietf.org/html/rfc6887.

[36] B. Ford, P. Srisuresh und D. Kegel, „Peer-to-Peer Communication Across Network Address Translators," *Proceedings of the 2005 USENIX Annual Technical Conference,* 2005.

[37] G. Huston, „Anatomy: A look inside network address translators," *The Internet Protocol Journal 7.3,* August 2004.

[38] A. Russell, J. Song und J. Archibald, „Service Workers," 25 Juni 2015. [Online]. Available: https://www.w3.org/TR/service-workers/. [Zugriff am 7 Juli 2016].

[39] A. Barth, „The Web Origin Concept," Dezember 2011. [Online]. Available: https://tools.ietf.org/html/rfc6454.

[40] Mozilla Foundation, „Firefox," [Online]. Available: https://www.mozilla.org/en-US/firefox/new/. [Zugriff am 9 Juli 2016].

[41] A. Deveria, „Service Workers," [Online]. Available: http://caniuse.com/#feat=serviceworkers. [Zugriff am 9 Juli 2016].

[42] „Firefox — Notes (44.0) — Mozilla," 26 Januar 2016. [Online]. Available: https://www.mozilla.org/en-US/firefox/44.0/releasenotes/. [Zugriff am 9 Juli 2016].

[43] Google Inc., „Chrome," [Online]. Available: https://www.google.com/chrome/. [Zugriff am 9 Juli 2016].

[44] „Chrome Releases: Stable Channel Update," 21 Januar 2015. [Online]. Available: http://googlechromereleases.blogspot.de/2015/01/stable-update.html. [Zugriff am 9 Juli 2016].

[45] „Functions and classes available to Web Workers," 25 März 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Functions_and_classes_available_to_workers. [Zugriff am 11 Juli 2016].

[46] F. Aboukhadijeh, „Add support for WebRTC Data Channel in Workers,“ 25 Mai 2015. [Online]. Available: https://github.com/w3c/webrtc-pc/issues/230. [Zugriff am 23 02 2016].

[47] M. S. Mikowski und J. C. Powell, Single Page Web Applications: JavaScript end-to-end, Manning Publications, 2013.

[48] F. Aboukhadijeh, „PeerCDN: WebRTC-based peer-to-peer CDN [video],“ 28 März 2013. [Online]. Available: https://news.ycombinator.com/item?id=5452780. [Zugriff am 6 Juli 2016].

[49] „PeerCDN Acquired by Yahoo!,“ 13 Dezember 2013. [Online]. Available: https://twitter.com/peerCDN/status/412994894458667008. [Zugriff am 11 Juli 2016].

[50] „PeerCdn.com Screenshot History,“ 30 Juni 2015. [Online]. Available: http://www.screenshots.com/peercdn.com/2015-06-30. [Zugriff am 6 Juli 2016].

[51] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble und H. M. Levy, „An analysis of internet content delivery systems,“ *ACM SIGOPS Operating Systems Review,* Nr. 36, pp. 315-327, 2002.

[52] „Peer5 - The Serverless P2P CDN For Video Live Streaming,“ [Online]. Available: https://www.peer5.com/. [Zugriff am 11 Juli 2016].

[53] „BasketBuild Blog - Downloads now powered by Peer5!,“ 9 Dezember 2014. [Online]. Available: https://web.archive.org/web/20150919121300/https://s.basketbuild.com/blog/post/downloads-now-powered-by-peer5. [Zugriff am 6 Juli 2016].

[54] C. Vogt, M. J. Werner und T. C. Schmidt, „Content-centric user networks: WebRTC as a path to name-based publishing,“ *21st IEEE International Conference on Network Protocols (ICNP),* pp. 1-3, 2013.

[55] E. Rescorla, „Introduction to distributed hash tables,“ *IAB plenary, IETF 65,* 2006.

[56] „Streamroot: client-accelerated streaming delivery optimization for VOD and live streams,“ [Online]. Available: http://streamroot.io/. [Zugriff am 11 Juli 2016].

[57] „Swarmify.com Fail-Proof Video Delivery,“ [Online]. Available: https://swarmify.com/. [Zugriff am 11 Juli 2016].

[58] B. Cohen, „The BitTorrent Protocol Specification,“ 10 Januar 2008. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html. [Zugriff am 28 Juni 2016].

[59] D. Flanagan, JavaScript: the definitive guide, O'Reilly Media, Inc., 2006.

[60] S. Tilkov und S. Vinoski, „Node. js: Using JavaScript to build high-performance network programs,“ *IEEE Internet Computing,* Nr. 14.6, pp. 80-83, 2010.

[61] The Chromium Project, „Chrome V8,“ [Online]. Available: https://developers.google.com/v8/. [Zugriff am 9 Juli 2016].

[62] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières und R. Morris, „Event-driven programming for robust software,“ *Proceedings of the 10th workshop on ACM SIGOPS European workshop,* pp. 186-189, 2002.

[63] T. Berners-Lee, R. T. Fielding und L. Masinter, „Uniform Resource Identifier (URI): Generic Syntax,“ Januar 2005. [Online]. Available: https://tools.ietf.org/html/rfc3986.

[64] „Secure Hash Standard (SHS),“ August 2015. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. [Zugriff am 8 Juli 2016].

[65] P. Rogaway und T. Shrimpton, „Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance,“ in *International Workshop on Fast Software Encryption*, Springer Berlin Heidelberg, 2004.

[66] R. Sleevi und M. Watson, „Web Cryptography API,“ 12 November 2015. [Online]. Available: https://dvcs.w3.org/hg/webcrypto-api/raw-file/tip/spec/Overview.html. [Zugriff am 28 Juni 2016].

[67] „Window.crypto - Web APIs | MDN,“ 25 November 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/crypto#Browser_compatibility. [Zugriff am 11 Juli 2016].

[68] M. Rouse, „Definition man-in-the-middle attack (MitM),“ Dezember 2015. [Online]. Available:

http://internetofthingsagenda.techtarget.com/definition/man-in-the-middle-attack-MitM. [Zugriff am 9 Juli 2016].

[69] D. Akhawe, F. Braun, F. Marier und J. Weinberger, „Subresource Integrity," 21 Juni 2016. [Online]. Available: https://w3c.github.io/webappsec-subresource-integrity/. [Zugriff am 1 Juli 2016].

[70] „Subresource Integrity - Web security | MDN," 5 November 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity #Browser_compatibility. [Zugriff am 11 Juli 2016].

[71] A. van Kesteren, „Cross-Origin Resource Sharing," 16 Januar 2014. [Online]. Available: https://www.w3.org/TR/cors/. [Zugriff am 11 Juli 2016].

[72] A. Luotonen und K. Altis, „World-wide web proxies," *Computer Networks and ISDN systems,* Bd. 2, Nr. 27, pp. 147-154, 1994.

[73] M. Cáceres, F. J. Moreno und I. Grigorik, „Network Information API," 2016 Mai 2016. [Online]. Available: http://wicg.github.io/netinfo/. [Zugriff am 7 Juli 2016].

[74] „NavigatorID.userAgent," 29 Februar 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/NavigatorID/userAgent. [Zugriff am 7 Juli 2016].

[75] „URL.createObjectURL() - Web APIs | MDN," 21 Januar 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL. [Zugriff am 11 Juli 2016].

[76] E. Uhrhane, „File API: Directories and System," 24 April 2014. [Online]. Available: https://dev.w3.org/2009/dap/file-system/pub/FileSystem/. [Zugriff am 7 Juli 2016].

[77] E. Uhrhane, „fileapi-directories-and-system/filewriter," 2 April 2014. [Online]. Available: http://lists.w3.org/Archives/Public/public-webapps/2014AprJun/0010.html. [Zugriff am 7 Juli 2016].

[78] „HTML," 7 Juli 2016. [Online]. Available: https://html.spec.whatwg.org/multipage/webstorage.html#dom-localstorage. [Zugriff am 7 Juli 2016].

[79] A. Deveria, „WebRTC Peer-to-peer connections,“ [Online]. Available: http://caniuse.com/#feat=rtcpeerconnection. [Zugriff am 9 Juli 2016].

[80] T. Bray, „The JavaScript Object Notation (JSON) Data Interchange Format,“ März 2014. [Online]. Available: https://tools.ietf.org/html/rfc7159.

[81] „MessagePack: It's like JSON. but fast and small,“ [Online]. Available: http://msgpack.org/. [Zugriff am 11 Juli 2016].

[82] F. Yergeau, „UTF-8, a transformation format of ISO 10646,“ November 2003. [Online]. Available: https://tools.ietf.org/html/rfc3629.

[83] „Babel · The compiler for writing next generation JavaScript,“ [Online]. Available: http://babeljs.io/. [Zugriff am 11 Juli 2016].

[84] Microsoft, „ChakraCore,“ [Online]. Available: https://github.com/Microsoft/ChakraCore. [Zugriff am 9 Juli 2016].

[85] „gulp.js - the streaming build system,“ [Online]. Available: http://gulpjs.com/. [Zugriff am 11 Juli 2016].

[86] S. Sorhus, „gulp-babel,“ [Online]. Available: https://www.npmjs.com/package/gulp-babel. [Zugriff am 9 Juli 2016].

[87] T. Schaub, „gulp-newer,“ [Online]. Available: https://www.npmjs.com/package/gulp-newer. [Zugriff am 9 Juli 2016].

[88] fractal, „gulp-rename,“ [Online]. Available: https://www.npmjs.com/package/gulp-rename. [Zugriff am 9 Juli 2016].

[89] F. Reiterer, „gulp-sourcemaps,“ [Online]. Available: https://www.npmjs.com/package/gulp-sourcemaps. [Zugriff am 9 Juli 2016].

[90] „Promise - JavaScript | MDN,“ 22 Juni 2016. [Online]. Available: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Zugriff am 11 Juli 2016].

[91] M. Ogden, „Callback Hell,“ [Online]. Available: http://callbackhell.com/. [Zugriff am 11 Juli 2016].

[92] B. McKelvey, „websocket,“ [Online]. Available: https://www.npmjs.com/package/websocket. [Zugriff am 9 Juli 2016].

[93] A. Kazemier, „ws,“ [Online]. Available: https://www.npmjs.com/package/ws. [Zugriff am 9 Juli 2016].

[94] „Selenium - Web Browser Automation,“ [Online]. Available: http://docs.seleniumhq.org/. [Zugriff am 11 Juli 2016].

[95] „The makers' choice for sysadmins, developers and desktop users,“ [Online]. Available: https://www.opensuse.org/. [Zugriff am 11 Juli 2016].

[96] I. Sysoev, „nginx news,“ [Online]. Available: http://nginx.org/. [Zugriff am 11 Juli 2016].

[97] „Frequently Asked Questions | NGINX,“ [Online]. Available: https://www.nginx.com/resources/wiki/community/faq/#how-do-you-pronounce-nginx. [Zugriff am 11 Juli 2016].

[98] „INFORMATION TECHNOLOGY – DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES – REQUIREMENTS AND GUIDELINES,“ September 1992. [Online]. Available: https://www.w3.org/Graphics/JPEG/itu-t81.pdf. [Zugriff am 6 Juli 2016].

[99] J. Mann, Z. Wang und A. Quach, „User Timing,“ 12 Dezember 2013. [Online]. Available: https://www.w3.org/TR/user-timing/#extensions-performance-interface. [Zugriff am 9 Juli 2016].

[100] „Navigator.sendBeacon() - Web APIs | MDN,“ 19 April 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/sendBeacon. [Zugriff am 11 Juli 2016].

[101] R. Hill, „gorhill/uBlock: uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean,“ [Online]. Available: https://github.com/gorhill/uBlock. [Zugriff am 11 Juli 2016].

[102] „Hyperlink-Auditing aka <a ping> and Beacon aka navigator.sendBeacon() | Wilders Security Forums,“ 12 Juni 2014. [Online]. Available: http://www.wilderssecurity.com/threads/hyperlink-auditing-aka-a-ping-and-beacon-aka-navigator-sendbeacon.364904/. [Zugriff am 11 Juli 2016].

[103] T. Ylonen und C. Lonvick, „The Secure Shell (SSH) Protocol Architecture,“ Januar 2006. [Online]. Available: https://tools.ietf.org/html/rfc4251.

[104] R. H. B. Netzer und B. P. Miller, „What are race conditions?: Some issues and formalizations,“ *ACM Letters on Programming Languages and Systems (LOPLAS),* Nr. 1.1, pp. 74-88, 1992.

[105] S. P. Romano und S. Loreto, Real-Time Communication with WebRTC, O'Reilly Media, Inc., 2014.

[106] C. Sommer, „Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)) algorithm,“ 4 November 2006. [Online]. Available: https://commons.wikimedia.org/wiki/File:STUN_Algorithm3.svg. [Zugriff am 18 April 2016].

[107] „Information technology -- Computer graphics and image processing -- Portable Network Graphics (PNG): Functional specification,“ 1 März 2004. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=29581. [Zugriff am 6 Juli 2016].

[108] Mozilla Foundation, „SpiderMonkey,“ [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. [Zugriff am 9 Juli 2016].

[109] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee, „Hypertext Transfer Protocol -- HTTP/1.1,“ Juni 1999. [Online]. Available: https://tools.ietf.org/html/rfc2616.

[110] P. Mockapetris, „DOMAIN NAMES - CONCEPTS AND FACILITIES,“ November 1987. [Online]. Available: https://tools.ietf.org/html/rfc1034.

[111] E. Rescorla, „HTTP Over TLS,“ Mai 2000. [Online]. Available: https://tools.ietf.org/html/rfc2818.

[112] M. Tuexen und R. R. Stewart, „UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication,“ Mai 2013. [Online]. Available: https://tools.ietf.org/html/rfc6951.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____