HAW HAMBURG

MASTER THESIS
Vincent Roederer

Peerunterstützte Verteilung von Webseiteninhalten

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Vincent Roederer

Peerunterstützte Verteilung von Webseiteninhalten

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science 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

Eingereicht am: 18. August 2023

Vincent Roederer

Thema der Arbeit

Peerunterstützte Verteilung von Webseiteninhalten

Stichworte

WebRTC, Service Worker, P2P, Peer, CDN, eCDN, Browser, Datenkanal, Signaling, Hashserver

Kurzzusammenfassung

Entwicklung und Evaluierung einer Lösung für den Einsatz auf Webseiten zur peerunterstützten Verteilung von Webseiteninhalten.

Vincent Roederer

Title of Thesis

Peer-assisted Distribution of Website Content

Keywords

WebRTC, Service Worker, P2P, Peer, CDN, eCDN, Browser, Data Channel, Signaling, Hashserver

Abstract

Development and evaluation of a solution for use on websites for peer-assisted distribution of website content.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Im World Wide Web surft man durch Abrufen von HTML-Dokumenten auf Webservern über das Hypertext-Übertragungsprotokoll (HTTP) im Internet. Der Kontakt zum Webserver entspricht dabei dem Client-Server-Modell. Mehrere Clients (Browser) rufen einen zentralen Server auf. Dieses Modell des Datenverkehrs kann zu einer Überlastung beim Server führen. Insbesondere wenn der Webserver nicht nur HTML-Dokumente liefert, sondern auch große Webinhalte wie Bilder und Videos, wodurch anfragende Browser mit längeren Auflösungszeiten zu rechnen haben. Um dem entgegenzuwirken, verwenden viele Webseiten ein Content Delivery Network (CDN). Ein CDN ist ein Netzwerk aus Servern, das unter anderem Webseiteninhalte wie Bilder und Videos auf eine Vielzahl von Servern repliziert. Dadurch werden die Ressourcen eines einzelnen Servers entlastet und das Problem wird auf mehrere Server skaliert. Eine weitere Möglichkeit besteht in der Anwendung einer Lösung zur peerunterstützten Verteilung von Webseiteninhalten. Dabei wird ausgenutzt, dass manche Browser bereits über Webseiteninhalte in ihrem Cache verfügen und gerade auch auf der jeweiligen Webseite surfen. Webseiteninhalte werden dann zwischen den Browsern untereinander geteilt und nicht von einem Server heruntergeladen. Möglich wird dies durch das Zusammenspiel mehrerer Browserschnittstellen wie WebRTC und den immer besser ausgestatteten Browsern hinsichtlich ihres Funktionsumfangs.

Um den Datenverkehr von Webservern zu reduzieren, soll eine Lösung entwickelt werden, die dies durch eine peerunterstützte Verteilung von Webseiteninhalten erzielt.

1.1 Zielsetzung

Das Ziel dieser Masterarbeit ist die Entwicklung und Evaluierung einer Lösung, die auf Webseiten eingesetzt werden kann, um eine peerunterstützte Verteilung von Webseiteninhalten zu ermöglichen. Wichtig dabei ist, dass diese auf bereits existierenden Webseiten

eingesetzt werden kann und keine weitreichenden Änderungen notwendig sind. Die Lösung soll wie ein Drop-in-Replacement funktionieren, welches nach Einsatz sämtliche Anfragen peerunterstützt auflöst. Außerdem sollen die Webserver entlastet werden und der Datenverkehr zum Teil auf die Browser verlagert werden, wodurch der Webseitenbetreiber mit Einsparungen rechnen kann, um so womöglich die Verfügbarkeit zu erhöhen. Weiter soll die peerunterstützte Auflösung auch dann funktionieren, selbst wenn der eigentliche Webserver nicht mehr erreichbar ist.

1.2 Einsatzgebiete

Die peerunterstützte Verteilung kann auf jeder Webseite eingesetzt werden, da es sich um eine generische Lösung handelt, die versucht, alle Anfragen peerunterstützt aufzulösen. Besonders gut geeignet ist der Einsatz auf Streaming-Webseiten oder webinhaltlastigen Webseiten, da sich dort sehr viel Datenverkehr beim Webserver einsparen lässt. Aber auch als Enterprise Content Delivery Network (eCDN) [1] ist diese Lösung interessant. Bei einem eCDN handelt es sich um ein CDN [2] in einem firmeninternen Netzwerk. Diese werden genutzt, um Internetdatenverkehr an den Grenzen des Firmennetzwerks einzusparen. Die Verwendung eines eCDNs gestaltet sich schwierig, da Server aufgesetzt und gewartet werden müssen. Womöglich sind Änderungen am Firmennetzwerk nötig und die Anwendungen der Mitarbeiter müssen erweitert werden, damit diese das eCDN verwenden. Ein peerunterstütztes eCDN kann einige dieser Probleme vereinfachen. Die Lösung ist für den einfachen Einsatz auf Webseiten vorgesehen und bis auf einen Koordinationsserver, benötigt die Lösung keine weiteren Server. Erweitert man die Webanwendungen der Mitarbeiter mit der Lösung, können Webinhalte lokal im Firmennetz geteilt werden, um so den Internetdatenverkehr am Firmennetz zu reduzieren.

1.3 Gliederung

Im folgenden Kapitel wird kurz auf die Funktionsweise der wichtigsten Technologien der Lösung eingegangen. Daraufhin werden ähnliche Lösungen in diesem Themenbereich vorgestellt, vor welchen Herausforderungen sie standen und wie sich die Lösung zu dieser unterscheidet. Dann wird im Hauptkapitel die Architektur einer Lösung zur peerunterstützten Verteilung von Webseiteninhalten ausführlich beschrieben. Wie sie funktioniert,

welche Limitierungen sie besitzt und welche Folgen der Einsatz hat. Im Experimententeil wird die Lösung in einer Testumgebung untersucht, wie gut sie funktioniert und mit welchen Datenübertragungsraten man zwischen den Peers rechnen kann. Dies wird hauptsächlich mit den Browsern Mozilla Firefox und Google Chrome untersucht. Schließlich werden die gewonnenen Erkenntnisse zusammengefasst und der Einsatz der Lösung eingeordnet.

2 Grundlagen

Im Folgenden werden grundlegende Technologien und Konzepte erklärt, auf die die Lösung zur peerunterstützten Verteilung aufbaut.

2.1 WebRTC

Web Real-Time Communication (WebRTC) [3] ist ein Sammelbegriff für Schnittstellen und Protokolle, die es ermöglichen, Peer-to-Peer-Verbindungen aufzubauen, auf denen Daten oder auch Video- und Audiostreams ausgetauscht werden können. Implementiert wird WebRTC hauptsächlich in Browsern, wie zum Beispiel in Google Chrome seit der Beta-Version 23 vom 2. Oktober 2012 [4]. Dadurch können Browser eine Direktverbindung zueinander aufbauen. WebRTC basiert auf den etablierten Netzwerkprotokollen UDP [5], DTLS [6], SRTP [7] und SCTP [8], die für den bidirektionalen Datenverkehr zuständig sind. Besonders bei der Verwendung von SCTP in WebRTC ist, dass SCTP nicht wie normalerweise neben UDP auf der Transport-Schicht des OSI-Modells [9] angesiedelt ist, sondern auf UDP betrieben wird. Grund dafür ist die fehlende Unterstützung in NAT-Systemen [10].

SCTP ist die Grundlage der WebRTC-Datenkanäle. Für die peerunterstützte Verteilung spielen sie eine entscheidende Rolle. Über die WebRTC-Datenkanäle lassen sich willkürliche Daten von Peer zu Peer übertragen. Neben diesen unterstützt WebRTC auch die gezielte Übertragung von Medien wie Video- und Audiostreams über SRTP. Alle Peer-to-Peer-Verbindungen sind im Rahmen von WebRTC Ende-zu-Ende verschlüsselt.

Neben Protokollen zur Datenübertragung, werden die Protokolle ICE [11], STUN [12] und TURN [13] in WebRTC für den Verbindungsaufbau verwendet, auf die im Weiteren eingegangen wird.

2.1.1 Network Address Translation

Nach Einführung der Network Address Translation (NAT) [14] in Routern zur Entschärfung der IPv4-Adressknappheit [15], ist der Verbindungsaufbau zwischen Peers zum Problem geworden, wodurch diverse Protokolle benötigt werden. Wird zum Beispiel außerhalb eines NAT-Systems versucht eine Verbindung zu einem Host innerhalb des NAT-Systems aufzubauen, weiß das NAT-System nicht, zu welchem Host die Verbindung aufgebaut werden soll, da sich aufgrund des NAT-Verfahrens mehrere Hosts eine einzelne IPv4-Adresse teilen. Das Ende-zu-Ende-Prinzip wird hierdurch verletzt [16].

Die Funktionsweise des NAT-Verfahrens beruht auf der Ersetzung der Quell-IP-Adresse und Port bei ausgehenden IP-Paketen mit der IP-Adresse des NAT-Systems und einem ausgewählten Port. Bei der Ersetzung wird eine Zuordnung im NAT-System angelegt, die beschreibt, mit welcher IP-Adresse und Port das ausgehende Paket, welches der interne Host versendet hat, ersetzt wurde. Erhält das NAT-System nun eingehende IP-Pakete, wird nachgeschaut, ob eine Zuordnung zu einem internen Host existiert. Falls eine Zuordnung existiert, wird die Ziel-IP-Adresse und Port des IP-Pakets mit denen des zugeordneten internen Hosts ersetzt. Dieser Funktionsweise ist zu entnehmen, dass ein interner Host zuerst eine Nachricht senden muss, damit er auf dem ersetzen IP-Adress- und Port-Paar Nachrichten von außen erhalten kann.

ICE definiert Methoden zur Durchdringung von NAT-Systemen, um die Verbindung zwischen Peers wieder zu ermöglichen (NAT-Traversal). Dazu kontaktieren Peers zunächst einen STUN-Server. Server befinden sich in der Regel nicht hinter einem NAT-System, wodurch Verbindungen zu ihnen ohne Probleme aufgebaut werden können. Dieser STUN-Server teilt den Peers ihre nach außen sichtbaren Verbindungsinformationen mit (IP-Adresse und Port). Einem weiteren Server, dem Signaling-Server, teilen die Peers ihre erhaltenen Verbindungsinformationen mit und erhalten jeweils die des anderen. Versuchen beide Peers nun eine Verbindung zueinander mit diesen Informationen aufzubauen, kann unter Umständen eine Verbindung hergestellt werden. Das wird dadurch erzielt, indem das NAT-System durch den selbst initiierten Verbindungsaufbaus eines Peers einen Eintrag in seiner NAT-Tabelle anlegt und dadurch beim Empfang der Nachrichten des anderen Peers, diese Kommunikation zugeordnet werden kann.

Entscheidend für die erfolgreiche Durchdringung des NAT-Systems ist dessen Funktionsweise. Die erfolgreiche Durchdringung beruht darauf, dass das NAT-System den ausgewählten Port bei der Ersetzung beibehält, wenn ein interner Host Nachrichten mit

demselben internen Port an unterschiedliche Hosts versendet. Ist dies nicht der Fall, wählt das NAT-System für die Verbindungen zum STUN-Server und zum Peer jeweils andere Ports aus, wodurch die Verbindungsinformationen vom STUN-Server, die der andere Peer erhält, nicht übereinstimmen. Für den anderen Peer ist es daher nicht möglich herauszufinden, welcher Port ausgewählt wurde, um zum Peer hinter dem NAT-System durchzudringen. In diesem Fall müsste der andere Peer den Port von insgesamt 2^{16} Ports erraten. Der Durchdringungsversuch wird an dieser Stelle meistens abgebrochen und es wird versucht eine Verbindung mithilfe eines Relay-Servers, dem TURN-Server, herzustellen. Im Rahmen dieser Arbeit ist die Verwendung eines TURN-Servers nicht zweckmäßig, da es einfacher ist, die peerunterstützte Verteilung abzubrechen und die Webinhalte direkt über den Webserver aufzulösen.

2.1.2 Verbindungsaufbau

Um eine Verbindung zu einem anderen Peer über WebRTC aufzubauen, sind vorher verschiedene Absprachen zwischen den Peers nötig. Dieser Prozess ist als Signaling bekannt und wird dem Anwender im Rahmen von WebRTC selbst überlassen.

Der Prozess des Verbindungsaufbaus zweier Peers beginnt damit, dass ein Peer ein SDP [17] erstellt. Listing 2.1 zeigt ein Beispiel eines SDP. Dabei handelt es sich um ein Textformat, das unter anderem beschreibt, wie der Peer zu erreichen ist und welche Datenkanäle und Medienstreams nach dem Verbindungsaufbau zu erwarten sind. Die Verbindungsinformationen, unter welchen Adressen der Peer erreichbar ist, werden mittels ICE [11] gesammelt und *ICE candidates* genannt. Die Erstellung des SDP ist als Offer bekannt. Diese muss nun zum anderen Peer übertragen werden. Das kann beispielsweise über einen Signaling-Server passieren, den beide Peers erreichen können. Erhält der andere Peer nun über den Signaling-Server das Angebot (Offer), wird das Angebot an die WebRTC-Schnittstelle übergeben und eine Antwort (Answer) erstellt. Diese Antwort muss nun zurück über den Signaling-Server zum Peer der das Angebot erstellt hat, damit dieser die Verbindungsinformationen des anderen Peers kennt und die Antwort ebenfalls an die WebRTC-Schnittstelle übergeben kann. Beide WebRTC-Instanzen versuchen nun aufgrund der Informationen des SDP eine Direktverbindung, womöglich durch ein NAT-System, zwischen den Peers aufzubauen.

Listing 2.1: Beispiel einer SDP-Beschreibung für den Verbindungsaufbau über WebRTC

```
v=0
o=mozilla...THIS_IS_SDPARTA-99.0 4525271934271926711 0 IN IP4 0.0.0.0
s=-
t=0 0
a=sendrecv
a=fingerprint:sha-256
 ↪  01:F8:A3:04:6E:AE:48:1B:C1:38:FB:8D:78:D9:60:3A:84:7E:5B:1C:39:79:0C:F9:95
a=group:BUNDLE 0 1 2
a=ice-options:trickle
a=msid-semantic:WMS *
m=application 56240 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 81.122.70.42
a=candidate:0 1 UDP 2122187007 268bbe93-4609-4415-b072-6c9d66f73ba2.local
 ↪  56240 typ host
a=candidate:2 1 UDP 2122252543 5567238c-497d-4219-8b20-432983ffd517.local
 ↪  56241 typ host
a=candidate:4 1 TCP 2105458943 268bbe93-4609-4415-b072-6c9d66f73ba2.local 9
 ↪  typ host tcptype active
a=candidate:5 1 TCP 2105524479 5567238c-497d-4219-8b20-432983ffd517.local 9
 ↪  typ host tcptype active
a=candidate:1 1 UDP 1685987327 81.122.70.42 56240 typ srflx raddr 0.0.0.0
 ↪  rport 0
a=candidate:3 1 UDP 1686052607 2003:e8:281a:d344:5e:ebb5:a542:96e7 56241 typ
 ↪  srflx raddr 0.0.0.0 rport 0
a=sendrecv
a=end-of-candidates
a=ice-pwd:b8369347d447718394666732f5c19261
a=ice-ufrag:a59ad01f
a=mid:2
a=setup:actpass
a=sctp-port:5000
a=max-message-size:1073741823
```

2.2 Service Worker

Bei einem Service Worker [18] handelt es sich um eine JavaScript-Datei [19] in einer speziellen Browserumgebung, die einer Webseite erweiterte Funktionalitäten ermöglicht. Ein Service Worker kann von jedem entwickelt werden und wird durch spezielle Anweisungen einer Webseite im Browser registriert. Bei erfolgreicher Registrierung wird der Service Worker im Hintergrund des Browsers ausgeführt und ist ab sofort für die registrierende Webseite zuständig. Das bedeutet, dass dieser selbst noch ausgeführt werden kann, obwohl die Webseite, die den Service Worker registriert hat, geschlossen wurde. Weiter kann ein registrierter Service Worker vom Browser selbst gestartet oder gestoppt

werden. Man erkennt hier schon den Unterschied zu normalen eingebundenen JavaScript-Anwendungen, die mit dem Schließen der Webseite beendet werden.

Mit dieser unterschiedlichen Lebensdauer ist aber noch nicht die erweiterte Funktionalität gemeint. Der Service Worker wird in einer speziellen Browserumgebung ausgeführt, in der der Service Worker beispielsweise über alle getätigten HTTP-Anfragen der registrierenden Webseite mittels Ereignisse informiert wird. Weiter erhält man die Möglichkeit, diese Anfragen selbst im Service Worker zu beantworten. Bei diesen Anfragen handelt es sich zum Beispiel um Anfragen zur Auflösung eines Bilds, das auf der Webseite über ein `img`-Element eingebunden wurde und dargestellt werden soll. Ohne registrierten Service Worker, würde der Browser selbst die Anfrage (bestehend aus der URL des Bilds) zum Webserver stellen und das Bild in diesem Fall selbst herunterladen.

In diesen Prozess lässt sich also mit einem Service Worker eingreifen. Verwendet wird dies unter anderem, um einen Offline-Modus für die eigene Webseite zu implementieren. Anfragen zu Webinhalten oder ganzen Unterseiten werden im Service Worker protokolliert und gegebenenfalls in einen eigenen Cache im Service Worker über die Cache-Schnittstelle [18] im Browser abgelegt. Ruft man beispielsweise ohne Internetverbindung eine Webseite auf, wird der Service Worker darüber benachrichtigt und dieser kann die Anfrage gegebenenfalls über den eigenen Cache beantworten, ohne dass die Anfrage durchs Internet gestellt werden muss.

Im Rahmen dieser Masterarbeit ist diese Funktionalität ein zentraler Bestandteil. Anfragen werden dabei im Service Worker abgefangen und es wird versucht, den angefragten Webinhalt über andere Peers aufzulösen und diesen im Service Worker wieder dem Browser zur Verfügung zu stellen.

2.3 Related Work

Im Themenbereich der peerunterstützten Verteilung existieren einige Lösungen, die zu ihrer Zeit mit weitgehenden Limitierungen in der Browserumgebung zu kämpfen hatten. Eine Lösung darunter ist Firecoral [20]. Firecoral war eine Browsererweiterung für Mozilla Firefox, die es ermöglicht hat, Webinhalte jeder Webseite mit anderen Peers – die auch die Erweiterung installiert haben – zu teilen und herunterzuladen. Die Komponenten ähneln stark der in dieser Masterarbeit vorgestellten Architektur. So verwendet Firecoral auch einen separaten Server, der Webinhalte selbst herunterlädt, um Hashes zu berechnen,

die den Peers für die Integritätsprüfung bereitgestellt werden können. Firecoral steht vor anderen Herausforderungen und unterscheidet sich zu dieser Arbeit in der Anwendung auf Webseiten und dem Datenaustausch über WebRTC.

Nach der Implementierung von WebRTC in Browsern, rund um das Jahr 2012, entstand eine Lösung unter dem Namen Maygh [21]. Maygh ist eine Anwendung, die auf einer Webseite eingebunden werden kann, um Daten, adressiert über ihren Hash, von anderen Peers, die sich auch gerade auf der Webseite befinden, herunterzuladen und zu teilen. Eine peerunterstützte Verteilung von Webinhalten lässt sich mit Maygh erreichen, indem die Webseite im Vorfeld Hashes zu ihren Webinhalten berechnet. Die Hashes werden dann auf einer Webseite eingebunden und über die JavaScript-Anwendung Maygh im Browser durch andere Peers aufgelöst. Dargestellt können die Daten daraufhin im Browser beispielsweise über `img`-Elemente, die mit den Daten gefüllt werden.

An diesem Punkt spiegeln sich die damaligen Limitierungen in der Browserumgebung wider. Es sind weitgehende Änderungen an der Webseite nötig, um Webinhalte peerunterstützt aufzulösen, da Anfragen noch nicht durch einen Service Worker abgefangen werden können. So kann beispielsweise kein Bild mehr über das `img`-Element mit dem `src`-Attribut eingebunden werden, weil der Browser dieses dann normal ohne Peers auflöst.

Eine neuere Lösung in diesem Bereich, die sowohl WebRTC als auch Service Worker verwendet, ist Arc [22]. Arc können Webseitenbetreiber auf ihrer Webseite einbinden, wodurch die Webseitenbesucher Teil eines globalen Arc-Netzwerks werden, das webseitenübergreifend Peers miteinander verbindet. Bereits diese Einbindung wird von Arc vergütet, denn die Webseitenbesucher werden dafür genutzt, Webinhalte von fremden Webseiten herunterzuladen und diese weiteren Peers zur Verfügung zu stellen. Mit Arc werden also auch Webinhalte peerunterstützt verteilt und aufgelöst. Jedoch wird durch die alleinige Einbindung von Arc noch kein Webinhalt der Webseite peerunterstützt aufgelöst oder verteilt. Erst wenn man Arc für die peerunterstützte Auflösung von Webinhalten bezahlt, werden die eigenen Webinhalte im Arc-Netzwerk an Peers verteilt, sodass diese für die peerunterstützte Auflösung verwendet werden können.

Viele weitere kommerzielle Lösungen [23] [24] [25] haben sich auf die peerunterstützte Verteilung von Video-Streams spezialisiert, da hier am meisten Datenverkehr beim Webserver eingespart werden kann. Anders als bei der allgemeinen Auflösung von Webinhalten, verwalten diese Lösungen selbst die Anfragen zu den Videos, was die Komplexität dieser Lösungen verringert.

WebTorrent WebTorrent [26] ist eine BitTorrent-Implementierung [27] fürs Web, mit der sich Torrents herunterladen und mit anderen Peers teilen lassen. Aufgrund der limitierenden Browserumgebung verwendet WebTorrent andere Protokolle, um den Datenaustausch im Web zu ermöglichen. Daher sind WebTorrent-Peers und BitTorrent-Peers nicht miteinander kompatibel.

WebTorrent teilt sich wesentliche Implementierungsaspekte mit der Lösung zur peerunterstützten Verteilung von Webseiteninhalten. Nämlich die Verwendung der WebRTC-Schnittstelle, die JavaScript-Anwendung im Browser und unter anderem auch der zentrale Server (Tracker), der die Peers zu den Torrents organisiert. Lösungen wie CacheP2P [28] haben versucht, auf Grundlage von WebTorrent, eine peerunterstützte Verteilung von Webseiten zu erreichen. Dazu musste CacheP2P unter anderem das Adressierungsproblem lösen. Denn Peers werden aufgrund der Hashes von Torrent-Dateien organisiert. Für die Anwendung auf einer Webseite, sind den Webseitenbesuchern nur die URLs von Webinhalten bekannt und können damit keine Peers finden. CacheP2P hat dafür die WebTorrent-Implementierung modifiziert und benutzt statt eines Hashes der Torrent-Datei den Hash der URL, um Peers zu organisieren. Mit dieser Entscheidung fällt allerdings eine zentrale Funktion von BitTorrent weg: Die Überprüfung, ob die erhaltenen Daten unverändert sind und dem Torrent entsprechen. Als Lösung wird vorgeschlagen, eine Liste von URLs mit einem Hash über ihren Inhalt zu pflegen, um die Integrität der Webinhalte überprüfen zu können.

Dies und weitere Limitierungen bei der Verwendung von WebTorrent als Grundlage für die peerunterstützte Verteilung von Webseiteninhalten, ist der Grund, wieso eine spezifischere Lösung im Rahmen dieser Masterarbeit entwickelt wurde.

3 Architektur

Zur Umsetzung der peerunterstützten Verteilung von Webseiteninhalten wurden drei Komponenten entwickelt: Der Koordinationsserver, das Webseitenskript und der Service Worker. Diese Komponenten sind essentiell für die Umsetzung der peerunterstützten Verteilung unter Berücksichtigung der Ziele (vgl. Kapitel 1.1). Der Koordinationsserver (im Folgenden Koordinator genannt) verwaltet dabei Peers und stellt wichtige Metainformationen zu Webseiteninhalten bereit. Die Client-Anwendung, die jeder Peer ausführt, besteht aus dem Webseitenskript und dem Service Worker. Beide Client-Komponenten spielen zusammen, um Limitierungen in der Browserumgebung zu umgehen, um so Abfragen des Browsers zu Webseiteninhalten abzufangen und diese peerunterstützt aufzulösen. Abbildung 3.1 stellt eine Architekturübersicht hinsichtlich der Kommunikation zwischen den Komponenten dar.

Im Folgenden wird ein grober Ablauf beschrieben, um die Funktionsweise der peerunterstützten Verteilung einordnen zu können. Technische Einzelheiten der Komponenten werden in den jeweiligen Kapiteln genauer erläutert. Einige Begriffe werden für das weitere Verständnis in Kapitel 3.1 definiert.

Die peerunterstützte Verteilung beginnt mit der Einbindung eines `script`-Elements im HTML-Dokument einer Webseite (mehr dazu in Kapitel 3.2). Das Skript, im Weiteren Clientskript genannt, ist hauptsächlich für die Herstellung von WebRTC-Verbindungen als auch für die Registrierung des Service Workers zuständig. Sobald ein Browser das Skript ausführt und der Service Worker registriert wurde, werden ab diesem Zeitpunkt alle Anfragen des Browsers zur zugehörigen Webseite über den Service Worker geleitet. Das bedeutet, dass die Verantwortung zur Auflösung der Anfragen vollständig an den Service Worker übergeben wird und nicht mehr nativ vom Browser gehandhabt wird. Alle vom Service Worker abgefangenen Anfragen werden von nun an an den Koordinator weitergeleitet, um zu erfahren, wie die Anfrage aufzulösen ist. Handelt es sich bei der Anfrage zum Beispiel um einen zwischenspeicherbaren Webseiteninhalt und es existieren

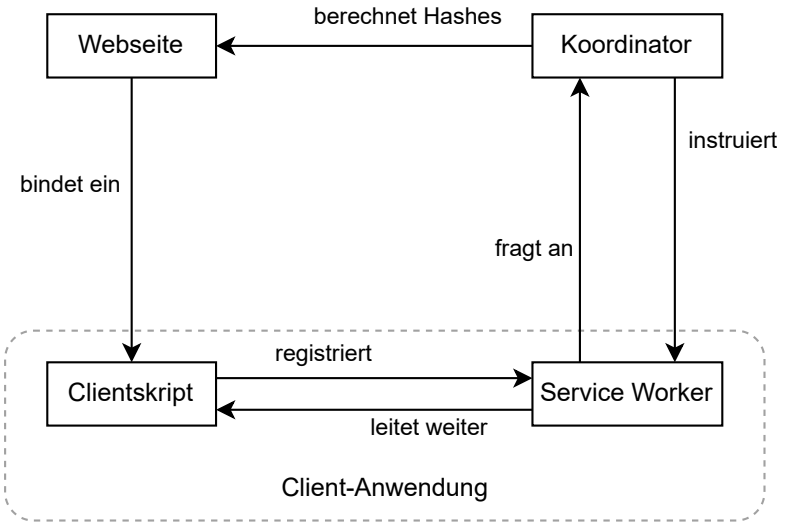

Abbildung 3.1: Architekturübersicht für die peerunterstützte Verteilung von Webseiteninhalten

Peers, die diesen Inhalt besitzen, antwortet der Koordinator mit Verbindungsinformationen zu den Peers, damit diese untereinander eine Direktverbindung über WebRTC zum Austausch des Webseiteninhalts herstellen können. Falls die Anfrage nicht zwischenspeichbar ist oder keine Peers existieren, weist der Koordinator den anfragenden Browser an, die Anfrage normal über HTTP vom Ursprungsserver herunterzuladen. Wird die Anfrage aber über Peers aufgelöst, senden diese dem anfragenden Browser Datenstücke des gefragten Webseiteninhalts. Die Datenstücke werden im Browser zusammengesetzt und mithilfe von Hashes, die vom Koordinator erhalten wurden, auf ihre Integrität überprüft. Ist alles in Ordnung, beantwortet der Service Worker die Anfrage mit den erhaltenen Daten der Peers und der Webseiteninhalt kann im Browser dargestellt werden.

Für eine derartige Funktionsweise mussten bisher immer Anwendungen, wie die Browsererweiterung Firecoral [20], von jedem Teilnehmer zuvor installiert werden, um Daten von Peer zu Peer austauschen zu können. Seit Einführung der WebRTC-Schnittstelle in Browsern ist dies nicht mehr notwendig, wodurch einfacher große P2P-Netze entstehen können. Dieser Umstand, das Zusammenspiel zwischen Clientskript und Service Worker und die Bereitstellung der Hashes über den Koordinator sind der Mittelpunkt dieser Masterarbeit. Mit dieser Architektur wird die peerunterstützte Verteilung von Webseiteninhalten erreicht, ohne vorherige Installation beim Anwender und ohne weitgehende Änderungen an Webseiten.

3.1 Begriffserläuterungen

Webseiteninhalt Mit Webseiteninhalt oder Webinhalt sind Inhalte wie Bilder, Videos oder Skripts gemeint, die über eine URL aufgerufen werden können.

Anfrage Damit ist eine HTTP-Anfrage gemeint, die vom Browser gestellt wird.

Normale Auflösung Die normale Auflösung einer Anfrage bedeutet, dass diese nicht peerunterstützt heruntergeladen wird, sondern normal über HTTP vom Ursprungsserver aufgelöst wird.

Ursprungsserver Mit Ursprungsserver ist der Webserver gemeint, von dem die Webinhalte ursprünglich stammen.

Clientskript Das Clientskript ist die JavaScript-Datei, die man auf einer Webseite über das `script`-Element einbindet.

Service Worker Eine JavaScript-Datei, welche im Hintergrund eines Browser ausgeführt wird und Webseitenanfragen auf Webseiten abfangen kann auf denen der Service Worker registriert wurde. Genaueres in Kapitel 2.2.

Das Durchlassen von Anfragen Diese Anfragen werden nicht an den Koordinator weitergeleitet und somit auch nicht peerunterstützt aufgelöst.

Auflösen von Anfragen Damit ist das Herunterladen und Darstellen von Anfragen im Browser gemeint.

GET-Anfrage Oder auch HEAD-Anfrage, bezeichnet eine HTTP-Anfrage mit der HTTP-Methode `GET` oder `HEAD`.

Peer Je nach Betrachtungsweise auch Client oder Browser. Dabei handelt sich um die gleiche Client-Anwendung. Aus Sicht zum Koordinator wird die Anwendung häufig als Client bezeichnet und beim Datenaustausch mit anderen Peers als Peer.

3.2 Einbindung der Lösung

Möchte man als Webseitenbetreiber die peerunterstützte Verteilung verwenden, müssen folgende Anforderungen erfüllt werden:

1. Die Webseite muss über HTTPS [29] verfügbar sein. Andernfalls verweigern Browser die Registrierung eines Service Workers.

2. Ein `script`-Element muss in den jeweiligen HTML-Dokumenten, die von den Webseitenbesuchern aufgerufen werden, hinzugefügt werden. Ein solches `script`-Element ist in Listing 3.1 gezeigt. Browser werden dadurch angewiesen, die JavaScript-Datei herunterzuladen und auszuführen. Besonders an diesem `script`-Element ist das `async`-Attribut. Die meisten Skripts, die auf Webseiten eingebunden werden, interagieren mit dem Document Object Model (DOM) [30]. Das DOM beschreibt die Struktur eines HTML-Dokuments. Die Interaktion mit dem DOM besteht zum Beispiel aus dem Registrieren eines Klick-Handlers auf Knöpfen. Dies gelingt allerdings nur, wenn das DOM für die Knöpfe bereits verfügbar ist. Um das zu garantieren, werden viele `script`-Elemente mit dem `defer`-Attribut versehen, um ihre Ausführung, bis das gesamte DOM verfügbar ist, zu verzögern. Die peerunterstützte Lösung interagiert jedoch nicht mit dem DOM, sondern nur mit der Auflösung von HTTP-Abfragen. Daher wird das `async`-Attribut verwendet, wodurch das Skript unverzüglich ausgeführt wird und zusätzlich das Parsen des DOM nicht blockiert wird.

3. Der für die peerunterstützte Verteilung entwickelte Service Worker, in Form einer JavaScript-Datei, muss unter der gleichen Origin[1] erreichbar gemacht werden, wie die Webseite, unter der die peerunterstützte Verteilung verwendet werden soll. Dies stellt womöglich die größte Hürde für viele Webseiten dar, weil die Datei beispielsweise zum Webserver hochgeladen werden muss. Aus Sicherheitsgründen kann ein Service Worker nicht von anderen Origins geladen werden. Denn sollte zum Beispiel

[1] Origin (deutsch: Herkunft) wird definiert durch das Schema, Hostnamen und Port der URL [31]. Man spricht von gleicher Herkunft (englisch: same-origin) zweier URLs, falls diese drei Werte übereinstimmen. Beispielsweise beschreibt `https://www.haw-hamburg.de` eine Origin.

Listing 3.1: Einbindung der Lösung zur peerunterstützten Verteilung von Webseiteninhalten in einem HTML-Dokument

```
1 <script async src="clientskript.js"></script>
```

eine Webseite eine XSS-Schwachstelle [32] besitzen, könnte so ein Angreifer einen eigenen Service Worker registrieren, der alle zukünftigen Anfragen des angegriffenen Browsers auf die Webseite abfangen und abändern kann, sodass Instruktionen zur Bereinigung des bösartigen Service Workers verhindert würden.

3.3 Koordinator

Der Koordinator ist ein WebSocket-Server [33], der folgende Aufgaben in der peerunterstützten Verteilung übernimmt:

- Er entscheidet, ob Webseiteninhalte vom Ursprungsserver oder über die peerunterstützte Verteilung aufzulösen sind. Dazu werden angefragte Webseiteninhalte selber vom Koordinator heruntergeladen und aufgrund ihrer Caching-Metadaten, die genauer in Kapitel 3.3.1 beschrieben werden, für geeignet oder ungeeignet eingestuft.
- Fungiert als Rendezvous-Punkt über den Verbindungsinformationen der Peers ausgetauscht werden. Diese Funktion ist auch bekannt als Signaling. Da Peers noch keine Informationen voneinander besitzen, muss der Koordinator die Peers miteinander bekannt machen. Durch den Austausch der Verbindungsinformationen können Peers eine Direktverbindung zueinander aufbauen. Dieses Verhalten ähnelt das einem BitTorrent-Tracker. Sowohl der Koordinator als auch der BitTorrent-Tracker verfügen über eine globale Ansicht aller teilnehmenden Peers und geben Verbindungsinformationen an anfragende Peers weiter.
- Die Berechnung und Bereitstellung von Hashes der angefragten Webseiteninhalte. Hashes werden benötigt, damit Peers ihre empfangenen Daten auf ihre Integrität überprüfen können. Die genaue Berechnung der Hashes wird in Kapitel 3.6 beschrieben.

Listing 3.2: Implizite und Explizite Verwendung eines CDN

```
1 <img src="https://www.haw-hamburg.de/bild.jpg">
2 <img src="https://d1342cf9za17q2.cloudfront.net/bild.jpg">
```

Damit die Sicherheit gewährleistet ist, sollte die WebSocket-Verbindung zum Koordinator verschlüsselt über HTTPS aufgebaut werden. Wird ein angefragter Webinhalt auch über HTTPS abgerufen, kann so auch die Authentizität der berechneten Hashes gewährleistet werden.

Anhand der Funktionsweise lässt sich erkennen, dass der Koordinator kein zentraler Dienst sein muss. Jeder Webseitenbetreiber kann einen eigenen Koordinator einsetzen. Dies bietet den Vorteil, dass die HTTP-Anfragen der Webseitenbesucher nicht an Dritte offenbart werden. Aber nicht nur der Koordinator kann in unabhängige Instanzen ausgelagert werden, auch die Aufgaben des Koordinators können in einzelne Dienste ausgelagert werden. Namen dieser Dienste könnten lauten: Signaling-Server, Proxy-Server, Hash-Server oder Tracker.

3.3.1 Eignung von Webseiteninhalten

Eine Herausforderung des Koordinators besteht darin, zu bestimmen, welche Webseiteninhalte für die peerunterstützte Verteilung geeignet sind. Vor ähnlichen Problemen stehen CDNs wie Cloudflare, die zwischen Webserver und Besucher geschaltet sind. Diese müssen aufgrund der Antworten vom Webserver ermitteln, was zwischengespeichert werden kann. Das entspricht einer impliziten Verwendung eines CDNs, weil nicht durch explizite URLs auf einen CDN verwiesen wird. Listing 3.2 zeigt in der ersten Zeile eine mögliche Verwendung eines impliziten CDN anhand einer Einbindung eines Bilds in einem HTML-Dokument der Webseite `https://www.haw-hamburg.de`. Zeile 2 zeigt die explizite Verlinkung zu einem CDN-Anbieter.

Vor einem ähnlichen Problem steht der Koordinator. Dieser erhält nämlich auch Anfragen der Browser, was der impliziten Verwendung eines CDNs entspricht. Allerdings kann im Rahmen der peerunterstützten Verteilung bereits eine erste Filterung aufgrund der HTTP-Methode der HTTP-Anfrage im Service Worker erfolgen. So, dass beispielsweise nur GET-Anfragen beim Koordinator angefragt werden. Denn nur solche Anfragen sind in der Regel zwischenspeicherbar und daher für die peerunterstützte Verteilung geeignet.

Erhält der Koordinator nun eine solche Anfrage (die unter anderem aus der URL besteht), prüft dieser, ob der Webseiteninhalt für die peerunterstützte Verteilung geeignet ist. Dazu setzt der Koordinator selbst die Anfrage zur angefragten URL ab und prüft dies aufgrund der Antwort vom Webserver. Der Algorithmus zur Feststellung, ob ein Webinhalt für die peerunterstützte Verteilung geeignet ist, entspricht größtenteils der Spezifikation des HTTP-Caching [34]. Ausgenommen sind jedoch Anfragen, die revalidiert werden müssen, beispielsweise durch Angabe von `Last-Modified` in den HTTP-Kopfdaten der Antwort. Bei den zwischenspeicherbaren Webinhalten sollte es sich um solche handeln, die über einen längeren Zeitraum in einem gemeinsamen Cache zwischengespeichert werden dürfen.

Abhängig von der Konfiguration der peerunterstützten Lösung (mehr dazu in Kapitel 3.4.5) erfordert jede GET-Anfrage zunächst eine Abfrage an den Koordinator, um festzustellen, ob die Anfrage für die peerunterstützte Verteilung geeignet ist oder nicht. Dies führt zu erhöhten Paketumlaufzeiten aller GET-Anfragen. Zwar könnte jeder Browser selbst in Erfahrung bringen ob eine Anfrage geeignet ist, indem eine HEAD-Anfrage auf dieselbe URL gestellt wird, jedoch verzögert es sich dabei auch um eine Paketumlaufzeit.

3.3.2 Absicherung des Koordinators

Da der Koordinator für die Bereitstellung der Hash-Werte und für die Prüfung von geeigneten Webinhalten zuständig ist, muss der Koordinator selbst Webinhalte herunterladen, die von Browsern erfragt werden. Dies stellt den Koordinator vor einige Herausforderungen, da jeder den Koordinator veranlassen kann, willkürliche GET-Anfragen zu stellen. Dadurch fungiert er ähnlich wie ein Proxy-Server [35]. Bei der Implementierung des Koordinators ist daher auf einige Punkte zu achten:

- Es muss überprüft werden, ob das Ziel der Anfrage eine Loopback-Adresse ist. Werden diese Anfragen nicht verhindert, können lokale Dienste des Koordinationsservers aufgerufen werden. In der Regel erlauben Programmbibliotheken zum Absetzen von HTTP-Anfragen die Angabe über eine URL. Für den Hostnamen der URL muss womöglich eine Namensauflösung durchgeführt werden, falls es sich nicht um eine IP-Adresse handelt, sondern um eine Domain. Hinter einer Domain kann sich allerdings immer noch eine Loopback-Adresse verbergen. Die Namensauflösung wird in den Programmbibliotheken meistens automatisch im Hintergrund

durchgeführt, wodurch man als Anwender der Bibliothek im Vorhinein nicht erfährt, ob es sich um eine Loopback-Adresse handelt. In solchen Fällen muss die Namensauflösung selbst durchgeführt werden, um die Prüfung durchzuführen.

- Webseiten können mit Anfragen vom Koordinator überflutet werden. Daher müssen Limitierungen eingeführt werden. Sowohl für den anfragenden Browser als auch für die jeweilige Webseite.
- Lädt der Koordinator bereits einen Webinhalt herunter und erhält in der Zeit eine erneute Anfrage auf den gleichen Webinhalt, so muss verhindert werden, dass der Koordinator mehrfach den Webinhalt herunterlädt. Dieses Problem ist bekannt als Cache-Stampede [36]. Sobald ein Webinhalt vollständig heruntergeladen wurde, kann erst ein Eintrag angelegt werden, der besagt, ob der Webinhalt für die peerunterstützte Verteilung geeignet ist und für wie lange. Erneute Anfragen zu Webinhalten können danach erst sofort beantwortet werden.
- Neben bösartigen Anfragen zum Koordinator können auch bösartige Webserver betrieben werden. Diese könnten beispielsweise unendlich große Antworten generieren oder versuchen die Verbindung möglichst lange offen zu halten (Slowloris-Angriff [37]) um die Ressourcen des Koordinators zu erschöpfen.
- Der Koordinator kann Anfragen an illegale Webinhalte stellen und diese herunterzuladen.

3.3.3 Auswahl der Peers

In Bezug auf die Auswahl der Peers für die peerunterstützte Verteilung bleibt der Optimierungsspielraum nahezu unbegrenzt [38]. Für BitTorrent-Tracker, die in der Peer-Auswahl dem Koordinator ähneln, wurde bereits über Traceroutes [39] versucht, die Auswahl zu verbessern, um kürzere Downloadzeiten zu erreichen [40] [41]. Dort wurden Peers aufgrund ihrer Hop-Anzahl [42] zueinander ausgewählt.

Im Web ist die Peer-Auswahl mit zusätzlichen Herausforderungen verbunden:

- Unterschiedliche Browser und Versionen zwischen den Peers, die sich auf die Performance auswirken können (vgl. Kapitel 4.2.2).
- Daten sollten primär sequentiell vom Anfang erhalten werden, um das Streamen zu ermöglichen (vgl. Kapitel 3.4.3).

- Viele Webseitenbesucher surfen über Mobilfunk, wodurch in vielen Fällen Datenverkehr direkte Kosten verursacht. Zusätzlich sind solche Peers häufig von Paketverlust, hohen Latenzzeiten und Jitter [43] betroffen, weshalb sie aussortiert werden sollten. Dies kann zum Beispiel aufgrund der IP-Adresse geschehen, indem bekannte Adressbereiche von Mobilfunkanbietern hinzugezogen werden. Außerdem kann im Browser die experimentelle Network-Information-Schnittstelle [44] verwendet werden, um den Verbindungstyp des Browsers festzustellen.

Weitere Eigenschaften der Peers, die zur Verbesserung der Auswahl hinzugezogen werden können, sind die Upload-Bandbreite, Latenzzeiten und Stabilität. Diese werden zum Beispiel in fybrrStream [45] hinzugezogen, um eine Punktzahl für Peers zu berechnen. Bei fybrrStream handelt es sich auch um eine WebRTC-Anwendung in einem ähnlichen Problemfeld, bei der Live-Video-Streams zwischen Peers geteilt werden.

Zusätzlich zu diesen Eigenschaften können bei der peerunterstützten Verteilung von Webseiteninhalten aufgrund des Surfverhaltens Verbindungen zu Peers im Vorfeld aufgebaut werden. Erkennt ein Koordinator, dass nach einer Anfrage immer eine weitere bestimmte Anfrage folgt, kann der Koordinator bereits Peers für die zweite Anfrage liefern, sobald ein Peer die erste Anfrage stellt.

Neben der Auswahl von Peers, spielt auch die Anzahl der zurückgegebenen Peers eine entscheidende Rolle. Denn lässt man einen Webinhalt durch eine Vielzahl von Peers auflösen, erhöht sich die Wahrscheinlichkeit, dass ein Peer während der Übertragung abspringt, wodurch es zu Verzögerungen kommen kann. Es besteht jedoch auch die Möglichkeit, dass mehrere Peers dieselben Teile eines Webinhalts übertragen. Dadurch entfällt im Falle eines Ausfalls eines Peers die Notwendigkeit für zusätzliche Verwaltungsmaßnahmen im Netzwerk.

3.4 Client-Anwendung

Die Anwendung, die die Browser ausführen, wird im gesamten als Client-Anwendung bezeichnet. Sie besteht aus zwei Teilen. Zum einem aus der JavaScript-Datei, die man auf einer Webseite einbindet. Dieses Webseitenskript wird im Rahmen dieser Masterarbeit als Clientskript bezeichnet. Und dem Service Worker, der über eine weitere JavaScript-Datei vom Clientskript eingebunden wird. Die Hauptaufgabe des Clientskripts ist es, den Service Worker im Browser zu registrieren und WebRTC-Verbindungen aufzubauen.

Im Service Worker wird eine WebSocket-Verbindung zum Koordinator aufgebaut. Die Aufgabe des Service Workers besteht neben dem Abfangen von HTTP-Anfragen auch als Relay, um Nachrichten zwischen Clientskript und Koordinator weiterzuleiten. Zum Beispiel leitet der Service Worker die WebRTC-Verbindungsinformationen vom Clientskript zum Koordinator weiter und umgekehrt. Die Einbindung der Client-Anwendung ist abwärtskompatibel, was bedeutet, dass die Auflösung von HTTP-Anfragen nicht verhindert wird, wenn keine Verbindung zum Koordinator oder zu Peers aufgebaut werden kann, wodurch HTTP-Anfragen weiterhin normal aufgelöst werden.

Wird die Client-Anwendung ausgeführt, werden ab diesem Zeitpunkt alle zwischenspeicherbaren Anfragen zu Webinhalten über die Cache-Schnittstelle [18] im Browser abgelegt, sodass auf diese später zugegriffen werden kann, damit sie mit anderen Peers geteilt werden können. Zusätzlich wird dem Koordinator mitgeteilt, welche Webinhalte im Cache abgelegt wurden. Von nun an kann es passieren, dass man vom Koordinator als Peer ausgewählt wird. In diesem Fall teilt der Koordinator Verbindungsinformationen eines Peers mit, der die Webinhalte aus dem Cache peerunterstützt auflösen möchte. Beide Peers versuchen daraufhin eine Direktverbindung über WebRTC zueinander aufzubauen. Nach erfolgreichem Aufbau, werden auf dem zuverlässigen und reihenfolgeerhaltenden WebRTC-Datenkanal die Webinhalte zwischen den Peers geteilt. Ist der Webinhalt vollständig übertragen, legt nach erfolgreicher Integritätsprüfung der anfragende Peer den Webinhalt selbst im Cache ab und meldet das dem Koordinator, sodass sich der Pool an Peers für diesen Webinhalt vergrößert. Der anfragende Peer hat nun den angefragten Webinhalt peerunterstützt heruntergeladen und kann diesen im Browser darstellen.

Bei erneuten Seitenaufrufen kann nun auf die abgelegten Webinhalte im Cache zur Beantwortung von Anfragen zugegriffen werden, denn diese werden von der Cache-Schnittstelle persistiert.

3.4.1 Zusammenspiel der Datenkanäle

Die Kommunikation zwischen zwei Peers erfolgt über zwei WebRTC-Datenkanäle. Der Grund dafür liegt an den zwei Arten von Nachrichten, die zwischen den Peers ausgetauscht werden. Zum einen werden Teile von Webinhalten übertragen und zum anderen Nachrichten, mit denen sich beide Peers abstimmen. Im Rahmen der Masterarbeit heißen diese Datenkanäle Webinhalt-Kanal und Kontroll-Kanal. Beide sind als zuverlässige

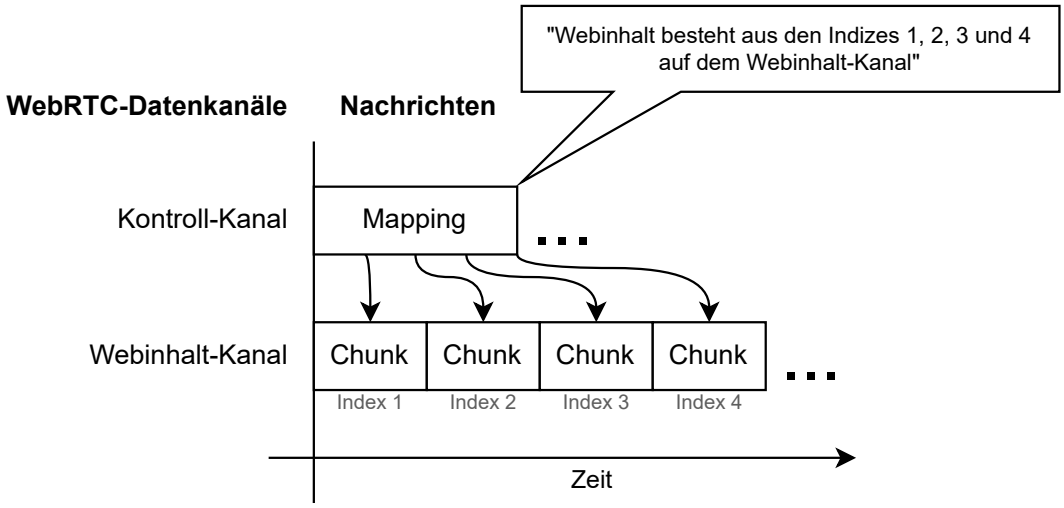

Abbildung 3.2: Mapping der Chunks auf dem Webinhalt-Kanal

Datenkanäle in der WebRTC-Schnittstelle konfiguriert. Die Verwendung von zwei Datenkanälen ist mit zwei Vorteilen verbunden:

- Während auf dem Kontroll-Kanal Nachrichten in einem bestimmten Datenformat, wie zum Beispiel JSON [46] ausgetauscht werden, können auf dem Webinhalt-Kanal die rohen Daten übertragen werden, ohne sie in ein Datenformat überführen zu müssen. Bei Verwendung von nur einem Datenkanal, muss jede Nachricht einem Datenformat entsprechen und kann je nach Format für zusätzlichen Overhead sorgen.
- Da die Datenkanäle auf SCTP basieren und SCTP Multistreaming unterstützt, kann durch die Verwendung mehrerer Datenkanäle Gebrauch davon gemacht werden. Das hat den Vorteil, dass zum Beispiel große Nachrichten auf dem Webinhalt-Kanal, die Nachrichten auf dem Kontroll-Kanal nicht blockieren, solange die große Nachricht noch übertragen wird (Head-of-Line-Blocking [47]). Allerdings ist diese Funktionalität in den Browsern noch nicht implementiert (vgl. Kapitel 3.4.6).

Da nun auf dem Webinhalt-Kanal keine Zusatzinformationen zu den Daten mitgeliefert werden, muss über den Kontroll-Kanal beschrieben werden, um was für Daten es sich auf dem Webinhalt-Kanal handelt. Anders als bei TCP [48] erhält man keinen Stream von Daten, sondern Nachrichten fester Größe auf dem Webinhalt-Kanal. Da die Nachrichten auch in der richtigen Reihenfolge eintreffen, kann man so ein Mapping erstellen,

welches aufgrund des Nachrichtenindex die Daten auf dem Webinhalt-Kanal dem Webinhalt zuordnet. Dieses Mapping erhält man von jedem Peer, der einem Daten auf dem Webinhalt-Kanal sendet. Abbildung 3.2 zeigt, wie die Nachrichten auf den WebRTC-Datenkanälen erhalten und über das Mapping zugeordnet werden.

3.4.2 Flusskontrolle

Die Datenübertragung in den WebRTC-Datenkanälen erfolgt mithilfe von SCTP [8]. SCTP ist ein nachrichtenorientiertes Protokoll, das ausschließlich mit festen Nachrichten arbeitet. Im Gegensatz dazu stehen streamorientierte Transportprotokolle wie TCP, bei denen ein Datenstrom in unterschiedlich großen Paketen kontinuierlich übertragen wird. Für die Übertragung von Webinhalten eignet sich ein streamorientiertes Protokoll, da die Daten in optimale Paketgrößen aufgeteilt werden können und der Empfänger diesen Datenstrom auf Byte-Ebene erhält. Dieser Unterschied spiegelt sich in der nachrichtenorientierten WebRTC-Schnittstelle der Datenkanäle wider. So kann beispielsweise kein Datenstrom des Webinhalts zur Übertragung durch den WebRTC-Datenkanal angegeben werden, sodass man von einer automatischen Flusskontrolle profitieren könnte. Stattdessen muss die Flusskontrolle und die Nachrichtengröße selber definiert und implementiert werden.

Die Wahl der Nachrichtengröße wird in Kapitel 3.4.6 genauer beschrieben. Für die Flusskontrolle kann man die `bufferedAmountLowThreshold`-Eigenschaft des Datenkanals verwenden. Gibt man für diese Eigenschaft einen Wert an, wird für den Datenkanal das `bufferedamountlow`-Ereignis ausgelöst, sobald die Bytegröße des Ausgabepuffers kleiner ist als der definierte Wert. Um einen Webinhalt nun so schnell wie möglich über den WebRTC-Datenkanal zu übertragen, legt man im Handler des Ereignisses kleinere Datenstücke des Webinhalts in den Ausgabepuffer, dass dieser mindestens die definierte Bytegröße der `bufferedAmountLowThreshold`-Eigenschaft erreicht, sodass das `bufferedamountlow`-Ereignis immer weiter ausgelöst wird, solange bis der Webinhalt übertragen wurde. Legt man stattdessen direkt den gesamten Webinhalt in den Ausgabepuffer, kann es passieren, dass der Browser diesen nicht annimmt und einen Fehler wirft.

3.4.3 Streamen von Webinhalten

Eine typische Strategie in P2P-Anwendungen ist es, zuerst Datenbereiche herunterzuladen, die nicht viele Peers besitzen. Dies ist als *rarest first order* bekannt. Dadurch erzielt man eine bessere Verteilung der Daten im Netzwerk und kann so den Download aller Peers verbessern.

Im Gegensatz dazu steht der Download über das Web. Hierbei werden Webinhalte in der Regel von Anfang an und in sequenzieller Reihenfolge heruntergeladen. Die Herausforderungen, denen P2P-Anwendungen beim Download gegenüberstehen, sind im Kontext des Webs weniger relevant.

Ein Vorteil der sequentiellen Datenübertragung ist das Streamen. Bestimmte Datenformate ermöglichen es, dass sie nach und nach dargestellt werden können, während sie noch übertragen werden.

Das Streamen von Webinhalten ist ein sehr wichtiges Feature, wenn es sich um Videos handelt. Diese möchte man schon während der Übertragung schauen können, noch bevor sie vollständig übertragen wurden. Insbesondere wenn es sich um große Videodateien handelt, bei denen der vollständige Download eine längere Zeit dauert.

Um diese Eigenschaften des Webs durch den Einsatz der peerunterstützten Verteilung von Webseiteninhalten nicht zu verändern, muss die Funktionsweise von P2P-Anwendungen und die des Webs miteinander vereint werden. Dazu muss die Download-Strategie dahingehend verändert werden, dass Datenbereiche möglichst sequentiell und womöglich von mehreren Peers heruntergeladen werden, um ein Streamen von Webinhalten zu erreichen. Ein Umstand, der nicht vereint werden kann, ist das Streamen einzelner Bytes. Während bei einem Webdownload die Daten gezielt von einem Server heruntergeladen werden, erhält man bei der peerunterstützten Verteilung Daten von willkürlichen Peers, die zuerst verifiziert werden müssen, wodurch einzelne Bytes zunächst nicht an den Browser weitergegeben werden. Stattdessen werden Webinhalte in Bereiche aufgeteilt zu denen Hashes für die Integritätsprüfung berechnet werden. Das bedeutet, dass die Daten nur in Stücken an den Browser weitergegeben werden und nicht in einzelnen Bytes. Die genaue Funktionsweise und Datenstruktur ist in Kapitel 3.6 beschrieben.

Implementierung in Browsern

Um das Streamen von Webinhalten mit der peerunterstützten Verteilung zu erreichen, ist die Verwendung einer Reihe von Schnittstellen notwendig. Oft wird die Browserfunktion `URL.createObjectURL` [49] verwendet, um Daten im Browser darzustellen. Diese Funktion erstellt mittels eines übergebenen Puffers eine spezielle lokale einzigartige URL, die beispielsweise als Ziel für ein Bild-Element angegeben werden kann, wodurch das Bild aufgrund des Puffers im Browser dargestellt wird. Eine solche URL besitzt folgende Struktur:

`blob:https://haw-hamburg.de/48810c85-dbca-4108-9263-1e469431c581`

Die Verwendung dieser Browser-Funktion ist mit zwei massiven Nachteilen verbunden: Zum einen muss die gesamte Datei als Puffer an den Browser übergeben werden, wodurch kein Streamen ermöglicht wird. Und zum anderen werden sämtliche peerunterstützt aufgelöste URLs von Elementen einer Webseite durch blob-URLs ersetzt. Webseitenbesucher, die sich dann zum Beispiel die URL eines Bilds anzeigen lassen wollen, erhalten nicht die originale URL, sondern die lokale blob-URL.

Beide Probleme lassen sich durch ein Zusammenspiel von Service Worker, Streams-Schnittstelle [50] und *transferable objects* [51] lösen. In Listing 3.3 ist ein minimalistisches Beispiel des relativ komplexen Prozesses im Service Worker gezeigt.

In der ersten Zeile wird ein Ereignishandler für das `fetch`-Ereignis registriert. Dieser Handler wird daraufhin für alle HTTP-Anfragen des Browsers ausgeführt. Um die Auflösung der Anfrage dem Browser abzunehmen – um diese peerunterstützt aufzulösen – muss im Handler `event.respondWith()` aufgerufen werden, mit einer Instanz der `Response`-Klasse. Diese `Response`-Klasse wird in Zeile 3 mit einem `ReadableStream` [50] instanziiert, der durch die `TransformStream`-Klasse in Zeile 2 erstellt wurde. Ein `TransformStream` besteht aus einem `ReadableStream` und einem `WritableStream`. Das bedeutet, dass man sowohl in diesen Stream schreiben, als auch von ihm lesen kann. Im Konstruktor des `TransformStream` wird normalerweise ein Handler definiert, der aufgerufen wird, wenn Daten in den Stream geschrieben werden. In diesem Fall wird kein Handler definiert, wodurch alle Daten unverändert durchgeleitet werden. Das dient dazu, um Daten an den Browser mittels des `ReadableStream` zu streamen, während in den `WritableStream` Daten der peerunterstützten Lösung geschrieben werden. Weiter wird in Zeile 10 der `WritableStream` an das Clientskript transferiert. Das bedeutet, dass der Service Worker diesen `WritableStream` nicht mehr

Listing 3.3: Abfangen von Anfragen im Service Worker mit Beantwortung der Anfrage über einen TransformStream mit gleichzeitiger Ablegung in den Cache und der Transferierung des WritableStream zum Clientskript

```
1  self.addEventListener('fetch', event => {
2    let stream = new TransformStream()
3    let response = new Response(stream.readable, {
4      headers: {
5        'content-type': 'image/jpeg',
6      },
7    })
8    event.respondWith(response.clone())
9    self.clients.get(event.clientId).then(client => {
10     client.postMessage(stream.writable, [stream.writable])
11   })
12   self.caches.open('p2pcache').then(cache => {
13     cache.put(event.request.url, response)
14   })
15 })
```

weiter verwenden kann. Diese Transferierung an das Clientskript ist notwendig, da der Datentransfer zwischen den Peers nicht im Kontext des Service Workers stattfindet, wodurch dieser keine Daten besitzt, die er in den `WritableStream` schreiben kann.

Neben der Beantwortung der abgefangenen HTTP-Anfrage durch `event.respondWith()` und der `Response`-Instanz, muss die Anfrage zusätzlich im Cache abgelegt werden (Zeile 13), damit auf diese im Nachhinein zugegriffen werden kann, um erneute Anfragen auf denselben Webinhalt schnell beantworten zu können oder den Webinhalt mit anderen Peers zu teilen. Aufgrund des `ReadableStream` der `Response`-Instanz kann eine solche Instanz jedoch nur einmal gelesen werden. Man kann also nicht nur eine einzelne `Response`-Instanz verwenden, um die Antwort sowohl im Cache abzulegen als auch dem Browser zur Verfügung zu stellen. Für solche Anwendungsfälle existiert die `clone`-Methode der `Response`-Klasse, wodurch mehrere `Response`-Instanzen geklont werden können, von denen jeweils gelesen werden kann. Es ist zu beachten, dass die Klone erstellt werden müssen, bevor vom `ReadableStream` gelesen wurde. Daher wird bereits in Zeile 8 ein Klon erstellt und in Zeile 13 erst die normale `Response`-Instanz verwendet.

Listing 3.4: Empfang und Schreiben in den vom Service Worker erhaltenden WritableStream im Clientskript.

```
1  let writer
2  navigator.serviceWorker.addEventListener('message', event => {
3    writer = event.data.getWriter()
4  })
5  peerConnection.dataChannel.addEventListener('message', event => {
6    writer.write(event.data)
7    if (done) {
8      writer.close()
9    }
10 })
```

Damit Browser Anfragen auch gestreamt darstellen, muss zusätzlich im Konstruktor der `Response`-Klasse noch der richtige MIME-Typ [52] angegeben werden (Zeile 5). In diesem Beispiel wurde dieser fest auf `image/jpeg` gesetzt. Da dem Service Worker der MIME-Typ des angefragten Webinhalts im Vorfeld nicht bekannt ist, muss der Koordinator diese Information bereitstellen. Andernfalls lässt sich über eine HEAD-Anfrage der MIME-Typ eines Webinhalts herausfinden.

Listing 3.4 zeigt nun die beispielhafte Gegenstelle im Clientskript. In der zweiten Zeile wird ein Ereignishandler registriert, der auf Nachrichten vom Service Worker horcht. Dort erhält das Clientskript exklusiven Zugriff auf den durchgereichten `WritableStream`, der mit dem `ReadableStream` im Service Worker verbunden ist. Die Zeilen 5 bis 10 zeigen einen Ereignishandler von Peer-Nachrichten auf dem Webinhalt-Kanal. In diesem Handler werden die erhaltenen Daten der Peers auf ihre Integrität überprüft und anschließend in Zeile 6 in den `WritableStream` geschrieben. Diese gestreamten Daten sorgen nun dafür, dass die abgefangene Anfrage im Service Worker nach und nach beantwortet und zusätzlich im Cache abgelegt wird.

3.4.4 Besondere Anfragen

Nicht alle HTTP-Anfragen, die vom Browser gestellt werden, sind einfache GET-Anfragen. Neben anderen HTTP-Methoden, die alle vom Service Worker zum Ursprungsserver durchgelassen werden, existieren besondere GET-Anfragen.

Range-Anfragen GET-Anfragen können Kopfdaten enthalten, die nur bestimmte Bereiche eines Webinhalts anfordern. Solche Anfragen können zum Beispiel durch JavaScript-Anwendungen oder vom Browser selbst gestellt werden. Das ist unter anderem der Fall, wenn eine Videodatei abgespielt und in dieser gespult wird. Die Unterstützung dieser Funktion müssen Webserver zuvor über Kopfdaten in ihren HTTP-Antworten signalisieren.

Die Auflösung solcher Anfragen gestaltet sich für die peerunterstützte Verteilung schwierig. Zusätzlich muss dem Koordinator zur URL auch der angeforderten Datenbereiche mitgeteilt werden. Peers, die der Koordinator verwaltet, müssen dann nicht nur pro URL organisiert werden, sondern auch pro Datenbereich, den sie besitzen. Das kann zu Peers führen, die nur Teile des Datenbereichs liefern können. Weitere Änderungen sind außerdem an den Protokollen zwischen den Peers und dem Koordinator nötig, um die Range-Anfragen zu unterstützen.

Es kann außerdem dazu führen, dass ein größerer Datenbereich als angefordert heruntergeladen werden muss, falls der Datenbereich nicht in die Grenzen eines Pieces fällt (vgl. Kapitel 3.6). Denn die Webseiteninhalte werden vom Koordinator in feste Pieces unterteilt und nur zu diesen werden Hashes berechnet. Fällt ein Datenbereich also in den Bereich eines Pieces, muss dieser vollständig heruntergeladen werden. Nur so lässt sich der Hash vom Client berechnen, um die Integrität überprüfen zu können.

Cross-origin Webinhalte Ein Service Worker wird immer für einen bestimmten Geltungsbereich registriert, für den er verantwortlich ist und Anfragen abfangen kann. Allerdings wird nicht jede Anfrage zum Service Worker geleitet, die in seinen Geltungsbereich fällt. Denn der Geltungsbereich bezieht sich nur auf den Ursprung der Anfrage und nicht auf das Ziel. Ist zum Beispiel in einem Browser ein Service Worker für `https://www.haw-hamburg.de` registriert und der Browser befindet sich auf `https://example.com`, wo zum Beispiel eine Anfrage aufgrund einer Einbindung eines Bilds zu `https://www.haw-hamburg.de/bild.jpg` ausgelöst wird, dann wird dem Service Worker diese Anfrage nicht mitgeteilt. Andersrum werden also Anfragen zu `https://example.com/bild.jpg` vom Service Worker abgefangen, wenn man sich auf `https://www.haw-hamburg.de` befindet und für diese Origin der Service Worker registriert ist.

Für den Service Worker bedeutet das also, dass dieser auch Anfragen zu anderen Origins abfangen kann (cross-origin). Das peerunterstützte Auflösen solcher Anfragen stellt kein

Problem dar. Allerdings kann die peerunterstützte Verteilung in solchen Fällen zu Problemen führen. Denn wird diese Anfrage peerunterstützt aufgelöst, erhält der Ursprungsserver keine Anfrage des Browsers, wodurch womögliche Erwartungen des Webseitenbetreibers nicht erfüllt werden, falls die Anfrage beispielsweise für Tracking verwendet wird.

3.4.5 Konfigurationsmöglichkeiten

Das Verhalten der peerunterstützten Verteilung lässt sich vom Webseitenbetreiber zu einem gewissen Maße einstellen. Hauptsächlich kann man festlegen, wie viel einer Webseite peerunterstützt verteilt werden soll. Sowohl im Koordinator als auch in der Client-Anwendung kann das Verhalten angepasst werden. Eine Reihe von Einstellungen, die in der Client-Anwendung angepasst werden können, sind:

- Die peerunterstützte Auflösung nur für bestimmte URLs zu erlauben. Beispielsweise alle URLs, die das Präfix `https://www.haw-hamburg.de/images/` besitzen.
- Eine Sperrliste von URLs, die nicht peerunterstützt aufgelöst werden sollen.
- URLs anderer Origins nicht peerunterstützt aufzulösen.
- Noch während die peerunterstützte Verteilung initialisiert wird, die ersten Bytes des Webinhalts normal vom Ursprungsserver herunterladen.

Einstellungen, die im Koordinator vorgenommen werden können, sind:

- Webinhalte erst peerunterstützt zu verteilen, wenn eine Anzahl von Peers diesen Webinhalt besitzen oder geeignet genug sind (vgl. Kapitel 3.3.3 und 3.8.1 für mögliche Probleme).
- Erst ab einer bestimmten Dateigröße des Webinhalts, diesen peerunterstützt zu verteilen.

3.4.6 Browserunterschiede

SCTP-Nachrichtengröße Die maximale Nachrichtengröße und deren Ermittlung war lange Zeit ein Problem für die Kommunikation auf dem WebRTC-Datenkanal. Aufgrund von Implementierungsunterschieden in den Browsern, galt 16 KiB als sichere maximale

Nachrichtengröße [53]. Der Grund dafür liegt in der Fragmentierung von Nachrichten und dass Nachrichten komplett gepuffert werden müssen. Eine Fragmentierung tritt auf, wenn eine Nachricht nicht in ein einziges SCTP-Paket passt, welches durch die Größe der PMTU [54] beschränkt ist. Diese Fragmentierung ist auch der Grund, wieso SCTP und die implementierenden Browser immer noch mit dem Problem des Head-of-Line-Blocking [47] zu kämpfen haben. Denn SCTP verhindert nur Head-of-Line-Blocking durch mehrere Streams, wenn die Nachrichten nicht fragmentiert sind. Das bedeutet, dass eine einzige große Nachricht alle Streams von SCTP blockieren kann. Die Browserentwickler haben daher keine Anstrengungen unternommen, die maximale Nachrichtengröße immer weiter zu erhöhen. Zur Lösung des Problems wurde eine Erweiterung für SCTP spezifiziert, die das Head-of-Line-Blocking aufgrund der fragmentierten Nachrichten verhindert [55]. Momentan wird diese Erweiterung in den Browsern implementiert [56] [57], die bis zur Fertigstellung weitere Probleme blockiert, wie zum Beispiel dass keine Nachrichten vom Datentyp `Blob` in Google Chrome versendet werden können [58].

Die Festlegung einer maximalen Nachrichtengröße des WebRTC-Datenkanals war deswegen so ein Problem, weil nicht ermittelt werden konnte, welche Nachrichtengrößen ein Peer unterstützt. Seit Version 113 vom 9. Mai 2023 ermöglicht Mozilla Firefox neben Google Chrome die Abfrage der maximalen Nachrichtengröße [59], die einige Zeit zuvor auch aus dem SDP gelesen werden konnte (siehe `max-message-size`-Attribut in Listing 2.1).

Lebensdauer Service Worker Service Worker werden je nach Browser unterschiedlich lange ausgeführt. So beendet Firefox Service Worker nach 30 Sekunden Inaktivität. In Google Chrome wird der Service Worker solange nicht beendet, bis alle Tabs für die der Service Worker zuständig ist, geschlossen sind. Das Verhalten von Firefox stellt für die peerunterstützte Verteilung ein Problem dar, weil die Verbindung zum Koordinator aus Performancegründen (vgl. Kapitel 3.5.2) im Service Worker aufgebaut wird und die Verbindung somit nach kurzer Zeit getrennt wird. Der Koordinator kann dadurch den Browser nicht mehr kontaktieren, wodurch ein möglicher Peer, der für die peerunterstützte Verteilung genutzt werden könnte, wegfällt. Ein Workaround besteht darin, den Verbindungsaufbau zum Koordinator in das Clientskript zu verlagern. Oder man versucht den Service Worker im Firefox aktiv zu halten. Dies lässt sich zum Beispiel durch das ständige Versenden von Nachrichten vom Clientskript zum Service Worker erreichen.

Datentypen Laut Spezifikation der WebRTC-Datenkanäle [3] entspricht der Standardwert für die `binaryType`-Eigenschaft des Datenkanals den Wert `blob`, wodurch die empfangenen Nachrichten den Datentyp `Blob` [49] aufweisen und nur dieser Datentyp zum Versenden genutzt werden kann. Alle auf Chromium [60] basierenden Browser, wie Google Chrome, weichen von diesem Standard ab [58]. Da Chromium den Datentyp `Blob` für die WebRTC-Datenkanäle nicht unterstützt, wird als Standardwert `arraybuffer` verwendet, wodurch die Nachrichten dem Datentyp `ArrayBuffer` [61] entsprechen. Firefox und Safari unterstützen beide Datentypen. Um die Kompatibilität zwischen den Browsern zu gewährleisten, kann die `binaryType`-Eigenschaft explizit auf `arraybuffer` gesetzt werden.

In Kapitel 4.2.4 wird untersucht, wie sich der Datentyp auf die Übertragungsrate im Firefox auswirkt.

3.4.7 Hürden in der Browserumgebung

Die Entwicklung der Client-Anwendung ist aufgrund der limitierenden Browserumgebung mit vielen Hürden verbunden.

Deaktivierte Schnittstellen in unsicheren Umgebungen Wird eine Webseite unverschlüsselt über HTTP aufgerufen, gilt sie als unsichere Umgebung. Browser deaktivieren den Zugriff auf die Web-Kryptographie-Schnittstelle [62] in solchen Umgebungen. Die Schnittstelle wird benötigt, um Hashes über die von den Peers erhaltenen Daten zu berechnen. Zur Prüfung der Integrität werden diese mit den Hashes vom Koordinator abgeglichen. Die Deaktivierung ist damit begründet, dass die Verwendung von Kryptographie in einer unsicheren Umgebung niemals zu mehr Sicherheit führen kann [63]. Dieser Schlussfolgerung wird zugestimmt, allerdings nur solange es sich um denselben Kommunikationskanal handelt, der abgesichert werden soll. Im Gegensatz dazu werden im Rahmen der peerunterstützten Verteilung jedoch zusätzliche Kommunikationskanäle zu Peers geöffnet. Diese zusätzlichen Kommunikationskanäle könnten kryptographisch abgesichert werden. Während der Angriffsvektor der unsicheren Umgebung auf einen Man-in-the-Middle-Angriff [64] beruht, können Peers ohne weiteres falsche Daten senden. Diesen Anwendungsfall konnte ehemaligen Google-Mitarbeitern, die sich für die Deaktivierung ausgesprochen haben, allerdings nicht klar gemacht werden [65].

Alternativ kann auch im Falle einer unsicheren Umgebung der Algorithmus zur Berechnung der Hashes in der Client-Anwendung mitgeliefert werden, sodass die Schnittstelle nicht benötigt wird.

Eine weitere Schnittstelle, die in unsicheren Umgebungen deaktiviert wird, ist die der Service Worker. In diesem Fall funktioniert die vorgestellte peerunterstützte Verteilung nicht, da HTTP-Anfragen nicht abgefangen werden können. Allerdings kann die WebRTC-Schnittstelle immer noch verwendet werden, um Daten zwischen Peers auszutauschen [66]. In Umgebungen, wo HTTP-Anfragen selbst durch eine Client-Anwendung gestellt werden, kann die peerunterstützte Verteilung daher immer noch genutzt werden. Dies ist zum Beispiel bei Web-Videoplayern der Fall, die Videostücke selbst herunterladen.

Deaktivierte Service Worker In manchen Umständen erlauben Browser keine Registrierung eines Service Workers. Dies ist unter anderem beim Firefox der Fall, wenn man in einem privaten Fenster surft. In Google Chrome hingegen, lassen sich Service Worker auch im Inkognito-Modus registrieren. Nur wenn man alle Cookies blockiert, verhindert Google Chrome die Registrierung eines Service Workers.

Kein Random Access auf Webinhalte Webseiteninhalte werden über die Cache-Schnittstelle [18] im Browser abgelegt. Möchte man auf diese Inhalte wieder zugreifen, um sie anderen Peers zur Verfügung zu stellen, können die Inhalte nur im gesamten oder sequentiell vom Anfang gelesen werden. Handelt es sich um große Webinhalte, kann das Lesen im gesamten den Arbeitsspeicher eines Browsers sehr beanspruchen, was sich negativ auf die Performance auswirken kann. Das sequentielle Lesen ist daher vorzuziehen, da immer nur Teile eines Webinhalts zu einem Zeitpunkt im Arbeitsspeicher gehalten werden.

Erhält man aber von mehreren Peers einen Webseiteninhalt, übermitteln diese verschiedene Teile davon. Beispielsweise übermittelt der erste Peer die erste Hälfte des Webinhalts und der zweite Peer die zweite Hälfte. In diesem Fall vermisst der zweite Peer einen Direktzugriff auf die Daten, um nur die zweite Hälfte lesen zu können. Aufgrund der limitierenden Browserumgebung muss entweder die erste Hälfte beim sequentiellen Lesen verworfen werden oder der gesamte Webinhalt in den Arbeitsspeicher geladen werden.

Fehlende Schnittstellen im Service Worker Das größte Problem der limitierenden Browserumgebung ist die fehlende WebRTC-Schnittstelle im Kontext eines Service Workers [67]. Abgefangene HTTP-Anfragen möchte man im Kontext des Service Workers peerunterstützt auflösen. Aufgrund der fehlenden WebRTC-Schnittstelle, werden als Workaround die WebRTC-Verbindungen stattdessen im Clientskript aufgebaut. Dadurch entsteht eine Abhängigkeit zum Clientskript, die zu einigen Problemen führt. Abgefangene HTTP-Anfragen können beispielsweise welche zum Clientskript selbst sein, wodurch diese eigene Abhängigkeit blockiert werden kann. Im folgenden Kapitel 3.5 werden alle daraus resultierenden Probleme genauer beschrieben.

3.5 Herausforderungen der fehlenden WebRTC-Schnittstelle

3.5.1 Zuordnung abgefangener Anfragen

Ein besonderes Problem tritt auf, wenn HTTP-Anfragen zum Clientskript selbst oder zum HTML-Dokument einer Webseite von einem bereits registrierten Service Worker abgefangen werden. Da Service Worker nicht an die Lebensdauer eines Browsertabs gebunden sind und nach Verlassen oder vor Besuch einer Webseite noch ausgeführt werden können, kann es passieren, dass Anfragen abgefangen werden, die (noch) nicht peerunterstützt Verteilt werden können, weil gerade kein Clientskript ausgeführt wird. Die notwendigen WebRTC-Verbindungen für die peerunterstützte Verteilung können in der Umgebung eines Service Workers nicht aufgebaut werden. Daher wird das Clientskript benötigt, um diese Schnittstellen bereitzustellen. Da alle abgefangenen Anfragen an den Koordinator weitergeleitet werden, kann es passieren, dass der Koordinator entscheidet, dass die Anfrage zum Clientskript für die peerunterstützte Verteilung geeignet ist. Dies sollte auch der Fall sein, da es sich beim Clientskript um ein statisches und zwischenspeicherbares Skript handelt. Tritt dieser Fall ein, versucht der Service Worker das Clientskript anzuweisen, die Anfrage – zum Clientskript – peerunterstützt herunterzuladen. Jedoch ist das Clientskript noch nicht ausgeführt und kann dem Service Worker daher nicht antworten. Abbildung 3.3 zeigt diesen Ablauf von Entscheidungen, unter welchen Bedingungen Anfragen peerunterstützt aufgelöst werden können.

Die Lösung dieses Problems besteht darin zu erkennen, um was es sich bei den abgefangenen HTTP-Anfragen handelt. Dazu existieren verschiedene Möglichkeiten:

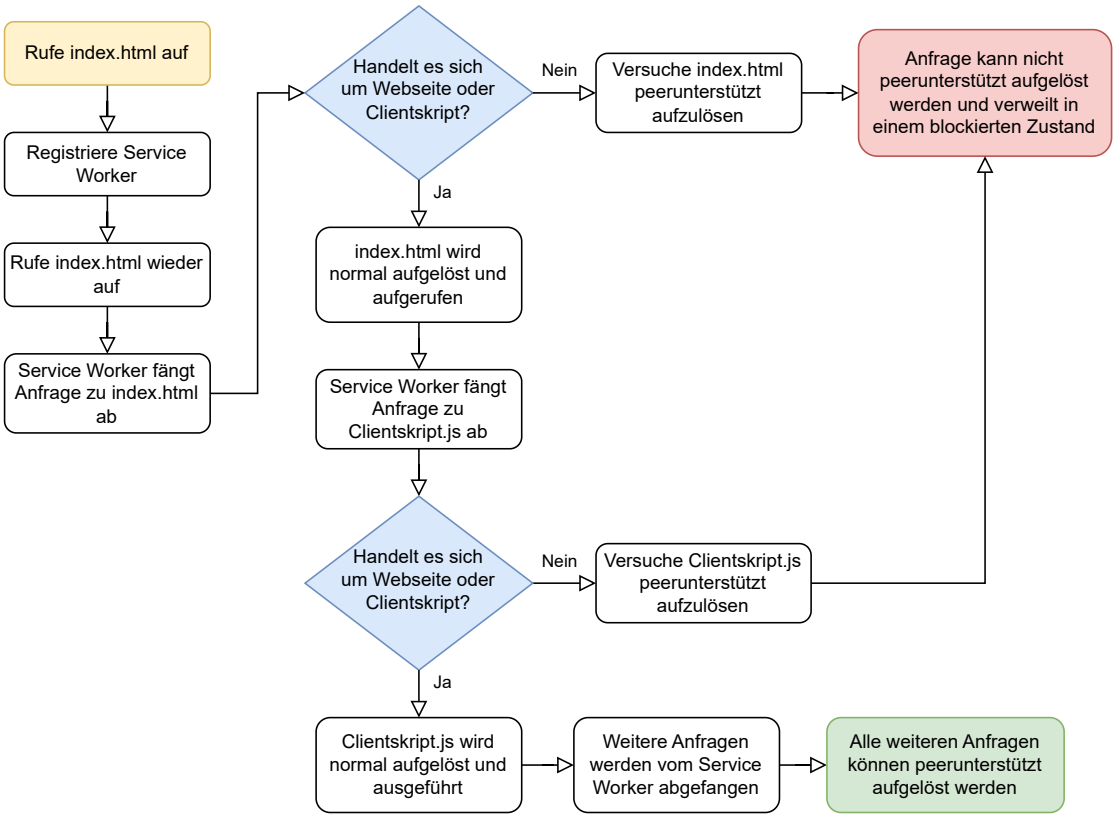

Abbildung 3.3: Ablaufdiagramm von Entscheidungen ob Anfragen peerunterstützt aufgelöst werden können

Listing 3.5: Wiedererkennung der Anfrage zum Clientskript aufgrund integrity-Attribut und spezieller Query-Parameter in der URL

```
1 <script async src="clientskript.js?tcDTfY3OsXB7"  integri-
  ↪  ty="sha256-2r4OI/yLDPUngnTUrrF7RPa9P85a2QHfK9k99C/V/xI=">
2 </script>
```

- Im Ereignishandler zum Abfangen von HTTP-Anfragen erhält der Service Worker ein Objekt, das Eigenschaften zur Anfrage enthält. Unter anderem die URL. Über diese kann man Informationen über die Anfrage enkodieren. Beispielsweise kann man einen speziellen Query-Parameter zur URL zum Clientskript hinzufügen. Wie das aussehen könnte ist in Listing 3.5 gezeigt. Im Service Worker kann nun die abgefangene Anfrage dahingehend überprüft werden, ob der in diesem Fall spezielle Query-Parameter `tcDTfY3OsXB7` existiert. Falls er existiert, wird die Anfrage nicht an den Koordinator weitergeleitet, sondern normal aufgelöst. Dem Browser steht nun nichts mehr im Weg das Clientskript auszuführen und so, von nun an die peerunterstützte Verteilung zu ermöglichen.
- Eine weitere Methode besteht darin das `integrity`-Attribut des `script`-Elements zu verwenden. Normalerweise wird dadurch sichergestellt, dass das zu ladende Skript denselben Hashwert besitzt wie der im `integrity`-Attribut angegebene Wert. Dadurch kann verhindert werden, dass Skripts ausgeführt werden, die verändert wurden. Im Service Worker lässt sich dies nun verwenden, da man zur abgefangenen Abfrage nämlich noch den Wert des `integrity`-Attributs erhält. Listing 3.5 zeigt neben dem Query-Parameter auch die Verwendung des `integrity`-Attributs zur Erkennung der Clientskripts. Hinterlegt man nun den Hashwert des Clientskripts im Service Worker und gibt ihn im `integrity`-Attribut an, kann beim Abfangen der Anfragen abgeglichen werden, ob das `integrity`-Attribut diesem Hashwert entspricht und so die Weiterleitung zum Koordinator verhindern.
- Das Clientskript kann von einem zentralen Punkt geladen werden. Die darauf zeigende URL wird im Service Worker hinterlegt und mit den URLs der abgefangenen Anfragen abgeglichen.

Man könnte meinen, dass sich das Problem auch durch die Signalisierung der Lebensdauer des Clientskripts lösen lässt. Zum Beispiel indem das Clientskript dem Service Worker

eine Nachricht sendet, sobald es initialisiert wurde. In der Zwischenzeit würde der Service Worker alle abgefangenen HTTP-Anfragen zurückhalten, bis sich das Clientskript meldet. Allerdings steht man auch hier wieder vor demselben Problem: Eines von den abgefangenen Anfragen ist die zum Clientskript; und dieses wird so lange zurückgehalten bis sich das Clientskript meldet. Auch hier würde das Clientskript niemals ausgeführt werden und kann sich daher nicht melden.

Wie anfangs erwähnt betrifft das Problem nicht nur die HTTP-Anfrage zum Clientskript, sondern auch die Anfrage zum HTML-Dokument, welches das Clientskript einbindet. Es handelt sich um dasselbe Problem. Weißt der Koordinator an die Anfrage zum HTML-Dokument peerunterstützt aufzulösen, wird die Ausführung des notwendigen Clientskripts verhindert und blockiert somit wieder alle Anfragen. Das bedeutet, dass die Anfrage zum HTML-Dokument auch erkannt werden muss, damit diese normal aufgelöst werden kann. Für diesen Fall existiert glücklicherweise das `mode`-Attribut im Objekt der abgefangenen HTTP-Anfrage. Enthält es den Wert `navigation`, bedeutet dies, dass für diese HTTP-Anfrage der Browser angewiesen wurde zu einer Webseite zu navigieren. Es handelt sich also um keine Anfrage, die aufgrund einer Einbindung in einem HTML-Dokument geschehen ist. Mithilfe dieser Information kann man nun solche Anfragen immer normal auflösen. Handelt es sich nämlich bei der Anfrage – ausgelöst durch das Navigieren – um ein HTML-Dokument, muss dieses sowieso normal aufgelöst werden. Und falls es sich nicht um ein HTML-Dokument handelt, sondern beispielsweise um ein Bild, welches direkt aufgerufen wurde, dann kann dies sowieso nicht peerunterstützt aufgelöst werden, da kein Clientskript ausgeführt wird.

Das Problem, im Service Worker zu entscheiden, welche HTTP-Anfragen normal aufgelöst und welche zum Koordinator weitergeleitet werden sollen, wird jedoch immer noch nicht gelöst, selbst wenn man die Anfragen zum Clientskript und den HTML-Dokumenten erkennt. Denn ist zum Beispiel ein Service Worker bereits registriert und man ruft ein HTML-Dokument der Webseite auf, welches das Clientskript nicht einbindet, dann hält der Service Worker alle Anfragen zurück, solange bis sich das Clientskript meldet. Dies kann auch passieren, wenn die Ausführung des Clientskripts durch den Browser blockiert wird, etwa durch Werbeblocker wie uBlock Origin [68].

Die Lösung besteht aus der Kombination beider Methoden. Zum einen das Durchlassen der Clientskript- und Navigations-Anfragen, als auch die Hinzuziehung der Lebensdauer des Clientskripts. Das Durchlassen ermöglicht die peerunterstützte Verteilung, während die Betrachtung der Lebensdauer des Clientskripts die Fälle berücksichtigt, in denen das

Clientskript aus diversen Gründen nicht ausgeführt wird. Da man allerdings nichts über die womöglich zukünftige Ausführung des Clientskripts sagen kann, muss mit Timeouts gearbeitet werden, wonach alle zurückgehaltenen HTTP-Anfragen normal aufgelöst werden.

Dieses Problem besteht also nur aufgrund der fehlenden WebRTC-Schnittstelle in der Umgebung des Service Workers und des daraus resultierenden Workarounds über das Clientskript.

3.5.2 Erkennung aktiver Peers

Eine weitere Folge der fehlenden WebRTC-Schnittstelle im Service Worker ist das Problem aktive Peers zu erkennen. Das bedeutet aus Sicht des Koordinators: Hinter welchen WebSocket-Verbindungen, die zu den Browsern gehalten werden, steckt ein aktives Clientskript, das für die peerunterstützte Verteilung genutzt werden kann? Denn wie zuvor erläutert, werden Service Worker und Clientskript nicht immer zur selben Zeit ausgeführt und aus Performancegründen wird die WebSocket-Verbindung nicht im Clientskript aufgebaut, sondern im Service Worker. Daher ist die Etablierung einer WebSocket-Verbindung kein Garant dafür, dass ein Clientskript aktiv ist.

Aber nicht nur das Fehlen eines aktiven Clientskripts ist ein Problem, sondern auch wenn das Clientskript neu geladen wird und die WebSocket-Verbindung zum Koordinator bestehen bleibt. Denn aufgrund des Neuladens sind die dem Koordinator mitgeteilten WebRTC-Verbindungsinformationen nicht mehr gültig und können für die peerunterstützte Verteilung nicht mehr verwendet werden. Daher müssen die Verbindungsinformationen pro Clientskript-Instanz im Service Worker organisiert werden, was auch dem Koordinator mitgeteilt werden muss. Dafür kann beispielsweise die `clientId`-Eigenschaft verwendet werden. Diese wird unter anderem in Ereignissen mitgeliefert, wenn das Clientskript mit dem Service Worker kommuniziert.

Um das Wegfallen eines aktiven Clientskripts festzustellen, kann es auf das `unload`-Ereignis einer Webseite einen Handler registrieren. Dieses Ereignis wird in vielen Fällen vom Browser ausgelöst, wenn die Webseite geschlossen wird. Im Handler kann man dann dem Service Worker mitteilen, dass die WebRTC-Verbindungsinformationen dieser Clientskript-Instanz nicht mehr gültig sind.

Leider ist diese Lösung mit zwei Problemen behaftet: Zum einen wird dieses Ereignis nicht immer vom Browser ausgelöst und zum anderen verhindert allein die Registrierung auf dieses Ereignis die Verwendung des bfcache [69]. Dies ist eine Browseroptimierung, die beim Navigieren Schnappschüsse anlegt, sodass beim Zurücknavigieren vorherige Webseiten sofort dargestellt werden können.

Die Erkennung aktiver Peers bleibt daher weiterhin ein Problem, dem durch ständige Heartbeat-Nachrichten entgegengewirkt werden kann.

Performancegründe Die Performancegründe für den Verbindungsaufbau zum Koordinator im Service Worker beruhen auf den wiederholten Verbindungsaufbau zum Koordinator, der mit jedem Webseitenaufruf einhergeht, wenn die WebSocket-Verbindung im Clientskript aufgebaut wird. Denn beim Surfen auf einer Webseite wird das Clientskript ständig beendet und erneut ausgeführt. Dadurch werden alle Verbindungen kurzzeitig getrennt. Der Service Worker ist von diesen Webseitenaufrufen nicht betroffen und kann die Verbindung aufrechterhalten. So werden Paketumlaufzeiten eingespart, wodurch der Koordinator schneller kontaktiert werden kann, um die peerunterstützte Verteilung durchzuführen. Außerdem wird immer nur eine Instanz des Service Workers ausgeführt, egal in wie vielen Browsertabs die Webseite geöffnet ist. Dadurch wird immer nur eine Verbindung zum Koordinator aufgebaut. Andernfalls kann mit einem Shared Worker [70] gearbeitet werden, um nur eine Verbindung aufzubauen, falls der Verbindungsaufbau nicht im Service Worker stattfinden soll und die Webseite mehrfach geöffnet ist.

Verbindungsaufbau im Clientskript Verlagert man den Verbindungsaufbau zum Koordinator in das Clientskript, wird die Verbindung mit jedem Seitenaufruf getrennt. sondern auch die WebRTC-Verbindungen zu den Peers. Ein weiteres Problem, geschuldet der fehlenden WebRTC-Schnittstelle im Service Worker. Dadurch ist mit erhöhten Auflösungszeiten der peerunterstützten Verteilung bei jedem Webseitenaufruf zu rechnen. Verhindert man das Neuladen der Webseite beim Surfen, zum Beispiel durch Verwendung einer Single-Page-Webanwendung (SPA) [71], können die WebRTC-Verbindungen aufrechterhalten werden und erreicht niedrigere Auflösungszeiten.

3.6 Datenstruktur

Aufgrund der Funktionsweise der peerunterstützten Verteilung müssen die Daten eines Webseiteninhalts genauer aufgeteilt werden. Empfängt man Daten von Peers, möchte man überprüfen können, ob diese auch wirklich dem Original entsprechen. Dazu wird jeder Webinhalt in sogenannte Pieces unterteilt. Dieser Begriff stammt vom prägenden Filesharing-Protokoll BitTorrent. Pieces sind anliegende Datenbereiche fester Größe und im Rahmen der Masterarbeit 256 KiB groß, wobei das letzte Piece womöglich kleiner sein kann. Diese Datenstruktur ist in Abbildung 3.4 dargestellt.

Sowohl das Clientskript als auch der Koordinator arbeiten mit dieser Einteilung in Pieces. Der Koordinator berechnet zu jedem Piece einen Hash und besitzt so zu jedem Webinhalt eine Liste von Hashes. Diese Hashes werden anfragenden Client-Anwendungen mitgeteilt. Durch die feinere Unterteilung und der Hash-Liste kann ein Client nun überprüfen, ob die erhaltenen Pieces von seinen Peers dem Original entsprechen, indem der Client auch über diese einen Hash berechnet und mit dem jeweiligen Hash vom Koordinator abgleicht.

In so einer Unterteilung ist es sinnvoll, dass jedes Piece immer nur von einem Peer empfangen wird. So kann im Fehlerfall, falls die Hashes nicht übereinstimmen, die Verbindung zu diesem Peer getrennt werden, da dieser offensichtlich falsche Daten übermittelt hat und kann versuchen dieses Piece von einem anderen Peer zu beziehen. Erhält man ein Piece von mehreren Peers, kann bei diesem Umstand nicht mehr eindeutig ermittelt werden, welcher Peer falsche Daten übermittelt hat.

Bei besonders großen Webinhalten im Gigabyte-Bereich und je nach gewählter kryptographischer Hashfunktion können Hash-Listen relativ groß werden. Beispielsweise beträgt die Größe einer Hash-Liste 80 KiB bei einer 1 GiB großen Datei mit einer Piece-Größe von 256 KiB und einer Hash-Größe von 20 B, wie sie zum Beispiel von SHA-1 [72] erzeugt wird. Da die meisten Nachrichten vom und zum Koordinator weniger als 1 KiB betragen, können diese großen Listen die Bandbreite des Koordinators erschöpfen. Um dieses Problem zu entschärfen, kann mit Hash-Bäumen gearbeitet werden. Dazu wird zusätzlich zur Hash-Liste ein Hash-Baum erzeugt, indem über mehrere Hashes aus der Hash-Liste ein einzelner weiterer Hash berechnet wird. Dies wird solange wiederholt, bis jeder Hash in der Liste zusammengefasst wurde. Das Resultat ist eine weitere Liste, die jedoch kürzer ist, je nachdem wie viele Hashes zusammengefasst wurden. Diese Liste wird wiederum zusammengefasst, so oft bis die resultierende Liste nur noch aus einem Hash besteht, dem Wurzel-Hash. Abbildung 3.4 zeigt die Berechnung eines Hash-Baums, bei

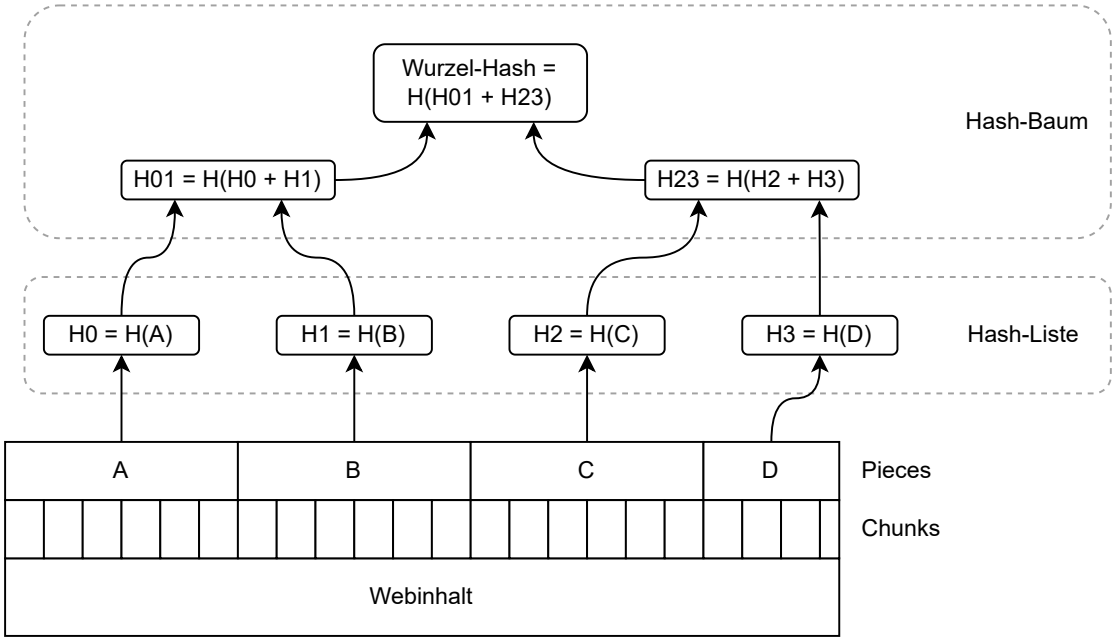

Abbildung 3.4: Aufteilung von Webinhalten und Berechnung des Hash-Baums

dem jeweils immer zwei Hashes zusammengefasst werden. Dieser sogenannte vollständige Binärbaum benötigt zusätzlich $n - 1$ Hashes, wobei n für die Anzahl der Pieces steht. Es können zum Beispiel auch jeweils drei Hashes zusammengefasst werden. Ein solcher Baum benötigt dann nur noch $(n - 1)/2$ zusätzliche Hashes, fasst aber größere Bereiche zusammen, wodurch falsche Daten ungenauer lokalisiert werden können.

Der Vorteil des Hash-Baums besteht nun darin, dass anfragenden Clients nicht immer die gesamte Hash-Liste mitgeteilt werden muss. Es reicht zunächst nur der Wurzel-Hash. Analog dazu berechnet ein Client auch diesen Hash-Baum basierend auf den empfangenen Daten seiner Peers. Der Client gleicht beide Wurzel-Hashes miteinander ab. Falls sie nicht übereinstimmen, können weitere Teile des Baums vom Koordinator angefordert werden, um die falschen Daten genauer lokalisieren zu können. Somit kann die Datenmenge reduziert werden, die der Koordinator den Clients, aufgrund der Hashes, übertragen muss.

Das Streamen von Webinhalten (vgl. Kapitel 3.4.3) kann von dieser Optimierung leider nicht profitieren, da die Integrität jedes Piece sofort überprüft werden muss, da der Webinhalt nicht vollständig, sondern nach und nach an den Browser übergeben wird.

In Abbildung 3.4 ist eine weitere Unterteilung der Pieces in Chunks zu sehen. Aufgrund von Problemen der maximalen Nachrichtengröße, die Browser auf dem WebRTC-Datenkanal versenden und empfangen können (vgl. Kapitel 3.4.6), kann nicht immer mit Nachrichten gearbeitet werden, die der Piece-Größe entsprechen. Daher werden die Pieces in noch kleinere Chunks unterteilt, um dem Problem entgegenzuwirken. Auch hier ist es möglich, dass der letzte Chunk kleiner ausfällt, falls der Webseiteninhalt nicht durch die Chunk-Größe teilbar ist. Die Größe der Chunks hängt von der maximalen Nachrichtengröße ab, die die Peers miteinander austauschen können. Es kann auch eine dynamische Chunk-Große verwendet werden, jedoch wird dadurch das Mapping (vgl. Kapitel 3.4.1) komplexer. Die Chunks werden vom Client zu einem Piece zusammengefügt, woraufhin erst der Hash über das Piece berechnet werden kann. Für die Implementierung sind Nachrichtengrößen, die der Piece-Größe entsprechen, aufgrund geringerer Komplexität zu bevorzugen, da dadurch die Unterteilung in Chunks wegfällt.

3.7 Abläufe

Dieses Kapitel zeigt die konkreten Abläufe zur peerunterstützten Auflösung von Webinhalten. In Abbildung 3.5 wird der Ablauf der Initialisierung und der Fall, dass keine Peers zur Verfügung stehen, gezeigt. Angefangen mit einem Browser der eine Webseite aufruft. Die Webseite gibt ein HTML-Dokument zurück, das die Lösung der peerunterstützten Verteilung einbindet. Wird diese ausgeführt, registriert sich ein Service Worker für die Webseite. In diesem Zustand können allerdings noch keine Anfragen abgefangen werden. Denn wenn zum ersten Mal ein Service Worker registriert wird, werden bis zum Neuladen der Webseite alle Anfragen weiterhin normal aufgelöst. Dieses Verhalten lässt sich durch den Aufruf der Browserfunktion `clients.claim` umgehen und es werden ab sofort alle Anfragen vom Service Worker abgefangen. Stellt der Browser nun eine Anfrage, wird diese abgefangen und beim Koordinator nachgefragt, wie diese Anfrage aufzulösen ist. In diesem Beispiel sind keine Peers verfügbar und das wird dem Browser mitgeteilt. Daraufhin lädt der Browser die Anfrage ganz normal vom Webserver herunter.

Abbildung 3.6 beschreibt nun den Ablauf nach Initialisierung der Lösung zur peerunterstützten Verteilung mit vorhandenen Peers. Angefangen wieder bei der abgefangenen Anfrage im Service Worker. Diese wird beim Koordinator angefragt, wie die Anfrage aufzulösen ist. In diesem Beispiel ist dem Koordinator bereits bekannt, dass die Anfrage für die peerunterstützte Verteilung geeignet ist (vgl. Kapitel 3.3.1) und hat diesen Webinhalt

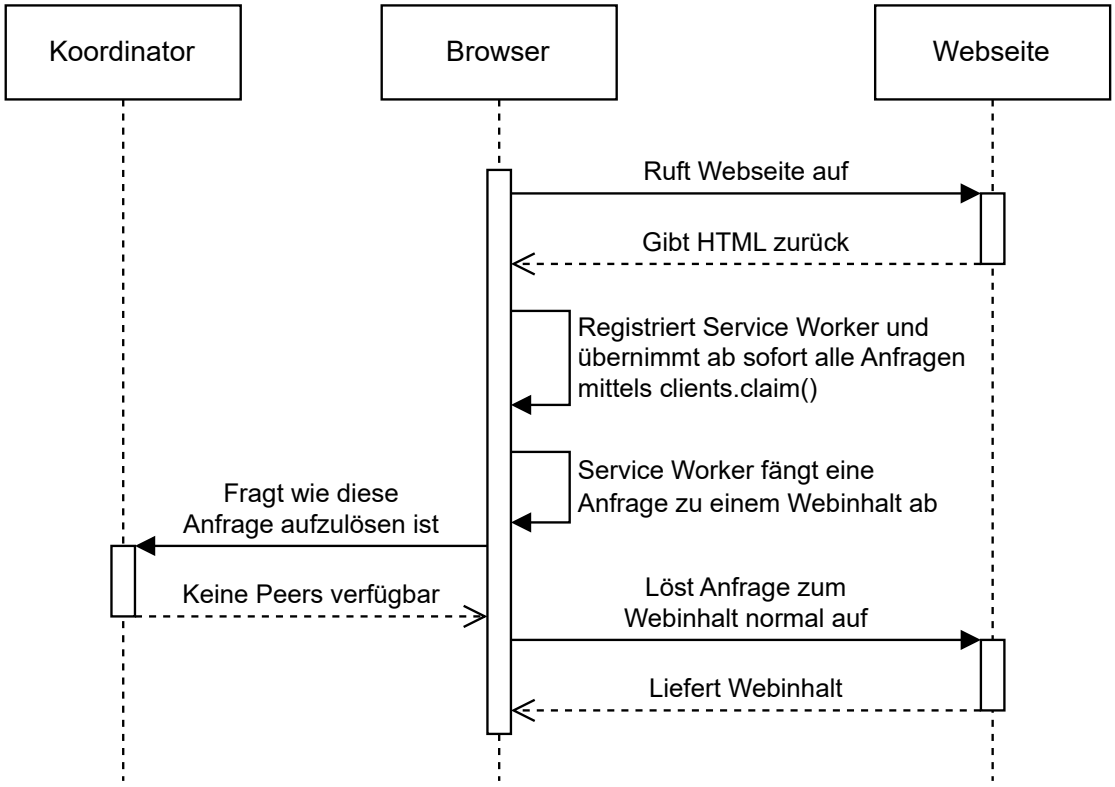

Abbildung 3.5: Ablauf der Initialisierung der peerunterstützten Verteilung ohne verfügbare Peers

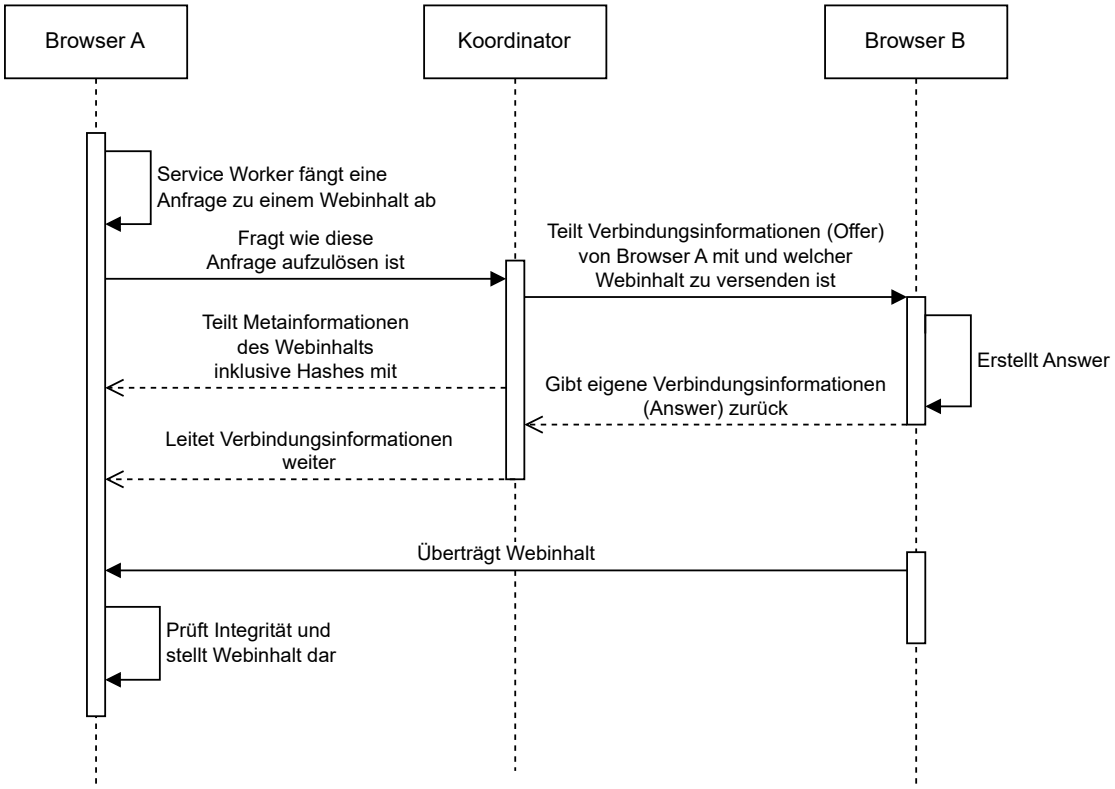

Abbildung 3.6: Ablauf der peerunterstützten Auflösung nachdem eine Anfrage abgefangen wurde und Peers vorhanden sind

bereits heruntergeladen und Hashes berechnet. Da in diesem Fall Peers zum Webinhalt existieren (Browser B), leitet der Koordinator die mitgesendeten Verbindungsinformationen von Browser A zu Browser B weiter (vgl. Kapitel 2.1.2). Außerdem wird Browser B noch mitgeteilt, welchen Webinhalt er Browser A zu übermitteln hat, damit sich beide Peers nicht noch nach dem Verbindungsaufbau absprechen müssen. Diese Funktionsweise spart Paketumlaufzeiten ein und entspricht dem Push-Prinzip [73]. Nach Erhalt der Verbindungsinformationen erstellt Browser B eine dazu spezifische Antwort und sendet sie durch den Koordinator zu Browser A. In der Zwischenzeit hat der Koordinator dem anfragenden Browser A Metainformationen bezüglich des angefragten Webinhalts, inklusive der Hashes, mitgeteilt. Beide Browser versuchen daraufhin eine Direktverbindung, womöglich durch NAT-Systeme, mittels WebRTC aufzubauen. Kam es zu einem erfolgreichen Verbindungsaufbau, übermittelt Browser B unverzüglich den von Browser A angefragten Webinhalt. Während des Empfangs prüft Browser A die Integrität der erhaltenen Teilstücke des Webinhalts mittels denen vom Koordinator mitgeteilten Hashes. Ist die Integritätsprüfung erfolgreich, werden die Teilstücke des Webinhalts nach und nach an den Browser gestreamt.

Dieser Ablauf hat gezeigt, wie ein Webinhalt peerunterstützt aufgelöst werden kann, ohne dass diese Anfrage dem Webserver gestellt werden musste. Webinhalte könnten also peerunterstützt aufgelöst werden, obwohl der ursprüngliche Webserver nicht mehr erreichbar ist.

3.8 Datenschutz

Als Teilnehmer der peerunterstützten Verteilung wird anderen Peers ungefragt die eigene IP-Adresse mitgeteilt und es findet P2P-Datenverkehr statt. Dieser Umstand ist den meisten unbekannt und mit einigen Problemen verbunden, denn man gibt nicht nur unwillentlich die IP-Adresse anderen Peers preis, sondern auch welche Webinhalte man herunterlädt.

3.8.1 Peers gezielt aufsuchen

Aufgrund der Funktionsweise von P2P-Anwendungen ist jedem Peer klar, mit welchem Peer er, hinsichtlich der IP-Adresse, verbunden ist. Diese Information ist in den meisten

Fällen unbrauchbar, um die Identität eines Peers festzustellen. Für die meisten Anwendungsfälle spielt diese Information keine Rolle, da aus einem scheinbar zufälligen Pool von Peers ausgewählt wird. Jedoch lässt sich dieser Pool im Rahmen der peerunterstützten Verteilung sehr genau begrenzen.

Angriffsszenario Möchte man als Angreifer die IP-Adresse eines Bekannten erfahren, kann wie folgt vorgegangen werden: Man besucht eine möglichst populäre Webseite, die die peerunterstützte Verteilung verwendet und die beim Bekannten Vertrauen erweckt. Danach navigiert man auf eine Unterseite, auf der sich hoffentlich gerade niemand anderes befindet, sodass man nur als einziger Peer für die Webseiteninhalte dieser Unterseite infrage kommt. Den Link zur Unterseite lässt man dem Bekannten zukommen. Folgt der Bekannte nun diesem Link und erfüllt dieser die Voraussetzungen für die peerunterstützten Verteilung, erhält der Angreifer kurze Zeit später die IP-Adresse des Bekannten und weiß, dass er dem Link gefolgt ist.

So ein Szenario ist beispielsweise für Bilderhosting-Dienste wie Imgur [74] denkbar, die bekannt sind und somit Vertrauen erwecken als auch vom Einsatz der peerunterstützten Verteilung profitieren würden. Eine Lösung des Problems besteht in der Konfiguration (vgl. Kapitel 3.4.5) des Koordinators, indem Webseiteninhalte erst für die peerunterstützte Verteilung in Betracht gezogen werden, wenn eine bestimmte Anzahl von Peers für diesen Inhalt erreicht sind. Dadurch sinkt die Wahrscheinlichkeit, dass der Angreifer an der peerunterstützten Auflösung des Bekannten beteiligt wird und es steigt die Wahrscheinlichkeit, dass auch andere Peers zu diesem Zeitpunkt diese Webseiteninhalte anfordern. Jedoch wird der Angreifer als Peer präferiert ausgewählt (vgl. Kapitel 3.3.3), wenn er gute Metriken, wie zum Beispiel, wenn er sich in der Nähe vom Bekannten befindet, aufweist.

3.8.2 WebRTC-Leak

Das Surfen im Internet mit aktiviertem JavaScript ist heutzutage zur Normalität geworden. Abgesehen davon, dass viele Webseiten nur noch mit aktiviertem JavaScript funktionieren, ziehen viele nicht in Betracht, welches Datenschutzrisiko sich daraus ergibt. Die WebRTC-Schnittstelle ist eine von vielen Schnittstellen im Browser, die über JavaScript genutzt werden kann, um Informationen über die Identität der Webseitensucher zu sammeln (Stichwort: Fingerabdruck). Dabei ist die WebRTC-Schnittstelle besonders

brisant, da es darüber unter Umständen möglich ist, die eigene öffentliche IP-Adresse des Webseitenbesuchers zu ermitteln, obwohl sich dieser hinter einem VPN [75] oder Proxy [35] verbirgt. Es handelt sich dabei um keinen Fehler in der Schnittstelle im Browser. Viele VPN- und Proxylösungen sind nicht dafür ausgelegt, die Anfragen, die das WebRTC-Framework stellt, umzuleiten. Erweiterungen und Proxyeinstellungen im Browser können meistens nur HTTP-Anfragen umleiten, aber keine Anfragen, die zur Ermittlung der *ICE candidates* gestellt werden (vgl. Kapitel 2.1.2), wodurch die eigene öffentlich IP-Adresse des Webseitenbesuchers offenbart werden kann. Bei VPNs wiederum, kann eine Fehlkonfiguration zu diesem WebRTC-Leak führen. Beispielsweise wenn nur IPv4-Datenverkehr durchs VPN geleitet wird, aber kein IPv6-Datenverkehr.

Zusätzlich zu den öffentlichen IP-Adressen, konnten auch private IP-Adressen der Webseitenbesucher ermittelt werden, die zum Erstellen eines Fingerabdrucks genutzt werden können. Dieses Problem ließ sich allerdings von den Browserentwicklern durch den Einsatz von Multicast DNS (mDNS) [76] lösen. Dabei liefert die WebRTC-Schnittstelle keine privaten IP-Adressen mehr zurück, sondern Multicast DNS Names. Diese Namen sind einzigartige Domains, die der Browser selber erstellt. Im Kontext von mDNS, reagiert der Browser auf diese erstellten Namen im lokalen Netzwerk. Eine solche Domain könnte wie folgt lauten:

2c3f71fb-f7de-48c8-b59a-19967213b4fd.local

Befinden sich zwei Peers im selben Netzwerk und tauschen diese Domains über den Koordinator miteinander aus, kann mithilfe von mDNS eine Verbindung zwischen ihnen hergestellt werden, ohne dass private IP-Adressen ausgetauscht werden müssen. Aufgrund der einzigartigen Domains, die der Browser erstellt, kann dies nicht zum Verfeinern des Fingerabdrucks verwendet werden, wie es bei den privaten IP-Adressen der Fall ist.

Um das Problem der öffentlichen IP-Adressen in den Griff zu bekommen, gingen viele dazu über, die WebRTC-Schnittstelle im Browser zu deaktivieren. Entweder durch Einstellungen, Browsererweiterungen wie Disable WebRTC [77] oder zwischenzeitlich als Standardfunktion im populären Werbeblocker uBlock Origin. Nach Einführung von mDNS wurde die Funktion in uBlock Origin allerdings wieder entfernt [78].

Für die peerunterstützte Verteilung sind diese Bedingungen optimal. Die Verwendung der WebRTC-Schnittstelle ist ohne Vorbedingungen möglich, anders wie es zum Beispiel beim Abspielen von Sounds der Fall ist. Diese Funktion wird in manchen Browsern erst

erlaubt, wenn zuvor mit der Webseite, wie unter anderem durch das Klicken innerhalb der Webseite, interagiert wurde.

3.9 Ethik

Setzt ein Webseitenbetreiber die peerunterstützte Verteilung ein, werden unweigerlich die Ressourcen der Webseitenbesucher verwendet, um die Webseite im Datenverkehr zu entlasten. Zu dieser Praktik kann man verschiedene Meinungen haben. Ärgerlich wird es, wenn man als Peer Webseiteninhalte verteilt und man mit dem Internet über Mobilfunk verbunden ist. Denn dabei können, neben einem erhöhten Stromverbrauch, zusätzliche Kosten entstehen. Zwar kann über die Network-Information-Schnittstelle [44] die Verbindungsart des Browsers ermittelt werden, jedoch unterstützen Firefox und Safari diese Schnittstelle noch nicht. Außerdem kann diese auch falsche Werte liefern, wenn man zum Beispiel über einen Hotspot surft, der mit dem Internet über Mobilfunk verbunden ist.

Wenn es um Ressourcenverbrauch der Webseitenbesucher geht, dann ist Arc [22] die Spitze des Eisbergs. Setzt ein Webseitenbetreiber diese Lösung ein, werden die Ressourcen der Webseitenbesucher an Arc verkauft. Diese nutzt Arc, um bestimmte Webinhalte, für die bezahlt wurde, in ihrem P2P-Netzwerk abzulegen, die die jeweilige Webseite dann peerunterstützt auflösen kann.

Für diese Praktik landete Arc auf einer Blockliste des Ressourcenmissbrauchs [79] im Werbeblocker von uBlock Origin. Mittlerweile wurde Arc wieder aus der Blockliste entfernt, nachdem man der Ausführung von Arc explizit zustimmen musste. Allerdings erfolgt diese Abfrage nur für Webseitenbesucher, die einen Werbeblocker verwenden. Andernfalls wird Arc ohne Zustimmung ausgeführt. Praktischerweise gilt diese Entscheidung auch webseitenübergreifend.

3.10 Future Work

Um den Werdegang der peerunterstützten Verteilung im Web positiv zu beeinflussen, könnten heutzutage Webserver bereits Hashes ihrer Webinhalte über Kopfdaten in ihren Antworten bereitstellen. Dadurch entfällt ein Großteil des Datenverkehrs beim Koordinator, da dieser sämtliche Webinhalte nicht mehr herunterladen muss. Außerdem erlangen

Webserver ihre Autorität zurück, da die Hashes sonst vom jeweiligen Koordinator bereitgestellt werden und dieser dann autoritär über die Webinhalte ist. Einige Cloud-Speicher, wie Backblaze B2 [80], geben bereits Hashes zu ihren Webinhalten zurück. Jedoch handelt es sich nur um einen Hash über den gesamten Webinhalt und nicht um eine Hash-Liste kleinerer Teilstücke, wodurch das Streamen (vgl. Kapitel 3.4.3) erschwert wird.

Eine weitere Idee ist die Entwicklung einer Browsererweiterung, die Webinhalte von sämtlichen Webseiten peerunterstützt auflöst. Mit Firecoral [20] wurde dies bereits probiert, jedoch ist dieser Versuch gescheitert. Durch die Möglichkeiten der heutigen Zeit könnte dies aber anders aussehen. P2P-Netze können heutzutage ohne vorherige Installation von Erweiterungen im Browser aufgebaut werden. Man ist also nicht mehr nur auf die Nutzer angewiesen, die auch Firecoral installiert haben, sondern kann auf bestehende P2P-Netze zugreifen, die Webseitenbetreiber zum Beispiel durch die Einbindung der peerunterstützten Verteilung von Webseiteninhalten erstellt haben.

Die größten Probleme der peerunterstützten Verteilung lassen sich lösen, indem Browser es ermöglichen, WebRTC-Verbindungen im Service Worker aufzubauen. Dadurch würden viele Probleme wegfallen, besonders das Problem des Verbindungsabbruchs zu allen Peers, der auftritt, wenn man durch die Webseite navigiert und es sich um keine SPA [71] handelt.

4 Experimente

Nach der Architekturbeschreibung, die die einfache Verwendung der peerunterstützten Verteilung auf Webseiten zeigt, ist das Ziel der Experimente herauszufinden, mit welchen Geschwindigkeiten man bei der Auflösung von Webinhalten rechnen kann.

4.1 Methodik

Die Experimente werden auf einer eigenen Implementierung der Architekturbeschreibung durchgeführt.

Koordinator Als Grundlage für den Koordinator wurde die Node.js-Laufzeitumgebung [81] verwendet. Programmiert ist der Koordinator in TypeScript [82], das zur Ausführung in der Node.js-Laufzeitumgebung nach JavaScript kompiliert wird. Für die WebSocket-Funktionalität wurde die Bibliothek ws [83] verwendet. Ansonsten wird kein Gebrauch von weiteren Bibliotheken gemacht.

Client-Anwendung Die Client-Anwendung, bestehend aus dem Clientskript und dem Service Worker, ist ebenfalls in TypeScript programmiert und wird zur Ausführung im Browser nach JavaScript kompiliert. Es werden keine Bibliotheken verwendet. Die WebSocket-Verbindung kann über die nativen Browserschnittstellen hergestellt werden.

4.1.1 Testumgebung

Zum automatischen Starten und Stoppen von Browserinstanzen wird Playwright [84] von Microsoft verwendet. Playwright ermöglicht es, verschiedene Browser über eine einheitliche Schnittstelle zu steuern. Dies sind zum Beispiel Google Chrome, Firefox und Safari,

Listing 4.1: Auslesen von Logdaten der Client-Anwendung über das Testskript.

```
async function getLog(page) {
  return await page.evaluate(() => window.Paw)
}
```

die im Headless-Modus betrieben werden. Um Playwright selbst zu steuern, wird die offizielle Node.js-Schnittstelle verwendet. Die einzelnen Experimente werden in JavaScript-Dateien programmiert, die im Weiteren als Testskript bezeichnet werden. Über Node.js lassen sich die Testskripts einfach starten, die daraufhin Browserinstanzen über Playwright fernsteuern.

Als Webserver, der die Lösung zur peerunterstützten Verteilung einbindet und Webinhalte bereitstellt, wurde der Koordinator um einen einfachen HTTP-Server erweitert. Dieser arbeitet wie ein Dateiserver und setzt zusätzlich Cache-Anweisungen in den Kopfdaten der HTTP-Antworten, die eine längere Zwischenspeicherung der Webinhalte erlauben, sodass diese für die peerunterstützte Verteilung genutzt werden können.

Eine Herausforderung in der Versuchsdurchführung besteht darin, die gemessenen Daten der Client-Anwendung aus den Browserinstanzen zu extrahieren. Denn die Browserinstanzen werden im Headless-Modus betrieben, wodurch die Messwerte nicht einfach abgelesen werden können. Die Messwerte jeder Browserinstanz müssen persistiert und organisiert werden. Eine Möglichkeit besteht darin, die Messwerte an den Koordinator zu übermitteln. Denn dieser ist, anders als die Browserinstanzen, imstande, die Messdaten zu persistieren. In einem Fehlerfall kann es allerdings passieren, dass die Messdaten oder auch wichtige Lognachrichten der Client-Anwendung nicht an den Koordinator übertragen werden. Daher wurde eine andere Methode zur Extraktion der Messwerte angewendet. Die Logdaten werden von der Client-Anwendung in einer globalen Variable gesammelt, die an spezifischen Zeitpunkten vom Testskript ausgelesen werden können. Ermöglicht wird dies durch die `evaluate`-Funktion von Playwright, deren Einsatz in Listing 4.1 gezeigt ist. Die Logdaten befinden sich in der globalen Variable `Paw` innerhalb einer Webseiteninstanz. Mit `evaluate` kann eine Funktion im Kontext einer Webseite ausgeführt werden. In diesem Fall besteht die Funktion nur aus dem Zurückgeben der globalen Variable. Diese wird von Playwright serialisiert und dem Testskript zur Verfügung gestellt.

Der Aufbau einer Testumgebung ist für die Entwicklung der Client-Anwendung von sehr großem Vorteil. Normalerweise kann man bei der Entwicklung von Anwendungen, die Anwendung neu starten, um Änderungen zu testen. Das ist bei der Entwicklung des Service Workers leider nicht der Fall. Dieser persistiert im Browser und ein Neuladen der Webseite führt nicht immer zur Aktualisierung des Service Workers. Der Browser ermittelt zu verschiedenen Zeitpunkten, ob eine neue Version des Service Workers existiert und versucht gegebenenfalls ein Update des Service Workers durchzuführen. Dies ist aber nicht immer möglich, wenn zum Beispiel noch Verbindungen zum Koordinator gehalten werden oder ausstehende abgefangene HTTP-Anfragen beantwortet werden müssen. In diesen Fällen kann der Service Worker, selbst manuell über die Entwicklungsumgebung der Browser, nicht einfach beendet werden. Das bedeutet, dass der Service Worker quasi zu zufälligen Zeitpunkten aktualisiert wird. Dieser Umstand führt zu einem erheblich höherem Entwicklungsaufwand, wenn man einen Service Worker mit den neuen Änderungen erwartet hat.

Testet man stattdessen über eine Browser-Automatisierungssoftware wie Playwright, können Browser gestartet und gestoppt werden, die beim erneuten Start über keinen persistenten Service Worker mehr verfügen.

4.1.2 Versuchsaufbau

Vor jeder Versuchsdurchführung wird der Koordinator gestartet. In diesem Zustand hat dieser keine Kenntnis über Webinhalte, sodass Hashwerte und die Tauglichkeit zur peerunterstützten Verteilung zu den Webinhalten noch ermittelt werden müssen. Für die Experimente ist dieser Ablauf uninteressant, weshalb dem Koordinator im Vorfeld die Anfragen mitgeteilt werden, sodass dieser bereits über Hashwerte und Tauglichkeit der Webinhalte verfügt.

Die Browserinstanzen, die von den Testskripts gestartet werden, navigieren vor jeder Versuchsdurchführung auf eine spezielle Unterseite, auf der zunächst nur der Service Worker registriert wird. Der Grund dafür liegt an der späten Registrierung des Service Workers, wodurch Webinhalte beim ersten Besuch einer Webseite unter Umständen noch ohne Peerunterstützung aufgelöst werden. Denn der Browser hat zunächst keine Kenntnis von einem Service Worker. Dieser wird erst registriert, wenn das eingebundene Clientskript heruntergeladen und ausgeführt wurde. In dieser Zeit löst der Browser Webinhalte

normal auf und deswegen kann die peerunterstützte Verteilung nicht zuverlässig in den Experimenten getestet werden.

Wurde der Service Worker auf der Unterseite erfolgreich registriert, wird das dem Testskript signalisiert. Das Ereignis dafür erhält das Skript, welches den Service Worker registriert. Dieses verändert daraufhin das DOM der Unterseite und legt ein `h1`-Element an. Playwright verfügt über die praktische `waitForSelector`-Funktion, mit der so lange gewartet werden kann, bis das `h1`-Element existiert. So wird dem Testskript mitgeteilt, dass der Service Worker registriert wurde. Das Testskript leitet den Browser daraufhin zum eigentlichen Experiment weiter.

Bei allen Versuchsdurchführungen handelt es sich um lokale Browserinstanzen. Die Experimente werden also unter idealen Bedingungen durchgeführt. Dabei werden Firefox Version 115 und Chromium Version 116 verwendet.

Das Problem der Erkennung aktiver Peers (vgl. Kapitel 3.5.2) wurde gelöst, indem die Browserinstanzen die Webseiten in den Experimenten nicht neu laden und der Service Worker mit ständigen Nachrichten aktiv gehalten wird. Dadurch besitzen das Clientskript und der Service Worker in etwa die gleiche Lebensdauer.

In allen Versuchsdurchführungen wird mit einer Nachrichtengröße von 256 KiB gearbeitet, die die Browser mittlerweile unterstützen. Dies entspricht der Piece-Größe, wodurch keine Unterteilung in Chunks nötig ist, die der Browser wieder zusammenfügen muss.

4.2 Ergebnisse

4.2.1 Verzögerungen

In diesem Experiment wird ermittelt, mit welchen zusätzlichen Verzögerungen bei der Auflösung von Webinhalten zu rechnen ist, wenn die peerunterstützte Verteilung eingesetzt wird. Dazu werden zwei Testwebseiten erstellt, die ein rund 2 MiB großes Bild über das `img`-Element einbinden, wovon eine zusätzlich die Client-Anwendung zur peerunterstützten Verteilung einbindet.

Die Auflösungszeit wird gemessen am `load`-Ereignis, das der Browser auslöst, wenn alle Webseiteninhalte geladen und ausgeführt wurden. Bei der peerunterstützten Verteilung schließt es das Laden und Ausführen der Client-Anwendung mit ein und natürlich die

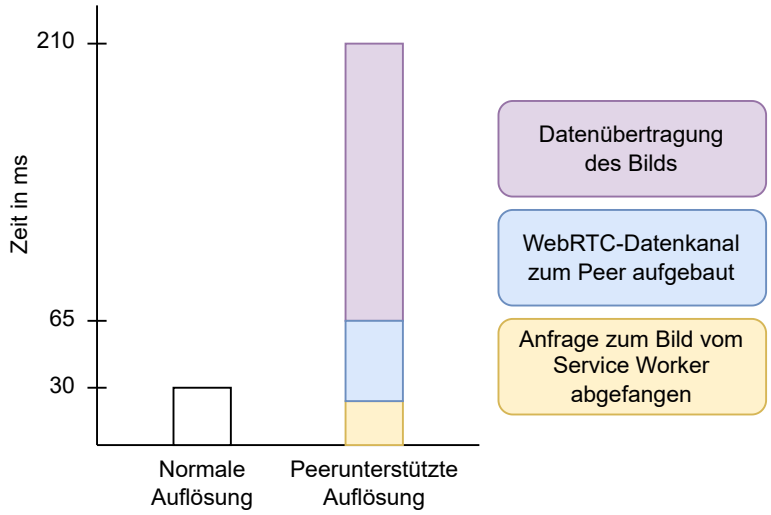

Abbildung 4.1: Zeit bis zum load-Ereignis der Testwebseiten

benötigte Zeit, um den Webinhalt über einen Peer aufzulösen. Die Zeit zur Registrierung des Service Workers wird im `load`-Ereignis aber nicht erfasst, weil wie zuvor erwähnt, der Service Worker auf einer Vorseite bereits registriert wurde.

Zur Feststellung des Startzeitpunkts, um die Differenz zum `load`-Ereignis zu berechnen, wird die Performance-Schnittstelle [85] im Browser verwendet. Diese erfasst unter anderem sämtliche Zeitpunkte des Lebenszyklus eines Navigation-Ereignisses im Browser und stellt die Zeit bis zum `load`-Ereignis auch relativ zum Startzeitpunkt der Navigation dar.

Für den peerunterstützten Versuchsdurchlauf steht bereits ein Peer bereit, der den Webinhalt dem anfragenden Browser übermittelt. Das Experiment wird unter optimalen Voraussetzungen, in einer lokalen Umgebung inklusive Browser, Webserver und Koordinator, durchgeführt.

Abbildung 4.1 stellt beide Auflösungszeiten dar und schlüsselt wichtige Ereignisse der peerunterstützten Verteilung weiter auf. Man erkennt bei der normalen Auflösung, dass selbst die einfache Darstellung eines Bilds im Browser in einer lokalen Umgebung relativ viel Zeit in Anspruch nimmt. Bis das Bild peerunterstützt aufgelöst wurde, sind 210 ms vergangen. Das ist rund 7-mal länger als bei der normalen Auflösung. Man darf aber nicht vergessen, dass Datenverkehr zum Webserver eingespart wurde und die Webinhalte weiterhin noch peerunterstützt aufgelöst werden können, selbst wenn der Webserver nicht mehr erreichbar ist.

4.2.2 Übertragungsgeschwindigkeit

Um nicht nur die Latenzen sondern auch die Übertragungsgeschwindigkeit testen zu können, wird in diesem Experiment eine rund 100 MiB große Videodatei peerunterstützt aufgelöst. Wieder von einem Peer, der das Video bereitstellt.

Das Experiment wird in vier Konstellationen durchgeführt, um Browserunterschiede zwischen Firefox und Chromium in der Implementierung feststellen zu können. Diese Konstellationen beinhalten die Versuchsdurchführungen von Firefox zu Firefox, von Chromium zu Chromium sowie zwischen Firefox und Chromium in beide Richtungen.

Die Übertragungsgeschwindigkeiten werden durch den zeitlichen Abstand der erhaltenen Chunks beim Empfänger ermittelt.

Abbildung 4.2: Empfangsrate von einem rund 100 MiB großen Video

Abbildung 4.2 zeigt die Datenübertragungsrate über Zeit bis zur vollständigen Auflösung des rund 100 MiB großen Videos zwischen zwei Chromium-Instanzen. Es wird eine Datenübertragungsrate von bis zu 25 MB/s erreicht und das Video wird in ca. 6 s aufgelöst. Die durchschnittliche Datenübertragungsrate beträgt knapp 18 MB/s. Für eine lokale Versuchsumgebung ist das ein sehr geringer Wert.

In Abbildung 4.3 sind die durchschnittlichen Datenübertragungsraten über fünf Durchgänge aller Konstellationen gezeigt. Die Datenübertragungsraten unterscheiden sich je nach verwendetem Browser und Konstellation kaum. Auch in diesem Experiment werden nur sehr geringe Werte erreicht.

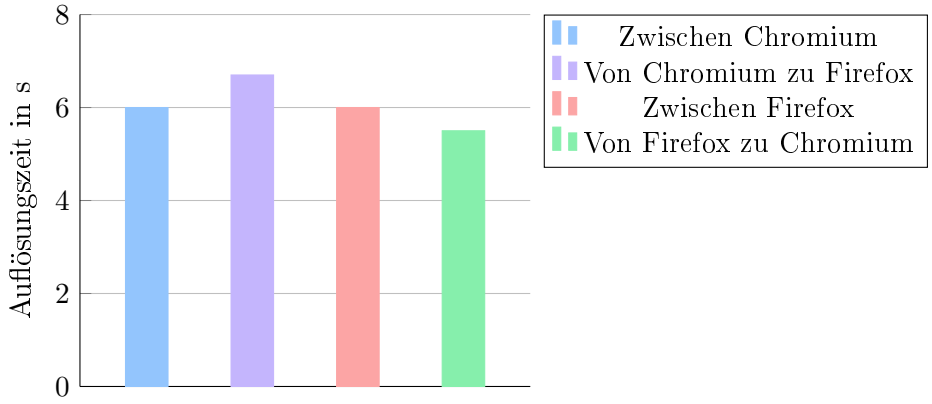

Abbildung 4.3: Auflösungszeiten eines rund 100 MiB großen Videos zwischen verschiedenen Browsern

4.2.3 Peer-Anzahl

Da im vorherigen Experiment nur sehr geringe Datenübertragungsraten gemessen wurden, wird in diesem Experiment untersucht, wie sich die Datenübertragungsraten verändern, wenn das rund 100 MiB große Video von mehreren Peers aufgelöst wird.

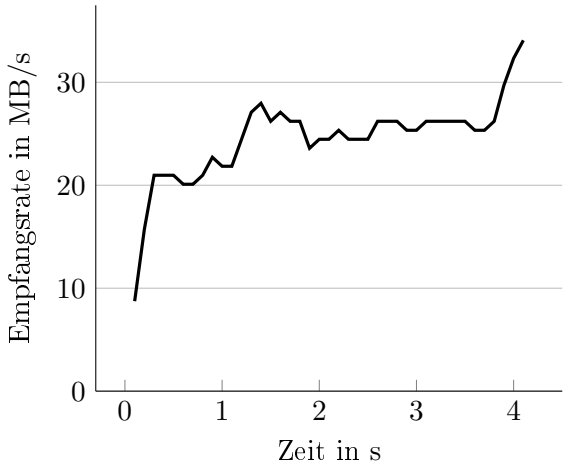

Abbildung 4.4: Empfangsrate von einem rund 100 MiB großen Video durch zwei Peers

Dazu teilt der Koordinator je nach Peer-Anzahl jedem Peer einen bestimmten Bereich zu, den er dem anfragenden Peer zu übermitteln hat. So wird beispielsweise bei zwei Peers der erste angewiesen, die erste Hälfte aller Chunks vom Webinhalt zu übermitteln, und der zweite Peer die andere Hälfte.

In diesem Experiment reduzierte sich die Auflösungszeit 6 s auf ca. 4,4 s, wenn das Video von zwei Peers heruntergeladen wurde. Die Datenübertragungsrate ist in Abbildung 4.4 gezeigt. Wurden mehr als zwei Peers verwendet, erhöhte sich die Auflösungszeit wieder.

Eine Erklärung dafür ist, dass das Senden und Empfangen von Daten auf dem WebRTC-Datenkanal mit erheblicher Prozessorlast verbunden ist. Zu dieser Schlussfolgerung kam auch eine weitere Evaluierung [86], bei der ähnliche Datenübertragungsraten von rund 19 MB/s erreicht wurden.

4.2.4 Blob vs. ArrayBuffer

Bisher wurden alle Experimente mit dem Datentyp `ArrayBuffer` (vgl. Kapitel 3.4.6) durchgeführt. Firefox unterstützt aber auch den Datentyp `Blob`, der auf den WebRTC-Datenkanälen versendet und empfangen werden kann.

In diesem Experiment wird untersucht, ob eine höhere Übertragungsgeschwindigkeit werden kann, wenn mit dem Datentyp `Blob` gearbeitet wird, um so die vermeintliche Prozessorbelastung zu verringern. Dazu steht wieder ein Peer bereit, der das rund 100 MiB große Video bereitstellt. Dieses mal handelt es sich aber um einen Firefox-Peer und für die Versuchsdurchführung wird auch eine Firefox-Instanz verwendet.

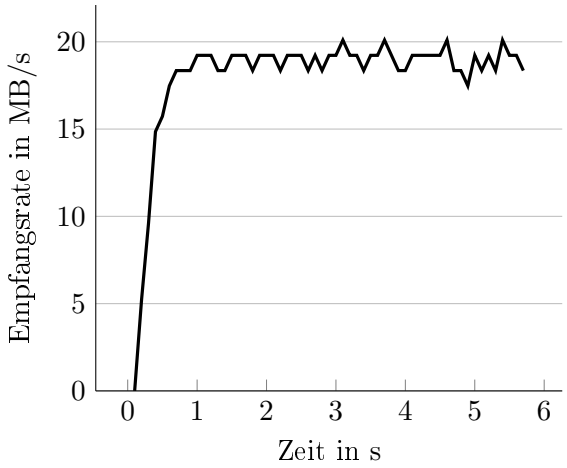

Abbildung 4.5: Empfangsrate von einem rund 100 MiB großen Video mit dem Blob-Datentyp zwischen zwei Firefox-Instanzen

Anhand der Datenübertragungsrate in Abbildung 4.5 sieht man, dass die Verwendung des Blob-Datentyps die Datenübertragungsrate nicht erhöhen konnte. Das Video wurde

wieder in rund 6 s aufgelöst. Allerdings erkannt man an diesem Graphen einen Unterschied in der Übertragung im Vergleich zu Chromium. In allen Versuchsdurchläufen fiel die Datenübertragungsrate in Chromium nach kurzer Zeit massiv ab, bevor Datenübertragungsraten von bis zu 25 MB/s erreichen wurden. Dieses Phänomen konnte in Firefox nicht beobachtet werden und es wurden nach kurzer Zeit relativ konstant knapp 20 MB/s erreicht.

4.2.5 Größe des Sendepuffers

Bezüglich der Flusskontrolle (vgl. Kapitel 3.4.2) wurde für die `bufferedAmountLowThreshold`-Eigenschaft bisher der Wert der Piece-Größe verwendet. Fällt diese zu niedrig aus, kann der Sendepuffer leerlaufen und dadurch die Datenübertragungsrate verringern. Um diesen Fall vorzubeugen, wird in diesem Experiment untersucht, ob sich die Datenübertragungsrate bei höheren Schwellenwerten erhöht.

Dazu wurde wieder das rund 100 MiB große Video von einer Chromium-Instanz zur Anderen übertragen. Eine höhere Datenübertragungsrate konnte mit höheren Schwellenwerten auch nicht erzielt werden und die Auflösungszeit betrug weiterhin rund 6 s. Dabei wurden Schwellenwerte bis zu 20-facher Piece-Größe verwendet, was einem Sendepuffer von 5 MiB entspricht.

5 Schlussfolgerungen

In dieser Masterarbeit wurde eine Lösung zur peerunterstützten Verteilung von Webseiteninhalten vorgestellt, implementiert und getestet. Folgende Erkenntnisse ließen sich dabei gewinnen:

1. Es wurde gezeigt, dass die Lösung zur peerunterstützten Verteilung von Webseiteninhalten einfach auf vorhandenen Webseiten eingebunden werden kann.
2. Webinhalte ließen sich peerunterstützt auflösen. Sind aber in der lokalen Testumgebung mit höheren Auflösungszeiten verbunden gewesen.
3. Datenverkehr zum Auflösen der Webinhalte konnte beim Webserver eingespart werden. Theoretisch sind Einsparungen von bis zu 100 % minus zwei Übertragungen möglich. Eine Übertragung zum Koordinator zur Berechnung der Hashes und eine zum ersten Peer.
4. Es wurde experimentell nachgewiesen, dass nur relativ geringe Übertragungsraten auf dem WebRTC-Datenkanal zu erreichen sind.

Für den produktiven Einsatz steht die Lösung allerdings immer noch vor einigen Herausforderungen. Der Datenverkehr von Mobilgeräten im Internet steigt immer weiter an und liegt laut Cloudflare Radar für Deutschland schon bei ca. 50 % [87]. Diese Geräte sind aus vielen Gründen nicht für die peerunterstützte Verteilung geeignet und reduzieren so den Pool an möglichen Peers.

Ein weiteres großes Problem für den breiten Einsatz auf Webseiten ist, dass alle Peer-Verbindungen beim Navigieren auf der Webseite abgebrochen werden. Daher macht der Einsatz nur für Single-Page-Webanwendungen oder Streaming-Webseiten Sinn. Auf Streaming-Webseiten konsumieren Webseitenbesucher längere Zeit einen Stream, wodurch Peer-Verbindungen länger gehalten werden können.

Literaturverzeichnis

[1] Bianca Galvez. What is an enterprise content delivery network (ecdn)? URL https://vimeo.com/blog/post/what-is-a-content-delivery-network/.

[2] Francesco Altomare. What is a content delivery network? cdn explained. URL https://www.globaldots.com/resources/blog/content-delivery-network-explained/.

[3] Cullen Jennings, Florent Castelli, Henrik Boström, and Jan-Ivar Bruaroey. WebRTC: Real-time communication in browsers, March 2023. URL https://www.w3.org/TR/2023/REC-webrtc-20230306/.

[4] Justin Uberti. Supporting new media experiences on the web. URL https://blog.chromium.org/2012/10/supporting-new-media-experiences-on-web.html.

[5] J. Postel. User Datagram Protocol. RFC 768, August 1980. URL https://www.rfc-editor.org/info/rfc768.

[6] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012. URL https://www.rfc-editor.org/info/rfc6347.

[7] Elisabetta Carrara, Karl Norrman, David McGrew, Mats Naslund, and Mark Baugher. The Secure Real-time Transport Protocol (SRTP). RFC 3711, March 2004. URL https://www.rfc-editor.org/info/rfc3711.

[8] Randall R. Stewart, Michael Tüxen, and karen Nielsen. Stream Control Transmission Protocol. RFC 9260, June 2022. URL https://www.rfc-editor.org/info/rfc9260.

[9] Hubert Zimmermann. Osi reference model-the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.

[10] Michael Tüxen and Randall R. Stewart. UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication. RFC 6951, May 2013. URL https://www.rfc-editor.org/info/rfc6951.

[11] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445, July 2018. URL https://www.rfc-editor.org/info/rfc8445.

[12] Philip Matthews, Jonathan Rosenberg, Dan Wing, and Rohan Mahy. Session Traversal Utilities for NAT (STUN). RFC 5389, October 2008. URL https://www.rfc-editor.org/info/rfc5389.

[13] Tirumaleswar Reddy.K, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 8656, February 2020. URL https://www.rfc-editor.org/info/rfc8656.

[14] Matt Holdrege and Pyda Srisuresh. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, August 1999. URL https://www.rfc-editor.org/info/rfc2663.

[15] Brad White and Andrew Robertson. Available pool of unallocated ipv4 internet addresses now completely emptied. Technical report, Tech. rep. Washington, DC: The Internet Corporation for Assigned Names and Numbers (ICANN), 2011. URL https://itp.cdn.icann.org/en/files/announcements/release-03feb11-en.pdf.

[16] Tony L. Hain. Architectural Implications of NAT. RFC 2993, November 2000. URL https://www.rfc-editor.org/info/rfc2993.

[17] Ali C. Begen, Paul Kyzivat, Colin Perkins, and Mark J. Handley. SDP: Session Description Protocol. RFC 8866, January 2021. URL https://www.rfc-editor.org/info/rfc8866.

[18] Jake Archibald and Marijn Kruisselbrink. Service workers, July 2022. https://www.w3.org/TR/2022/CRD-service-workers-20220712/.

[19] Stefan Koch. Javascript: Einführung, programmierung und referenz. dpunkt-verlag, 6, 2011.

[20] Jeff Terrace, Harold Laidlaw, Hao Eric Liu, Sean Stern, and Michael J Freedman. Bringing p2p to the web: security and privacy in the firecoral network. In *IPTPS*, page 7, 2009.

[21] Liang Zhang, Fangfei Zhou, Alan Mislove, and Ravi Sundaram. Maygh: Building a cdn from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 281–294, 2013.

[22] Arc. Arc - the peer-to-peer cdn that pays you. URL https://arc.io/.

[23] Viblast. Peer-to-peer advanced video delivery for live hd streaming. URL https://viblast.com/.

[24] Strive Media GmbH. Scalable enterprise video delivery. URL https://strivecast.com/.

[25] Lumen Technologies. Mesh delivery. URL https://www.lumen.com/en-us/edge-cloud/mesh-delivery.html.

[26] Feross Aboukhadijeh. Webtorrent - streaming browser torrent client. URL https://webtorrent.io/.

[27] Bram Cohen. The bittorrent protocol specification. URL https://www.bittorrent.org/beps/bep_0003.html.

[28] Carlos Guerrero. Cachep2p. URL https://web.archive.org/web/20170214195200/http://www.cachep2p.com/.

[29] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Semantics. RFC 9110, June 2022. URL https://www.rfc-editor.org/info/rfc9110.

[30] Anne van Kesteren, Aryeh Gregor, and Ms2ger. Dom standard. URL https://dom.spec.whatwg.org/.

[31] Adam Barth. The Web Origin Concept. RFC 6454, December 2011. URL https://www.rfc-editor.org/info/rfc6454.

[32] Shashank Gupta and Brij Bhooshan Gupta. Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8:512–530, 2017.

[33] Alexey Melnikov and Ian Fette. The WebSocket Protocol. RFC 6455, December 2011. URL https://www.rfc-editor.org/info/rfc6455.

[34] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Caching. RFC 9111, June 2022. URL https://www.rfc-editor.org/info/rfc9111.

[35] Ari Luotonen and Kevin Altis. World-wide web proxies. *Computer Networks and ISDN systems*, 27(2):147–154, 1994.

[36] Andrea Vattani, Flavio Chierichetti, and Keegan Lowenstein. Optimal probabilistic cache stampede prevention. *Proceedings of the VLDB Endowment*, 8(8):886–897, 2015.

[37] Robert Hansen, John Kinsella, and Hugo Gonzalez. Slowloris http dos. URL https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris.

[38] Vassilis Moustakas, Hüseyin Akcan, Mema Roussopoulos, and Alex Delis. Alleviating the topology mismatch problem in distributed overlay networks: A survey. *Journal of Systems and Software*, 113:216–245, 2016.

[39] Matthew Luckie, Young Hyun, and Bradley Huffaker. Traceroute probe method and forward ip path inference. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 311–324, 2008.

[40] Wei Li, Shanzhi Chen, and Tao Yu. Utaps: An underlying topology-aware peer selection algorithm in bittorrent. In *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, pages 539–545. IEEE, 2008.

[41] Fenglin Qin, Ju Liu, Lina Zheng, and Liansheng Ge. An effective network-aware peer selection algorithm in bittorrent. In *2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 418–421. IEEE, 2009.

[42] Margaret Rouse. Hop count, . URL https://www.techopedia.com/definition/26127/hop-count.

[43] L Rizo-Dominguez, D Torres-Roman, D Munoz-Rodriguez, and C Vargas-Rosales. Jitter in ip networks: a cauchy approach. *IEEE Communications Letters*, 14(2): 190–192, 2010.

[44] Ilya Grigorik. Network information api. URL https://wicg.github.io/netinfo/.

[45] Debajyoti Halder, Prashant Kumar, Saksham Bhushan, and Anand M Baswade. fybrrstream: A webrtc based efficient and scalable p2p live streaming platform. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2021.

[46] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017. URL https://www.rfc-editor.org/info/rfc8259.

[47] K-J Grinnemo, Torbjörn Andersson, and Anna Brunstrom. Performance benefits of avoiding head-of-line blocking in sctp. In *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services-(icas-isns' 05)*, pages 44–44. IEEE, 2005.

[48] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, August 2022. URL https://www.rfc-editor.org/info/rfc9293.

[49] Marijn Kruisselbrink. File API, February 2023. URL https://www.w3.org/TR/2023/WD-FileAPI-20230206/.

[50] Adam Rice, Domenic Denicola, Mattias Buelens, and Takeshi Yoshino. Streams standard. URL https://streams.spec.whatwg.org/.

[51] MDN. Transferable objects. URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Transferable_objects.

[52] Ned Freed and Dr. Nathaniel S. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046, November 1996. URL https://www.rfc-editor.org/info/rfc2046.

[53] Lennart Grahl. Demystifying webrtc's data channel message size limitations. URL https://lgrahl.de/articles/demystifying-webrtc-dc-size-limit.html.

[54] Dr. Steve E. Deering and Jeffrey Mogul. Path MTU discovery. RFC 1191, November 1990. URL https://www.rfc-editor.org/info/rfc1191.

[55] Randall R. Stewart, Michael Tüxen, Salvatore Loreto, and Robin Seggelmann. Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol. RFC 8260, November 2017. URL https://www.rfc-editor.org/info/rfc8260.

[56] Google. Support ndata for sctp, . URL https://bugs.chromium.org/p/webrtc/issues/detail?id=5696.

[57] Mozilla. Enable sctp ndata to prevent data channel send queue monopolisation, . URL https://bugzilla.mozilla.org/show_bug.cgi?id=1381145.

[58] Google. Support sending blob over datachannel, . URL https://bugs.chromium.org/p/webrtc/issues/detail?id=2276.

[59] Mozilla. Firefox 113 for developers, . URL https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/113.

[60] Ben Goodger. Welcome to chromium. URL https://blog.chromium.org/2008/09/welcome-to-chromium_02.html.

[61] ECMA International. Standard ecma-262 - ecmascript language specification. URL https://tc39.es/ecma262/.

[62] Mark Watson. Web cryptography API, January 2017. URL https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/.

[63] Jake Archibald. You can't make an insecure thing secure if the implementation is shipped insecurely. here's why. URL https://gist.github.com/jakearchibald/c4297f4191eb60484a6a14f5f5e5ea64.

[64] Subodh Gangan. A review of man-in-the-middle attacks. *arXiv preprint arXiv:1504.02115*, 2015.

[65] pfeffervince. Vince auf twitter. URL https://twitter.com/pfeffervince/status/1229399659774382082.

[66] GitHub. Should webrtc be [securecontext], . URL https://github.com/w3ctag/design-reviews/issues/228.

[67] GitHub. Add support for webrtc data channel in workers, . URL https://github.com/w3c/webrtc-pc/issues/230.

[68] Raymond Hill. ublock origin. URL https://github.com/gorhill/uBlock/.

[69] Philip Walton and Barry Pollard. Back/forward cache. URL https://web.dev/bfcache/.

[70] Ian Hickson, Simon Pieters, Anne van Kesteren, Philip Jägenstedt, and Domenic Denicola. Html standard. URL https://html.spec.whatwg.org/.

[71] Madhuri A Jadhav, Balkrishna R Sawant, and Anushree Deshmukh. Single page application using angularjs. *International Journal of Computer Science and Information Technologies*, 6(3):2876–2879, 2015.

[72] James H Burrows. Secure hash standard. *Federal information processing standards publication*, pages 180–1, 1995.

[73] Margaret Rouse. Push technology, . URL https://www.techopedia.com/definition/5732/push-technology.

[74] Alan Schaaf. Imgur. URL https://imgur.com/.

[75] Manfred Lipp. *VPN-virtuelle private Netzwerke: Aufbau und Sicherheit*. Pearson Deutschland GmbH, 2006.

[76] Stuart Cheshire and Marc Krochmal. Multicast DNS. RFC 6762, February 2013. URL https://www.rfc-editor.org/info/rfc6762.

[77] Chris Antaki. Disable webrtc. URL https://addons.mozilla.org/en-US/firefox/addon/happy-bonobo-disable-webrtc/.

[78] GitHub. Remove webrtc leak prevention, . URL https://github.com/uBlockOrigin/uBlock-issues/issues/1723.

[79] GitHub. Remove arc.io from the resource abuse list, . URL https://github.com/uBlockOrigin/uAssets/pull/8874.

[80] Backblaze. B2 cloud storage. URL https://www.backblaze.com/cloud-storage.

[81] OpenJS Foundation. Node.js. URL https://nodejs.org/.

[82] Microsoft. Typescript: Javascript with syntax for types, . URL https://www.typescriptlang.org/.

[83] ws: a node.js websocket library. URL https://www.npmjs.com/package/ws.

[84] Microsoft. Fast and reliable end-to-end testing for modern web apps | playwright, . URL https://playwright.dev/.

[85] Noam Rosenthal and Yoav Weiss. Navigation timing level 2, June 2023. https://www.w3.org/TR/2023/WD-navigation-timing-2-20230607/.

[86] Rasmus Eskola and Jukka K Nurminen. Performance evaluation of webrtc data channels. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 676–680. IEEE, 2015.

[87] Cloudflare. Cloudflare radar. URL https://radar.cloudflare.com/de.

A Anhang

A.1 Quelltext

Der Quelltext der entwickelten Lösung und der Testumgebung samt Testskripts befinden sich auf der beigelegten CD im ZIP-Archiv `Quelltext.zip`.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original